TIMING ANALYSIS IN EXISTING AND EMERGING CYBER PHYSICAL SYSTEMS

BY

JUNG-EUN KIM

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2017

Urbana, Illinois

Doctoral Committee:

Professor Lui Sha, Chair/Director of Dissertation
Professor Tarek Abdelzaher
Professor Alex Kirlik
Doctor Richard Bradford, Rockwell Collins, Inc.

Abstract

A main mission of safety-critical cyber-physical systems is to guarantee timing correctness. The examples of safety-critical systems are avionic, automotive or medical systems in which timing violations could have disastrous effects, from loss of human life to damage to machines and/or the environment.

Over the past decade, multicore processors have become increasingly common for their potential of efficiency, which has made new single-core processors become relatively scarce. As a result, it has created a pressing need to transition to multicore processors. However, existing safety-critical software that has been certified on single-core processors is not allowed to be fielded on a multicore system as is. The issue stems from, namely, serious inter-core interference problems on shared resources in current multicore processors, which create non-deterministic timing behavior. Since meeting the timing constraints is the crucial requirement of safety-critical real-time systems, the use of more than one core in a multicore chip is currently not certified yet by the authorities. Academia has paid relatively little attention to non-determinism due to uncoordinated I/O communications, as compared with other resources such as cache or memory, although industry considers it as one of the most troublesome challenges. Hence we focused on I/O synchronization, requiring no information of Worst Case Execution Time (WCET) that can get impacted by other interference sources. Traditionally, a two-level scheduling, such as Integrated Modular Avionics system (IMA), has been used for providing temporal isolation capability. However, such hierarchical approaches introduce significant priority inversions across applications, especially in multicore systems, ultimately leading to lower system utilization. To address these issues, we have proposed a novel scheduling mechanism called budgeted generalized rate monotonic analysis (Budgeted GRMS) in which different applications' tasks are globally scheduled for avoiding unnecessary priority inversions, yet the CPU resource is still partitioned for temporal isolation among applications. Incorporating the issues of no information of WCETs and I/O synchronization, this new scheduling paradigm enables the "safe" use of multicore processors in safety-critical real-time systems.

Recently, newly emerging Internet of Things (IoT) and Smart City applications are becoming a part of cyber-physical systems, as the needs are required and the feasibility are getting visible. What we need to pay attention to is that the promises and challenges arising from IoT and Smart City applications are providing new research landscapes and opportunities and fundamentally transforming real-time scheduling. As mentioned earlier, in traditional real-time

systems, an instance of a program execution (a process) is described as a scheduling entity, while, in the emerging applications, the fundamental schedulable units are chunks of data transported over communication media. Another transformation is that, in IoT and Smart City applications, there are multiple options and combinations to choose to utilize and schedule since there are massively deployed heterogeneous kinds of sensing devices. This is contrary to the existing real-time work which is given a fixed task set to be analyzed. For that reason, they also suggest variants of performance or quality optimization problems.

Suppose a disaster response infrastructure in a troubled area to ensure safety of humanitarian missions. Cameras and other sensors are deployed along key routes to monitor local conditions, but turned off by default and turned on on-demand to save limited battery life. To determine a safe route to deliver humanitarian shipments, a decision-maker must collect reconnaissance information and schedule the data items to support timely decision-making. Such data items acquired from the time-evolving physical world are in general time-sensitive - a retrieved item may become stale and no longer be accurate/relevant as conditions in the physical environment change. Therefore, "when to acquire" affects the performance and correctness of such applications and thus the overall system safety and data timeliness should be carefully considered. For the addressed problem, we explored various algorithmic options for maximizing quality of information, and developed the optimal algorithm for the order of retrievals of data items to make multiple decisions. I believe this is a significant initial step toward expanding timing-safety research landscapes and opportunities in the emerging CPS area.

This dissertation is dedicated

to my God,

to my father, Kwang-Ho Kim, my mother, Min-Kyung Kim, and my sister, Ji-Eun Kim, and

to my partner and colleague, Man-Ki Yoon.

Acknowledgments

First of all, I have been fortunate enough to be advised by my adviser, Professor Lui Raymond Sha. I would like to deliver my deepest gratitude to him. Without his support and patience, I could have neither started nor finished this long journey. I realize everyday how pioneering his vision is and how right his direction is, and how precious his advising and teaching have been. I would like to say that anything good is thanks to him and anything not good is because of me in this dissertation and my work. I hope he will be there forever as my mentor who I can rely on and respect as he is.

I appreciate the environment that I could work with Professor Lui Sha and also with Professor Tarek Abdelzaher. Working with Professor Tarek Abdelzaher has been just joy. His encouraging words led me to enjoy research in good times and bad times. Also, his passion for research gave me every time a fresh inspiration. I sincerely appreciate that he presented me more chances to work and guided me toward newer and exciting research opportunities. I believe exiting more and more chances to work with him will keep coming for worthwhile fruit.

I also deeply indebted to Doctor Richard Bradford for his endless supporting. I am grateful to God for that I have met Doctor Richard Bradford, worked with him and shared many good things and moments. He always helped me realize what I cannot see. Hence, I was always looking forward to the next meeting and talking with him. He is one of my role models intellectually and personality-wise, hence working with him let me try to improve myself to be a better researcher and person. I hope he would keep working with me so I can keep striving.

Lastly, I would like to say that I appreciate Professor Alex Kirlik's generous help for my doctoral path and my forward steps. Also, I am thankful to him for improving the quality of this dissertation. I hope to have the opportunity to interact and collaborate with him in the future.

Therefore, I emphasize again that this dissertation has been completed by being indebted by those people and others that I could not explicitly mention here.

Table of Contents

Part I

Timing Analysis in Existing CPS

Chapter 1

Introduction

Over the last decade, multicore processors have become increasingly common for their efficiency, which has made new single-core processors relatively scarce. This trend has created a pressing need to transition to multicore processors. However, existing previously-certified software for safety-critical applications underwent rigorous certification processes on single-core processors, and in the case of the Federal Aviation Administration (FAA) is not allowed to be fielded on a current-generation multicore system except under extremely restrictive conditions. System designers and providers wish to avoid costly recertification, making the certification of safety-critical software for multicore processors a problem of very significant interest to industry.

The issue stems from serious inter-core interference problems on shared resources. This has not happened on separate single-core chips, creating non-deterministic timing behavior. To address the issue, we have conducted underlying research of a scheduling approach for interference isolation among the tasks in a multicore system in [Yoon et al., 2011]. Later on, we have focused on exclusive I/O transactions since I/O transactions need to be exclusively synchronized to avoid interference across cores. As compared to other resources, there has been little attention paid in the literature to non-determinism due to uncoordinated I/O communications. In [Kim et al., 2013, Kim et al., 2014, Sha et al., 2016] we have solved this problem so that it is backward compatible to the Integrated Modular Avionics (IMA) architecture that is widespread in modern avionics [Kim et al., 2015b]. To the best of our knowledge, this is the first work that schedules exclusive I/O transactions on a multicore IMA system. (See Chapter 2 and 3.)

Although an IMA system provides a strong isolating capability, there is a limitation due to its inflexible Time Division Multiple Access (TDMA) schedule: high-priority tasks in one inactive application need to wait for other applications to finish, even if lower-priority tasks are running. This ultimately leads to lower system utilization. To avoid such issues, we proposed a framework that globally assigns priorities across all tasks on a core, irrespective of application, while enforcing per-application CPU budgets. This provides the foundation for replacing IMA for future multicore avionics.

In real-time systems, most of the existing work assumes a mature and complete system, and thus, supposes a given value for each task's worst-case execution time (WCET). Unfortunately, in the course of development or migration of a system, the WCET of a task may not (accurately) be available, although a system designer or developer needs an

assessment of timing feasibility of the system that is being developed.

In order to incorporate the two issues above, in [Kim et al., 2015a], we developed a new type of methodology to check timing correctness, intended for use in the software development cycle, *i.e.*, with no knowledge of task execution times, and combining the isolation benefits of partitioning with the utilization benefits of global priority assignment (see Chapter 4). The work has since been incorporated with I/O issues on multicore platforms and presented in [Kim et al., 2017] (see Chapter 5). That is, we suggested a new scheduling paradigm for the real-time multicore area, which can provide system developers/designers in a development cycle with quick results and higher utilization.

Chapter 2

I/O Synchronization in Multicore Systems

In the existing literature, the interference impacts stemming from cache, memory and bus are extensively analyzed. However, as compared to such resources, there has been little attention paid in the literature to non-determinism due to uncoordinated I/O communications. Hence in this chapter, we consolidate I/O transactions that synchronize applications across cores and can be feasibly scheduled on a core so as not to interfere with other applications. We propose an approach to solving a problem in scheduling partitions (applications) in a multicore system, namely that of preventing conflicts among I/O transactions from applications residing on different cores. One novel point of this work is that we dedicated a core for synchronizing I/O transactions. We formalize the problem as a partition scheduling problem that serializes I/O partitions. Although this problem is strongly NP-complete, we formulate it as a Constraint Programming (CP) problem. Since the CP approach scales poorly, we propose a heuristic algorithm that outperforms the CP approach in scalability. The work of this chapter is published in [Kim et al., 2014].

2.1 Introduction

In recent years, microprocessor vendors have been transitioning from single-core to multicore processors due to their potential for more easily achieving continued performance improvements. This trend poses a significant challenge for industries that develop safety-critical applications whose behavior must be characterized and verified in great detail to meet certification requirements [Kinnan, 2009,Bieber et al., 2012,Huyck, 2012]. Many of these companies have a large installed base of software applications that were certified on single-core processors. This represents a huge investment in single-core processor technology, because of the high costs of certification. Such a company may eventually have little choice but to transition to multicore, however, since it may become increasingly difficult to obtain single-core processors as the technology continues to evolve. These companies naturally want to minimize the potentially high costs of re-certifying their software for multicore processors. In support of this goal, we propose an approach to solving a fundamental problem that arises when migrating legacy software applications to multicore systems, namely that of preventing interference among I/O transactions from applications residing on different cores. We handle this issue solely by scheduling applications without modifying any system structure or hardware component.

Integrating software applications onto a multicore system supports the efficient utilization of system resources by allowing the applications to share common devices. As a consequence, however, the complexity of correlations among I/O transactions becomes greater. For example, some hardware devices such as graphics cards or storage units do not admit multiple accesses. Unpredictable interference among I/O transactions could occur if multiple devices sharing an I/O channel perform transactions simultaneously.[1]

The Integrated Modular Avionics (IMA) architecture [air, 1991, Rushby, 1999] has been widely adopted by the avionics industry due to its strong isolation properties. In IMA, real-time applications run within a *partition* which is the basic execution environment of software applications according to the ARINC 653 standard [ARI, 2010]. Since IMA supports the partitions' temporal and spatial isolation from one another, the various real-time avionics functions (having various safety-assurance levels) can be developed independently. In addition, the IMA partitioning mechanism has helped ease the certification process for mixed-criticality avionics systems. Thanks to the IMA architecture, partitioning can provide the basis for migrating from single-core to multicore avionics systems. The partitions in a set of single-core systems can be modularized and migrated one by one to a multicore system. If done correctly, this IMA partition-based migration can avoid substantial recertification costs.

As mentioned above, synchronization challenges arise in the course of migrating multiple single-core IMAs to a multicore system; these arise mainly because of sharing I/O devices and channels. Thus, we address the problem of scheduling IMA-type partitions on multicore systems for *conflict-free* I/O, which serializes I/O transactions. For managing I/O transactions, we apply the concept of *Zero partition* also known as the *device management partition* [Rushby, 1999] in IMA structure. The primary motivation for IMA partitioning is fault containment: preventing a failure in one partition from propagating to cause a failure in some other partition. Traditional locking protocols (e.g., semaphore, mutex) can be problematic in this sense – especially, for example, when an errant partition leaves a shared device locked. This would cause a type of partition-to-partition failure propagation, which can prevent other partitions from using the shared device afterwards. For this reason, traditional locking protocols are usually avoided in standard IMA systems [Rushby, 1999].

Sharing a device through a zero partition, on the other hand, does not allow such a failure propagation since a faulty partition's ownership of a shared device is always released before switching to another. Thus, zero partitions have been used as special-purpose 'I/O partitions' to perform I/O transactions in a consolidated fashion and thus have simplified both implementation and management of I/O operations [Krodel, 2004, Parkinson and Kinnan, 2007]. As a result, the I/O synchronization problem now reduces to the problem of I/O Partition synchronization. In essence, the contribution of this work lies in generating a *multi-IMA partition schedule* in which each I/O partition runs exclusively (i.e. no two I/O partitions overlap in time). Additionally, the I/O partitions are consolidated in a dedicated core, to

[1] Some current users of multicore systems resort to disabling all but one core in order to resolve compatibility problems with legacy software. Such an approach fails to take full advantage of potential performance improvements from the multicore technology.

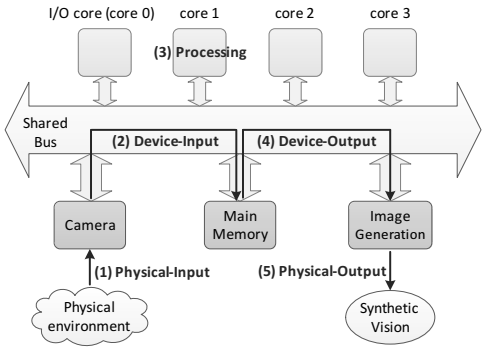


Figure 2.1: Physical-I/O and Device-I/O flows.

simplify the management of the partitions. [2]

We consider non-preemptive scheduling of partitions to ensure temporal isolation among partitions;[3] partitions follow a cyclic schedule table constructed by partition periods and pre-specified time durations. However, this presents another challenge in the migration; since partitions are integrated from different systems, their application periods may not be compatible in general. Thus, supporting multiple rate groups is another important challenge in the migration, which increases the difficulty of synchronization. Our approach to address this issue is discussed in detail in Sec. 2.2. We do not consider the application-to-core assignment problem [Kim et al., 2013]; the assignment is assumed to be known a priori. In addition, the cores are assumed to be synchronized either by the operating system or by other architectural support.

The partition scheduling problem that serializes I/O partitions for exclusive executing is strongly NP-complete, which would imply that it does not admit an efficient, scalable algorithm [Korst et al., 1996]. We first formulate it as a Constraint Programming (CP) problem (Sec. 2.4). However, when the input size (such as the number of partitions or the I/O load) is large, the search space becomes substantially larger, and the CP is no longer efficient. Thus, we propose a heuristic algorithm, called *Hierarchical Offset Selection* (HOS), that outperforms the CP approach in scalability. The details and the evaluation results of the proposed algorithm are presented in Sec. 2.5 and 4.4, respectively.

2.2 System Model

We overview the features of the system in advance of formally describing the partition scheduling problem.

2.2.1 Physical-I/O vs. Device-I/O

We classify I/O transactions into *Physical-I/O* and *Device-I/O*. Physical-I/O is for reading and writing raw data between the physical environment and an I/O device. For example, as illustrated in Fig. 2.1, the transaction of a camera

[2]The original single-core IMA structures are assumed to exhibit the separation between I/O and application logic, as described in Sec. 2.2.1. Accordingly, our approach does not incur significant additional recertification costs, since it does not require modifications of application logic.

[3]This is not a must but one conventional option for modeling an IMA system.

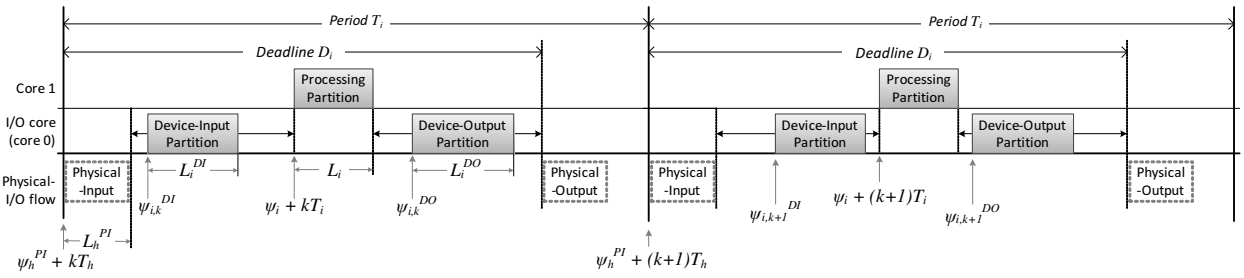


Figure 2.2: The IMA application modeled in this work; The notation is given in Table 2.1. This figure illustrates the precedence dependency between partitions, semi-periodicity of Device-I/O, and notation used for offsets and durations (In general, the durations of Physical-I/O and Device-I/O are relatively short even though the figure does not describe it in that way).

taking pictures from the physical environment is a *Physical-Input*. A *Device-Input* then relays the buffered data for pictures to the main memory, where they are then processed by a *Processing* application running on a core. The output of the processing is buffered in the main memory and then transferred to an output device through a *Device-Output* transaction. The last transaction is a *Physical-Output* in which the resulting image is scattered on a final entity such as a synthetic vision system. Thus, these five transactions have a precedence relationship as shown in Fig. 2.2. Meanwhile, their executions do not need to be 'back-to-back'; Device-I/O operations are allowed to be executed at any arbitrary times as long as the results are available on time.

2.2.2 IMA Application

We consider that a series of executions of Device-Input, Processing, and Device-Output transactions forms an *IMA Application*. [4] Each transaction is modeled and scheduled as an individual IMA partition. As in Sec. 2.1, the concept of zero partition is applied to Device-I/O activity, thus, it is required to be an *exclusive region*, which means that only one Device-I/O partition is allowed to be scheduled at a time instant. We achieve this exclusivity by dedicating one core for Device-I/O partitions only, and we designate it as the *I/O core* (that is core 0 in Fig. 2.1). While Device-I/O partitions are not shared, Physical-Input devices can be shared by multiple applications in an avionics system, as is done in practice. For example, in an avionics system the data from an Attitude Heading Reference System (AHRS) is used by multiple components such as Primary Flight Display (PFD), the Autopilot, and the Flight Management System (FMS).[5] Thus, our model allows Physical-Input devices to be shared while Device-I/O partitions are not to be shared. Processing partitions can generate memory traffic, which is another source of interference in multicore [Fedorova et al., 2010, Schliecker and Ernst, 2011, Yoon et al., 2011, Yoon, 2011]. In this work, however, we limit ourselves to the problem for conflict-free I/O. In other words, this is a work which solves the problem of scheduling partitions in

[4] We assume that there are no precedence relations or data flows between IMA applications.

[5] AHRS consists of sensors providing heading, attitude, and yaw information for avionics system; PFD is a system for fusing and displaying all information critical to flight; FMS is an avionics system that reduces flight crew workload by automating various in-flight tasks.

Table 2.1: List of symbols used in Chapter 2.

Symbol	Description	in/out
π_i	Processing partition i	
$\pi_{i,k}^{DI}$	k_{th} instance of π_i's Device-Input partition	
$\pi_{i,k}^{DO}$	k_{th} instance of π_i's Device-Output partition	
$C(\pi_i)$	core where π_i is in	
MC	major cycle: the least common multiple of all periods	
T_i	period of Processing partition i	input
T_h	period of Physical-Input device h	input
L_i	length of Processing partition i	input
L_h^{PI}	duration of Physical-Input device h	input
L_i^{DI}	length of Device-Input partition i	input
L_i^{DO}	length of Device-Output partition i	input
D_i	relative deadline of Application i	input
$PI_h(i)$	boolean variable: it is $\mathbf{T}$ if Λ_i uses device h	input
ψ_i	offset of Processing partition i	output
ψ_h^{PI}	offset of Physical-Input device h	output
$\psi_{i,k}^{DI}$	offset of $\pi_{i,k}^{DI}$	output
$\psi_{i,k}^{DO}$	offset of $\pi_{i,k}^{DO}$	output
	※ 'input' and 'output' are for CP and heuristic.	

order for I/O traffics not to conflict each other.

2.2.3 Strictly Periodic vs. Semi-Periodic

Since Physical-I/O is periodically operated with a fixed frequency at each I/O device, it is *strictly periodic* in that the time distance between any two consecutive operations of Physical-Inputs (Outputs) is constant. On the other hand, Device-I/O transactions can be buffered, and thus we handle and schedule them as *semi-periodic*, which means that each invocation's offset can vary from period to period as illustrated in Fig. 2.2. Accordingly, we can regard the set of invocations of a Device-I/O partition as a set of strictly periodic partitions, each invoked once per major cycle (MC) of the global IMA schedule table.[6] The primary reason for modeling them as such is to accommodate a greater number of rate groups of applications into the system. Meanwhile, we require a Processing partition to be strictly periodic so as to avoid jitter in its real-time applications. We assume the period of a Processing partition is a multiple of the period of Physical-I/O which it uses.

[6]The Major Cycle (MC) is the least common multiple of all periods of partitions; it is also referred as MAjor time Frame (MAF). The schedule of partitions repeats every major cycle. In the case that the complexity (size) of the schedule would be an issue, a technique to reduce MC, such as one proposed in [Ripoll and Ballester-Ripoll, 2013], could be applied to reduce the complexity (size) of the schedule.

2.2.4 Formal Model

Table 2.1 summarizes the notation used in this chapter.[7] We consider a multicore system consisting of $N^{\mathbf{C}}$ homogeneous cores, $\mathbf{C} = \{C_0, C_1, \ldots, C_{N^{\mathbf{C}}-1}\}$. We use an I/O core, which, without loss of generality, we assume to be C_0; all Device-I/O partitions are assigned on C_0 and Processing partitions are assigned on the rest of the cores. We then consider a set of $N^{\mathbf{\Lambda}}$ IMA applications, $\mathbf{\Lambda} = \{\Lambda_0, \Lambda_1, \ldots, \Lambda_{N^{\mathbf{\Lambda}}-1}\}$, each of which consists of Physical-I/O, Device-I/O partition, and Processing partition as explained in Sec. 2.2.2. The partition sizes are assumed to be predefined [8]. A Processing partition π_i is represented by (T_i, L_i, ψ_i); the first execution of π_i occurs at offset ψ_i and π_i then executes with a period of T_i for a length of L_i. Accordingly, π_i executes during the time interval

$$\forall_{k \in \mathbb{Z}_0^+ \leq (\frac{MC}{T_i} - 1)}, \quad [\psi_i + kT_i, \psi_i + kT_i + L_i), \tag{2.1}$$

where $\mathbb{Z}_0^+$ is the set of non-negative integers. Due to the semi-periodicity of Device-I/O partitions as explained in Sec. 2.2.3, each invocation of the Device-I/O partition is handled individually throughout the major cycle. That is, each needs to be denoted with its order: the k^{th} invocation (k starts from 0) of Device-Input and Device-Output partitions of Application i are denoted as $\pi_{i,k}^{DI} = (MC, L_i^{DI}, \psi_{i,k}^{DI})$ and $\pi_{i,k}^{DO} = (MC, L_i^{DO}, \psi_{i,k}^{DO})$, respectively. Accordingly, $\pi_{i,k}^{DI}$ and $\pi_{i,k}^{DO}$ execute during the time intervals

$$[\psi_{i,k}^{DI}, \psi_{i,k}^{DI} + L_i^{DI}) \text{ and } [\psi_{i,k}^{DO}, \psi_{i,k}^{DO} + L_i^{DO}), \tag{2.2}$$

respectively, for all $k \in \mathbb{Z}_0^+ \leq (\frac{MC}{T_i} - 1)$. We define the relative deadline D_i of Application i as the time instant by which the Device-Output transaction must be completed. That is, the Physical-Output transaction of Λ_i is performed at every D_i measured from the start of Physical-Input (Fig. 2.2). Thus, the four transactions (Physical-Input, Device-Input, Processing, and Device-Output) need to be completed in a duration of D_i.

Each application Λ_i is mapped to a Physical-Input device h. Each Physical-Input transaction is characterized by the offset, period, and length, as if it is an IMA partition.[9] Thus, its interval is represented as follows:

$$[\psi_h^{PI} + kT_h, \psi_h^{PI} + kT_h + L_h^{PI}). \tag{2.3}$$

$PI_h(i)$ is a boolean variable whose value is $\mathbf{T}$ (True) if Λ_i uses device h. Intuitively speaking, if $PI_h(i) = \mathbf{T}$, Λ_i consumes the raw input data produced by device h at every $\psi_h^{PI} + kT_h + L_h^{PI}$. As aforementioned, a Physical-Input

[7] Although there are a large number of symbols used, this is a consequence of the richness of the scheduling model (i.e., the number of relevant parameters for which we will derive values).

[8] In migration from single-core systems to a multicore system, a partition size will be adjusted due to, e.g., different processor speeds. Thus the partition sizes are assumed to be predefined. The interested reader in partition parameter selection problem is referred to [Davis and Burns, 2005, Yoon et al., 2013].

[9] A Physical-I/O is not a partition but a time duration for raw I/O operation.

device can be shared by multiple applications. Thus, if $PI_h(i) = \mathbf{T}$ and $PI_h(j) = \mathbf{T}$, device h is shared between Λ_i and Λ_j. In this case, the applications take the same parameters for Physical-Input h, such as L_h^{PI} and ψ_h^{PI}. We assume that $T_i = kT_h$ for some $k \in \mathbb{N}$ if $PI_h(i) = \mathbf{T}$. Meanwhile, we do not explicitly model Physical-Output; it just defines application deadlines as previously explained. This does not mean that it does not have a duration or length. But it means that they are not considered into the partition scheduling since they do not conflict with other transactions or partitions in our model. Lastly, a Physical-I/O transaction can overlap any other Physical-I/O transactions in time because it is a local operation independent from others.

2.3 Problem Description

Given a set $\mathbf{\Lambda}$ of IMA applications, the Application-Core assignments, and the Application-Device requirements, the problem is to generate a schedule of the partitions that satisfies the constraints presented in the previous sections, which are formalized as follows:

$$\forall_{i,j,i\neq j}, \forall_h, \forall_{k\in\mathbb{Z}_0^+\leq(\frac{MC}{T_i}-1)}, \forall_{l\in\mathbb{Z}_0^+\leq(\frac{MC}{T_j}-1)},$$

Constraint 1. Precedence requirements among Physical-I/O, Device-I/O, and Processing partitions of Application i that uses device h (i.e., $PI_h(i) = \mathbf{T}$) and the deadline requirement

$$\begin{aligned}
&\psi_h^{PI} + kT_i + L_h^{PI} \leq \psi_{i,k}^{DI}, \\
&\psi_{i,k}^{DI} + L_i^{DI} \leq \psi_i + kT_i, \\
&\psi_i + kT_i + L_i \leq \psi_{i,k}^{DO}, \\
&\psi_{i,k}^{DO} + L_i^{DO} \leq \psi_h^{PI} + kT_i + D_i.
\end{aligned} \tag{2.4}$$

Constraint 2. Processing core's feasibility – Processing partitions on the same core must not overlap each other.

$$\begin{aligned}
&[\psi_i + kT_i, \psi_i + kT_i + L_i) \cap [\psi_j + lT_j, \psi_j + lT_j + L_j) = \varnothing, \\
&\text{if} \quad \forall_i \forall_{j\neq i} C(\pi_i) = C(\pi_j).
\end{aligned} \tag{2.5}$$

Constraint 3. I/O core's feasibility – Device-I/O partitions on the I/O core must not overlap each other.

$$\begin{aligned}
&[\psi_{i,k}^D, \psi_{i,k}^D + L_i^D) \cap [\psi_{j,l}^D, \psi_{j,l}^D + L_j^D) = \varnothing, \\
&\text{where the superscript 'D' can be 'DI' or 'DO'.}
\end{aligned} \tag{2.6}$$

The problem formulated above is a Constraint Satisfaction Problem (CSP) whose objective is to find the offsets of partitions and those of Physical-Inputs satisfying the constraints above. This problem is NP-complete in the strong sense even if we are given only two cores and all Device-Input and Device-Output partitions have length of at most one. Its strong NP-completeness can be shown via a reduction from the *3-Partition Problem*, which is a well-known strongly NP-complete problem asking whether a given set of integers can be partitioned into triples with the same sum [Garey and Johnson, 1975]. Even if Processing partitions are scheduled on a single-core, setting aside exclusiveness of I/O partitions, the non-preemptive scheduling of IMA partitions does not admit an efficient algorithm [Jeffay and Stanat, 1991, Cai and Kong, 1996]. Strict Periodicity adds greater complexity to the problem [Korst et al., 1996, Al Sheikh et al., 2012].

To address this problem, we first formulate it with Constraint Programming (CP) to provide a basis for comparison. The CP formulation, however, does not scale well with the problem size (e.g., number of applications, core counts, I/O loads) due to the exponential number of constraints. Thus, in Sec. 2.5, we present a hierarchical search heuristic algorithm that scales well even with problems of interesting size.

We explain the non-overlapping condition for IMA partitions, which we apply to our problem from [Korst et al., 1996]. The property states that any two non-preemptive strictly periodic tasks do not overlap each other if they satisfy the following condition:

$$L_i \leq \ (\psi_i - \psi_j) \bmod \ \gcd(T_i, T_j) \ \leq \ \gcd(T_i, T_j) - L_j, \tag{2.7}$$

where $\gcd$ returns the greatest common divisor of the given values. This property can be used to check if any two partitions (regardless of being Processing or Device-I/O partitions) overlap each other. This property can explain how the schedulability of partitions can be improved by the semi-periodicity of Device-I/O partitions; the $\gcd$ term becomes MC and thus the condition can hold for higher values of L_i and L_j. Similarly, for fixed L_i and L_j, the increased $\gcd$ term allows a wider selection range of the offsets, ψ_i and ψ_j. This helps the system accommodate multiple rate groups of applications.

2.4 Constraint Programming Formulation

In this section, we present the Constraint Programming (CP) formulation for the partition offset selection problem described in the previous section. This CP formulation is given to show the necessity of a scalable heuristic proposed in Sec. 2.5.

2.4.1 Decision Variables(Output)

- ψ_h^{PI}: offset of device h, $0 \leq \psi_h^{PI} \leq \min_{i, PI_h(i)=\mathbf{T}}(T_i - D_i)$.
- ψ_i: offset of π_i, $L_i^{PI} + L_i^{DI} \leq \psi_i \leq T_i - (L_i + L_i^{DO})$.
- $\psi_{i,k}^{DI}$ and $\psi_{i,k}^{DO}$: the offset of π_i^{DI}'s and π_i^{DO}'s k^{th} invocation, respectively, $k \in \mathbb{Z}_0^+ \leq (\frac{MC}{T_i} - 1)$.

2.4.2 Constraints

Precedence and deadline requirements

Same as Constraint 1 in Sec. 2.3.

Processing core's feasibility

$$\forall_i \forall_{j \neq i} \; (C(\pi_i) = C(\pi_j)),$$
$$L_i \leq (\psi_j - \psi_i) \mod \gcd(T_i, T_j) \leq \gcd(T_i, T_j) - L_j.$$

I/O core's feasibility

For any two Device-I/O partitions of different applications, the condition (2.7) must hold. That is,

$$\forall_{i,j,i \neq j}, \forall_{n \in \mathbb{Z}_0^+ \leq (\frac{MC}{T_i} - 1)}, \forall_{m \in \mathbb{Z}_0^+ \leq (\frac{MC}{T_j} - 1)},$$
$$L_i^D \leq (\psi_{j,m}^D - \psi_{i,n}^D) \mod MC \leq MC - L_j^D,$$

where the superscript 'D' can be 'DI' or 'DO'. Note that one can eliminate a significant number of such constraints by considering the possible maximum range of each Device-I/O partition; no constraint is made for any two Device-I/O partitions that can *never* overlap. For example, let $min_{i,k}^{DI}$ and $max_{i,k}^{DI}$ be the minimum and maximum possible offsets of $\psi_{i,k}^{DI}$, respectively. $min_{i,k}^{DO}$ and $max_{i,k}^{DO}$ are similarly defined for $\psi_{i,k}^{DO}$. Then, the maximum possible ranges are

$$[min_{i,k}^{DI}, max_{i,k}^{DI}] = [kT_h + L_h^{PI}, (k+1)T_h - L_i^{DI} - L_i - L_i^{DO}],$$
$$[min_{i,k}^{DO}, max_{i,k}^{DO}] = [kT_h + L_h^{PI} + L_i^{DI} + L_i, (k+1)T_h - L_i^{DO}].$$

Now, for any two Device-I/O partitions of different applications, we apply the constraints above if the maximum possible ranges overlap. This can be checked by using (2.7) with $L_i = max_{i,k}^D - min_{i,k}^D + 1$, $\psi_i = min_{i,k}^D$, and $T_i = MC$, where the superscript 'D' can be DI or DO. Note, however, that the reduction in the number of constraints

can be small when there are significant differences among the applications periods. This is because the maximum possible Device-I/O range of a long period application can overlap many ranges of small periods.

2.4.3 Complexity

The presented CP, however, is not scalable with large inputs such as a large number of applications and a high ratio of MC to application periods. Specifically, the number of decision variables is highly dependent on the major cycle length and the application periods, since, for each application i, $\frac{2MC}{T_i}$ variables are generated for its Device-I/O partitions. Thus, the closer the periods are to harmonic, the fewer the number of decision variables generated. This also holds for Constraint 1 in Sec. 2.3. The major computational complexity is associated with the constraints for I/O core feasibility; the problem size can explode because of the non-overlapping conditions for Device-I/O partitions. As previously explained, the constraint is made for any pair of Device-I/O partitions of different applications. Thus, we need as many as $\sum_{i,j\neq i} 4\frac{MC}{T_i}\frac{MC}{T_j}$ such constraints, which can make the search space grow exponentially with respect to the number of applications and their MC to period ratios. For these reasons, the CP is no longer efficient for those large inputs, which motivates us to propose a scalable heuristic algorithm in the following section.

Algorithm 1 HOS$(\mathbf{C}, \mathbf{\Lambda})$

```
for k = 1, ..., MaxTries do
    Cand ←ConstructInitialCandidate(C, Λ)
    Sol ←DevIO_OffsetSearch(Cand, C, Λ)
    if Sol is not Null then return Sol                ▷ A feasible solution is found
    end if
end forreturn Null                                    ▷ Failed to find a solution
```

2.5 Hierarchical Offset Selection

We now present a heuristic algorithm, called *Hierarchical Offset Selection (HOS)*, for the partition offset selection problem formulated in Sec. 2.3. The key idea of the heuristic is to start with a random but partially feasible assignment of the offsets of all Physical-I/Os and Processing partitions and then to find a complete solution by determining the offsets of all Device-I/O partitions. The reason the heuristic starts with an initial assignment of the strictly periodic offsets is that this helps narrow the initial search space by focusing on a smaller set of decision variables; recall that for each application, only one pair of variables for the Physical-Input and the Processing partition offsets is required. Thus, we need only a polynomial number of constraints for the processing core's feasibility (see Constraint 2 in (2.5)).[10] More importantly, as will be seen shortly, once the offsets are fixed, the search space for the rest, i.e., the Device-I/O partition offsets, can be represented as a set of periodic intervals, which reduces the problem size considerably.

[10] In the worst-case, i.e., all applications are on the same core, $\binom{N^{\mathbf{\Lambda}}}{2} \sim O(N^{\mathbf{\Lambda}^2})$ constraints are generated.

Through this hierarchical searching, the algorithm quickly finds a solution on average and scales well with the problem size even though it does not guarantee that it could find a solution even if one exists.

2.5.1 Algorithm Overview

The pseudo-code in Algorithm 1 illustrates the structure of our HOS algorithm. It is an iterative procedure, in which each iteration is composed of an *initial candidate construction* phase, where the offsets of Physical-I/O and Processing partitions are randomly assigned without destroying feasibility, and *local search* phases, where the offsets of Device-I/O partitions are searched. This process is similar to the Greedy Randomized Adaptive Search Procedure (GRASP) [Feo and Resende, 1995] except that in HOS, (i) a *partially* initialized solution is constructed at the first stage, (ii) the offsets (of Physical-I/O and Processing partitions) that were assigned in the construction stage remain fixed during the local search phase, and (iii) the algorithm terminates whenever a feasible solution is found (recall that our problem is not a numerical optimization problem). If no solution could be constructed upon the initial candidate solution during the local search phase, the same procedure above repeats until either a solution is found or the stopping criterion, `MaxTries`, is met. Meanwhile, if we fail to construct a candidate solution (Line 2), the iteration starts over (not shown in the pseudo-code) as long as the maximum trial number has not been reached.

2.5.2 Randomized Initial Candidate Construction

The key task in this phase is to assign the offsets of all Physical-I/O and Processing partitions, i.e., $\{\psi_h^{PI}\}$ and $\{\psi_i\}$, while guaranteeing the processing core feasibility (Constraint 2 in Sec. 2.3). Note that it is known that the problem of testing if a set of non-preemptive strictly periodic tasks (Processing partitions in our case) is schedulable even on a single processor is strongly NP-complete [Korst et al., 1996]. Thus, we take the following greedy approach to initial solution construction. First, for each Physical-Input Device h, its offset, ψ_h^{PI}, is randomly chosen between 0 and the minimum value of $T_i - D_i$ among all applications that use device h as in Sec. 2.4.1. Then, we assign Processing partition offsets for each core. To be more specific, for each core C_j, we try to assign the Processing partition offsets $\{\psi_i | C(\pi_i) = C_j\}$ in the increasing order of their periods. This is because placing a partition of longer period first fragments the available periodic spaces for the remaining partitions of shorter periods [Korst et al., 1996]. Now, for each π_i, the possible range of its Processing partition offset, ψ_i, is limited to

$$[\psi_h^{PI} + L_i^{PI} + L_i^{DI}, \psi_h^{PI} + D_i - L_i^{DO} - L_i],$$

if $PI_h(i) = \mathbf{T}$ by the precedence and deadline requirements in (2.4). The search for a feasible ψ_i begins from a randomly selected value in the given range and proceeds with the next value until π_i does not overlap other partitions that have already been placed when π_i is placed at ψ_i. If no such offset is found in the range, it may try different

ψ_h^{PI} and the corresponding $\{\psi_i | PI_h(i) = \mathbf{T}\}$ or just return `Null` so that the construction starts over. Note that this construction procedure is not an exhaustive search in that it does not attempt all possible sets of offsets of Physical-I/O and Processing partitions until one is found. This is not only because no efficient search algorithm exists (as aforementioned) but also because even if a candidate solution is found via an exhaustive search, it does not guarantee that a feasible complete solution will be found in the local search phase. Thus, instead of trying to construct a feasible candidate at every attempt, the phase simply starts over, with the goal of constructing a feasible one in a future attempt.

Algorithm 2 DEVIO_OFFSETSEARCH(`Cand`, $\mathbf{C}$, $\mathbf{\Lambda}$)

$\mathcal{I}$: set of all intervals in `Cand` found by (2.10)
$\mathcal{C} \leftarrow$BUILDCONFLICTGROUPS($\mathcal{I}$)
for all conflict group c in $\mathcal{C}$ **do**
    $\mathcal{I}_c$: Intervals in c
    $S^{\mathcal{I}_c}$: Sorted list of $\mathcal{I}_c$ in increasing order of ψ_j^I and decreasing order of w_j
    ψ^* : the earliest available offset ▷ Initialized by 0
    while $S^{\mathcal{I}_c}$ is not empty **do**
        $I_j \leftarrow$FINDNEXTINTERVAL($S^{\mathcal{I}_c}, \psi^*$)
        if $I_j =$ `Null` **then return** `Null`
        else
            Place π_j^I at $\max(\psi^*, \psi_j^I)$ on I_j
            $\psi^* \leftarrow \max(\psi^*, \psi_j^I) + L_j^I$
            if $\psi^* > \psi_j^I + w_j$ **then return** `Null`
            end if
            Remove I_j from $S^{\mathcal{I}_c}$
        end if
    end while
end forreturn `Sol` ▷ i.e., partition offsets found above

2.5.3 Local Search for Device-I/O Offsets

In the construction phase, we did not consider how Device-I/O partitions would feasibly be placed on the I/O core. However, if constructed successfully, an initial solution guarantees spaces for Device-I/O partitions and the precedence and deadline requirements of every application. Given such a candidate solution, this phase tries to construct a complete solution by searching a set of Device-I/O partition offsets satisfying the I/O core feasibility condition; the search is defined in Algorithm 2.

Periodic intervals and conflict groups

Suppose now we have a set of feasible ψ_h^{PI} and ψ_i assigned in the construction phase. Then, for each Λ_i such that $PI_h(i) = \mathbf{T}$, the ranges in which Device-I/O partitions can be placed are determined by (2.4). That is, the k^{th}

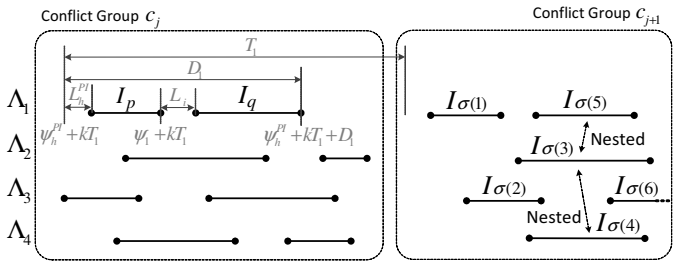


Figure 2.3: Periodic intervals, conflict groups, and nested intervals.

Device-Input partition of Λ_i, i.e., $\pi_{i,k}^{DI}$, can be placed somewhere in

$$[\psi_h^{PI} + kT_i + L_h^{PI}, \psi_i + kT_i). \tag{2.8}$$

Similarly, $\pi_{i,k}^{DO}$ can be placed somewhere in

$$[\psi_i + kT_i + L_i, \psi_h^{PI} + kT_i + D_i). \tag{2.9}$$

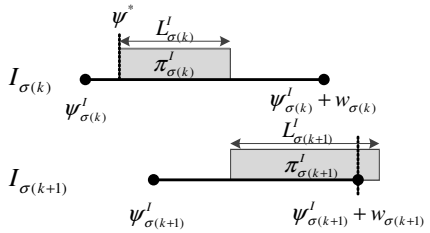
Observe here that each range is *periodic* with T_i for $k = 0, \ldots, \frac{MC}{T_i} - 1$. Thus, the set of such ranges can be represented as *periodic intervals*.[11] Now, the problem becomes placing total $\sum 2\frac{MC}{T_i}$ Device-I/O partitions on each of its intervals. We denote an interval as $I_j = (\psi_j^I, w_j, L_j^I)$, where ψ_j^I and w_j are the starting offset and the width of the interval, respectively, and L_j^I is the length of the partition that needs to be allocated in the interval. Depending on whether I_j corresponds to Device-Input or Device-Output, ψ_j^I and w_j are determined by (2.8) or (2.9). If I_j is the k^{th} Device-I/O partition of Λ_i and $PI_h(i) = \mathbf{T}$, then

$$I_j = \begin{cases} (\psi_h^{PI} + kT_i + L_h^{PI}, \psi_i - \psi_h^{PI} - L_h^{PI}, L_i^{DI}), \\ (\psi_i + kT_i + L_i, \psi_h^{PI} + D_i - \psi_i - L_i, L_i^{DO}). \end{cases} \tag{2.10}$$

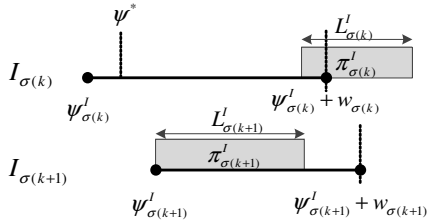
Fig. 2.3 illustrates an example of how periodic intervals are defined. For example, Λ_1's Physical-Input and Processing partition offsets, i.e., ψ_h^{PI} and ψ_1, determine the intervals, I_p and I_q, where its Device-Input and Device-Output partitions, respectively, will be placed. These intervals, such as I_p and $I_{\sigma(1)}$, repeat with a period of T_1. From this point, however, we do not need to consider whether each interval is for input or output and to which application it belongs.

The set of all intervals, $\mathcal{I}$, forms a set of disjoint *conflict groups*, $\mathcal{C} = \{c_1, \ldots, c_{N^C}\}$. A *conflict group* is a maximal set of intervals in which each interval overlaps at least one another interval in the same group. As shown in Fig. 2.3, since no two intervals in the different conflict groups overlap each other, the search is independently performed in

[11] The periodic interval in this work is different from the ones in [Korst et al., 1996].



(a) $\pi_{\sigma(k)}$ first



(b) $\pi_{\sigma(k+1)}$ first

Figure 2.4: Proof of Lemma 1 for case $\psi^* < \psi^I_{\sigma(k+1)}$.

each conflict group.[12] In what follows, we limit ourselves to a single conflict group.

Local search in a conflict group

Given a conflict group c, we perform the following heuristic to find a feasible assignment of Device-I/O partition offsets. First, let $\mathcal{I}_c$ be the set of intervals that appear in c. We build a sorted list $S^{\mathcal{I}_c}$ of $\mathcal{I}_c$ by ordering intervals in increasing order of their starting offsets, ψ^I_j; break ties by shorter interval width, w_j (see the conflict group c_{j+1} in Fig. 2.3). Then, the algorithm iterates over the intervals (Line 7–19 in Algorithm 2), and each time it places a partition on an interval selected by FINDNEXTINTERVAL (Line 8, which is detailed in Algorithm 3). Partitions are placed as near the beginning of their intervals as possible, and this is done by keeping the earliest available offset, denoted as ψ^* (Line 12–13). Now, the following lemma suggests how the partitions should be placed on the intervals so as to maximize the possibility of finding a feasible schedule for the partitions.

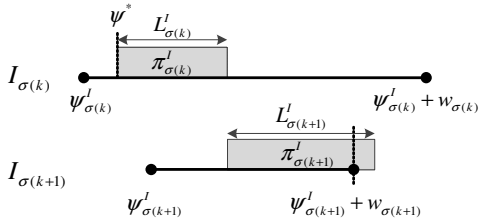
Lemma 1. *Given a set of intervals $\mathcal{I}_c$, if its sorted list $S^{\mathcal{I}_c} = \{I_{\sigma(1)}, \ldots, I_{\sigma(|\mathcal{I}_c|)}\}$ satisfies*

$$\psi^I_{\sigma(k)} \leq \psi^I_{\sigma(k+1)} \text{ and } \psi^I_{\sigma(k)} + w_{\sigma(k)} \leq \psi^I_{\sigma(k+1)} + w_{\sigma(k+1)}, \tag{2.11}$$

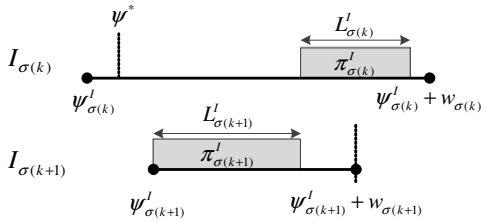
for all $1 \leq k \leq |\mathcal{I}_c - 1|$, then placing partitions at the earliest possible offset in the same order as the intervals are sorted is optimal in the sense that if they cannot be scheduled with this placement, no feasible schedule exists.

Proof. Let us consider two intervals $I_{\sigma(k)}$ and $I_{\sigma(k+1)}$ in $S^{\mathcal{I}_c}$. For the simplicity of representation, let $\pi^I_{\sigma(k)}$ be the (Device-I/O) partition that needs to be placed on $I_{\sigma(k)}$. Now, suppose the partitions $\pi^I_{\sigma(1)}, \ldots, \pi^I_{\sigma(k-1)}$ have been

[12] $\mathcal{C}$ can be built simply by iteratively merging conflict groups until no more merges can occur. Each individual interval is initially a conflict group by itself. Two conflict groups merge if any two intervals from them overlap each other.



(a) $\pi_{\sigma(k)}$ first: unschedulable



(b) $\pi_{\sigma(k+1)}$ first: schedulable

Figure 2.5: Example for nested intervals.

placed according to the placement rule described above. Then, we can find the earliest available offset ψ^* ($\geq \psi_{\sigma(k)}$) on $I_{\sigma(k)}$ where it can accommodate $\pi^I_{\sigma(k)}$ of length $L^I_{\sigma(k)}$. The placement rule above says that $\pi^I_{\sigma(k)}$ should be placed before $\pi^I_{\sigma(k+1)}$.

Now, for a contraposition, suppose that this placement makes the schedule infeasible as in Fig. 2.4 (a). That is,

$$\psi^* + L^I_{\sigma(k)} + L^I_{\sigma(k+1)} > \psi^I_{\sigma(k+1)} + w_{\sigma(k+1)}. \tag{2.12}$$

(Note: We only need to consider the case when $\psi^* + L^I_{\sigma(k)} > \psi^I_{\sigma(k+1)}$ because it is the only problematic case.) We show that the other order of placement, i.e., $\pi^I_{\sigma(k+1)}$ before $\pi^I_{\sigma(k)}$, does not turn the schedule feasible: $\pi^I_{\sigma(k)}$ is not schedulable on $I_{\sigma(k)}$. First, the earliest available offset for $I_{\sigma(k+1)}$ is determined by $\max(\psi^*, \psi^I_{\sigma(k+1)})$. Now, we consider the following two cases (Fig. 2.4 (b) illustrates the first case):

(i) $\psi^* < \psi^I_{\sigma(k+1)}$: $\pi^I_{\sigma(k+1)}$ is placed at $\psi^I_{\sigma(k+1)}$. Thus,

$$\begin{aligned}\psi^I_{\sigma(k+1)} + L^I_{\sigma(k+1)} + L^I_{\sigma(k)} &> \psi^* + L^I_{\sigma(k+1)} + L^I_{\sigma(k)} \\ &> \psi^I_{\sigma(k+1)} + w_{\sigma(k+1)}.\end{aligned}$$

(ii) $\psi^* \geq \psi^I_{\sigma(k+1)}$: $\pi^I_{\sigma(k+1)}$ is placed at ψ^*. Thus,

$$\psi^* + L^I_{\sigma(k+1)} + L^I_{\sigma(k)} > \psi^I_{\sigma(k+1)} + w_{\sigma(k+1)}.$$

The last inequality in each case follows from (2.12). In any case, the *finishing time* of $\pi^I_{\sigma(k)}$, i.e., $\max(\psi^*, \psi^I_{\sigma(k+1)}) + L^I_{\sigma(k+1)} + L^I_{\sigma(k)}$, exceeds that of $I_{\sigma(k+1)}$, which proves that $\pi^I_{\sigma(k)}$ is not schedulable on $I_{\sigma(k)}$ since $\psi^I_{\sigma(k)} + w_{\sigma(k)} \leq \psi^I_{\sigma(k+1)} + w_{\sigma(k+1)}$ by (2.11). □

Therefore, if the intervals in a conflict group can be ordered satisfying (2.11), for example, c_j in Fig. 2.3, we can optimally place partitions on the intervals as long as a feasible schedule exists. However, the condition in (2.11) may not be satisfied by some conflict groups such as c_{j+1} in Fig. 2.3. In the example, $I_{\sigma(3)}$ and $I_{\sigma(4)}$, $I_{\sigma(3)}$ and $I_{\sigma(5)}$, and $I_{\sigma(4)}$ and $I_{\sigma(5)}$ do not satisfy (2.11) because one interval is *nested* by the other. In such a case, the ordering rule described above is no longer optimal as illustrated in Fig. 2.5. Thus, we need to examine both cases. If both are feasible, we choose the first case, because this would increase ψ^*, the earliest available offset, by the smallest amount, which thus maximizes the possibility of finding a feasible schedule for the rest of the intervals.

Algorithm 3 FINDNEXTINTERVAL($S^{\mathcal{I}}, \psi^*$)

1: $I_{\sigma(1)}$: the first interval in $S^{\mathcal{I}}$
2: $I_{\sigma(j)}$: the smallest-indexed interval nested by $I_{\sigma(1)}$, $j > 1$
3: **if** no such $I_{\sigma(j)}$ exists **then return** $I_{\sigma(1)}$
4: **else**
5: **if** $\max(\psi^*, \psi^I_{\sigma(1)}) + L^I_{\sigma(1)} + L^I_{\sigma(j)} \leq \psi^I_{\sigma(j)} + w_{\sigma(j)}$ **then return** $I_{\sigma(1)}$
6: **else if** $\max(\psi^*, \psi^I_{\sigma(j)}) + L^I_{\sigma(j)} + L^I_{\sigma(1)} \leq \psi^I_{\sigma(1)} + w_{\sigma(1)}$ **then return** $I_{\sigma(j)}$
7: **elsereturn** `Null` ▷ Cannot find an interval to feasibly schedule
8: **end if**
9: **end if**

Algorithm 3 summarizes the interval selection procedure. As described above, initially it tries to select the first interval, $I_{\sigma(1)}$, in the given sorted list. If no interval is nested by $I_{\sigma(1)}$, it is chosen for the optimality as described in Lemma 1. If another interval is nested by it, we still try to keep the ordinary ordering, i.e., $I_{\sigma(1)}$ first, (Line 6–7) as aforementioned. The nested one is selected only when this would maintain feasibility (Line 8–9). Now, it could be the case that more than two intervals are nested by $I_{\sigma(1)}$. Instead of attempting all possible cases, we consider only $I_{\sigma(j)}$, the smallest-indexed interval nested by $I_{\sigma(1)}$. It should be noted, however, that this may no longer guarantee optimality especially if intervals are recursively nested, e.g., $I_{\sigma(3)}, I_{\sigma(4)}$, and then $I_{\sigma(5)}$ in c_{j+1} in Fig. 2.3; it could be that the only feasible schedule has $\pi^I_{\sigma(5)}$ preceding the others. Even though HOS does not perform such a *look-ahead search* in consideration of performance and simplicity, it finds a solution quickly even for the problems of large size as we will see in the next section.

2.6 Evaluation

In this section, we evaluate the proposed algorithm, HOS, first comparing it with the CP approach presented in Sec. 2.4 and performing an in-depth evaluation of its performance. The results are based on randomly generated input instances with the following parameters: application count per core is in $[3, 7]$; T_i is in $[100, 3000]$; L_i is in $[10, 50]$; L_i^{PI} is in $[1, 5]$; both L_i^{DI} and L_i^{DO} are in $[3, 10]$; D_i is $0.95T_i$; all values are integers. As mentioned in Sec.2.2, the durations of Physical-I/O and Device-I/O are relatively short, thus the experimental ranges of the parameters are not wide. For each input instance, we ran HOS 20 times and averaged the results because of the randomness involved in HOS (see

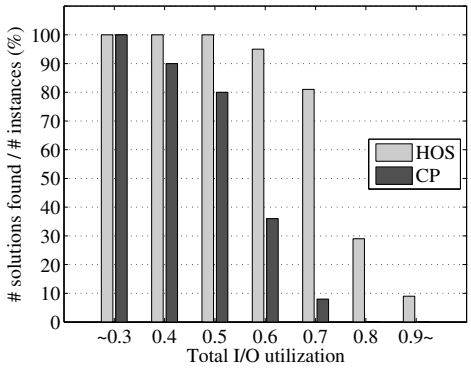


Figure 2.6: The ratio of the number of instances whose solution is found to that of instances according to the total I/O utilization.

Sec. 2.5.2). We set the maximum number of trials of HOS, i.e., `MaxTries` in Algorithm 1, to 3000. All experiments were performed on an Intel® Core™ i7-2600 CPU 3.40 GHz with 16GB RAM. For CP, we used IBM ILOG CPLEX CP Optimizer. To the best of our knowledge there is no existing work which can be compared with our work.

2.6.1 Comparison with Constraint Programming

In this evaluation, we compare the scalability of HOS with that of CP. For this evaluation, we generated 300 random input instances with 3 to 5 cores.[13] We limited the core count to 5 since CP could not handle cases with more than 5 cores. Due to the large number of inputs, we set a timeout of 10 hours for CP; when it terminates with timeout, we consider that no solution for the given instance is found even if one may exist. However, we observed that the solving times were much smaller than the limit in most cases when solutions are found.

Fig. 2.6 compares the number of solutions found by each method. We grouped input sets according to the I/O utilization defined by $\sum_i \frac{L_i^{DI}+L_i^{DO}}{D_i}$, since the I/O utilization rather than the total utilization is the bottleneck on the system. We grouped input sets according to the I/O utilization by rounding to the nearest 0.1 (each group has approximately 50 instances except the last two groups which have 39 and 13 instances, respectively). Each bar represents the ratio of the number of instances whose solution is found to that of instances in each group. As we can see, HOS found more solutions than CP did in every utilization group, and in fact, every input instance for which a solution was found by CP was also solved by HOS. In addition, the difference became greater as the I/O utilization increased. This stems from the fact that instances with higher I/O utilizations tend to include larger numbers of cores and applications and hence of Device-I/O partitions to be scheduled. As discussed in Sec. 2.4.3, this greatly affects the complexity by generating a huge number of constraints that CP cannot resolve within the time limit. In addition, longer Device-I/O partitions with shorter periods may have contributed to a higher I/O utilization. In such cases, it is

[13] This includes an I/O core. The choice of the range is for evaluation purposes only, not realism. The core count starts from 3 since a count of 2 (one I/O core and one processing core) is trivial.

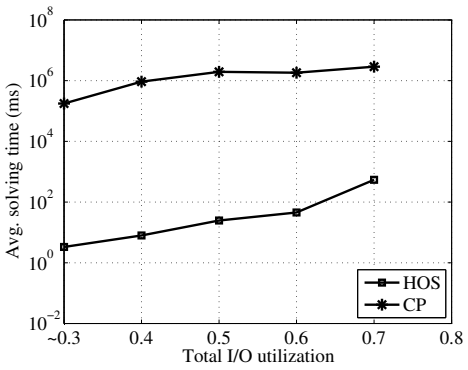


Figure 2.7: The average solving time according to the total I/O utilization.

simply harder to schedule them, or no feasible solution may exist,[14] which is the reason why the number of solutions found by HOS is low for I/O utilizations of 0.8 and above.

Next, Fig. 2.7 compares the average solving time for each instance whose solution is found by both methods. The x-axis is again I/O utilization and the y-axis is average solving time in milliseconds in log scale. The results for I/O utilization groups of 0.8 and above were not drawn due to insufficient numbers of solutions found. The result clearly shows that HOS found solutions much faster than CP did by orders of magnitude; for those instances with I/O utilization of around 0.5, HOS found solutions in an average of about 25 ms while CP took an average of around 32 minutes. In the worst-case HOS did not take more than 2.7 seconds, whereas CP took up to 133 minutes. Now, the performance difference between the methods becomes more evident if we compare them in instance level. In Fig. 2.8, each mark represents the ratio of the solving time of CP to that of HOS for each input instance. The x-axis this time is $\sum_{i,j \neq i} 4 \frac{MC}{T_i} \frac{MC}{T_j}$, the worst-case number of constraints for I/O core feasibility made by CP, in log scale (see Sec. 2.4.3). We can see that the performance gap between the methods significantly grows with the constraint count. We note that this trend also holds when the x-axis is replaced by the I/O utilization or the number of applications due to high correlations among them.

2.6.2 In-depth Evaluation of HOS

We now evaluate the performance of HOS for more general problems by experimenting with expanded ranges of parameters. The results are based on 500 randomly generated input instances with 4 to 8 cores. Other parameters and assumptions are the same as for the previous evaluation.

Fig. 2.9 and Table 2.2 show (i) the average number of trials (i.e., iterations in Algorithm 1) that occurred before HOS found a solution (bar plot, in log scale) and (ii) the average failure rate of constructing an initial candidate solution

[14] Some input instances might be infeasible in the first place. The exact number of feasible ones is unknown because this requires an optimal method.

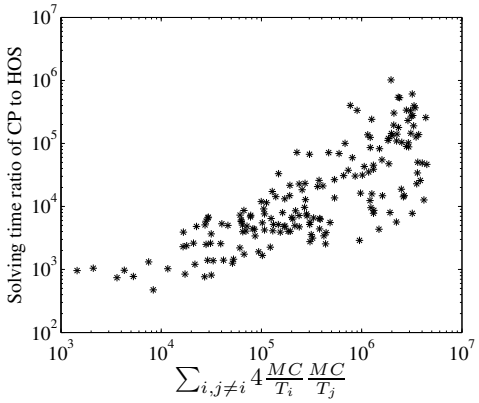


Figure 2.8: The solving time ratio of CP to HOS according to the worst-case number of constraints made for I/O core feasibility.

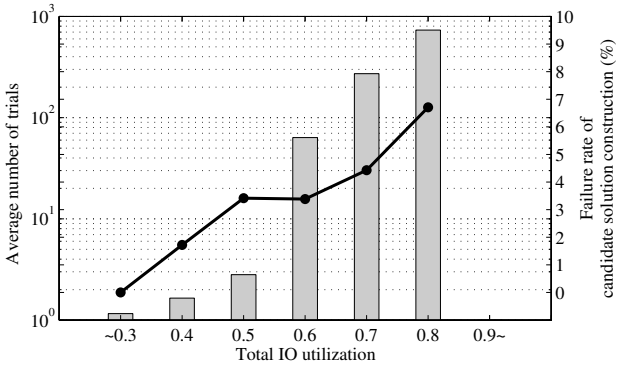


Figure 2.9: The average number of trials of HOS until a solution is found (bar plot) and the failure rate of constructing an initial candidate solution (line plot).

Table 2.2: The numerical data plotted on Fig. 2.9.

I/O Util group	~ 0.3	0.4	0.5	0.6	0.7	0.8
Avg. # of trials	1.16	1.65	2.81	63.5	271	732
Fail % of init. sol.	0.00	1.72	3.41	3.39	4.43	6.71

Table 2.3: The number of instances and found solutions on Fig. 2.9.

I/O Util group	~0.3	0.4	0.5	0.6	0.7	0.8	0.9~
# of instances	49	79	90	73	61	82	66
# of sol. found	49	79	90	73	57	46	5

at CONSTRUCTINITIALCANDIDATE (line plot).[15] Again, those instances might be infeasible in the first place. Similar to the above, the instances were grouped by I/O utilization. The results reveal that when the I/O utilization is low, a small number of iterations is enough for HOS to find a feasible solution. For example, for the first group, HOS never failed to construct an initial candidate solution and could find a complete solution almost right away through local searches. For the group of utilization 0.5, an average of 3 iterations were enough to find a feasible solution. If we look only at the failure rate of CONSTRUCTINITIALCANDIDATE, it does not rapidly grow with I/O utilization. This is because the phase is decoupled from searching for Device-I/O offsets, and hence is less sensitive to I/O loads. The still increasing trend with the I/O utilization is simply because higher I/O loads are mainly due to bigger application sets, which makes it harder to schedule Processing partitions on each core (even without considering I/O partitions). Nevertheless, we can see that HOS performs well in constructing initial solutions even for higher I/O utilizations; e.g., it failed once in every 15 tries on average for the utilization group of 0.8. This low failure rate tells us that most of the retrials were caused by failing in the local search phase especially for higher I/O loads. This can be attributed to difficulties in (i) handling many overlapping intervals (which results in fewer conflict groups) and thus more nested intervals and (ii) constructing a *good* initial candidate in the construction phase. One can expect better efficiency of the local search by trying to find a better candidate such as one that creates fewer overlaps and nested intervals, which we leave for future work.

In Fig. 2.10, we plot the average solving time of HOS for individual input instances (drawn in log scale). The plot at the top of the figure shows the trend of exponentially increasing solving times with the total I/O utilization that we can foresee from the previous evaluations. The similar trend can be observed in the next plot where the same results are drawn according to the worst-case number of constraints for I/O core feasibility that would have been made for CP. Meanwhile, we observed that the outliers in $[0.25, 0.5] \times 10^7$ along the x-axis were simply due to exceptionally higher I/O utilization than the others having similar x values. Although not shown explicitly, HOS took less than 27 seconds for 90% and 80 seconds for 99% of the all instances and less than 100 seconds in the overall worst-case. Considering the size of the inputs used, this result, along with the previously discussed results, shows how well our HOS scales for problems of practical application.

[15] Fig. 2.9 and Table 2.2 do not show the results for the I/O utilization group of 0.9 due to the insufficient number of samples (see Table 2.3).

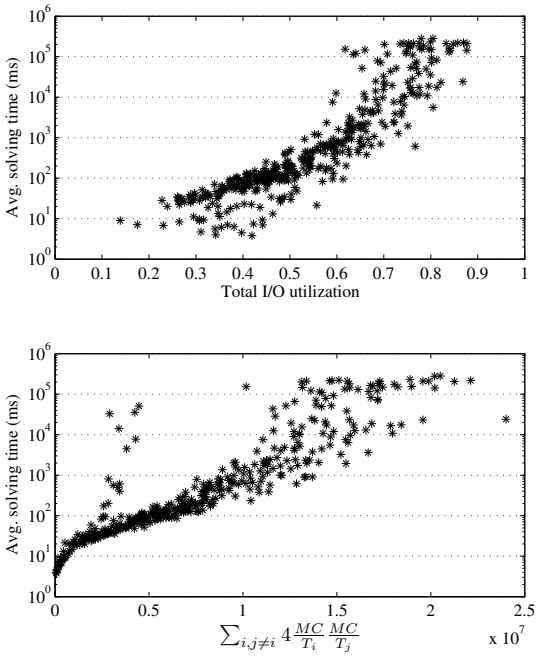


Figure 2.10: The average solving time of HOS according to (i) the total I/O utilization and (ii) the worst-case number of CP's constraints made for I/O core feasibility (see Sec. 2.4.3).

Chapter 3

Schedulability Bound for Integrated Modular Avionics Partitions

As mentioned earlier, in the avionics industry, as a hierarchical scheduling architecture Integrated Modular Avionics (IMA) System has been widely adopted for its isolating capability. In this chapter, we present a method to calculate schedulability bound for IMA systems. In practice, in an early development phase, a system developer does not know much about task execution times, but only task periods and IMA partition information. In such a case the schedulability bound for a task in a given partition tells a developer how much of the execution time the task can have to be schedulable. Once the developer knows the bound, then the developer can deal with any combination of execution times under the bound, which is safe in terms of schedulability. We formulate the problem as linear programming that is commonly used in the avionics industry for schedulability analysis, and compare the bound with other existing ones which are obtained with no period information. The work of this chapter is published in [Kim et al., 2015b].

3.1 Introduction

The avionics industry has widely adopted the Integrated Modular Avionics (IMA) architecture [air, 1991, Rushby, 1999], which enables real-time functions to run within partitions that are temporally and spatially isolated from one another. Real-time tasks run within an IMA partition which is an execution environment of software applications according to the ARINC 653 standard [ARI, 2010]. Since IMA supports the partitions' temporal and spatial isolation from one another, the various real-time avionics functions can be developed independently. In addition to that, the mechanism has helped ease the certification process for mixed-criticality avionics systems. If done correctly, this modular approach can avoid substantial re-certification costs.

In practice, in an early development phase of IMA systems, IMA partition parameters and task periods are pre-determined while the worst-case execution times of the tasks are partially or even completely unavailable. Such a situation is common. For example, when receiving data from a sensor, a task may know the sensing frequency in advance but not the worst-case execution time since determining the worst-case execution time is not a simple job. In such a case, if we could know the schedulability bound with no information on task execution time, that would be

helpful to determine whether certain tasks are able to be integrated into a system or not. Besides, in the course of system development, if a system designer would add/modify a task to/on an existing system, the designer needs to check the schedulability of the system including the new/modified task. If it is schedulable the new/modified task can be included into the system. In such cases the schedulability test provides a quick, abstracted assessment of the system under development.

Hence, we present an analysis determining the schedulability bound for a set of tasks in an IMA partition, when IMA partition information and *task periods are known while task execution times are unavailable*. With the bound, we can determine *how much of execution times the tasks can have to be schedulable in the given periods* in a hierarchical IMA system. Since we can determine the bound when there is no information about task execution times at all, we can also trivially do it when only a subset of information for task execution times is available.

To derive the bound, we formulate the problem as a linear programming (LP) problem that can be solved by a linear programming solver which is commonly used in the avionics industry for schedulability analysis. If the final bound is above or equal to the total utilization of all tasks in the partition, the tasks are determined to be schedulable. Throughout this chapter, we assume task execution on an IMA system is based on Rate-Monotonic (RM) scheduling [Liu and Layland, 1973] which is the optimal scheduling policy on fixed priority scheduling and supported by open standards in avionics systems. RM assigns higher priority on the task with a shorter period.

In order to see the impact of more known information, we conduct comparisons in Sec. 3.4. There are two works which look for a utilization bound in a hierarchical environment with resource information in the upper level but without task information. In [Sha, 2003], a scheduling bound is presented when only Real-Time Virtual Machine (RTVM) sizes are given while task information is not. RTVM can be seen as a similar hierarchical concept with IMA system. The bound shows the maximally possible utilization of tasks in an RTVM. In [Shin and Lee, 2003], considering a periodic resource, a utilization bound for RM is presented when only resource information but no task information (workload) is given (only the number of tasks, the shortest task period and the ratio between periods are known). The resource is similar to IMA partition in the perspective of a task in our context. Since the utilization bound can show the maximum load of tasks under a certain resource requirement, it can be used for determining whether to accept a (new) task into a system or not (admission test). Since our presented bound is a result based on 'more' information (task periods) than that of [Sha, 2003] or [Shin and Lee, 2003], our bound is supposed to be higher than theirs.

On the stream of the research of temporally partitioned hierarchical scheduling there have been two main issues - one is how much of task utilization can be supported by the allocated resource requirement. That is, as what this work is about, to find the maximal utilization of tasks which run under a given partition (e.g., server, resource, RTVM) requirement. Besides to ours, [Sha, 2003] and [Shin and Lee, 2003] reside in this category. The another issue on the

stream is, the other way around, how much of the computational resource needs to be allocated to each partition in order for the system to be optimized for a certain metric. For instance, it is desirable in the system design process to minimize the system utilization while guaranteeing the timing requirements of both partitions (servers) and their applications. That is obtaining the partition or server parameters to sustain the given task. Relevant work can be found in [Shin and Lee, 2003, Almeida and Pedreiras, 2004, Lipari and Bini, 2003, Lipari and Bini, 2005, Davis and Burns, 2005, Davis and Burns, 2008, Yoon et al., 2013, Saewong et al., 2002, Easwaran, 2007, Shin and Lee, 2008, Dewan and Fisher, 2010].

Our LP formulation is based on the one in [Lee et al., 2004] which used the formulation for QoS management in admission control, thus the formulation is partly found in [Lee et al., 2004]. However, the formulation works on only a *non-hierarchical* environment, while we formulate the LP to deal with hierarchical IMA structure. The authors [Park et al., 1995] presented an LP formulation which is not polynomial for an analogous problem with one in [Lee et al., 2004]. In non-hierarchical system for this problem which regards no period information, the authors of [Kuo and Mok, 1991, Kuo and Mok, 1997] showed the utilization bound depending on the number of harmonic groups of tasks. On the other hand the authors in [Lauzac et al., 2003] presented another bound regarding the ratio between the longest and shortest period. An exponential complexity algorithm in the integer domain for that problem was presented in [Chen et al., 2003]. More other relevant work can be found in [Han and Tyan, 1997, Liebeherr et al., 1995, Kuo et al., 2003].

3.2 System Model

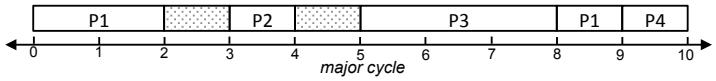
In thie section, we overview IMA system, and present system and problem description.

3.2.1 Integrated Modular Avionics System

In IMA system, a partition is a hierarchical running environment for tasks. Application tasks belong to an IMA partition and run only when their partition is active, which achieves temporal isolation between partitions. Once their assigned partition finishes, any running task should be suspended. Since partitions are based on cyclic executive, a partition is assigned by certain time slots to run, and the assigned schedule repeats every *major cycle*. For example in Fig. 3.1 (a), slot 3 is assigned to partition 2 and slots 5–7 are assigned for partition 3 while slot 2 and 4 are not assigned to any partition, and this partition schedule repeats every 10.

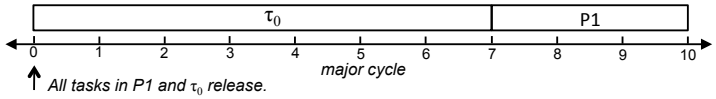
Tasks in different partitions do not share the same address space or any global variables, and an IMA system does not allow dynamic resource allocation such as dynamic memory allocation. These have partitions be spatially isolated, and prevent any failure propagation from a partition to another, which is the aim of designing IMA system [Rushby, 1999]. Also, device I/O transaction atomically begins and finishes within a single slot.[1] In addition to that, each

[1] Actually, an I/O transaction is done in a special-purpose partition called zero partition or device management partition to perform I/O transac-



(a) Partition schedule in a major cycle.

(b) An example rearrangement of partitions in (a).



(c) Overall system schedule in the perspective of a task in P1 when all tasks and τ_0 release at time t=0, i.e., in the worst-case. Note that τ_0 has the highest priority.

Figure 3.1: Partition schedule.

IMA partition can have its own scheduling policy, which eases partition-level development, migration or certification. Throughout this chapter, we assume that tasks run on RM scheduling policy.

3.2.2 System Description

At the higher level in a hierarchical IMA system, a partition contains a set of periodic application tasks $\mathbf{\Gamma} = \{\tau_i | i = 1, \ldots, n\}$. Each τ_i is represented by $\tau_i := (e_i, p_i)$ where e_i is the worst-case execution time and p_i is the period which is equal to the relative deadline. Without loss of generality, tasks are sorted according to their priorities in decreasing order, i.e., τ_1 is the highest priority and τ_n is the lowest, that is, p_1 is the shortest and p_n is the longest according to RM policy. We assume no task release jitter. Context switching overheads of application tasks and partitions are assumed to be zero.

The total utilization of all the application tasks in a single partition is denoted by U_{total} and defined as follows:

Definition 1. $U_{total} = \sum_{i=1}^{n} \frac{e_i}{p_i}$.

Also, IMA *partition capacity* is defined as follows:

Definition 2. *Partition capacity* $= \dfrac{\textit{\# of assigned slots}}{\textit{major cycle}}$.

For instance, in Fig. 3.1 (a) and (b), partition 1's capacity is $\frac{3}{10} = 0.3$ and partition 2's is $\frac{1}{10} = 0.1$. In our model, the major cycle is not longer than any task's period. However, this *never* implies that if the major cycle is longer than a task's period the task misses its deadline. For example, if we are given task $\tau_i = (e_i, p_i) = (1, 5)$, (deadline$= 5$) and a partition with major cycle of 10, the task always meets its deadline wherever the partition slots are allocated as long as the partition takes 6 or more slots (*i.e.*, partition's capacity ≥ 0.6).

tions in a consolidated fashion and thus has simplified both implementation and management of I/O operations. However, it is out of scope of this work. Interested readers can refer to [Rushby, 1999, Krodel, 2004, Parkinson and Kinnan, 2007, Kim et al., 2013] for the full details.

This issue is also connected with the worst-case partition slot arrangement. A task in a partition experiences the worst-case response time when it releases at (*i.e.*, just right after) the finishing instant of its partition and all the partition slots aggregate together (see [Davis and Burns, 2008, Sha, 2003]). This is visually described in Fig. 3.1 (c). Partition 1's slots are consecutively allocated from $t = 7$ to 10, and a task in partition 1 releases at $t = 0$ ($= 10, \because$ the major cycle is 10), which is the worst-case situation for the task regardless of other partitions' arrangement, since the task's slot allowed to run was just past, thus it should wait to run until the next cycle.

In order to model the worst-case situation, we apply the concept of VM Periodic Task in [Sha, 2003] to a single task, τ_0. τ_0 is relatively defined for the tasks in each partition.

Definition 3. *τ_0: in the perspective of a task in a partition,*

- *τ_0 is the highest priority task in the entire system, and*
- *τ_0's execution time is the sum of the slots not assigned to its own partition, and*
- *τ_0's period is the major cycle.*

For example, whatever the original schedule was one such as Fig. 3.1 (a) or (b), in the perspective a task in partition 1, τ_0's execution time $e_0 = 7$ and period $p_0 = 10 =$ major cycle, which can be represented as Fig. 3.1 (c). Since τ_0 holds the highest priority, the task experiences the worst-case response time when it simultaneously releases with τ_0 at $t = 0$ by the critical instant theorem of Liu & Layland [Liu and Layland, 1973]. In a hierarchical partition scheduling, a task cannot run on the unassigned slots (other partition's assigned slots and the empty slots) but can run only on its own partition's assigned slots. By keeping the highest priority and having execution time as much as the sum of unassigned slots for the partition, τ_0 realizes the effect. Since τ_0 inherently models the worst-case effect in the schedulability of a task, now an arrangement of partitions does not affect the schedulability and thus can be flexible. That is, the partition schedule in Fig. 3.1 (a) can be converted into the one in Fig. 3.1 (b) or anything as long as all the partitions' capacities are still the same, which means that we only need partition capacities but not an actual arrangement.

3.2.3 Problem Description

The problem is stated as follows: for a partition, we derive the maximal sufficient bound, $U_{bound}(\tau_i)$ for task τ_i, that minimizes the total tasks' utilization when

- major cycle and partition capacity (*i.e.*, $\tau_0 = (e_0, p_0)$), and
- application tasks' periods, i.e., $\{p_k | k = 1, \ldots, i\}$

are given. Then the maximal sufficient bound for all the tasks is denoted by U_{bound} and represented by

$$U_{bound} = \min_{1 \le i \le n} \Big(U_{bound}(\tau_i) \Big).$$

3.3 Maximal Sufficient Utilization Bound

We start our analysis by deriving the schedulability bound of each task in a partition, $U_{bound}(\tau_i)$. Then the minimum among each resulted bound is *the* bound, U_{bound}. If $U_{bound} \geq U_{total}$, the set of tasks in the partition is schedulable, otherwise it is not. Hence, we start deriving the schedulability bound for each task τ_i $(1 \leq i \leq n)$ in a partition.

3.3.1 Bound for Each Task in a Partition, $U_{bound}(\tau_i)$

A trivial sufficient bound to guarantee schedulability is an arbitrarily small value which is not useful in practice. Thus, we need to obtain the maximal sufficient bound for task τ_i. To find the maximal sufficient condition for task τ_i, we search for a task set with the minimal utilization in a partition characterized by the following three conditions identified by Liu and Layland in [Liu and Layland, 1973]. We shall refer them as ***L&L conditions***.

- **Condition 1:** The preemption to task τ_i is maximized by having all the tasks simultaneously begin at time $t = 0$.
- **Condition 2:** Task τ_i completes before its first deadline.
- **Condition 3:** The processor is fully utilized in $[0, p_i]$.

Since task periods are given, with those constant periods the maximal sufficient bound can be obtained by a linear programming as being subject to L&L conditions.

Condition 1 is easily addressed by indexing all the tasks' execution start times at $t = 0$. Condition 2 is also easy to be represented, which can be checked by summing up all the preemptions on task τ_i and τ_i's execution time, and then ensuring if the sum is larger than τ_i's deadline $(= p_i)$ or not.

However, Condition 3 cannot be easily addressed since it would need a potentially large number of constraints. Since Condition 3 ensures that the bound is *the maximal* sufficient bound not just a sufficient one, it requires the time duration $[0, p_i]$ to be fully utilized. To satisfy the condition, no idle time is supposed to be in $[0, p_i]$. If there is an idle time in $[0, p_i]$ the bound would not be maximally sufficient, since that means there is still room to be utilized as much as the idle time. Thus, execution times can increase until the duration $[0, p_i]$ gets filled – we call this process as a *maximalization* process which leads a sufficient bound to the maximally sufficient one.

Linear Programming Formulation

For the first step to represent Condition 3, let us account for the preemption in a form of linear constraints. In Fig. 3.2, we are interested in the schedulability of τ_3. From the perspective of task τ_3, the preemption from each higher priority task can be divided into two portions - *overflow* part and *non-overflow* part. Since the deadline of

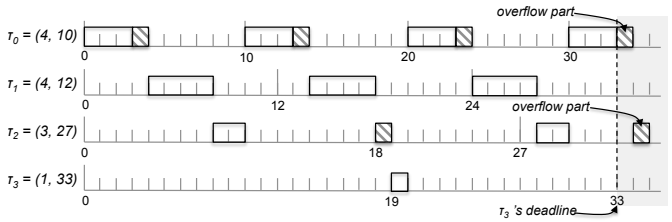


Figure 3.2: Preemption on τ_3, by non-overflow and overflow part.

τ_3 is 33, the execution of the higher priority tasks of τ_3 which occurs after 33 is classified as overflow part, while the remaining execution which happens before 33 is non-overflow part. This classification applies to any instance. For example, since one slot of τ_0's forth instance runs after τ_3's deadline 33, in every τ_0's instance overflow part for τ_3 denoted by $e^{over}_{0\to3}$ is 1 (diagonally striped in Fig. 3.2), while the non-overflow part denoted by $e^{non}_{0\to3}$ is 3. Likewise, τ_1's overflow part, $e^{over}_{1\to3}=0$ and τ_2's $e^{over}_{2\to3}=1$. For a bound of task τ_i, generally, for its higher task τ_h, $e^{over}_{h\to i}=\max\left(\left(\left\lfloor\frac{p_i}{p_h}\right\rfloor\cdot p_h+e_h\right)-p_i,0\right)$, and $e^{non}_{h\to i}=e_h-e^{over}_{h\to i}$.

In the example, the non-overflow part of τ_0 preempts τ_3 four times, while its overflow part preempts τ_3 only three times. Generally, the total preemption from a high priority task τ_h to τ_i is $\left\lceil\frac{p_i}{p_h}\right\rceil e^{non}_{h\to i}+\left\lfloor\frac{p_i}{p_h}\right\rfloor e^{over}_{h\to i}$. Hence, the total preemption to task τ_i from all the higher priority tasks can be represented as $\sum_{h=0}^{i-1}\left(\left\lceil\frac{p_i}{p_h}\right\rceil e^{non}_{h\to i}+\left\lfloor\frac{p_i}{p_h}\right\rfloor e^{over}_{h\to i}\right)$. Now, suppose that there exists a set of linear constraints $\mathbf{R}^*$ (will be shown later), in which $e^{non}_{h\to i}(0\le h\le i-1)$ still stay as non-overflow parts during a maximalization process and the processor is fully utilized in the interval $[0,p_i]$. Now let us define $U_{bound}(\tau_i)$.

Definition 4. $\mathbf{U_{bound}}(\tau_\mathbf{i})$*: denotes the maximal sufficient bound for task* τ_i *in the set of tasks* $\{\tau_k|1\le k\le i\}$.

Lemma 2. *The maximal sufficient bound* $U_{bound}(\tau_i)$ *for scheduling* τ_i *is obtained by the linear programming problem,*

$$
\begin{aligned}
&U_{bound}(\tau_i)=\min\left(\sum_{k=1}^{i}\frac{e_k}{p_k}\right)\\
&\textit{subject to } \mathbf{R}^*\\
&\textit{and}\qquad \sum_{h=0}^{i-1}\left(\left\lceil\frac{p_i}{p_h}\right\rceil e^{non}_{h\to i}+\left\lfloor\frac{p_i}{p_h}\right\rfloor e^{over}_{h\to i}\right)+e_i=p_i. \qquad (3.1)\\
&\textit{where}\qquad e^{non}_{0\to i}, e^{over}_{0\to i},\ \textit{and } p_k(0\le k\le i)\ \textit{are constant};\\
&\qquad\qquad e_k(1\le k\le i)\ \textit{are decision variables}.
\end{aligned}
$$

Proof. It directly follows L&L conditions - (3.1) represents Condition 1 and 2, and $\mathbf{R}^*$ does Condition 3. □

Before identifying $\mathbf{R}^*$, we will eliminate all overflow variables in (3.1) by setting the values to zero, except that of τ_0. Since for every task p_i is constant, the overflow part of e_0 does not change. Thus, after maximalization process,

we would not need to explicitly list the overflow variables (except τ_0's).

Lemma 3. *When the solution of the linear programming problem in Lemma 2 is attained,* $e_{h\to i}^{over} = 0 \quad (1 \le h \le i-1)$.

Proof. (Although similar proving can be found in Lemma 1 in [Lee et al., 2004], we prove it differently here for our context.) Assume that the minimal utilization task set has at least one overflow variable which is greater than 0, *i.e.*, $e_{h\to i}^{over} > 0$, for some h. Then the processor utilization contributed by $e_{h\to i}^{over}$ is $\frac{e_{h\to i}^{over}}{p_h}$ and the preemption provided by $e_{h\to i}^{over}$ is $\left\lfloor \frac{p_i}{p_h} \right\rfloor e_{h\to i}^{over}$. We keep the processor busy during $[0, p_i]$ by setting $e_{h\to i}^{over} = 0$ and increasing e_i by $\left\lfloor \frac{p_i}{p_h} \right\rfloor e_{h\to i}^{over}$. Then the reduced utilization is $\frac{e_{h\to i}^{over}}{p_h}$ and the increased utilization is $(\left\lfloor \frac{p_i}{p_h} \right\rfloor e_{h\to i}^{over})/p_i$. Accordingly, the net processor utilization is reduced since

$$\frac{p_i}{p_h} e_{h\to i}^{over} \ge \left\lfloor \frac{p_i}{p_h} \right\rfloor e_{h\to i}^{over},$$

$$\frac{e_{h\to i}^{over}}{p_h} \ge \frac{\left\lfloor \frac{p_i}{p_h} \right\rfloor e_{h\to i}^{over}}{p_i},$$

$$\text{reduced utilization} \ge \text{increased utilization}.$$

Hence, the net is reduced while $[0, p_i]$ is fully utilized. This contradicts the assumption that the minimal utilization task set has a non-zero overflow variable except for τ_0. □

According to Lemma 3, now we have $e_{h\to i}^{over} = 0$ $(1 \le h \le i-1)$, and thus in Lemma 2 we can eliminate overflow variables, $e_{h\to i}^{over} (1 \le h \le i-1)$. Applying Lemma 2, the rewritten form of Lemma 2 is in Lemma 4 as follows:

Lemma 4. *The maximal sufficient bound* $U_{bound}(\tau_i)$ *for scheduling* τ_i *is obtained by the linear programming problem,*

$$U_{bound}(\tau_i) = \min \left(\sum_{k=1}^{i} \frac{e_k}{p_k} \right)$$

subject to $\mathbf{R}^*$

$$\textit{and } \left(\left\lceil \frac{p_i}{p_0} \right\rceil e_{0\to i}^{non} + \left\lfloor \frac{p_i}{p_0} \right\rfloor e_{0\to i}^{over} \right) + \sum_{h=1}^{i-1} \left(\left\lceil \frac{p_i}{p_h} \right\rceil e_h \right) + e_i = p_i \tag{3.2}$$

where

$e_{0\to i}^{non}, e_{0\to i}^{over}$, *and* $p_k (0 \le k \le i)$ *are constant*; $e_0 = e_{0\to i}^{non} + e_{0\to i}^{over}$;

$z_k (1 \le k \le \sum_{h=0}^{i-1} \left\lceil \frac{p_i}{p_h} \right\rceil)$: *series of all arrival instants in* $(0, p_i)$;

$e_k (1 \le k \le i)$ *are decision variables*.

Proof. It directly follows Lemma 2 and Lemma 3. □

The variables for arrival instants, z_k, are needed for $\mathbf{R}^*$, of which idea is also found in the formulation of Theorem 2 in [Lee et al., 2004]. Let us first identify $\mathbf{R}^*$ which ensures that the processor is busy in $[0, p_i]$ and the non-

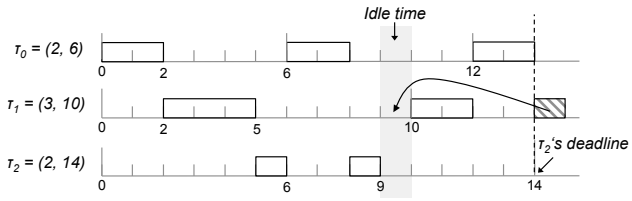


Figure 3.3: Interval $[0, 14]$ is not fully utilized (there is idle time) in spite of satisfying the equality constraint (3.2).

overflow variables are indeed non-overflow parts during the maximalization process. To do so, we first remove the $\mathbf{R}^*$ constraint from Lemma 4 and observe the implication. In Fig. 3.3. Consider the case of three tasks, $\tau_0(2, 6), \tau_1(3, 10)$ and $\tau_2(2, 14)$. These three tasks satisfy the equality constraint (3.2) in Lemma 4 as follows:

$$\left(\left\lceil \frac{p_2}{p_0} \right\rceil e^{non}_{0\to 2} + \left\lfloor \frac{p_2}{p_0} \right\rfloor e^{over}_{0\to 2}\right) + \left(\left\lceil \frac{p_2}{p_1} \right\rceil e_1\right) + e_2 = p_2,$$
$$\left(\left\lceil \frac{14}{6} \right\rceil 2 + \left\lfloor \frac{14}{6} \right\rfloor 0\right) + \left(\left\lceil \frac{14}{10} \right\rceil 3\right) + 2 = 6 + 6 + 2 = 14.$$

However, as we can see there is an idle time in [9,10] since the 2^{nd} invocation of τ_1 overflows. This cannot be checked only by the constraint (3.2) since it just ensures that the aggregate sum of all executions is p_i=14. Hence, we need Lemma 5 for non-overflow variables not to represent overflow executions.

Lemma 5. *By equality constraint (3.2) in Lemma 4, overflow in a high priority task's execution time implies that there exists an idle time in the interval* $[0, p_i]$.

Proof. Since the total processor time requested by all the tasks is equal to p_i, if any part of a high priority task executes after p_i, there must be an idle interval in $[0, p_i]$. □

Corollary 1. *By equality constraint (3.2) in Lemma 4, having no idle time in interval* $[0, p_i]$ *implies that all decision variables in Lemma 4 have no overflow parts, i.e.,* $e^{over}_{h\to i} = 0$ $(1 \le h \le i - 1)$.

Proof. This is the contrapositive of Lemma 5. Since Lemma 5 is true, this proposition is true, as well. □

Therefore, constraint $\mathbf{R}^*$ can be represented by a set of linear constraints which forces all possible gaps, *i.e.*, idle time intervals, to be filled with executions. In Fig. 3.3, the task arrival instants (except $t = 0$) are $z_1 = 6, z_2 = 10$ and $z_3 = 12$. Note that once an idle interval starts (if any), it must terminate at the right next task arrival instant anyway. Because at that point, the next new task arrives. In the example, the idle interval beginning at $t = 9$ terminates at $t = 10$. Hence, the constraint ensuring that the total processor demand at an arrival instant is greater than or equal to (the instant $- 0$), should be checked at every arrival instant. That is represented at (3.4) in Theorem 1.

Theorem 1. *The maximal sufficient bound $U_{bound}(\tau_i)$ for scheduling τ_i is obtained by the linear programming problem,*

$$U_{bound}(\tau_i) = \min\left(\sum_{k=1}^{i}\frac{e_k}{p_k}\right)$$

subject to

$$\left(\left\lceil\frac{p_i}{p_0}\right\rceil e_{0\to i}^{non} + \left\lfloor\frac{p_i}{p_0}\right\rfloor e_{0\to i}^{over}\right) + \sum_{h=1}^{i-1}\left(\left\lceil\frac{p_i}{p_h}\right\rceil e_h\right) + e_i = p_i \tag{3.3}$$

and

$$\sum_{h=0}^{i-1}\left(\left\lceil\frac{z_k}{p_h}\right\rceil e_h\right) + e_i \geq z_k \quad \left(1 \leq k \leq \sum_{h=0}^{i-1}\left\lfloor\frac{p_i}{p_h}\right\rfloor\right) \tag{3.4}$$

where

$e_{0\to i}^{non}, e_{0\to i}^{over}$, *and* $p_k (0 \leq k \leq i)$ *are constant*; $e_0 = e_{0\to i}^{non} + e_{0\to i}^{over}$;

$z_k (1 \leq k \leq \sum_{h=0}^{i-1}\left\lceil\frac{p_i}{p_h}\right\rceil)$: *series of all arrival instants in* $(0, p_i)$;

$e_k (1 \leq k \leq i)$ *are decision variables*.

Proof. The proving follows a part of proof of Theorem 2 in [Lee et al., 2004]. If there is an idle interval in $[0, p_i]$, it will be terminated by either an arrival of a higher priority task before $t = p_i$ or at $t = p_i$. However, either case does not happen since it contradicts the linear programming constraints that there cannot be an idle time before each task arrives. By Corollary 1, disappearing of idle time makes an overflow impossible. Finally, the equality constraint (3.3) ensures that task τ_i is schedulable in $[0, p_i]$. In summary, Theorem 1's linear constraints satisfy all the L&L conditions. Furthermore, there cannot be any overflow (except for τ_0), and thus we do not need overflow variables. It follows that $U_{bound}(\tau_i)$ is the maximal sufficient bound. □

3.3.2 Bound for All Tasks in a Partition, U_{bound}

Now we derive the maximal sufficient bound, U_{bound}, for all the tasks $\{\tau_i | i = 1, \ldots, n\}$ in a partition, by using the bound for each task, $U_{bound}(\tau_i)$, that we obtain in Section 3.3.1.

Theorem 2. *A set of tasks $\{\tau_i | i = 1, \ldots, n\}$ is schedulable if its total utilization U_{total} is less than or equal to U_{bound}, i.e., $U_{total} \leq U_{bound}$, where $U_{bound} = \min_{1\leq i\leq n}(U_{bound}(\tau_i))$.*

Proof. According to that $U_{total} \leq U_{bound}$ and $U_{bound} = \min_{1 \leq i \leq n}(U_{bound}(\tau_i))$,

$$U_{total} \leq U_{bound} = \min_{1 \leq i \leq n}(U_{bound}(\tau_i)). \tag{3.5}$$

$$\text{For } 1 \leq i \leq n, \quad \sum_{k=1}^{i} \frac{e_k}{p_k} \leq \sum_{k=1}^{n} \frac{e_k}{p_k} = U_{total}, \quad \text{and} \tag{3.6}$$

$$\min_{1 \leq i \leq n} \left(U_{bound}(\tau_i) \right) \leq U_{bound}(\tau_i). \tag{3.7}$$

According to (3.5), (3.6) and (3.7) for $1 \leq i \leq n$,

$$\sum_{k=1}^{i} \frac{e_k}{p_k} \leq \sum_{k=1}^{n} \frac{e_k}{p_k} = U_{total} \leq U_{bound} = \min_{1 \leq i \leq n} U_{bound}(\tau_i) \leq U_{bound}(\tau_i).$$

Finally, we can obtain $\sum_{k=1}^{i} \frac{e_k}{p_k} \leq U_{bound}(\tau_i) \quad (1 \leq i \leq n)$,

which shows that each task's scheduling bound is met. Thus, Theorem 2 holds. □

According to Theorem 2, ultimately the maximal sufficient utilization bound for a schedulable set of tasks in a partition, U_{bound}, is obtained by picking the minimum among $U_{bound}(\tau_1)$, $U_{bound}(\tau_2)$, $\ldots$, $U_{bound}(\tau_n)$. Let us see how the presented method works with a numerical example.

Example 1. *Suppose a partition and the major cycle is 10. In the partition there are two tasks, τ_1 and τ_2: τ_1's period p_1 is 12 and τ_2's period p_2 is 41.*

The resulted bounds according to partition capacity are shown in Table 3.1. According to Theorem 1 and 2, U_{bound} is derived by picking the minimum among τ_1's and τ_2's bound, $U_{bound} = \min(U_{bound}(\tau_1), U_{bound}(\tau_2))$. For instance, for capacity 0.9, *i.e.*, $\tau_0 = (e_0, p_0) = (1, 10)$, $U_{bound} = \min(U_{bound}(\tau_1), U_{bound}(\tau_2)) = \min(0.83, 0.82) = 0.82$. Table 3.1 also shows the bound from [Sha, 2003] (referred to as SHA's) and [Shin and Lee, 2003] (referred to as Shin's).[2] More discussion continues in the next section.

Table 3.1: Utilization bounds according to partition capacity for Example 1. No information on task is given for SHA's and Shin's while we are given task periods. The more known information leads to a higher bounds. ($U_{bound} = min(U_{bound}(\tau_1), U_{bound}(\tau_2))$)

partition capacity	$U_{bound}(\tau_1)$	$U_{bound}(\tau_2)$	U_{bound}	SHA's	Shin's
0.1	0.08	0.08	0.08	0.05	minus
0.3	0.25	0.25	0.25	0.16	0.001
0.5	0.41	0.43	0.41	0.28	0.12
0.7	0.58	0.62	0.58	0.43	0.33
0.9	0.83	0.82	0.82	0.59	0.64

[2]For Shin's, the bound and theorem presented in [Shin and Lee, 2003] (full proof presented in [Shin and Lee, 2010]) contain errors thus the authors corrected them on [Shin and Lee, 2010] for their extended journal version [Shin and Lee, 2008]. Ultimately, bounds presented in Table 3.1 are obtained from (32) and (33) presented in the updated version of [Shin and Lee, 2010].

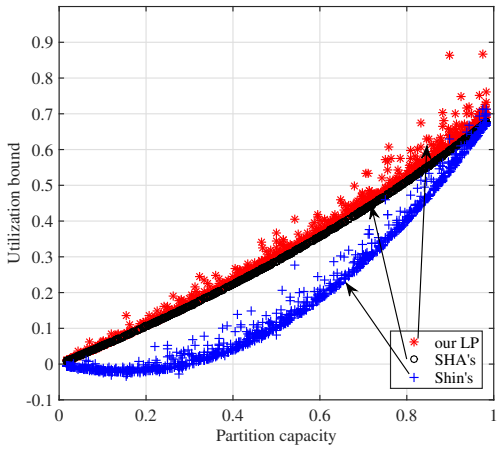


Figure 3.4: (Shown in color) Comparison of utilization bounds only for task sets in which the ratio between any two periods is less than 2. Thus, the task sets are *favorable* for SHA's and Shin's.

3.4 Evaluation

In Fig. 3.4 we compared the resulted bound from our LP with SHA's and Shin's for 1,000 samples: each sample task set contains 3–100 tasks with periods randomly generated from uniform distribution over 50–99 and major cycle over 30–60. As we can see the ratio between any two periods is less than 2 since Shin's bound (formulation (33) in [Shin and Lee, 2010]) can be applied only for such cases.[3] For that reason, our resulted bound is not that much higher than SHA's. Because in SHA's such task sets are assumed and then the bound is derived, which is favorable for SHA's bound. Hence, for a fair comparison, we ran another experiment with general periods for our LP and SHA's.

Fig. 3.5 shows the difference in utilization bound between our LP and SHA's according to a partition capacity. Differently from Fig. 3.4, in Fig. 3.5 ratio of a task's period to another can reach 30. Each task set also contains 3–100 tasks with periods randomly generated from uniform distribution over 10–300 and major cycle over 10–100. We ran 1,000 sample sets. The result shows that our LP enhances SHA's utilization bound since SHA's needs to suppose the worst-case periods and thus tries to reserve enough room for those tasks. Accordingly, the bound gets lower, *i.e.*, more conservative. On the other hand, since we are given the information for the task periods, we do not need to make the worst-case assumption on the task periods, which enables to spare more utilization for the tasks and thus raise the bound.

One of the issues we empirically found in using the bound is that large remainders between a task's period and its higher priority tasks' lower the bound. The smaller the difference, the likelier the task is to have a higher bound. For an example, just change the period of τ_2 from 41 to 60 in Example 1 to make the remainders zero, that is, $\{p_0 = 10, p_1 = 12, p_2 = 60\}$. In the case of the partition capacity for 0.9, *i.e.*, $\tau_0 = (1, 10)$, τ_1's bound is 0.83 while

[3] In [Shin and Lee, 2010], formulation (32) can be applied to a general case of periods, however, it can accommodate only 2 tasks.

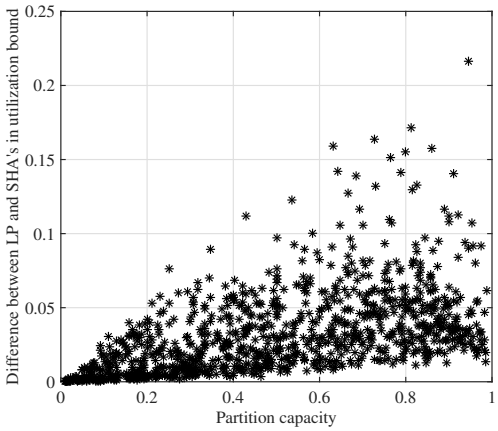


Figure 3.5: The difference in utilization bound between LP solution and SHA's, for general cases of task sets in which the ratio between any two periods is not restricted by 2. This shows how much 'known information (= known periods)' enhances the bound.

τ_2's bound is 0.9 which is the full utilization of the partition capacity. That is because, τ_2's period is divisible by τ_0 and τ_1. Such a case also corresponds to the known fact that in RM harmonic tasks achieve 100% of utilization. For the reason, if we were given a task set with large remainders, for an enhanced utilization bound we could consider the use of the period transformation method such as the one described in [Sha and Goodenough, 1990], which transforms a period into a smaller period which is harmonic in the task set, if it is possible to apply to the application.

Chapter 4

Budgeted Generalized Rate-Monotonic Analysis (Budgeted GRMS)

While a backward compatible solution for an IMA system greatly facilitates the aviation industry's transition from single core chip based systems to multicore based systems, there is a need to build a foundation for future systems with hundreds of cores. It is common in real-time project development, in accordance with good software engineering practice, that CPU budgets are allocated to different application teams early on, as a means of separation of concerns. CPU budgets are allocated to different applications and each application is composed of multiple periodic tasks that must share the same budget. Physical application requirements impose specifications on task periods and deadlines from the very beginning, but unlike the common assumption in traditional response time analysis, task execution times are not known. This is because task execution times depend on the exact system implementation, which is not finalized until later in the development cycle. Questions facing designers become: will my task meet its deadline given lack of knowledge of other tasks' execution times? What is the smallest deadline that my task can meet? These questions are traditionally addressed by using a two level scheduler, such as Periodic server or TDMA (*e.g.*, IMA): CPU is partitioned and assigned to application, and task priorities are determined within the scope of an application, and when server becomes active it schedules the tasks locally. Such two level scheduling approach introduces priority inversion across applications. In our approach, different applications' tasks are globally scheduled and yet the CPU resource is still partitioned and assigned to applications as a CPU budget. We schedule all the tasks globally while enforcing application budgets. The proposed new form of response time analysis is called *budgeted generalized rate-monotonic analysis* to compute the maximum response time for each task given only application budgets and task periods, but *without* knowledge of task execution times. We formulate this schedulability problem as a mixed integer linear programming problem and demonstrate a solution that computes the exact worst-case response times. Evaluation shows that our solution outperforms, in terms of schedulability, both global utilization bounds and mechanisms that attain temporal modularity via resource partitioning. The work of this chapter is published in [Kim et al., 2015a].

4.1 Introduction

Much research in real-time scheduling literature focused on response time analysis when the exact implementation of all tasks is available or, more specifically, when worst-case task computation times are known. While it is extremely valuable to be able to demonstrate temporal correctness before run-time, doing response time analysis at an advanced stage of development could be too late, since fixing schedulability problems is costly late in the development cycle. An important branch of response time analysis literature, therefore, is to analyze worst-case response times *early* in the development cycle, when many task execution parameters are unknown.

In this chapter, we present a new type of response time analysis, intended for use *early in the software development cycle*, called budgeted generalized rate monotonic analysis (Budgeted GRMS). We consider a system composed of multiple software subsystems, which we call applications. Each application is composed of multiple periodic tasks of possibly different priority. We further assume a constrained deadline model, where task deadlines are no larger than periods. All tasks share a single processor. The question addressed is to estimate a task's worst-case response time, despite lack of knowledge of computation times of (some of) the other tasks.

It is good in this context to define a notion of temporal modularity. Specifically, for the purposes of this work, we define temporal modularity as the *ability to compute task response times of one application without knowledge of task execution times of other applications*. In current practice, temporal modularity is attained via some form of resource partitioning. The resource partitioning approach ensures that response times of tasks depend only on other tasks in their own partition. For example, Integrated Modular Avionics (IMA) [air, 1991, Rushby, 1999, ARI, 2010], support isolation via IMA partitions. Within an IMA partition, an application is broken into tasks of different priorities that run only when the partition associated with their application is scheduled. Another example is servers, such as the Periodic Server [Davis and Burns, 2005], scheduled at a priority that corresponds to its period and allowing other tasks to be multiplexed on top. As we show later in the chapter, partitioning leads to priority inversion, where high-priority tasks of one application, whose partition (or server) is inactive, need to wait for another partition (or server) to finish, even if it serves lower-priority tasks.

A better approach from a schedulability perspective is to use a single global priority assignment across all tasks. In this case, however, temporal modularity is a problem because response-time of a task becomes a function of all higher priority tasks, some of which possibly belong to other applications. Hence, in the absence of constraints on individual applications, it becomes impossible to tell, early in the development cycle, if one task will meet its deadline when computation times of tasks in other applications are not known.

In order to gain the schedulability benefits of a global priority assignment while ensuring the temporal modularity of partitioning techniques, *utilization budgets* may be assigned to different applications. As long as the sum of utilizations of different applications (comprising sets of independent tasks) remains below the Liu and Layland bound [Liu

and Layland, 1973], the composition of these applications will be schedulable using a rate-monotonic priority assignment. Unfortunately, these bounds are pessimistic and require that deadlines be equal to periods. In most modern control systems, values from distributed sensors are sent to a controller on the network and actuator commands are sent to distributed actuators such as control surfaces in a plane, making the constrained deadline model more advantageous. Even though some tasks could have release jitter, in which case deadline minus monotonic priority order is optimal [Zuhily and Burns, 2007], in this work we limit our discussion with no release jitter.

To reconcile the high schedulability afforded by global priority-based scheduling with the clean temporal modularity afforded by resource partitioning, without incurring the limitations of utilization bounds, we propose a *partitioned, yet globally scheduled, enforced* uniprocessor task scheduling model. The model gives each application a utilization budget. *Exact response time analysis* is performed that assumes that all tasks will be globally scheduled in priority order using a single-level scheduler. The approach avoids priority inversion of partitioned systems and pessimism of utilization bounds. It also solves the aforementioned response time analysis problem exactly, producing the actual worst-case response time, given only task periods and budgets, but no computation times. We demonstrate, in the evaluation section, that our analysis leads to a higher utilization compared to the Liu and Layland bound as well as improved response times compared to resource partitioning and server-based techniques.

Note that, as development moves forward, more and more worst-case execution times become known (as the implementation of corresponding tasks is finished) and the maximum response-times computed for different tasks can be updated accordingly. Our analysis guarantees that the computed worst-case response-times never increase when more information on worst-case task execution times becomes known. This property is critical to the cost and schedule in real-time system development.

4.2 The Partitioned, yet Globally Scheduled, Enforced Model

Below we describe the scheduling model and offer a few examples to illustrate why it improves schedulability.

4.2.1 The Task Scheduling Model

The partitioned, yet globally scheduled, enforced uniprocessor (scheduling) model considers a set of N periodic tasks $\mathbf{\Gamma} = \{\tau_{i,j} | j = 0, \ldots, N-1\}$ in which each task $\tau_{i,j}$ belongs to an application i, App_i, $(i = 0, \ldots, M-1)$. Each task $\tau_{i,j}$ is represented by the tuple $\tau_{i,j} = (e_{i,j}, p_{i,j})$, where $e_{i,j}$ is the unknown execution time, and $p_{i,j}$ is the known period. Task $\tau_{i,j}$'s relative deadline, $d_{i,j}$, is assumed to be less than or equal to its period. Each application App_i is assigned a budget B_i at design time. Hence, for each App_i, when development is complete, the code should satisfy that $\sum_j \frac{e_{i,j}}{p_{i,j}} \leq B_i$. (We shall address the enforcement mechanism shortly, below.) It is then assumed that all tasks

will be scheduled at run-time in a generalized rate monotonic fashion. Hence, without loss of generality, we assume that tasks are sorted according to their priorities in a decreasing order, such that $\tau_{*,0}$ is the highest priority task and $\tau_{*,N-1}$ is the lowest priority one.[1] The above model possesses a few interesting properties:

- *Partitioned:* We call our model "partitioned" because, as a way of ensuring temporal modularity, each application is assigned its total CPU *utilization budget*, B_i, at design time, consistently with current engineering practice. All tasks belonging to that application must fit within its utilization budget in that their combined utilization must be less than or equal to the budget. Note that, application utilization budgets, used in our model, refer to a logical constraint only. These budgets do not correspond to an additional entity (such as a server) and do not need other parameters, such as a replenishment period.[2] Instead, budgets are fully specified by a utilization value.

- *Globally scheduled:* We call our model "globally scheduled" because all tasks of all applications are scheduled globally in a fixed-priority manner - no matter what application a task belongs to, it is scheduled by a fixed priority preemptive scheduling with deadline-monotonic priority assignment [Leung and Whitehead, 1982] on the uniprocessor. In scheduling, the absence of per-application resource partitions or servers significantly improves schedulability, as we show in the evaluation section.

- *Enforced:* We call our model "enforced" because when the system is ultimately implemented, the worst case execution time of each task will be known. It is therefore straightforward to check prior to deployment that the worst-case execution times and periods of tasks in each application comply with the application utilization budget constraint. Furthermore, since worst-case task execution times are known by deployment time, these worst-case execution times can be enforced for each task using existing operating system enforcement mechanisms (*e.g.*, by blocking tasks whose maximum execution time is exceeded [Herman et al., 2012] or wcet watchdog timers). Hence, one indirectly enforces the total utilization constraint of each application at run-time without resorting to application-wide partitions and other hierarchical scheduling mechanisms that result in unnecessary priority inversion.

The challenge is to estimate the worst-case response time of each task, early in the development cycle, when only task periods and application budget constraint are known, and before execution times of all tasks become available. Clearly, exact response time analysis should offer better solutions than the Liu and Layland utilization bound. Below, we also illustrate why the proposed model improves schedulability over two-level fashion scheduling.

Finally, while our analysis treats tasks as independent, it should be clear that the results also apply in schedulability tests that account for blocking over shared resources, for example, using priority inheritance or priority ceiling pro-

[1] For the sake of notational simplicity, we can use '*' in subscripts to represent 'any/some/all applications' depending on the context.
[2] As we show shortly, this does not mean they are not enforced.

tocols (assuming no deadlock) [Sha and Goodenough, 1990]. This is because the analytic treatment of blocking over shared resources simply lies in adding a calculated blocking time to the worst-case execution time of the task whose schedulability is analyzed. With that modification, the tasks are then treated as independent. Since our analysis does not assume specific values of computation times, they are trivially valid for *any* value of computation time, including transformed values that account for blocking.

4.2.2 Example

In this section, we present an illustrative example that compares our model to a periodic server [Shin and Lee, 2003, Davis and Burns, 2005, Yoon et al., 2013] (that restricts each application at run-time to a CPU share given by server bandwidth), and to resource partitioning via TDMA. Consider a system composed of two applications, whose task parameters are described in Table 4.1. Tasks within an application are indexed in global rate-monotonic priority order from highest priority to lowest. Note that, the utilization of this task set adds up to (1/3)/3 + 4/9 + 2/5 $\simeq$ 0.956 (i.e., 95.6%), which is above the Liu and Layland bound. Figure 4.1 shows the schedule using each of the three approaches compared.

Table 4.1: An Application Example.

	Task	Execution Time	Period (= Deadline)
App_1	Task $\tau_{1,1}$	1/3	3
	Task $\tau_{1,3}$	4	9
App_2	Task $\tau_{2,2}$	2	5

Figure 4.1(a) shows the schedule produced when the three tasks are scheduled in a rate-monotonic fashion. In the budgeted generalized rate-monotonic scheme, in order to produce this schedule, App_1 needs a utilization budget of (1/3)/3 + 4/9 = 0.556 (i.e., 55.6%) and App_2 needs a utilization budget of 2/5 = 0.4 (i.e., 40%), which is the sum of task utilizations within the respective applications.

Figure 4.1(b) shows what happens when App_1 and App_2 are mapped to periodic servers [Davis and Burns, 2005] of utilization 55.6% and 40%, respectively (i.e., the same values as computed above). We further assume that the period of each server is set to the smallest of the periods of application tasks mapped to that server. Hence, App_1 is mapped to a server of period=3 and execution budget of 3 * 0.556 $\simeq$ 1.67, whereas App_2 is mapped to a server of period=5 and an execution budget of 5 * 0.4 = 2. Servers are scheduled in rate-monotonic fashion. Note that, when both servers are invoked at the same time, task $\tau_{2,2}$ misses its deadline at time=5. This is because of the priority inversion that results from lumping the high-priority task, $\tau_{1,1}$, and the low-priority task, $\tau_{1,3}$, of App_1 into the same server, which ends up blocking the intermediate priority task $\tau_{2,2}$ of App_2.

Figure 4.1(c) shows what happens when a TDMA scheme is used. In this scheme, App_1 and App_2 are given alternating slices of utilization 55.6% and 40%, respectively, measured on a cycle of 3 time units. In this case, $\tau_{2,2}$

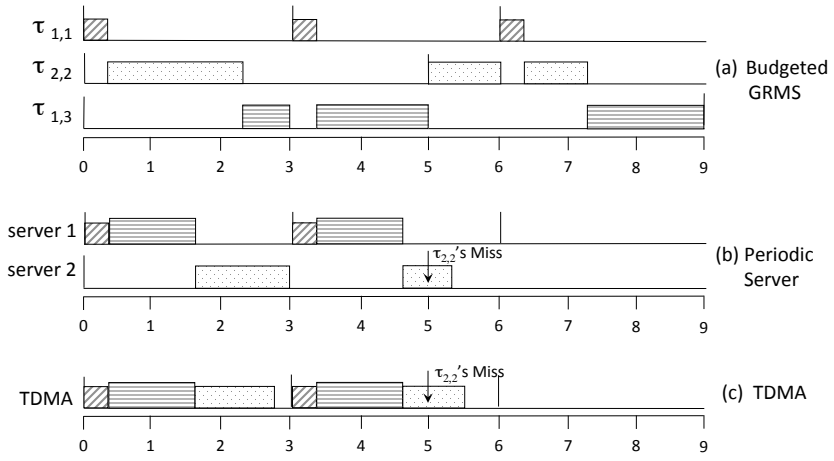


Figure 4.1: A comparison of three scheduling schemes for temporal modularity.

misses its deadline at time=5. The problem is attributed to both priority inversion as well as the mismatch between the TDMA cycle (3 time units) and the period of $\tau_{2,2}$. The difficulty in mapping tasks of different periods (or more generally, different rate groups) to a common TDMA cycle is a well-known problem with TDMA [Hsueh and Lin, 1996, Sha, 2003].

The example demonstrates the disadvantages of resource partitioning (Figure 4.1(b) and Figure 4.1(c)) compared to global rate monotonic scheduling (Figure 4.1(a)) on uniprocessors. The contribution of this work lies in solving the exact worst-case response-time analysis problem (i) in the absence of knowledge of task computation times, and (ii) without requiring resource partitioning at run-time, hence improving schedulability as suggested by Figure 4.1(a) above. Table 4.2 summarizes the comparison between the three approaches.

Table 4.2: A comparison of three scheduling schemes.

	TDMA	Periodic server	Our Budgeted GRMS
temporal modularity/isolation	√	√	√
flexibility (supporting multiple rate groups)		√	√
Absence of priority inversion			√

4.3 Budgeted Deadline-Monotonic Analysis

In the following subsections, we find an expression for the maximum response time of a task in our model, derive an expression for delay caused by higher priority tasks, understand the constraints imposed by busy periods, then derive a solution that solves the expressions for the worst-case scenario subject to the stated constraints. We formulate the problem as a Mixed Integer Linear Programming (MILP) since the objective of the problem is response time. As

known well the exact response time analysis is an iterative method [Joseph and Pandya, 1986]. That is because we do not know where the busy period would converge or even whether it would actually converge or not. For the issue, we introduce a variable called *preemption count*. Finally the variable is replaced by a different variable according to a technique called *elimination of products of variables*.

4.3.1 The Maximum Response Time of a Task, $\tau_{i,j}$

We formulate the proposed problem as a Mixed Integer Linear Programming. Here is its skeleton:

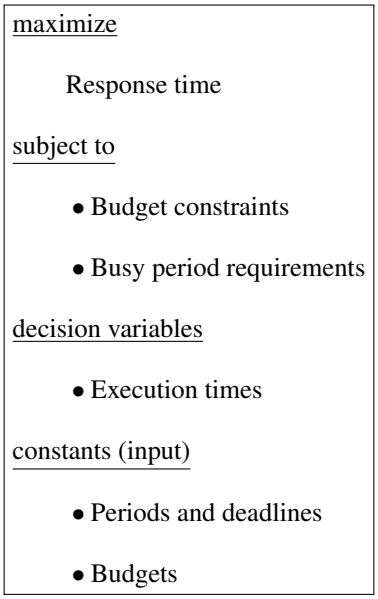


Figure 4.2: A skeleton of MILP formulation of the problem.

We take periods and deadlines of tasks and budget constraints for each application as input, then obtain the maximum response time which is achieved with a combination of possible task execution times subject to the budget constraints. As a result, the resulting objective value, $R_{i,j}$, is compared with the deadline, $d_{i,j}$. If

- $R_{i,j} \leq d_{i,j}$, then $\tau_{i,j}$ is schedulable,
- if not ($R_{i,j} > d_{i,j}$), then $\tau_{i,j}$ is not schedulable, since $\tau_{i,j}$ misses its deadline. That is, this formulation checks the schedulability of $\tau_{i,j}$ with considering all tasks in the system while the tasks are globally scheduled by generalized RM.

Our formulation is able to come up with a solution with no knowledge on task execution times at all. If some of the execution times are already known, the problem can be solved with the known values by trivially setting those execution time variables as constants. One thing we need to note is that our resulting execution times are the things obtained along with achieving the maximum value on response time of a single task to check its schedulability. In

other words, the execution times are not for being used in the final software. Rather than that, it means that any other combinations of the task execution times cannot make a longer response time than the obtained maximum one subject to the given constraints.

4.3.2 Budget Constraints

The representation of *the budget constraint* is straightforward. As we show in Lemma 6, it is represented as

$$\forall g: \quad \sum_{l\,:\,\tau_{g,l} \in App_g} \frac{e_{g,l}}{p_{g,l}} \leq B_g. \tag{4.1}$$

Lemma 6. *The representation of inequality* (4.1) *represents the utilization budget constraints which are given for the problem.*

Proof. As mentioned in Sec. 4.2 each application is assigned its total CPU utilization budget, B_i, at design time. All tasks belonging to that application must fit within its utilization budget in that their combined utilization must be less than or equal to the budget. Since these application utilization budgets refer to a logical constraint only, budgets are fully specified by a utilization value. Accordingly, inequality (4.1),

$$\forall g: \quad \sum_{l\,:\,\tau_{g,l} \in App_g} \frac{e_{g,l}}{p_{g,l}} \leq B_g,$$

directly represents the utilization budget constraints. □

4.3.3 Preemption Count from a Higher Priority Task, $I_{*,h}$

In order to calculate the exact response time of a task, the formula introduced in [Joseph and Pandya, 1986] is usually used. The exact method calculates the maximum possible preemption on the task from the higher priority tasks for a certain length of window and adds up the task's execution time. Then it repeats the calculation as iteratively increasing the window size until the window size exceeds the task's relative deadline (in this case, the task is determined to be unschedulable) or until the window size is stable. Then, the window size is the *busy period* or $R_{i,j}$ of the task and the maximum preemption from a higher priority task, $\tau_{*,h}$ is calculated as $\left\lceil \frac{R_{i,j}}{p_{*,h}} \right\rceil \cdot e_{*,h}$. However, such iterative method can only be applicable to brute-force optimization. Because of the ceiling function with unknown value, busy period, the term cannot be in our formulation. That is also why the formulation is an MILP not just an LP. Thus, in order to represent a busy period, we define a new variable, $I_{*,h}$ which is *the number of preemptions* from a higher priority task, $\tau_{*,h}$, as follows:

$$I_{*,h} = \left\lceil \frac{R_{i,j}}{p_{*,h}} \right\rceil. \tag{4.2}$$

Then, the response time can be represented as follows,

$$R_{i,j} = e_{i,j} + \sum_{h=0}^{j-1} I_{*,h} \cdot e_{*,h}, \tag{4.3}$$

where $0 \leq I_{*,h} \leq \lceil \frac{p_{i,j}}{p_{*,h}} \rceil + 1$. If $I_{*,h} > \lceil \frac{p_{i,j}}{p_{*,h}} \rceil$, $\tau_{i,j}$ misses its deadline as we show in Lemma 7. For task's schedulability test, we just determine if a task could miss its deadline or not, but do not care the value if the task misses its deadline anyway. Accordingly, in order to take into account cases that deadline is missed, we analyze it in $0 \leq I_{*,h} \leq \lceil \frac{p_{i,j}}{p_{*,h}} \rceil + 1$.

Lemma 7. *For a higher priority task $\tau_{*,h}$ of $\tau_{i,j}$, if $I_{*,h} > \lceil \frac{p_{i,j}}{p_{*,h}} \rceil$, $\tau_{i,j}$ misses its deadline.*

Proof.

$$\begin{aligned}
\text{If } I_{*,h} > \lceil \frac{p_{i,j}}{p_{*,h}} \rceil,\ \lceil \frac{R_{i,j}}{p_{*,h}} \rceil > \lceil \frac{p_{i,j}}{p_{*,h}} \rceil \because I_{*,h} = \lceil \frac{R_{i,j}}{p_{*,h}} \rceil \\
\lceil \frac{R_{i,j}}{p_{*,h}} \rceil \geq \lceil \frac{p_{i,j}}{p_{*,h}} \rceil + 1 \because \text{ both sides are integers,} \\
\text{then, } \frac{R_{i,j}}{p_{*,h}} + 1 > \lceil \frac{R_{i,j}}{p_{*,h}} \rceil \geq \lceil \frac{p_{i,j}}{p_{*,h}} \rceil + 1 \geq \frac{p_{i,j}}{p_{*,h}} + 1, \\
\therefore R_{i,j} > p_{i,j} \geq d_{i,j}.
\end{aligned}$$

Accordingly, since $R_{i,j} > d_{i,j}$, $\tau_{i,j}$ misses its deadline if $I_{*,h} > \lceil \frac{p_{i,j}}{p_{*,h}} \rceil$. □

4.3.4 Busy Period Requirements

Basically, since the scheduling is priority-driven and work-conserving, from a perspective of a higher priority task, $\tau_{*,h}$, at the instant when $\tau_{*,h}$ tries to start executing,

- it *preempts* $\tau_{i,j}$ if $\tau_{i,j}$ did not complete its execution yet (*i.e.*, if $\tau_{i,j}$'s busy period was not finished yet), or
- it would *not* be invoked if $\tau_{i,j}$ already completed its execution by/at the instant (*i.e.*, if $\tau_{i,j}$'s busy period was already finished).

This is the *busy period requirements*. That is, a higher priority task needs to check if $\tau_{i,j}$ has been completed or not at every instant of $\tau_{*,h}$ invocation. To satisfy this requirement, we introduce a binary variable $I^k_{*,h}$ which indicates if k^{th} instance of $\tau_{*,h}$ is invoked or not, as follows:

$$I^k_{*,h} = \begin{cases} 1 & \text{if } k^{th} \text{ invocation of } \tau_{*,h} \text{ preempts } \tau_{i,j}, \\ 0 & \text{otherwise,} \end{cases} \text{ and}$$

$$I^1_{*,h} \geq I^2_{*,h} \geq \ldots \geq I^{\lceil \frac{p_{i,j}}{p_{*,h}} \rceil + 1}_{*,h}. \tag{4.4}$$

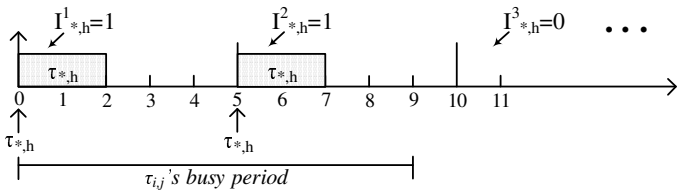


Figure 4.3: $I^k_{*,h}$ values are shown on busy period of 9 where $p_{*,h}$=5. Since $\tau_{*,h}$ can be invoked two times for 9, $I^1_{*,h} = I^2_{*,h} = 1$ and all the others for other k are 0.

Since busy period is a continuous duration time, the requirement (4.4) is needed. Accordingly, we can represent $I_{*,h}$ as a series of $I^k_{*,h}$ as follows:

$$I_{*,h} = I^1_{*,h} + I^2_{*,h} + \ldots + I^{\lceil \frac{p_{i,j}}{p_{*,h}} \rceil + 1}_{*,h} = \sum_{k=1}^{\lceil \frac{p_{i,j}}{p_{*,h}} \rceil + 1} I^k_{*,h}. \tag{4.5}$$

That is, $I_{*,h}$ indicates how many invocations contributes to the busy period and $I^k_{*,h}$ indicates which invocation does it. For example, for a busy period 9 of $\tau_{i,j}$, $\tau_{*,h}$ with period 5 can preempt $\tau_{i,j}$ 2 times ($I_{*,h} = 2$), with the first 2 invocations ($I^1_{*,h} = 1$ and $I^2_{*,h} = 1$) as shown in Figure 4.3. That is,

$$\begin{array}{ccccccc} I_{*,h} & = & I^1_{*,h} & + & I^2_{*,h} & + & I^3_{*,h} + \ldots + I^{\lceil \frac{p_{i,j}}{p_{*,h}} \rceil + 1}_{*,h}, \text{ and} \\ \| & & \| & & \| & & \| \\ 2 & & 1 & & 1 & & 0 \end{array}$$

$$\begin{array}{ccccccc} I^1_{*,h} & \geq & I^2_{*,h} & \geq & I^3_{*,h} & \geq \ldots \geq & I^{\lceil \frac{p_{i,j}}{p_{*,h}} \rceil + 1}_{*,h}. \\ \| & & \| & & \| & & \| \\ 1 & & 1 & & 0 & & 0 \end{array}$$

Through the notation of $I_{*,h}$ and $I^k_{*,h}$ we can represent the *busy period requirements* as follows:

for each higher priority task, $\tau_{*,h}$, $0 \leq h \leq j-1$, and $k = 1, 2, \ldots, \lceil \frac{p_{i,j}}{p_{*,h}} \rceil + 1$,

$$e_{i,j} + \sum_{z=0}^{j-1} \left\lceil \frac{(k-1)p_{*,h}}{p_{*,z}} \right\rceil e_{*,z} > I^k_{*,h} \cdot p_{*,h} \cdot (k-1), \tag{4.6}$$

$$R_{i,j} > (I_{*,h} - 1) \cdot p_{*,h}, \tag{4.7}$$

Lemma 8. *Inequality* (4.6) *ensures the busy period requirements.*

Proof. In inequality (4.6), $p_{*,h} \cdot (k-1)$ represents the instants where $\tau_{*,h}$ is invoked. Until the instants if $\tau_{i,j}$ did not complete yet, $\tau_{*,h}$ preempts $\tau_{i,j}$, which is $I^k_{*,h} = 1$. In other words, the relative window from release to an instant

should be less than the all requested executions in the window, *i.e.*,

$$\Big(\text{all requested executions in } p_{*,h} \cdot (k-1)\Big) > p_{*,h} \cdot (k-1)$$
$$e_{i,j} + \sum_{z=0}^{j-1} \left\lceil \frac{(k-1)p_{*,h}}{p_{*,z}} \right\rceil e_{*,z} > p_{*,h} \cdot (k-1).$$

As mentioned above, since $I^k_{*,h}$ is an invocation indicator that $\tau_{*,h}$ preempts $\tau_{i,j}$, we multiply $I^k_{*,h}$ on the right hand side - since $I^k_{*,h} = 1$ in this case, it does not affect the result. Thus, we can get the form of inequality (4.6). However, by/at every instant, if $\tau_{i,j}$ already completed, a higher priority task, $\tau_{*,h}$ should not be invoked, which is $I^k_{*,h} = 0$. In this case, the inequality returns true anyway. As a result, in either case, inequality (4.6) is satisfied and thus is included in the formulation to represent the busy period requirements. □

We can look into inequality (4.6) by the previous example. In Figure 4.3, since $\tau_{*,h}$'s period $p_{*,h}$ which is 5 is less than the busy period, the amount required until time 5 should be larger than 5. If the amount required until time 5 was less than 5, the busy period must have finished before time 5. For that reason, the 2^{nd} instance ($k = 2$) of $\tau_{*,h}$ would be invoked and thus $I^2_{*,h} = 1$ in (4.6),

$$e_{i,j} + \left\lceil \frac{(2-1) \cdot 5}{5} \right\rceil e_{*,h} > I^2_{*,h} \cdot 5 \cdot (2-1).$$
$$e_{i,j} + e_{*,h} > 5.$$

But right before the 3^{rd} invocation *i.e.*, at time 10, since the busy period already finished, it does not satisfy (4.6). Thus $I^3_{*,h} = 0$, and (4.6) does not semantically mean anything while it is mathematically true for the formulation, *i.e.*,

$$e_{i,j} + \left\lceil \frac{(3-1) \cdot 5}{5} \right\rceil e_{*,h} > I^3_{*,h} \cdot 5 \cdot (3-1),$$
$$e_{i,j} + 2 \cdot e_{*,h} > 0.$$

This is the reason why non-negative integer $I_{*,h}$ is represented by a series of binary $I^k_{*,h}$. By applying the substitution (4.5), the response time is rewritten as shown in Lemma 9.

Lemma 9. *The response time of $\tau_{i,j}$ is represented as $R_{i,j} = e_{i,j} + \sum_{h=0}^{j-1} \left(\sum_{k=1}^{\lceil \frac{p_{i,j}}{p_{*,h}} \rceil + 1} I^k_{*,h} \right) \cdot e_{*,h}$.*

Proof.

$$
\begin{aligned}
R_{i,j} &= e_{i,j} + \sum_{h=0}^{j-1} I_{*,h} \cdot e_{*,h}, \quad \text{by (4.3)} \\
&= e_{i,j} + \sum_{h=0}^{j-1} \left(I_{*,h}^1 + I_{*,h}^2 + \ldots + I_{*,h}^{\lceil \frac{p_{i,j}}{p_{*,h}} \rceil + 1} \right) e_{*,h}, \\
&= e_{i,j} + \sum_{h=0}^{j-1} \left(\sum_{k=1}^{\lceil \frac{p_{i,j}}{p_{*,h}} \rceil + 1} I_{*,h}^k \right) \cdot e_{*,h}, \quad \text{by (4.5)}.
\end{aligned}
$$

□

At last, inequality (4.7) ensures that response time should be larger than the number of periods before finishing time. Without this constraint, the preemption count can be over-counted which results an over-calculated response time.

Lemma 10. *By (4.2),* $I_{*,h} = \left\lceil \frac{R_{i,j}}{p_{*,h}} \right\rceil$*, we can get* $(I_{*,h} - 1) \cdot p_{*,h} < R_{i,j}$ *which is (4.7).*

Proof. In inequality (4.7), the left hand side,

$$
\begin{aligned}
(I_{*,h} - 1) \cdot p_{*,h} &= \left(\left\lceil \frac{R_{i,j}}{p_{*,h}} \right\rceil - 1 \right) \cdot p_{*,h}, \quad \text{by (4.2)} \\
&< \left((\frac{R_{i,j}}{p_{*,h}} + 1) - 1 \right) \cdot p_{*,h}, \\
&< \frac{R_{i,j}}{p_{*,h}} \cdot p_{*,h} = R_{i,j}.
\end{aligned}
$$

Therefore, $(I_{*,h} - 1) \cdot p_{*,h} < R_{i,j}$. □

This inequality makes the preemption count $I_{*,h}$ not unnecessarily explode and thus ensures that the response time duration is packed with real executions with no empty spaces. Now we can write the formulation with the busy period requirements and the budget constraint as shown in Fig. 4.4.

Theorem 3. *The problem of finding the maximal response time of task* $\tau_{i,j}$ *in a busy period, subject to budget constraints, reduces to the problem formulation shown in Fig. 4.4.*

Proof. To show that the theorem (and, hence, the problem formulation) is correct, we use the lemmas covered so far. First, note that, the expression of the objective function (*i.e.*, response time), in the presence of preemptions, given by Equation (4.8) in the problem formulation, is from Lemma 9. We have two constraints in the original problem formulation (see page 3). The first is the budget constraint. The second is the fact that execution constitutes a busy period (due to work-conserving scheduling), which we call the busy-period constraint. Constraint (4.9) represents the budget constraint by Lemma 6. Constraint (4.10) represents the busy-period constraint by Lemma 8. The remaining

constraints simply stem from the definitions of the used variables. Specifically, the number of preemptions by a higher priority task, $I_{*,h}$ is given by Equation (4.2). This equation implies the linear Constraint (4.11), according to Lemma 10. Finally, Constraint (4.12) reflects the fact that the invocation indicator variables are all 1s (for invocations that delay $\tau_{i,j}$) followed by all zeros (for those that do not). Hence, for any two consecutive invocation indicator variables, the earlier one is always larger than or equal to the latter. The sum of these variables is exactly the total number of preemptions, $I_{*,h}$, per Equation (4.13). It is desired to find the task execution times and indicator variables (i.e., corresponding preemption patterns) that maximize the stated objective function subject to the above constraints. This completes the problem formulation. □

maximize

$$R_{i,j} = e_{i,j} + \sum_{h=0}^{j-1} \Big(\sum_{k=1}^{\lceil \frac{p_{i,j}}{p_{*,h}} \rceil + 1} I_{*,h}^{k} \Big) \cdot e_{*,h} \tag{4.8}$$

subject to

- $\forall g$:

$$\sum_{l \,:\, \tau_{g,l} \in App_g} \frac{e_{g,l}}{p_{g,l}} \leq B_g \tag{4.9}$$

- for $h = 0, 1, \ldots, j-1$ and $k = 1, 2, \ldots, \lceil \frac{p_{i,j}}{p_{*,h}} \rceil + 1$:

  - $e_{i,j} + \sum_{z=0}^{j-1} \Big\lceil \frac{(k-1)p_{*,h}}{p_{*,z}} \Big\rceil e_{*,z} > I_{*,h}^{k} \cdot p_{*,h} \cdot (k-1)$ (4.10)

  - $(I_{*,h} - 1) \cdot p_{*,h} < R_{i,j}$ (4.11)

  - $I_{*,h}^{1} \geq I_{*,h}^{2} \geq \ldots \geq I_{*,h}^{\lceil \frac{p_{i,j}}{p_{*,h}} \rceil + 1}$ (4.12)

  - $I_{*,h} = \sum_{k=1}^{\lceil \frac{p_{i,j}}{p_{*,h}} \rceil + 1} I_{*,h}^{k}$ (4.13)

decision variables

- $\forall m, n : 0 \leq e_{m,n} \leq p_{m,n}$
- for $h = 0, 1, \ldots, j-1$ and $k = 1, 2, \ldots, \lceil \frac{p_{i,j}}{p_{*,h}} \rceil + 1$:

  $I_{*,h}^{k} \in \{0, 1\}$

constants (input)

- $\forall m, n : p_{m,n}$ and $d_{m,n}$
- $\forall g : B_g$

Figure 4.4: Formulation of the problem.

4.3.5 Elimination of Products of Variables

Now we eliminate a product of variables in the objective (4.8) by a technique called *Elimination of products of variables* described in [Bisschop, 2011]. [3] That is substituting a product of a binary variable and a continuous one into a new single variable with range requirements for the new variable. By the substituting the formulation still returns exactly the same solution, as we show in Lemma 11 and Theorem 4. In the formulation, we substitute $I^k_{*,h} \cdot e_{*,h}$ with $X^k_{*,h}$. That is, the value of $X^k_{*,h}$ is 0 or $e_{*,h}$, and thus $X^k_{*,h}$ represents a preemption duration by $\tau_{*,h}$'s k^{th} invocation. Accordingly,

$$X^k_{*,h} \le p_{*,h} \cdot I^k_{*,h}, \qquad X^k_{*,h} \le e_{*,h},$$
$$X^k_{*,h} \ge e_{*,h} - p_{*,h}(1 - I^k_{*,h}), \qquad X^k_{*,h} \ge 0.$$

The objective (4.8) is now rewritten as follows:

$$\begin{aligned} R_{i,j} &= e_{i,j} + \sum_{h=0}^{j-1} \Big(\sum_{k=1}^{\lceil \frac{p_{i,j}}{p_{*,h}} \rceil + 1} I^k_{*,h} \Big) \cdot e_{*,h} \\ &= e_{i,j} + \sum_{h=0}^{j-1} \Big(\sum_{k=1}^{\lceil \frac{p_{i,j}}{p_{*,h}} \rceil + 1} I^k_{*,h} \cdot e_{*,h} \Big) = e_{i,j} + \sum_{h=0}^{j-1} \Big(\sum_{k=1}^{\lceil \frac{p_{i,j}}{p_{*,h}} \rceil + 1} X^k_{*,h} \Big). \end{aligned}$$

Accordingly, by substituting $I^k_{*,h} \cdot e_{*,h}$ into $X^k_{*,h}$, we have the MILP formulation as shown in Fig. 4.5.

[3] Interested readers can refer to Sec. 7.7 Elimination of products of variables for the details in the book.

maximize

$$R_{i,j} = e_{i,j} + \sum_{h=0}^{j-1} \Big(\sum_{k=1}^{\lceil \frac{p_{i,j}}{p_{*,h}} \rceil + 1} X^k_{*,h} \Big)$$

subject to

- $\forall g$:

$$\sum_{l\,:\,\tau_{g,l} \in App_g} \frac{e_{g,l}}{p_{g,l}} \leq B_g$$

- for $h = 0, 1, \ldots, j-1$ and $k = 1, 2, \ldots, \lceil \frac{p_{i,j}}{p_{*,h}} \rceil + 1$:
  - $R_{i,j} > (I_{*,h} - 1) \cdot p_{*,h}$
  - $e_{i,j} + \sum_{z=0}^{j-1} \Big\lceil \frac{(k-1)p_{*,h}}{p_{*,z}} \Big\rceil e_{*,z} > I^k_{*,h} \cdot p_{*,h} \cdot (k-1)$
  - $I^1_{*,h} \geq I^2_{*,h} \geq \ldots \geq I^{\lceil \frac{p_{i,j}}{p_{*,h}} \rceil + 1}_{*,h}, \quad I_{*,h} = \sum_{k=1}^{\lceil \frac{p_{i,j}}{p_{*,h}} \rceil + 1} I^k_{*,h}$
  - $X^k_{*,h} \leq p_{*,h} \cdot I^k_{*,h}$ (4.14)
  - $X^k_{*,h} \leq e_{*,h}$ (4.15)
  - $X^k_{*,h} \geq e_{*,h} - p_{*,h}(1 - I^k_{*,h})$ (4.16)
  - $X^k_{*,h} \geq 0$ (4.17)

decision variables

- $\forall m, n : 0 \leq e_{m,n} \leq p_{m,n}$
- for $h = 0, 1, \ldots, j-1$ and $k = 1, 2, \ldots, \lceil \frac{p_{i,j}}{p_{*,h}} \rceil + 1$:

  $I^k_{*,h} \in \{0, 1\}, \quad X^k_{*,h}$

constants (input)

- $\forall m, n : p_{m,n}$ and $d_{m,n}$
- $\forall g : B_g$

Figure 4.5: Substituting $I^k_{*,h} \cdot e_{*,h}$ in the formulation in Fig. 4.4 into $X^k_{*,h}$.

Lemma 11. *Substituting $I^k_{*,h} \cdot e_{*,h}$ into $X^k_{*,h}$ does not change the solution of the problem in Fig. 4.4.*

Proof. Once the product $I^k_{*,h} \cdot e_{*,h}$ in which $0 \leq e_{*,h} \leq p_{*,h}$ is replaced by an additional continuous variable, $X^k_{*,h}$, the following constraints must be added to force $X^k_{*,h}$ to take the value of $I^k_{*,h} \cdot e_{*,h}$ as in (4.14), (4.15), (4.16) and

(4.17);

$$
\begin{aligned}
&X^k_{*,h} \leq p_{*,h} \cdot I^k_{*,h},\\
&X^k_{*,h} \leq e_{*,h},\\
&X^k_{*,h} \geq e_{*,h} - p_{*,h}(1 - I^k_{*,h}),\\
&X^k_{*,h} \geq 0.
\end{aligned}
$$

Suppose that $e_{*,h}$ has the value of w and $I_{*,h}$ is 0. Then, the above constraints result in

$$
\begin{aligned}
&X^k_{*,h} \leq 0,\\
&X^k_{*,h} \leq w,\\
&X^k_{*,h} \geq w - p_{*,h},\\
&X^k_{*,h} \geq 0.
\end{aligned}
$$

Since $w \leq p_{*,h}$, the only value of $X_{*,h}$ that can satisfy the above constraints is 0. Now suppose that $I_{*,h}$ is 1. Then,

$$
\begin{aligned}
&X^k_{*,h} \leq p_{*,h},\\
&X^k_{*,h} \leq w,\\
&X^k_{*,h} \geq w,\\
&X^k_{*,h} \geq 0.
\end{aligned}
$$

In this case, $X_{*,h}$ must be w in order to satisfy all the constraints. As a result, subject to (4.14), (4.15), (4.16) and (4.17), the substitution does not change the result in all possible cases. □

Theorem 4. *The problem of finding the maximal response time of task $\tau_{i,j}$ in a busy period, subject to budget constraints reduces to the problem formulation shown in Fig. 4.5.*

Proof. We have shown that the problem of finding the maximal response time of task $\tau_{i,j}$ in a busy period, subject to budget constraints is equivalent to the problem formulation shown in Fig. 4.4. Then according to Theorem 3 and Lemma 11, since the formulation in Fig. 4.4 is equivalent to the formulation in Fig. 4.5, the problem formulation shown in Fig. 4.5 is equivalent to the problem of finding the maximal response time of task $\tau_{i,j}$ in a busy period, subject to budget constraints. □

The formulation in Fig. 4.5 is equivalent to the one in Fig. 4.4 and can now be solved using a standard MILP solver.

Table 4.3: Timing Data of Generic Avionics Software in [Locke et al., 1990]. The WCRT of each task is calculated by our formulation. All the tasks are schedulable in the current set.

	Application	Budget		Function	Period = Deadline	WCRT
App_0	Navigation	0.20	$\tau_{0,6}$	Aircraft flight data	55	**34.0**
			$\tau_{0,7}$	Steering	80	**39.0**
App_1	Radar Control	0.05	$\tau_{1,1}$	Radar tracking	40	**3.0**
			$\tau_{1,8}$	Radar search	80	**46.0**
App_2	Targeting	0.10	$\tau_{2,2}$	Target tracking	40	**7.0**
App_3	Weapon control	0.10	$\tau_{3,0}$	Weapon release	10	**1.0**
			$\tau_{3,9}$	Weapon trajectory	100	**57.0**
App_4	Controls and Displays	0.25	$\tau_{4,3}$	HUD display	52	**22.0**
			$\tau_{4,4}$	MPD HUD display	52	**22.0**
			$\tau_{4,5}$	MPD tactical display	52	**22.0**
App_5	RWR Control and Threat response	0.05	$\tau_{5,10}$	RWR program	100	**73.8**
			$\tau_{5,11}$	Threat response display	100	**73.8**
			$\tau_{5,12}$	Poll RWR	200	**88.0**
6 applications		total = 0.75	13 tasks			all schedulable

Theorem 5. *A task is guaranteed to be schedulable subject to budget constraints if an only if its maximum response time, i.e., the MILP solution, is no longer than the deadline.*

Proof. ($\Rightarrow$) If a task that meets budget constraints is always schedulable, then its response time is always no larger than its deadline and therefore the maximum of all its possible response times subject to budget constraints is not larger than its deadline, which means that the MILP solution for this task is not larger than its deadline. This logic applies to every task in the task set.
($\Leftarrow$) If the MILP solution for a task is no larger than its deadline, then the maximum response time of this task subject to budget constraints is not larger than its deadline and hence any possible response time of this task subject to budget constraints is no larger than its deadline, meaning that this task is always schedulable. □

Note that, if some task execution times are known, we can set them as constants and solve for the remaining ones. Clearly, setting some computation times as constants reduces the set of all solutions considered over which response time is maximized. The maximum response time over the resulting subset of solutions therefore cannot be higher than the maximum over the original set. This observation suggests that the computed maximum response time either decreases or stays the same as more computation times become known.

4.3.6 Practical Example - Timing Data of Generic Avionics Software

We apply our formulation to a practical example presented in [Locke et al., 1990]. The example is a set of timing data of generic avionics software and shown in Table 4.3. All tasks' period information comes from there, but we assign the budget values of the applications based on the presented CPU times. Then we applied our formulation for each task to get its worst-case response time (WCRT) – thus we did 13 times of schedulability analyses since there are 13 tasks. Each WCRT is obtained by assigning the other tasks' execution times – which make the task of interest suffer

the worst response time. Thus for each WCRT value of each respective task, the other tasks' execution times might be assumed to be different. Through the analyses we obtained the WCRT of each task and we can see all the tasks are schedulable as the rightmost column shows in Table 4.3. The MILP has run on Intel Core i5 1.4GHz dual-core processor with 8GB RAM by using IBM CPLEX V12.6., and as an instance it took 0.08 sec to run this example for the schedulability for the lowest priority task, $\tau_{5,12}$.

For implementation purpose only, a task with lower ID (ID indicates the 2^{nd} subscript number) gets a higher priority among those who have the same period. For example in Table 4.3, $\tau_{3,9}$ obtains higher priority than $\tau_{5,10}$, no matter which application they are in. MPD is the abbreviation for Multi-Purpose Display, HUD is for Head Up Display, and RWR is for Radar Warning Receiver.

Trivially and obviously, for instance, since $\tau_{3,0}$ is the highest priority one, there is no preemption from any other higher priority task. Thus $e_{3,0} = 1.0$ and $e_{3,9} = 0.0$ is the optimal decision to make $\tau_{3,0}$'s WCRT longest and at the same time to satisfy the budget requirement of App_3, 0.10. Hence the $\tau_{3,0}$'s WCRT is 1.00. Any other combinations of the execution times cannot make the WCRT longer than 1.00 as long as the values satisfy the budget requirement, 0.10.

On the other hand, as we increase a budget, the WCRT of a task in a budget becomes longer and finally starts to be unschedulable. That is because as the budget capacity increases, the execution times of the tasks would grow as much as possible the budget requirement allows, and they make the worst-case response time of the task of interest finally miss its deadline.

4.4 Evaluation

In this section, we compare our response time analysis solution to several alternatives. The objective is to demonstrate two take-home points. First, the approach results in lower worst-case response times than the more traditional approaches based on resource partitioning or servers. This is because the approach allows individual tasks to be globally scheduled, hence eliminating priority inversion across tasks belonging to different application servers or resource partitions. Second, in the case where deadlines are equal to periods, the approach allows budgets to add up to more than the Liu and Layland utilization bound. This is because the analysis takes advantage of our knowledge of task periods. These points are illustrated in the two subsections below.[4]

4.4.1 Improved Response Time

As noted above, the proposed response time analysis achieves lower worst-case response times because tasks are globally scheduled. To demonstrate this point, we compare the response time computed by our approach to those

[4] All experiments ran on Intel Core i5 1.4GHz dual-core processor with 8GB RAM, and for MILP we used IBM CPLEX V12.6.

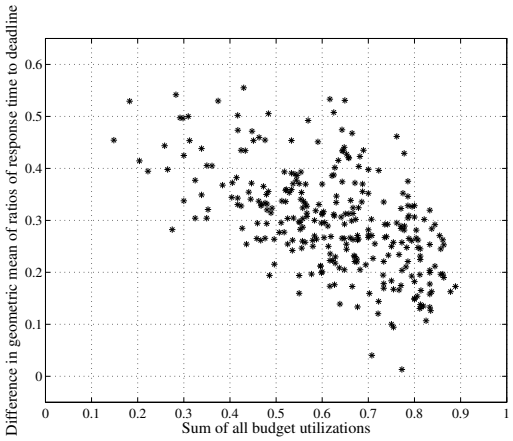


Figure 4.6: PS vs. Budgeted GRMS: difference in the geometric mean of ratios of the response time to the deadline. One dot in the graph represents difference of the means of a sample task set for PS and Budgeted GRMS.

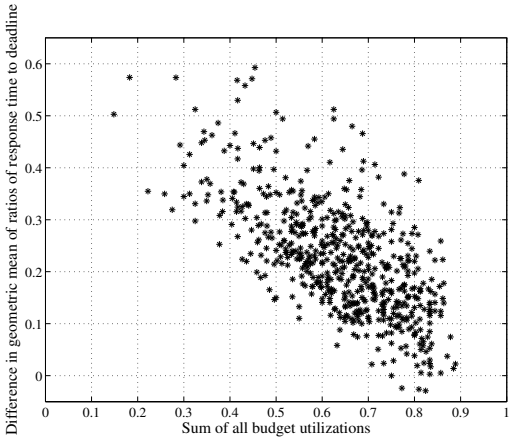


Figure 4.7: TDMA vs. Budgeted GRMS: difference in the geometric mean of ratios of the response time to the deadline. One dot in the graph represents difference of the means of a sample task set for TDMA and Budgeted GRMS. The input in PS is used but the server periods were transformed into a major cycle here.

computed for periodic server and strict resource partitions. More specifically, we compare the following techniques:

- *Partitioned, yet Globally Scheduled (Budgeted GRMS):* This is our model, where response times are computed using the MILP formulation detailed earlier in the chapter.

- *Periodic Server (PS):* In this approach, each application is assigned to a server. The period of the server is the greatest common divisor of the periods of tasks that belong to the server's application. The utilization of the server is equal to the application budget. This allows us to compute server parameters. The analysis in [Davis and Burns, 2005] (by R. Davis and A. Burns) is then used to compute response times. In principle, this analysis requires more information than ours; namely, to compute the response time of a task, it needs to know execution times of other tasks in the same application. For fairness, the execution times used here by

PS are also applied in Budgeted GRMS as constants. The point is to compare the worst case response times under the same conditions regarding computation times. This is meaningful, for example, when the application designer knows their application's task execution times, but not those of other applications.

- *Time Division Multiple Access (TDMA):* When TDMA is used, each application is assigned a TDMA partition. The utilization of the partition is equal to the application budget. For fairness, the input used in PS is also applied here for TDMA, but those server periods in PS are transformed to a multiple of the shortest period. Then the greatest common divisor of the periods is used as a major cycle for a TDMA system. However, we still keep the utilization of the corresponding TDMA partition as the same as the original server utilization in PS. The task information is still the same with the case of PS.

Figure 4.6 plots the differences between response times computed by our approach and those computed by PS (specifically, PS less ours), versus sum of budget utilizations. For ease of comparison, response time differences are normalized by corresponding task deadlines. Positive measurements indicate that our approach has lower normalized response times. Each point corresponds to the computed difference for one task set. We exclude points, where either approach fails to find a bounded response time for all tasks in the set. In the generated sets, the number of applications (and hence, servers) was randomly chosen between 2 and 10. The number of tasks per application was varied between 2 and 5. Since tasks belonging to the same application often belong to the same rate group, we generated task periods to be integer multiples (from 1 to 5) of a single base period (randomly chosen between 3 and 20), where the base period depends on the application. This base period of each application was then taken as the corresponding server period. Note that, across applications, base periods were random. Deadlines were chosen to be less than or equal to periods in the range between 0.8 and 1.0 period. Figure 4.6 shows that our response times are significantly lower.

It should be noted that the difference in response times between the two approaches is not due to lack of optimality of either schedulability analysis method. Both analysis approaches offer the exact solution for their respective scheduling models. The difference is attributed to the advantages of the partitioned, yet globally scheduled model. Namely, it avoids priority inversion at run-time, while achieving the same temporal modularity as resource partitioning via periodic servers.

Figure 4.7 repeats the above experiment, now comparing Budgeted GRMS versus TDMA. As before, positive normalized differences indicate that we have a lower response time. The input samples are the same (i.e., the same task sets are considered), except that server parameters are transformed into TDMA partition parameters. Since TDMA requires a unified cycle, the length of the cycle was chosen to be the smallest of the base periods for each task set. As before, we exclude points where either approach fails to find a bounded response time for all tasks in the set. Figure 4.7 shows that our response times are lower than those of TDMA for almost all of the task sets.

Figure 4.8 compares the number of tasks found schedulable in the above experiment for different ranges of to-

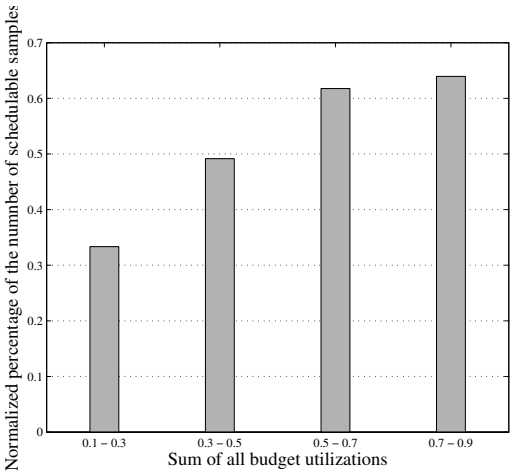


Figure 4.8: Normalized percentage of the number of schedulable samples, *i.e.*, $\frac{\text{\# of schedulable samples by TDMA}}{\text{\# of schedulable samples by Budgeted GRMS}}$, according to the sum of all budget utilizations.

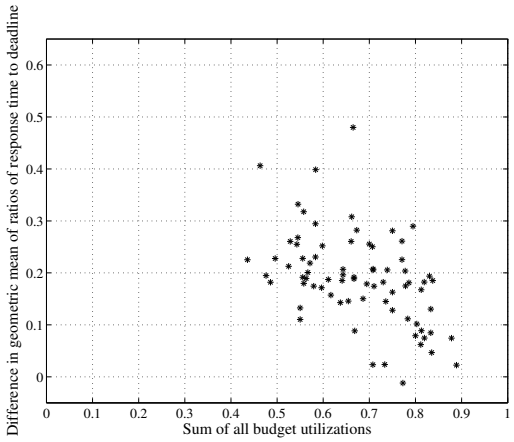


Figure 4.9: TDMA vs. Budgeted GRMS: the same experiment with Fig. 4.7, but task periods are transformed accordingly to be multiples of the major cycle.

tal (sum of budgets) utilization. Each bar represents the ratio of the number of schedulable samples by TDMA to the number of schedulable samples by Budgeted GRMS for the corresponding utilization range. It shows that our approach can find more task sets schedulable, and performs comparatively better as utilization increases. The latter observation is not a contradiction with Figure 4.7, where the two approaches appear to become more comparable at higher utilization. Rather, it is attributed to the fact that, as utilization increases, schedulability inefficiencies cause unbounded response times with a higher likelihood. Hence, the pool of task sets for which *both* approaches succeed at finding a bounded response time narrows down, making them look more similar when constrained to that pool. This does not preclude one approach from finding many more tasks sets schedulable, when the other fails to find bounded response times.

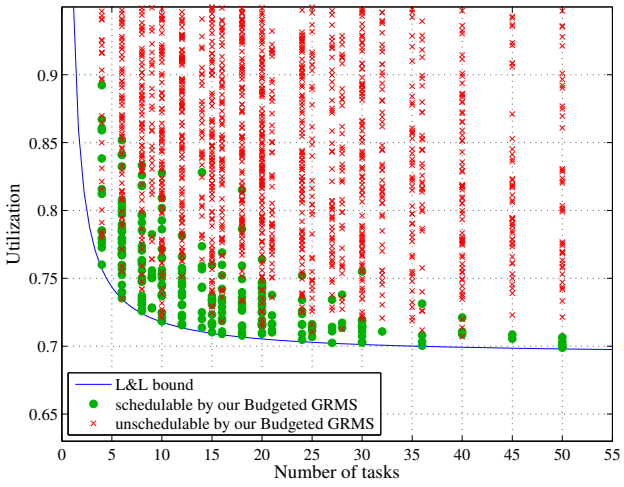


Figure 4.10: Achieved utilizations above L&L's utilization bound by our globally-scheduled approach with known periods, when deadline is equals to period. (Shown in color as well)

Figure 4.9 repeats the comparison of normalized response times of TDMA versus Budgeted GRMS for the same task sets, except that, in this experiment, for the TDMA schedule, tasks periods are decreased to the nearest multiple of the TDMA's major cycle. The change has two conflicting effects. The first is that task schedulability is improved, since all periods become harmonic. The second is that task set utilization increases thanks to the decreased periods. The figure shows that the latter effect dominates. Many task sets fail to fit in the given budget and hence suffer unbounded response times. Only task sets with bounded response times by both policies are shown Figure 4.9.

We note that, our MILP approach took 0.09 sec on average with 1.59 standard deviation to analyze sets of 5-50 tasks. Thus, our approach solves problems of reasonable size in seconds, which is an acceptable amount of time for offline analysis.

4.4.2 Improved Utilization

An alternative to resource partitioning is to use global scheduling at run-time, in which case the sum of task utilizations must be less than the utilization bound for schedulability. Since utilization bounds exist only for the task model where deadlines are equal to periods, we consider this model in this section, and compare the Liu and Layland utilization bound to the actual sum of budgets for task sets deemed schedulable (and those deemed unschedulable) according to our response time analysis. The same task sets were considered as in the previous experiments. Figure 4.10 shows the results.

The figure shows the number of tasks in the set on the x-axis. It shows the total (sum of budgets) utilization on the y-axis. If Budgeted GRMS finds the task set schedulable, a green circle is plotted at the corresponding utilization

and number of tasks. If it finds it unschedulable, a red cross is plotted. For comparison, the expression for the Liu and Layland utilization bound is plotted as well. Note that, all task sets below the bound are trivially schedulable. Hence, we focus on the region above the bound. It can be seen that a strip exists where the bound fails but Budgeted GRMS is able to find the task sets schedulable most of the time. As utilization increases, the green circles gradually give way to red crosses, indicating the schedulability limits of budgeted GRMS were reached. The figure demonstrates that our analysis allows the total schedulable utilization to exceed the bound. This is expected, since the analysis takes the specific task periods into account.

The results confirm that the partitioned, yet globally scheduled scheduling model is both superior to other partitioned approaches in terms of response time, and to globally scheduled approaches in terms of utilization.

Chapter 5

Schedulability Test with Budgeted GRMS and Synchronized I/O on Multicore Systems

In this chapter, the work of the previous chapter, Budgeted GRMS, is incorporated with I/O issues on multicore platforms and presented in this chapter. In the previous chapter, it proposed a response time analysis for schedulability analysis and in this chapter an analyzing method for schedulability bound is provided. Hence, this chapter presents a schedulability test for safety-critical software undergoing a transition from single-core to multicore systems - a challenge faced by multiple industries today. Our migration model, consisting of a schedulability test and execution model, is distinguished by three aspects consistent with reducing transition cost. First, it assumes externally-driven scheduling parameters, such as periods and deadlines, remain fixed (and thus known), whereas exact computation times are not. Second, it adopts a globally synchronized conflict-free I/O model that leads to a decoupling between cores, simplifying the schedulability analysis. Third, it employs global priority assignment across all tasks on each core, irrespective of application, where budget constraints on each application ensure isolation. These properties enable us to obtain a utilization bound that places an allowable limit on total task execution times. Evaluation results demonstrate the advantages of our scheduling model over competing resource partitioning approaches, such as Periodic Server and TDMA. The work of this chapter is presented in [Kim et al., 2017].

5.1 Introduction

This chapter presents a schedulability test to support migration of safety-critical software from single-core to multicore systems. The work is motivated by the advent of multicore processors over the last decade, with increasing potential for efficiency in performance, power and size. This trend has made new single-core processors relatively scarce and as a result, has created a pressing need to transition to multicore processors. Existing previously-certified software, especially for safety-critical applications such as avionics systems, underwent rigorous certification processes based on an underlying assumption of running on a single-core processor. Providers of these certified applications wish to avoid changes that would lead to costly recertification requirements when transitioning to multicore processors.

Our work provides a significant step toward supporting multicore solutions for safety-critical applications. It does this by building on three separate analysis methods that previously had not been applied together to multicore systems.

These are:

- Utilization bound analysis using task period information,
- Conflict-free I/O scheduling, and
- Global priority assignment across all tasks on a core, irrespective of application (defined by a group of tasks), while enforcing application budgets

Our schedulability analysis can be viewed as an extension to the classical Liu and Layland (L&L) schedulability bound [Liu and Layland, 1973]. When known values of task periods are used in the analysis, the bound becomes even better (*i.e.*, less restrictive), often significantly so. This is because the L&L analysis makes worst-case assumptions about task periods; actual periods are unlikely to resemble the worst case (for example, the ratio of two task periods will often be a whole number, as opposed to the square root of 2, as derived by L&L).

Conflict-free I/O scheduling treats I/O transactions as non-preemptive and globally synchronizes them in a conflict-free schedule. In the analysis, I/O transactions are regarded as having the highest priority, since this is the most pessimistic assumption for other tasks' schedulability. This eliminates cross-core interference due to I/O and leads to a decoupling between cores, simplifying the schedulability analysis.

In addition, the model assigns CPU utilization budgets to each application (*i.e.*, a group of tasks), yet it schedules tasks globally across applications sharing a core. Evaluation in a single-core model showed that this architecture significantly improves schedulability over TDMA and Periodic Server, while maintaining isolation properties. We build on this model [Kim et al., 2015a], providing an overview in Sec. 5.2.1.

Our utilization bound and global priority assignment with enforced application budgets are *complementary*; the former is useful early in the development process (indeed, even before coding begins) or during migration, whereas the latter is applicable when development is complete and all tasks' Worst Case Execution Times (WCET)s can be identified accurately. During development, and before the code is instrumented completely enough to determine WCETs with interference effects, developers can still execute the code under approximately worst-case conditions and measure processor idle time; this allows a quick and easy estimation of application utilization for comparison with the utilization bound.

5.2 Software Migration to Multicore Systems

We propose a task execution model and a corresponding schedulability analysis test, motivated by the need to transition safety-critical software certified on single-core systems to multicore systems. Toward that end, we make three important assumptions motivated by likely transition realities and design choices: (i) task periods, deadlines, and I/O

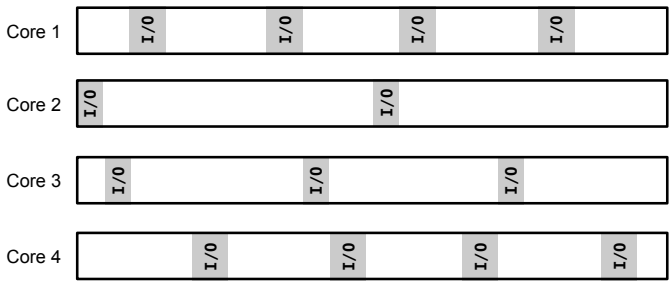


Figure 5.1: Conflict-free I/O section schedule over multiple cores. I/O sections are non-preemptive and strictly periodic.

durations are known since they are tied to system specifications or derived from physical constraints and data size, but our schedulability analysis assumes exact execution times are not yet known, (ii) all I/O transactions are globally scheduled in a conflict-free manner, and (iii) global priority assignment with application budgets enforced is employed on individual cores. Our model attempts to remove all timing dependencies across applications to support portability of applications. We provide a solution to the schedulability problem given the above model.

5.2.1 Task Execution Model

Schedulability Analysis with Task Period Data: we assume that an allocation of application software to cores has already taken place: we focus on scheduling instead. We are given M cores. In each core, m, we consider scheduling a set, $S_{(m)}$, of periodic tasks, where each task, $\tau_{m,i} \in S_{(m)}$, is described by a known period, $T_{m,i}$, a known relative deadline, $D_{m,i}$, and a known I/O duration, $IO_{m,i}$, but the worst case computation time of the task, denoted by $C_{m,i}$, may *not* be known. Once development is complete, the various factors and details that affect WCETs, including timing interference and delay due to shared resources (*e.g.*, bus, cache, memory), are assumed to be abstracted (by techniques such as [Rosén et al., 2007, Paolieri et al., 2009a, Yoon et al., 2011, Yoon, 2011, Ward et al., 2013, Chisholm et al., 2015]) and incorporated in the final WCETs. However, the utilization bound in our analysis framework allows for WCETs that are not yet known and still obtains a bound on allowable application utilization. We assume that tasks are indexed such that a lower number implies a higher priority in a core.

Conflict-free I/O: A key requirement for achieving isolation among cores is to ensure non-interference due to I/O. Hence, I/O transactions are scheduled such that they are conflict-free. As a result, all I/O activity occurs strictly periodically and non-preemptively, which makes the implementation and analysis easier [Mok, 1983]. I/O scheduling thus reduces to choosing phasing for the I/O transactions. I/O sizes tend to be relatively short, hence their strictly periodic scheduling does not seriously degrade system schedulability - we show this property in the evaluation. I/O transactions are modeled as periodic tasks with period $T_{m,i}$ and execution time $IO_{m,i}$. To ensure isolation and due to their relatively small size, I/O transactions are analyzed as having the *highest priority*, and being globally scheduled

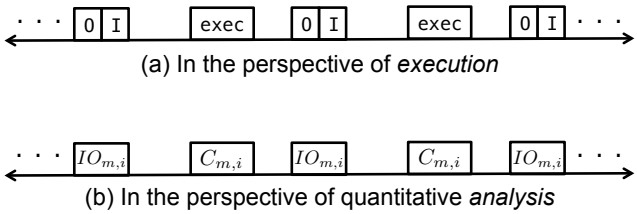


Figure 5.2: Top: task execution model with I/O sections, bottom: quantitatively lumping input and output time in the analysis.

in a conflict-free manner, such that only a single section executes at a time *across all cores* as shown in Figure 5.1. Hence no I/O on any core will ever be blocked, preempted, or otherwise delayed by I/O from another core.

In our model, an I/O transaction must first occur to acquire input, the processing component of a task then runs, followed by I/O to deliver the output. The I/O transactions are supposed to occur strictly periodically at a pre-designated instant, even though *raw* I/Os from external sources are asynchronous. We assume that output and input occur at period boundaries back-to-back, thus combining the output and input into a contiguous interval of I/O processing. In this chapter, we use the term *I/O section* to refer to such an object, having total duration, $IO_{m,i}$, for task $\tau_{m,i}$, as shown in Fig. 5.2. The I/O section's duration is relatively easy to bound since it depends mainly on data needs of control loops and so can be known. This duration can of course be affected by interference on shared resources such as bus and cache. We assume such interferences can be bounded by techniques such as [Rosén et al., 2007, Paolieri et al., 2009a, Yoon et al., 2011, Yoon, 2011, Ward et al., 2013, Chisholm et al., 2015] and incorporated into the I/O duration estimation.[1]

Processing tasks and I/O transactions constitute separately schedulable entities. The processing component runs at a known fixed priority value, $Pr_{m,i}$, whereas (as we explain later) I/O is regarded as *top priority*. Summing up I/O and execution time, we define the task's total utilization, $u_{m,i} = (C_{m,i} + IO_{m,i})/T_{m,i}$. The total number of tasks allocated to core m is $|S_{(m)}| = N_{(m)}$. Each task belongs to an application α_z, $(z = 1, \cdots, Z_{(m)})$, where $\alpha(\tau_{m,i})$ denotes the application to which task $\tau_{m,i}$ belongs to. Table 5.1 summarizes the notation used in this chapter.

Global priority assignment, yet enforcing application budgets: Each application (*i.e.*, a group of tasks) is assigned to one core. Note that, in principle, an application might be allocated to span several cores. However, we do not expect this to be the common case when migrating safety-critical software certified on single-core systems to multicore platforms. This is because individual applications comprising the original single-core system must have been certified to run on a single-core. While the allocation of applications might change upon transition, we expect that in order to minimize re-certification cost, it makes sense that tasks belonging to the same application should be assigned together

[1] Because of interference at run time, the start of an I/O section could be delayed slightly. We assume such delays (along with other context-switching delays) are captured in the assumed duration of an I/O section. Note that such interference could come only from non-I/O tasks; as explained herein, I/O sections cannot interfere with each other.

Table 5.1: Notation

Symbol	Description	
$\tau_{m,i}$	task i in core m	
$T_{m,i}$	period of $\tau_{m,i}$	
$C_{m,i}$	computation time of $\tau_{m,i}$	not given
$Pr_{m,i}$	priority of $\tau_{m,i}$	
$IO_{m,i}$	duration of I/O transaction of $\tau_{m,i}$	
$\psi_{m,i}$	offset of I/O transaction of $\tau_{m,i}$	
$u_{m,i}$	utilization of $\tau_{m,i}$	
U_m	utilization of core m	
α_z	application z	
$\alpha(\tau_{m,i})$	application to which $\tau_{m,i}$ belongs	
B_z	CPU utilization budget assigned for α_z	
$\mathcal{A}^n_{(m)}$	a set of applications to which $\tau_{m,n}$'s higher priority tasks belongs but excluding to which $\tau_{m,n}$ belongs	
M	core count	
$N_{(m)}$	task count in core m	

on the same core [2].

We further assume that Core m's utilization, U_m, is given by $U_m = \sum_{i=1}^{N_{(m)}} u_{m,i}$. At design time, each application α_z is assigned a budget B_z, defined as the maximum CPU utilization allowable for the sum of its tasks. Hence, for each α_z, when development is complete, the code should satisfy

$$\sum_{\forall \tau_{m,i} s.t. \alpha(\tau_{m,i})=\alpha_z} u_{m,i} \leq B_z. \tag{5.1}$$

Observe that the budget, B_z, of application α_z is *a design-time constraint*, not a run-time resource partitioning mechanism. Compliance with application budgets is checked repeatedly throughout the software development process. In cases of noncompliance, either software must be refactored, or else new schedules must be computed. For fielded software, WCET bounds for individual tasks will be enforced, thereby indirectly enforcing application budget compliance. Such WCET-enforced tasks will be scheduled using regular fixed priority scheduling. Hence, this mechanism indirectly allows enforcement of resource budgets, without employing resource partitioning mechanisms at run-time. [Kim et al., 2015a] The mechanism avoids inefficiencies of resource-partitioned systems, such as the Periodic Server and TDMA, arising due to priority inversion when a high-priority task in one partition must wait because the CPU is presently allocated to another partition (where a lower-priority task might be executing). See Figure 5.3. [3] Tasks' schedulability is analyzed in a fixed-priority fashion no matter which application they belong to.

[2] We would *always* expect system developers to avoid - if at all possible - breaking an application across cores. To do otherwise would invite additional complications without any additional benefit. Indeed, some processors have no shared cache between cores, so two threads of the same application running on different cores lose the advantage of caching, resulting in a significant performance loss. Meanwhile, much additional analysis would be required to manage the timing of thread execution and of resource availability on separate cores. Breaking large applications may become unavoidable for some future migrations, but it is outside the scope of this work.

[3] In this experiment, I/O sections are considered. The detailed information can be found in Sec. 5.6.

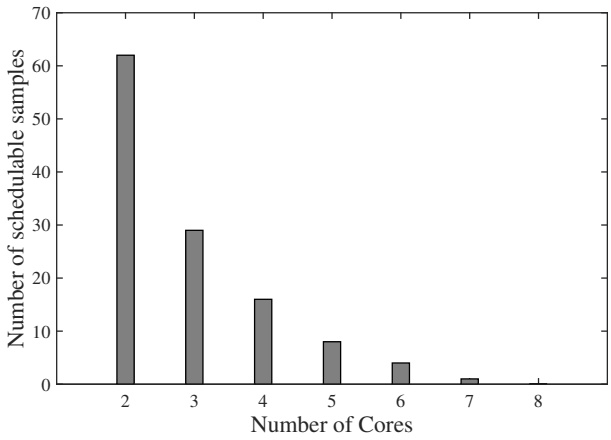


Figure 5.3: Low scalability of TDMA: the number of instances schedulable by TDMA according to core count.

5.2.2 An Equivalent Independent Task Model

Before developing our schedulability test, we note that the task model above can be transformed to one of scheduling independent tasks on each core. By assumption, we require I/O to be non-preemptible, and we require the following precedence constraints involving execution and I/O tasks to be satisfied for every invocation of every task: (i) the processing component does not begin until after the sub-task of acquiring input is complete, (ii) the sub-task of delivering the output does not begin until after the processing component is completed, and (iii) the sub-task of acquiring input for the next period's invocation of the task does not begin until after the sub-task of delivering the output from the current period's invocation is completed. (Note that the very first invocation of the task in the global schedule does not require a predecessor.) The following theorem shows that using the concept of I/O sections allows these precedence constraints to be satisfied automatically.

Theorem 6. *If a feasible schedule exists with I/O sections scheduled strictly periodically and conflict-free, then there exists a feasible schedule in which the precedence constraints in our task execution model are satisfied.*

Proof. Consider an arbitrary task $\tau_{m,i}$. In a feasible schedule with I/O sections scheduled periodically and conflict-free, each invocation of $\tau_{m,i}$ gets a total I/O processing time of $I_{m,i}+O_{m,i}$ within each period. In addition, $\tau_{m,i}$ gets an allocation of at least $C_{m,i}$ units of processor time in each period. If I/O sections consist of an output sub-task followed by an input sub-task, and if each invocation of the processing task follows after the I/O section, then the precedence constraint (i) above is satisfied. Since the processing task gets at least $C_{m,i}$ units of processor time in each period, each invocation of the processing component can complete before the next I/O section begins; hence precedence constraint (ii) is satisfied. Finally precedence constraint (iii) is satisfied by the construction of I/O sections. □

Accordingly, we are eliminating cross-core interference due to I/O and obtaining the utilization bound to place an allowable limit on total task execution times including other interference effects. As a result, our schedulability

problem is distilled into two subproblems:

- Ensure that I/O sections are scheduled strictly periodically and conflict-free.
- Analyze task schedulability on each core separately.

5.3 Schedulability Analysis with Budget Constraints

Per the discussion above, in this section, we analyze the schedulability of tasks on each core. We do so by analyzing schedulability of one task at a time, considering its application budget constraint.

5.3.1 Overview of Approach

A valid utilization bound for an *individual task*, say $\tau_{m,n}$ on core m, denoted by $U^n_{m,bound}$ means that it is schedulable whenever the overall utilization of the task set on core m satisfies $U_m \leq U^n_{m,bound}$. Since periods, relative deadlines, priorities, and I/O sections are known, the bound is computed by minimizing the utilization of a *critically schedulable*[4] task set on core m, $\sum_{i=1}^{n} u_{m,i}$, over all possible values of computation times $C_{m,i}$ for $1 \leq i \leq n$.

Consider the critical time zone[5] of task $\tau_{m,n}$, of application α_z, where the task arrives at time t=0 together with its all higher priority tasks. Suppose that the invocations in this time interval are a critically schedulable task set. Since scheduling is work-conserving, it follows that the time interval $0 \leq t < D_{m,n}$ is continuously busy. At the same time, budget constraints limit the utilization of all the tasks in α_z up to B_z. However, these two constraints conflict with each other since budgets could be too small to make the critical interval continuously busy with no gaps, and thus could make $U^n_{m,bound}$ not obtainable.

To tackle this issue, we release (*i.e.*, remove) the budget constraint for the application $\alpha(\tau_{m,n})$. Let the resultant bound be $U^n_{m,released}$. Note that removing a constraint in a minimization problem cannot lead to a higher-value solution, because the optimal solution to the problem *before* the removal remains feasible for the problem *after* the removal. Therefore,

$$U^n_{m,released} \leq U^n_{m,bound}. \tag{5.2}$$

Define set $\mathcal{A}^n_{(m)}$ as the set of applications, excluding $\alpha(\tau_{m,n})$, on core m containing higher priority tasks than $\tau_{m,n}$. Then, budget constraints for the applications are as follows,

[4] A task set is critically schedulable if any increase in execution time of any task would make the set unschedulable [Liu and Layland, 1973].

[5] A critical time zone is defined as the time interval between a critical instant and the end of the response to the corresponding request of the task [Liu and Layland, 1973].

$$\forall \alpha_z \in \mathcal{A}^n_{(m)}, \alpha(\tau_{m,i}) \neq \alpha_z, \sum_{i=1}^{n-1} u_{m,i} \leq B_z.$$

Let us denote $B^n_{m,total}$ as the budget sum of the applications to which $\tau_{m,n}$ and its higher priority tasks belong, *i.e.*,

$$B^n_{m,total} = \sum_{1 \leq z \leq Z_{(m)}} B_z, \ \forall \alpha_z \in (\mathcal{A}^n_{(m)} \cup \{\alpha(\tau_{m,n})\}).$$

Then, if $B^n_{m,total}$, is less than or equal to $U^n_{m,released}$, $\tau_{m,n}$ is determined to be schedulable by the following theorem.

Theorem 7. *If $\tau_{m,n}$ is compliant with its budget and $B^n_{m,total}$ is less than or equal to $U^n_{m,released}$, $\tau_{m,n}$ is schedulable.*

Proof. If $B^n_{m,total} \leq U^n_{m,released}$, by (5.2)

$$B^n_{m,total} \leq U^n_{m,released} \leq U^n_{m,bound},$$

Then $B^n_{m,total} \leq U^n_{m,bound}$. It means that $\tau_{m,n}$ is schedulable by the definition of utilization bound for schedulability. □

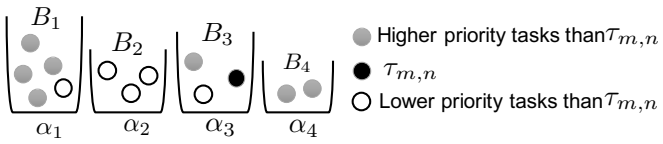
This test is applied to one task at a time, and a core is determined to be schedulable if all tasks on the core are schedulable. The procedure is illustrated in Fig. 5.4 and an illustrative example is presented in Sec. 5.5. In the next section, we show how to compute $U^n_{m,released}$.

5.3.2 The Utilization Bound

It remains to compute the utilization bound, $U^n_{m,released}$, which is the lowest possible utilization when task $\tau_{m,n}$ and its higher priority tasks are critically schedulable. It is computed over all possible values of the executions times of the higher-priority tasks on the same core. This is formulated as a linear programming (LP) problem and solved by a standard LP solver.

[Constraint 1] – Critically schedulable:

For task $\tau_{m,n}$ and its higher priority tasks to be *critically schedulable*, computation times need to *fully* utilize the available processor time within the critical time zone from the critical instant to the deadline. Hence, as all higher priority tasks and $\tau_{m,n}$ release at time 0, collectively their maximum possible amount of computation times *must add up to* $D_{m,n}$:



(0. Test if there exists a feasible I/O schedule using the MILP formulation in Sec. 5.4.)
1. Calculate $U^n_{m,released}$ with α_1 and α_4.
2. Calculate $B^n_{m.total}$ which is $B_1 + B_3 + B_4$.
3. If $B^n_{m,total} \leq U^n_{m,released}$, $\tau_{m,n}$ is schedulable.

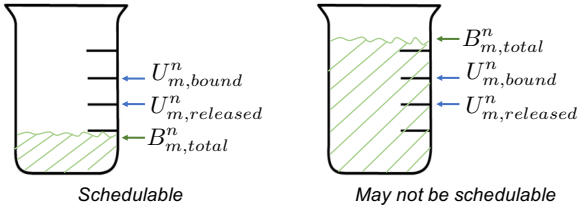


Figure 5.4: The overview of schedulability analysis with budget constraints, and the relationship between $U^n_{m,released}$, $U^n_{m,bound}$ and $B^n_{m,total}$.

$$C_{m,n} + \sum_{i=1}^{n-1} \lceil \frac{D_{m,n}}{T_{m,i}} \rceil C_{m,i} + \sum_{i=1}^{n} \lceil \frac{D_{m,n}}{T_{m,i}} \rceil IO_{m,i} = D_{m,n}.$$

[Constraint 2] – Fully utilized:

Even though Constraint 1 is satisfied, there could be empty gaps prior to $D_{m,n}$, which violates the assumption of fully (*i.e.*, continuously) utilizing the processor time. To prevent such a situation we need an additional constraint which checks, at every arrival ($l \cdot T_k$) of a task, if the cumulative demand up to time $l \cdot T_k$ is greater than or equal to $l \cdot T_k$ [Lehoczky et al., 1989, Park et al., 1995]:

$$\forall 1 \leq k \leq n, \forall 1 \leq l \leq \lfloor \frac{D_{m,n}}{T_{m,k}} \rfloor,$$
$$C_{m,n} + \sum_{i=1}^{n-1} \lceil \frac{l \cdot T_{m,k}}{T_{m,i}} \rceil C_{m,i} + \sum_{i=1}^{n} \lceil \frac{l \cdot T_{m,k}}{T_{m,i}} \rceil IO_{m,i} \geq l \cdot T_{m,k}. \tag{5.3}$$

For testing, $\tau_{m,n}$, $\sum_{k=1}^{n} \lfloor \frac{D_{m,n}}{T_{m,k}} \rfloor$ constraints are generated. This number can be reduced by using $\mathcal{P}_i(t)$ presented in [Bini and Buttazzo, 2004], which is defined as follows (see (6) in [Bini and Buttazzo, 2004]):

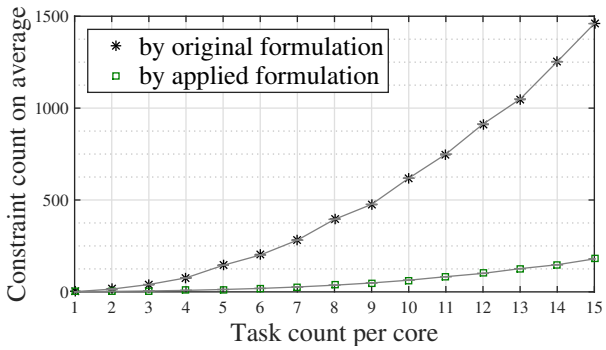


Figure 5.5: Comparison of the number of constraints generated by original formulation of Constraint 2 vs. redundancy removed from (5.4). The result is shown for the data from Fig. 5.7 in the evaluation section.

$$\mathcal{P}_0(t) = \{t\},\ \ \mathcal{P}_i(t) = \mathcal{P}_{i-1}(\lfloor \frac{t}{T_i} \rfloor T_i) \cup \mathcal{P}_{i-1}(t).$$

Accordingly, not all the arrivals at $l \cdot T_k$ ($\forall 1 \leq k \leq n, \forall 1 \leq l \leq \lfloor \frac{D_{m,n}}{T_{m,k}} \rfloor$) but only the subset of them, $t \in \mathcal{P}_{n-1}(D_{m,n})$, are considered as follows:

$$\begin{aligned} & t \in \mathcal{P}_{n-1}(D_{m,n}), \\ & C_{m,n} + \sum_{i=1}^{n-1} \lceil \frac{t}{T_{m,i}} \rceil C_{m,i} + \sum_{i=1}^{n} \lceil \frac{t}{T_{m,i}} \rceil IO_{m,i} \geq t. \end{aligned} \tag{5.4}$$

In the worst case, the number of constraints can be 2^n [Bini and Buttazzo, 2004]. However, set $\mathcal{P}_i(t)$ generates redundantly identical values. Hence we remove the redundancy and thus have fewer constraints. Fig. 5.5 shows the average number of constraints generated by the original formulation (5.3) of Constraint 2 and by (5.4) (but after removing redundant values).

Finally, we formulate the LP problem of finding $U^n_{m,released}$ for a given task $\tau_{m,n}$ as follows:

[Find $\mathbf{U^n_{m,released}}$] Minimize $\sum_{i=1}^{n} u_{m,i}$

Subject to:

- $\sum_{i=1}^{n-1} u_{m,i} \leq B_z, \qquad \forall \alpha_z \in \mathcal{A}^n_{(m)}, \alpha(\tau_{m,i}) \neq \alpha_z.$
- $C_{m,n} + \sum_{i=1}^{n-1} \lceil \frac{D_{m,n}}{T_{m,i}} \rceil C_{m,i} + \sum_{i=1}^{n} \lceil \frac{D_{m,n}}{T_{m,i}} \rceil IO_{m,i} = D_{m,n}.$
- $C_{m,n} + \sum_{i=1}^{n-1} \lceil \frac{t}{T_{m,i}} \rceil C_{m,i} + \sum_{i=1}^{n} \lceil \frac{t}{T_{m,i}} \rceil IO_{m,i} \geq t,$

  where $t \in \mathcal{P}_{n-1}(D_{m,n})$.

The return value of the problem above is $U^n_{m,released}$ which minimizes the total utilization of all higher priority tasks and $\tau_{m,n}$. If $B^n_{m,total} \leq U^n_{m,released}$, then $\tau_{m,n}$ is determined to be schedulable by Theorem 7.

5.4 Conflict-Free I/O

Since a bus is shared by multiple cores commonly on multicore processors, unpredictable interference among I/O sections could occur if conflicted. Hence we schedule I/O sections in a way that only one I/O section must be executed at a time *across all cores*. The I/O sections are non-preemptive and strictly periodic as can be seen in [Rushby, 1999, Krodel, 2004, Parkinson and Kinnan, 2007, Kim et al., 2013, Kim et al., 2014]. In [Korst et al., 1996], a necessary and sufficient condition that any two non-preemptive and strictly periodic intervals do not overlap each other was presented. We apply this condition to our problem for any two I/O sections of $\tau_{p,i}$ and $\tau_{q,j}$ on *any* core p and q as follows (core p and q may or may not be same):

$$\begin{aligned} IO_{p,i} &\leq (\psi_{q,j} - \psi_{p,i}) \mod \texttt{gcd}(T_{p,i}, T_{q,j}) \\ &\leq \texttt{gcd}(T_{p,i}, T_{q,j}) - IO_{q,j} \end{aligned}$$

where $\psi_{*,x}$ denotes the initial offset of $IO_{*,x}$ (appearing at every $\psi_{*,x} + kT_{*,x}$, $k = 0, 1, \ldots$) and gcd is the greatest common divisor function. The current form of the inequality above is not linear due to the modulo operation. Hence, we reformulate it as the following mixed integer linear programming problem:

$$IO_{p,i} \leq (\psi_{q,j} - \psi_{p,i}) - \texttt{gcd}(T_{p,i}, T_{q,j}) \cdot K \tag{5.5}$$

$$(\psi_{q,j} - \psi_{p,i}) - \texttt{gcd}(T_{p,i}, T_{q,j}) \cdot K \leq \texttt{gcd}(T_{p,i}, T_{q,j}) - IO_{q,j} \tag{5.6}$$

where K is a new real-valued variable bounded in

$$\left\lfloor \frac{1 - T_{p,i}}{\texttt{gcd}(T_{p,i}, T_{q,j})} \right\rfloor - 1 \leq K \leq \left\lfloor \frac{T_{q,j} - 1}{\texttt{gcd}(T_{p,i}, T_{q,j})} \right\rfloor + 1.$$

If a feasible conflict-free I/O schedule exists, we individually obtain $U^n_{m,released}$ to test schedulability of each task on each core, as explained in the previous section (Sec. 5.3).

5.5 An Illustrative Example

Table 5.2: Task set for an illustrative example

		budget		period	deadline	I/O
core 1	application 1	0.5	$\tau_{1,2}$	12	12	2
			$\tau_{1,3}$	16	16	1
	application 2	0.25	$\tau_{1,1}$	8	8	1
core 2	application 1	0.9	$\tau_{2,1}$	24	21	1

To illustrate how to apply the approach explained in the previous sections, let us consider an example which has 4 tasks in three applications running on two cores, as shown in Table 5.2.

We first check if a conflict-free I/O schedule exists. For this, we solve the mixed integer linear program in (5.5) and (5.6) for every pair among $IO_{1,1}$, $IO_{1,2}$, $IO_{1,3}$ and $IO_{2,1}$. Then, one of possible solutions is: $\psi_{1,1} = 1$, $\psi_{1,2} = 2$, $\psi_{1,3} = 5$, and $\psi_{2,1} = 16$. Hence the existence of a global I/O schedule is checked. Below we shall compute $U^3_{1,released}$ which is the theoretical bound on utilization for checking the schedulability of task $\tau_{1,3}$ on core 1.

The objective function of the optimization problem (presented at the end of Sec. 5.3) is

$$\sum_{i=1}^{3} \frac{C_{1,i}}{T_{1,i}} + \frac{IO_{1,i}}{T_{1,i}} = \frac{C_{1,1}}{8} + \frac{C_{1,2}}{12} + \frac{C_{1,3}}{16} + \frac{1}{8} + \frac{2}{12} + \frac{1}{16}.$$

The budget constraint is:

$$\frac{C_{1,1} + IO_{1,1}}{T_{1,1}} = \frac{C_{1,1} + 1}{8} \leq 0.25 \Rightarrow C_{1,1} \leq 1$$

Note that $\tau_{1,1}$ is the only higher-priority task than $\tau_{1,3}$ in other applications on the same core.

For $\tau_{1,3}$ to be critically schedulable:

$$C_{1,3} + \lceil \frac{16}{8} \rceil C_{1,1} + \lceil \frac{16}{12} \rceil C_{1,2} + \lceil \frac{16}{8} \rceil \cdot 1 + \lceil \frac{16}{12} \rceil \cdot 2 + \lceil \frac{16}{16} \rceil \cdot 1 = 16$$

$$C_{1,3} + 2C_{1,1} + 2C_{1,2} = 9$$

For $\tau_{1,3}$'s critical zone to be fully utilized:

$$C_{1,3} + \lceil \frac{8}{8} \rceil C_{1,1} + \lceil \frac{8}{12} \rceil C_{1,2} + \lceil \frac{8}{8} \rceil \cdot 1 + \lceil \frac{8}{12} \rceil \cdot 2 + \lceil \frac{8}{16} \rceil \cdot 1 \geq 8,$$

$$C_{1,3} + \lceil \frac{12}{8} \rceil C_{1,1} + \lceil \frac{12}{12} \rceil C_{1,2} + \lceil \frac{12}{8} \rceil \cdot 1 + \lceil \frac{12}{12} \rceil \cdot 2 + \lceil \frac{12}{16} \rceil \cdot 1 \geq 12.$$

Then,

$$C_{1,3} + C_{1,1} + C_{1,2} \geq 4,$$

and

$$C_{1,3} + 2C_{1,1} + C_{1,2} \geq 7.$$

Finally, solving the linear program above results in the computation times of $C_{1,1} = 0$; $C_{1,2} = 2$; $C_{1,3} = 5$. It should be noted that a solution to this optimization problem does not necessarily correspond to a realistic task set; instead, it gives us a limiting case that defines a sufficient condition for schedulable utilization. The corresponding utilization bound is

$$U^3_{1,released} = \frac{C_{1,1}}{8} + \frac{C_{1,2}}{12} + \frac{C_{1,3}}{16} + \frac{1}{8} + \frac{2}{12} + \frac{1}{16} \simeq 83.333\%.$$

Comparing this with the allowed total budget on core 1, we have $75\% \leq 83.333\%$. This means $\tau_{1,3}$ is schedulable as long as its execution time when development is complete (and also estimated bounds on any inter-core interference are taken into account) is compliant with its application budget. For instance, suppose $C_{1,2}$ and $C_{1,3}$ are finally bounded to 1 and 3, respectively. This is good because the utilization is under the budget. On the other hand, if $C_{1,2} = 2$ and $C_{1,3} = 2$, they are not compliant with the budget and thus the task set might not be schedulable.

The bounds $U^1_{1,released}$ and $U^2_{1,released}$ are computed similarly. Trivially, $U^1_{1,released}$ = 100%, and thus $\tau_{1,1}$ is schedulable. For $U^2_{1,released}$, we can obtain $C_{1,1} = 1$ and $C_{1,2} = 6$. The resulting bound $U^2_{1,released}$ is

$$\frac{1+1}{8} + \frac{2+6}{12} \simeq 91.667\%.$$

Since $B^2_{1,total} = 75\%$, $\tau_{1,2}$ is schedulable. As a result, since all three tasks are schedulable, core 1 is schedulable. In addition, since $\tau_{2,1}$ is schedulable as its utilization bound is 87.5% while $B^1_{2,total} = 0.9$, core 2 might not be schedulable.

In this example, the worst-case scenario ended up with $C_{1,1} = 0$. As mentioned earlier, such solutions are derived to obtain the extreme utilization 'bound' with the worst-case combination; no lower bound can be found by other combinations. In other words, if there is any increase on the resulting task execution times, the utilization bound would not be lower than the current utilization. For example, if we increase $C_{1,1}$ to 1 in the above example, and solve

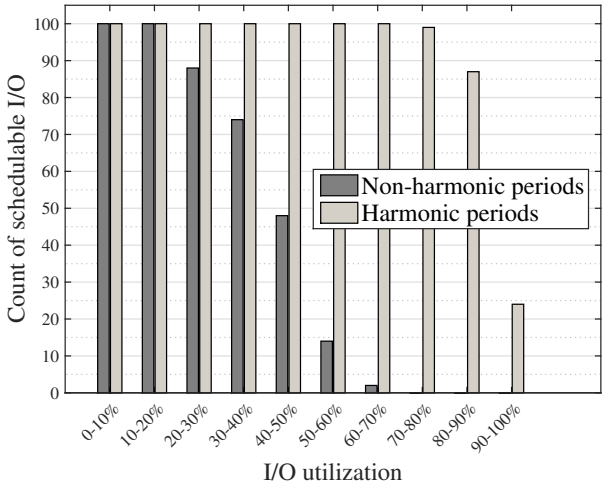


Figure 5.6: Number of task set instances that have schedulable I/O when task periods are non-harmonic vs. harmonic.

for the other execution times, we could get $C_{1,2} = 1$ and $C_{1,3} = 5$. The resulting utilization is then

$$\frac{C_{1,1}}{8} + \frac{C_{1,2}}{12} + \frac{C_{1,3}}{16} + \frac{1}{8} + \frac{2}{12} + \frac{1}{16} = 87.5\%$$

which is higher than what is computed before (83.333%). Hence, it is not the lowest-utilization critically-schedulable scenario. In short, we only aim to obtain the lowest 'bound' to compare for schedulability.

5.6 Evaluation

We first evaluate the impact of I/O on schedulability then compare our new schedulability test for individual cores to the Liu and Layland bound, showing the impact of using information on periods and deadlines. Lastly, we apply these results in comparing the count of schedulable instances using our approach, Periodic Server and TDMA. The results show that our approach can schedule more instances since it can avoid any unnecessary priority inversion. We evaluate with synthetic data since actual avionics systems have yet to be certified when *multiple cores* are running. However, we model tasks similar to actual ones - tasks are periodic, deadline-constrained, and contain I/O sections. All experiments ran on an Intel Core i7-2600 CPU 3.40 GHz with 16GB RAM, using IBM ILOG Cplex Optimizer for the linear programs, and solved essentially instantaneously.

5.6.1 I/O Schedulability

We compare the cases of non- and harmonic periods:

- NON-HARMONIC PERIODS: we generated 1,000 task sets from ten groups having, respectively, total I/O uti-

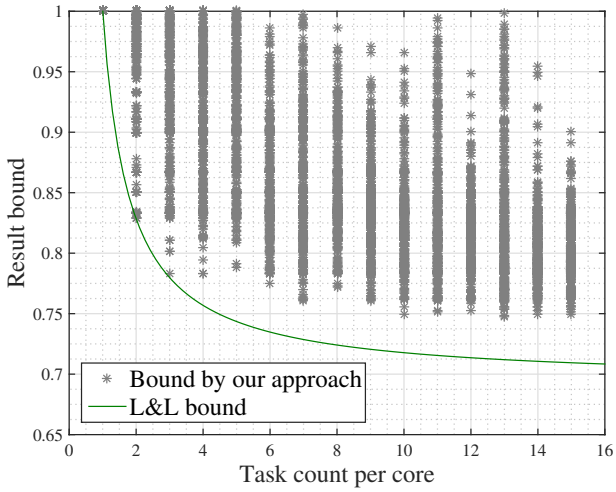


Figure 5.7: Utilization bounds per core with known periods (points) are higher than the Liu and Layland bound (line) (when deadline is equal to period).

lization (across the entire system), 0-10%, $\cdots$, 90-100%, 100 instances per group. For each instance, there are 4 cores, and up to 60 tasks. Each task period is a divisor of 54000 which is $2^4 \cdot 3^3 \cdot 5^3$ and I/O duration is randomly drawn but not longer than half of the `gcd` of the periods.

- HARMONIC PERIODS: instances are generated same as the non-harmonic period case, but task periods are harmonic (each period is either a divisor or a multiple of any other).

Figure 5.6 shows the number of task sets that have a feasible schedule of I/O section for different I/O utilization. As the I/O load increases, the schedulable rate of non-harmonic periods decreases, and there is no single schedulable instance when utilization is over 70%. Harmonic I/Os can achieve much higher utilization; nevertheless, because of their non-preemptivity, the I/O sections cannot always be schedulable even when the utilization is 100%. Beyond the comparison between harmonic and non-harmonic cases, we can see that relatively lower I/O utilization would not impact the entire system schedulability. Even in the non-harmonic case, I/O sections are always schedulable with I/O load of up to about 20%. As we have found that I/O utilization in practice is small (*e.g.*, less than 5%), it is reasonable to conclude our model of non-preemptive I/O sections is not overly restrictive.

5.6.2 Utilization Bound

As an alternative to existing resource partitioning schemes, we used a global scheduling approach that ignores application boundaries and schedules tasks according to their fixed priorities on each core. For per-core bound, we have obtained

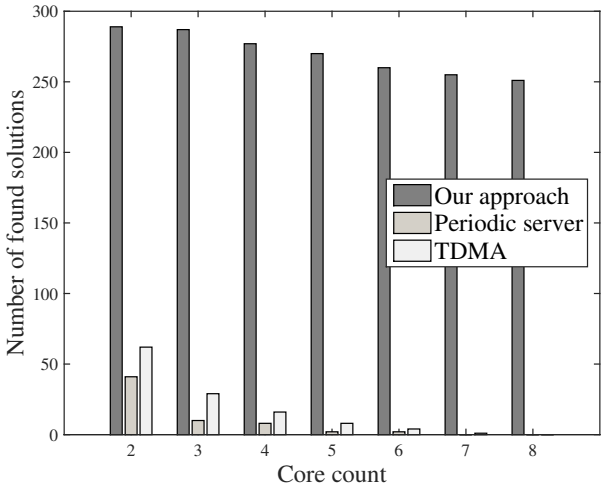


Figure 5.8: The number of instances schedulable by our approach, Periodic Server, and TDMA.

$$\min_{1 \leq n \leq N_{(m)}} U^n_{m,released}$$

on each core m. In Figure 5.7, each point corresponds to per-core bound for each task set. For this experiment, we used the same parameters as the non-harmonic period case in Sec. 5.6.1 above, but kept I/O utilization in 1%-5%. We generated 1,500 task sets with evenly-distributed numbers of tasks per core, *i.e.*, 100 instances per task count. For comparison, the Liu and Layland utilization bound is plotted as well (line plot), when deadlines are equal to periods.

As seen from the result, the bounds calculated by our approach are above the L&L bound. This is because the analysis takes advantage of the information on task periods. In practice, as development proceeds and teams gain more information on task WCETs, they can recalculate the utilization. As long as the eventual task WCETs yield utilization levels that comply with the bounds, the tasks will be schedulable.

5.6.3 Our Approach vs. Other Resource Partitioning Mechanisms

The same parameters were used as in the previous cases except that here we vary core count from 2 to 8 – we generated 300 instances for each core count. Fig. 5.8 shows the number of task set instances that are schedulable by our approach, Periodic Server, and TDMA. In order for an instance to be schedulable, all tasks in all applications (*i.e.*, partitions and servers) on every core must be schedulable. In Periodic Server approach, each application is assigned its own server. The period of the server is the greatest common divisor of the periods of tasks that belong to the server, which is essentially a best-case assumption for Periodic Server. The utilization of the server is equal to the application budget. This allows us to compute server parameters. The analysis in [Davis and Burns, 2005] is used to test the schedulability. This analysis requires execution time information. For fairness, the execution times and all other data

(if applicable) used here by Periodic Server are also applied in our approach and TDMA as well. When TDMA is used, each application is assigned a TDMA partition. The utilization of the partition is equal to the application budget. Then the greatest common divisor of the periods is used as a major cycle for a TDMA schedule. The schedulability of TDMA was analyzed by the method presented in [Sha, 2003].

From Fig. 5.8, we can see that our approach schedules more task sets than the TDMA or Periodic Server can. This is because our approach avoids priority inversions by scheduling tasks irrespective of their assignments to applications. All the three approaches show also that task sets with higher core count are less schedulable. Our result should not be read as saying Periodic Server is inferior to TDMA, because in some instances Periodic Server can successfully schedule them while TDMA does not and vice versa.

Chapter 6

Related Work

In the real-time multicore research, a great deal of effort has been devoted to address the optimization of shared resource allocation and arbitration in multicore architectures. For on-chip memory partitioning, Suhendra *et al.* [Suhendra et al., 2006] proposed an ILP formulation that finds the optimal scratchpad memory partition and task allocation/scheduling which minimize tasks' execution times. In [Suhendra and Mitra, 2008], the authors examined the impacts of different combinations of cache locking and partitioning schemes on the system utilization. In [Bui et al., 2008], Bui *et al.* proposed a genetic algorithm that can find near optimal cache partition and task-to-partition assignments that minimize the system utilization.

Another line of research has focused on shared bus arbitration methods. Rosén *et al.* [Rosén et al., 2007] and Andrei *et al.* [Andrei et al., 2008] addressed TDMA-based bus access policies that is tightly coupled with the worst-case execution paths of tasks. They proposed an optimization problem that finds the optimal TDMA schedule which minimizes the global delay of tasks, and extended it to deal with average-case delays [Rosén et al., 2011]. Additionally, Schranzhofer *et al.* [Schranzhofer et al., 2010] analyzed the worst-case response time of real-time tasks under different cache access models for TDMA-based bus arbitration policies. Although it is not addressed here, the issue of shared memory contention is also receiving increasing attention [Mutlu and Moscibroda, 2007, Paolieri et al., 2009b].

Some work has addressed temporal modularity for resource partitioning. In [Sha, 2003], the author presented a schedulability bound when only information of higher-level partitions is given. The work was based on TDMA and assumes that periods are the same as deadlines. Shin and Lee [Shin and Lee, 2003] proposed the periodic resource model for hierarchical scheduling. They proposed schedulability analysis of tasks mapped to a periodic resource supply (server). The authors presented the exact schedulability analysis under RM and EDF scheduling and derived the corresponding utilization bounds. Davis *et al*. [Davis and Burns, 2005, Davis and Burns, 2008] presented the exact worst-case response time analysis under the periodic server, sporadic server, and deferrable server. Differently from their work, as presented in Chapter 4 and Chapter 5, we assign only CPU utilization for each application (a group of tasks) and then globally schedule tasks (*i.e.*, ignore application-level isolation).

In another line of work, people made effort to find good parameters for a higher-level resource (*e.g.*, server, partition, or budget) in hierarchical scheduling. That is to calculate the minimal amount for the allocated partitioned

resource so the system is schedulable. Almeida *et al*. [Almeida and Pedreiras, 2004] analyzed a periodic server model by introducing the server availability function. They also developed a heuristic algorithm for server (resource) parameter optimization achieving the minimum system utilization. Lipari *et al*. [Lipari and Bini, 2003] also considered the server parameter optimization problem in a hierarchical scheduling system with a different approach of schedulability analysis. Yoon *et al*. [Yoon et al., 2013] considered multiple resources for parameter optimization achieving the minimum system utilization. In that work, the authors solved the problem with Geometric Programming. The addressed approach captures variable interferences among multiple resources in the resource bound. In [Saewong et al., 2002], Saewong *et al*. presented a response time analysis for real-time guarantees for deferrable and sporadic servers. Davis *et al*. [Davis and Burns, 2005, Davis and Burns, 2008] presented the exact worst-case response time analysis under the periodic server, sporadic server, and deferrable server. We compared our model with the periodic server analysis in Chapter 4. The authors in [Davis and Burns, 2005, Davis and Burns, 2008] also presented a greedy algorithm to select multiple server parameters and addressed optimal selection as a holistic problem. This line of research is complementary to ours. A better budget allocation could enhance the schedulability of the system.

In addition to the research mentioned above, the Explicit Deadline Periodic (EDP) resource model was presented in [Easwaran, 2007] by Easwaran. It is a generalized periodic resource model, and the author proposed an exact algorithm for determining the optimal resource parameter that minimizes the ratio of length to the period for an EDP resource. The same problem for the periodic resource model was addressed by Shin *et al*. [Shin and Lee, 2008], in which the authors presented a polynomial-time sufficient algorithm. Both problems were addressed by Dewan *et al*. [Dewan and Fisher, 2010] and Fisher [Fisher, 2009] by proposing fully-polynomial-time approximation algorithms that improve both the optimality and time complexity.

As one practical applications of resource partitioning, specifically TDMA, there have also been studies on IMA system-level optimization. In [Lee et al., 2000b, Lee et al., 2000a], Lee *et al.* considered an IMA system in which multiple processors are connected over Avionics Full-Duplex Switched Ethernet. The authors presented a method to find a cyclic schedule for both IMA partitions and bus channels that guarantees the timing requirements of tasks and messages. Tămaş–Selicean *et al.* [Tămaş-Selicean and Pop, 2011a, Tămaş-Selicean and Pop, 2011b] considered an optimization problem of scheduling of mixed-criticality partitioned resources in a distributed architecture. The authors developed a Tabu search-based algorithm that finds the assignments of tasks and partitions as well as their schedules that satisfy application schedulability while minimizing the design cost. Although not considering IMA architecture, Nemati *et al.* [Nemati et al., 2011] addressed a problem of migrating existing independently-developed systems onto a multicore system in a different aspect. Each existing system is migrated to each core and may use a different scheduling policy. The authors proposed a queuing-based synchronization protocol for resource sharing among such independent systems. They also derived a schedulability test in which sharing of global resources among tasks in each

system can be abstracted. Rufino *et al.* in [Rufino et al., 2010] addressed a uniprocesor-based Temporal and Spatial Partitioning system that includes a partitioning concept that is relevant for IMA. The approach prevents partitions from interfering with each other temporally and spatially for shared resources, such as memory, in an environment where a partition schedule may change according to the operational mode.

There has been work on scheduling optimization for *non-preemptive* and *strictly periodic* tasks on multiprocessor systems. In [Korst et al., 1994, Korst et al., 1996], Korst *et al.* addressed the problem of scheduling strictly periodic and non-preemptive tasks on a multiprocessor with the minimum number of processors. The authors showed that the problem is NP-complete in the strong sense even for a uniprocessor case. Al Sheikh *et al.* [Al Sheikh et al., 2011] addressed a similar problem by proposing a Game Theory based algorithm that not only finds a feasible schedule of tasks but also maximizes the relative distances between them. The authors then extended it to support harmonic and near-harmonic periods in [Al Sheikh et al., 2012] in the language of IMA. Kermia *et al.* [Kermia and Sorel, 2007] presented a greedy heuristic approach to find a non-preemptive multiprocessor schedule of strictly periodic tasks with precedence constraints. Their algorithm also considers communication among tasks and finds the schedule that minimizes the global execution time of communications. All these works aim to find a feasible schedule by assigning tasks to processors and scheduling each one appropriately. Since exclusiveness in resource sharing across processors is not modeled in these works, the tasks on different processors become independent once they are assigned to each processor. However, by limiting to a single processor (a *core*), one can adapt one of these ideas to the initial solution construction phase (Sec. 2.5.2) in HOS algorithm. In the stream, the Pinwheel problem [Holte et al., 1989] is a special class of hard real-time scheduling guaranteeing the occurrence of each symbol within any sequence of a given length of consecutive intervals. In [Han et al., 1996], Han *et al.* proposed a similar task model, a distance-constrained task system, in which the distance between the finishing times of consecutive executions are upper-bounded by a threshold.

In fact, the exclusive I/O execution model is motivated by the concept called a *device management partition* introduced by Rushby in [Rushby, 1999]. The original discussion arose from the need for a synchronization mechanism among IMA partitions communicating over a shared bus in distributed systems. Due to the exclusiveness on the communication line, the necessity of a global schedule that excludes simultaneous executions of device management partitions on different processors is addressed. Although I/O is one of the most serious interference sources on multicore processors, it has not been extensively researched in real-time computing literature, whereas bus, cache and memory related issues have been extensively addressed in [Rosén et al., 2007, Paolieri et al., 2009a, Ward et al., 2013, Chisholm et al., 2015]. Especially as shown in [Mok, 1983], the strictly periodic and non-preemptive model of I/O sections makes implementation and analysis easier. In our work, we tackle the I/O issue by a *conflict-free I/O* model, which is motivated by concepts such as zero partition or device management partition [Rushby, 1999, Krodel, 2004, Parkinson and Kinnan, 2007] that are used in IMA (Integrated Modular Avionics) systems.

In Chapter 4, we proposed a partitioned, yet globally scheduled, enforced uniprocessor task scheduling model. Our work is compared with other existing hierarchical scheduling techniques as we show in its evaluation section. We assume that utilization budgets are given, whereas task execution times are not. The goal is to perform exact worst-case response time analysis. With a similar motivation, in [Sha, 2003], the author presented a scheduling bound when only information of higher-level partitions is given. The work was based on TDMA and assumes that periods are the same as deadlines. Shin *et al*. [Shin and Lee, 2003] proposed the periodic resource model for hierarchical scheduling. They propose schedulability analysis of tasks mapped to a periodic resource supply (server). The authors presented the exact schedulability analysis under RM and EDF scheduling and derived the corresponding utilization bounds. Differently from their work, we calculate the maximum response time for the partitioned, yet globally scheduled scheme.

Perhaps the earliest work in a single-core system for utilization bound to account for unknown execution times was a generalized utilization bound for fixed-priority scheduling by Park *et al.* [Park et al., 1995] that takes into account task periods, deadlines, and arbitrary fixed priorities. The underlying optimization problem was later simplified, leading to a solution that was not tight [Park et al., 1996]. Chen *et al.* proposed other approximations [Chen et al., 2003], together with a tight utilization bound computed in exponential time for a periodic task model with known periods and implicit deadlines, under rate monotonic scheduling. A more recent approach to solve the problem offered another linear programming formulation [Lee et al., 2004]. In [Bini and Buttazzo, 2004], Bini and Buttazzo proposed a method to reduce the number of constraints that should be considered. However, they did not consider the impact of budgets.

Part II

Timing Analysis in Emerging CPS

Chapter 7

Introduction

The Internet of Things heralds a new generation of data-centric applications, where controllers connect to large numbers of heterogeneous sensing devices. This work is motivated by such increased connectivity among physical devices with sensing and actuation capabilities, leading to such visions as smart cities, green buildings, intelligent transportation, and personalized healthcare, among others. Several government, industry, and academic research initiatives emerged in recent years to address the IoT challenge.

At their core, the challenges of IoT largely intersect with cyber-physical systems research. In the US, the National Institute of Standards and Technology (NIST) launched the "Global City Teams Challenge" to develop the groundwork for smarter city technologies of the future.[1] According to NIST, "Smart cities rely on effective networking of computer systems and physical devices. These Internet of Things (IoT) and cyber-physical systems (CPS) currently account for more than $32 *trillion* in global economic activity, a number that is projected to grow". NIST asserts that "Communities ranging from small towns to megacities are looking to the power of emerging Internet of Things technologies to better manage their resources and improve everything from health and safety to education and transportation. They can meet their smart city needs with cyber physical systems (CPS) - interconnected hybrids of engineered and IT systems - if certain engineering, security, and measurement challenges can be addressed". A recent report by IBM,[2] emphasized the role of cyber-physical systems research in realizing future smart city and IoT visions. IBM also announced that it will invest $3 billion over four years to establish a new Internet of Things unit.[3]

In academia, IoT applications are investigated that bring about many new challenges, ranging from safety and security to predictability and real-time guarantees [Weber, 2010, Roman et al., 2011, Hu et al., 2015]. Of particular interest, for the purposes of this work, are challenges that impact real-time scheduling. There are several ways that IoT systems change the landscape of real-time scheduling problems addressed in more traditional applications, such as avionics and process control. Specifically, the following features emerge:

- *Data, not processes:* Traditional real-time scheduling literature mostly emphasizes computational tasks as the main scheduled entities. Such tasks might have other needs such as memory, data, or I/O, but the formulation

[1] http://www.nist.gov/cps/sagc.cfm
[2] http://www.ibm.com/developerworks/library/ba-cyber-physical-systems-and-smart-cities-iot/index.html
[3] http://www.eweek.com/database/ibm-launches-new-iot-software-services-ecosystem.html

often thinks of schedulable units as pieces of *code*, together with their resource requirements. The emerging world of IoT will instead revolve around *data*. Hence, the fundamental schedulable unit will often be chunks of *data*, moving over networks, together with their resource requirements (including processing and communication bandwidth).

- *Multiple choice problems:* Scheduling problems in current literature usually have a notion of a fixed task set whose schedulability is analyzed. In some cases, tasks arrive dynamically and schedulability decisions are made incrementally one task at a time. In contrast, with so many devices connected together to a common medium, in IoT applications choices will arise in the application workflow. For example, multiple providers may offer similar data, data objects may be available at different levels of quality, and application goals may be met by exploiting one of multiple alternative data sets. Problem formulations will thus adopt models where achieving a goal (such as computing some outputs by respective deadlines) will entail multiple choice regarding the requisite input set of data processing tasks used or their quality. Hence, we might see novel parallel task models with an "OR" structure (instead of an "AND") between subsets of branches, and possibly a quality trade-off that depends on the chosen branch.

- *Soft real-time, not hard real-time:* Unlike traditional (closed) real-time systems, where applications are often safety-critical, software is trusted, and component interactions are highly controlled, in a context where devices communicate over open networks, hard real-time guarantees may not always be possible. This changes how one formulates scheduling problems from "all-or-nothing" schedulability problems to a variant of performance or quality optimization problems. Quality will often be mentioned as an attribute of (collected or derived) information.

We present the problem of scheduling the acquisition of pertinent data items (objects) to support real-time *decision-making*. We assume that making an informed decision (*e.g.*, by a control mechanism) requires the acquisition of a specified set of data items that furnish the requisite information. Since devices on the Internet of Things, such as sensors, may have limited battery power and bandwidth, we assume a normally-off sensor model: no measurements of the environment are made until they are requested for a decision. When a data item is requested (which typically represents a measurement of the environment), sampling is started, and measurements are periodically sent. When a decision is made, all devices that furnished the data are de-activated again. We assume that each data item has a *validity interval* after which it is considered stale. The validity interval determines the sampling period.

Decision-making must obey two constraints. First, each decision must be made by a deadline. We call this constraint the *schedulability constraint*. Second, at the time a decision is made, all data items it is based on must be within their validity intervals. Otherwise, the decision would be based on stale information. We call it the *validity*

constraint (or *freshness constraint*, interchangeably).

Firstly, in Chapter 8, we consider the challenge of maximizing the quality of information collected to meet decision needs of real-time Internet-of-Things applications. We adopt an on-demand object retrieval model and assume that data may have several alternative quality levels. Each object at each level may have a different validity interval after which is it consider stale, thereby imposing a per-object scheduling constraint. We then discuss a family of resulting data-centric scheduling problems and show that they are generally intractable (because multiple-choice knapsack, which is NP-hard, can be reduced to a special case of the problem). We then present several scheduling heuristics and compares their performance in simulation, drawing conclusions on aspects of the solution space.

In Chapter 9, we derive the optimal scheduling policy that meets both schedulability and validity constraints in the presence of multiple decision tasks where i) data objects are independent and distinctive across multiple decision tasks and ii) each data has a single quality level. We show that when only a single decision task is present at a time, the *optimal* data retrieval policy is to retrieve the *Least Volatile item First (LVF)* (where an item is said to be less volatile if it has a *longer* sampling period). We also prove that, if multiple decision tasks are present, the *optimal* policy is a form of hierarchical scheduling policy that retrieves first the items belonging to the task with the earliest constraint (the smallest minimum of a validity interval expiration and a decision deadline) and retrieves the least volatile item first among the items needed for the same decision task. We call it *Earliest Deadline or Expiration First - Least Volatile First (EDEF-LVF)*. In addition to that, we explore the impact of inter-decision data dependency by proposing several heuristic algorithms to schedule dependent data objects that are used by multiple decision tasks. The evaluation result shows that exploiting the benefit of considering shared data items thus reusing such eligible shared items enhances the network resource utilization and thus schedulability.

Chapter 8

On Maximizing Quality of Information for the Internet of Things

In this chapter, we consider the challenge of maximizing quality of information collected to meet decision needs of real-time Internet-of-Things applications. A novel scheduling model is proposed, where applications need multiple data items to make decisions, and where individual data items can be captured at different levels of quality. We assume the existence of a single bottleneck over which data objects are collected and schedule the transmission of these objects over the bottleneck to meet decision deadlines and data validity constraints, while maximizing quality. A family of heuristic algorithms is presented to solve this problem. Their performance is empirically compared leading to insights into the solution space.

8.1 Introduction

This chapter presents the problem of maximizing quality of information collected for making real-time decisions in Internet-of-Things (IoT) applications. The work is motivated by increased connectivity among physical devices with sensing and actuation capabilities, leading to such visions as smart cities, green buildings, intelligent transportation, and personalized healthcare, among others. As mentioned earlier, the emerging IoT systems change the landscape of real-time scheduling problems addressed in more traditional applications, such as avionics and process control in the following three points: *Data, not processes*, *Multiple choice problems* and *Soft real-time, not hard real-time*.

The above changes suggest a new set of real-time scheduling problems that maximize *quality of information* (QoI), subject to timing constraints. Indeed, information is one of the main commodities in the IoT world. It may reflect measurements of the physical world that applications use for various decision purposes, or some derivate products thereof. In networking and data fusion contexts, the use of the term *quality of information* (as distinct from the term, *quality of service*) was made popular by military applications [Perry et al., 2004]. The term was coined to explicitly separate quality attributes of content objects from quality attributes of the channel (or *service*) given to the communication flow. Quality of information refers to the former and is often simply defined as "the fitness for use of the information provided".[1] While many dimensions of information quality were proposed (including accuracy, timeliness, completeness, and security, to name a few) [Miller, 1996], we shall henceforth use the simpler generic notion of "fitness for

[1] See https://en.wikipedia.org/wiki/Information_quality.

use", abstracted by a single information utility value. Hence, we consider a *unidimensional* information utility optimization problem. As a simple starting point, we further specialize in a subset of unidimensional information utility optimization problems featuring data communication over a *single* bottleneck. The purpose is to emphasize the novel aspects of the problem, such as sporadic on-demand sensor tasking and its implications on scheduling constraints. More work is ultimately needed to explore the general multidimensional quality of information optimization problem space, as well as problems involving multiple resource bottlenecks.

While IoT brings new interest into the above problem space, the underlying optimization bears similarity to prior work in at least two domains. The first body of related work is research on real-time databases, where data was also the main commodity and scheduling problems were motivated by data management needs [Song and Liu, 1995, Adelberg et al., 1995, Adelberg et al., 1996, Lee et al., 1996, Kang et al., 2002b, Kang et al., 2002a, Kao et al., 2003, Gustafsson and Hansson, 2004a, Gustafsson and Hansson, 2004b, Xiong and Ramamritham, 2004, Kang et al., 2004, Xiong et al., 2005, Xiong et al., 2008]. Prior work separated instances of data use from instances of data updates [Adelberg et al., 1995, Kang et al., 2002b, Kang et al., 2004, Xiong et al., 2008], resulting in algorithms that have the freedom to schedule access and update transactions separately. Much work assumed periodic updates. In contrast, a novel aspect of a category of IoT devices lies in their *on demand activation*. This model is motivated by sensing devices that have limited power and connection bandwidth, and hence lie idle until activated by the need for data. When sensing devices (that update the data) are activated *on demand*, data update occurs on data access. The resulting linkage of access and update times removes a previously assumed degree of freedom, making it harder to satisfy all constraints; a challenge addressed in this chapter.

The second body of work that resembles the information utility optimization problem lies in early quality of service (QoS) research [Lee et al., 1999, Nahrstedt and Smith, 1995, Abdelzaher et al., 2000]. The work developed algorithms for negotiating attributes of data flows such that some notion of utility is optimized. Inspired by multimedia applications, quality attributes were generally defined summarily for (all objects of) the *entire flow* and not separately for *individual objects*. For example, preventing jitter in multimedia playback required that all video frames suffer similar end-to-end delays. In contrast, in IoT applications, the items in question will often represent heterogeneous sensor data collected from large numbers of different sensors. Individual objects may therefore have *different* constraints such as different validity intervals. The existence of per-object (as opposed to per-flow) constraints, together with the on-demand object update model change the nature of the scheduling and information utility optimization problem.

In this chapter, we adopt an on-demand object retrieval model and assume that data may have several alternative quality levels. Each object at each level may have a different validity interval after which is it consider stale, thereby imposing a per-object scheduling constraint. We then discuss a family of resulting data-centric scheduling problems and show that they are generally intractable (because multiple-choice knapsack, which is NP-hard, can be reduced

to a special case of the problem). We then present several scheduling heuristics and compares their performance in simulation, drawing conclusions on aspects of the solution space.

8.2 Motivating Example

To motivate the problem addressed in this chapter, consider a disaster response scenario in a smart city. The city is equipped with emergency cameras that can send pictures of afflicted areas in an emergency to help rescue efforts. In the aftermath of some natural disaster, such as an earthquake or a hurricane, a team of first-responders must deliver help to different city locations. Since the conditions of streets are unknown, pictures are solicited to determine the best route to use to each target location. There may be multiple alternative routes that lead to each destination. Each route is composed of multiple segments covered by different cameras. These cameras are tasked with sending a fresh picture of the corresponding route segment to the command center. The commander uses these pictures to decide on the best routes to take. There may be a level of urgency associated with sending help to each of the respective locations. It can be approximated by a decision deadline.

Note how this scenario exhibits multiple features of IoT scheduling problems, described in the introduction. Following the presented bulleted list of IoT system features, observe that, first, the scheduled commodity here is chunks of data (the pictures), not processes, flowing from cameras to the command center. Second, there will ordinarily be choice regarding the cameras used or quality of objects. For example, the same street might be covered by cameras of different resolution, or the same camera may be able to deliver pictures of different size. Finally, the problem at hand is more of an optimization problem than a hard guarantee problem. In this case, the higher the quality of collected data the better, but decisions can generally be made with low-quality pictures as well.

Comparing with earlier work on similar optimization problems, observe that the above example features a sporadic data retrieval model. Cameras need not report (or even take pictures) until tasked to do so. Also, individual objects may have different per-object constraints. In this case, they have different validity intervals that stem from differences in the dynamics of reported observations. For example, pictures of areas that are further from the path of damage will likely have longer validity than pictures taken near active problems such as active fires or water leaks. Hence, the problem is one of optimizing information utility accrued from retrieving data objects (from sensors) on demand, while meeting per-object scheduling constraints and overall decision deadlines.

8.3 Problem Formulation

Consider the data object retrieval problem to support decision-making. We assume that making a decision requires retrieval of multiple objects that supply the requisite data. We call the task of retrieving all objects needed for a decision,

the *decision task*. An object can be retrieved in one of multiple forms, called *versions*, that differ in information utility. For example, retrieved images can differ in resolution offering a different utility when it comes to a given use such as threat detection. Each object version also has a validity interval after which it becomes stale. Since lower-quality objects carry less detail, they normally need less resources to deliver and may also have longer validity intervals (because details that can become invalid sooner have been removed). Each decision itself has a deadline. A decision can be made only when all objects needed for it have been retrieved. We call this time instant, the *finish time* of the decision task. It refers to finishing retrieval of the requisite objects. The decision is *valid* if all objects retrieved to make that decision remain within their validity intervals at the finish time of the decision task. A decision is *timely* if the finish time is before the decision deadline. The goal is to develop algorithms for deciding on the retrieved version of each object retrieved, such that (i) decision deadlines are met, (ii) decisions are valid, and (iii) the accrued information utility is maximized.

More formally, consider the set of decision tasks, $\tau_{total}(t)$, that have arrived since system start time and until the current time, t. Each decision task, $\tau_j \in \tau_{total}(t)$, is described by an arrival time, A^j, a relative deadline D^j, an absolute deadline $d^j = A^j + D^j$, and a set of objects it needs to collect, denoted by O^j, where $O^j = \{O_1^j, O_2^j, \ldots, O_{m_j}^j\}$ and $m_j = |O^j|$ is the number of items in that set. Let the finish time of decision task, τ_j, called F^j, be defined as the time instant when retrieval of all objects $O_i^j \in O^j$ is complete.

Object retrieval occurs over a bottleneck, called the *channel*. Each object, O_i^j, has multiple versions of QoI levels.[2] Retrieving version k of object, O_i^j, denoted by $O_i^j[k]$, incurs a retrieval time, $C_i^j[k]$. The retrieved object has a validity interval, $I_i^j[k]$, and information utility, $Q_i^j[k]$ ($0 \leq Q_i^j[k] \leq 1$). The validity interval denotes the interval of time during which the content remains fresh (and hence, valid). This interval starts at the instant when retrieval of the object begins. Let us call the instant when retrieval of $O_i^j[k]$ begins, time t_i^j. We assume that object retrieval is non-preemptive. Without loss of generality, we index the different versions of each object such that $Q_i^j[1] < Q_i^j[2] < \ldots$. We also define an additional virtual QoI level $k = 0$, corresponding to skipping the retrieval of object O_i^j. Hence, $O_i^j[0]$ is NULL, and $Q_i^j[0] = Penalty$, where $Penalty$ is the negative utility associated with not being able to retrieve the object. For each decision task, τ_j, the following constraints must be satisfied:

Timeliness: $F^j \leq d^j$

Validity: $\forall i : F^j \leq t_i^j + I_i^j$

The notations defined above are summarized in Table 8.1.

The total information utility of retrieved objects, $Q(t)$, for all decision tasks in set $\tau_{total}(t)$ is given by:

[2] In the rest of this chapter, we shall use the terms *level* and *version* interchangeably.

Table 8.1: Notation

Symbol	Description
τ_j	task j
A^j	Arrival time of τ_j
D^j	Relative deadline of τ_j
d^j	Absolute deadline of τ_j, $(d_j = A_j + D_j)$
F^j	Finish time of τ_j
O^j	A set of objects needed for τ_j; $Q^j = \{O_1^j, O_2^j, \ldots, O_{m_j}^j\}$
$Q_i^j[k]$	Information utility of O_i^j at QoI level k
$C_i^j[k]$	Retrieval time of O_i^j at QoI level k
$I_i^j[k]$	Freshness interval of O_i^j at QoI level k
Q	Total quality of the objects retrieved

$$Q(t) = \sum_{\tau_j \in \tau_{total}(t)} \sum_{i=1}^{m_j} Q_i^j[k_i^j] \tag{8.1}$$

where k_i^j is the retrieved version of object O_i^j.

Our objective is to find level k_i^j for each object, O_i^j, of each arrived task, τ_j, such that $Q(t)$ is maximized, while meeting timeliness and validity constraints. This can be expressed as the following maximization problem:

Maximize $Q(t)$ subject to:

$\forall j$: $F^j \leq d^j$ and $\forall i, j : F^j \leq t_i^j + I_i^j$

8.4 Solution Algorithms

Since we do not know future task arrivals, we cannot solve the above problem offline. Instead, we need to develop a solution that can be updated dynamically as new tasks arrive. Hence, at any given time, we focus on scheduling the retrieval of objects of only those tasks that have arrived but not yet finished. In other words, we focus on tasks in the current *busy period*, defined as a period of continuous utilization of the channel. We call them *current tasks*. Let $\Gamma(t)$ denote the set of current tasks at time t. Formally, $\Gamma(t)$ is defined below.

Definition 1. Current tasks: The set of current tasks, $\Gamma(t)$, is defined as follows. If at time, t, the channel is idle, then $\Gamma(t) = \emptyset$. Otherwise, $\Gamma(t) = \{\tau_j | A^j \leq t \leq F^j\}$.

It is also useful to define the total set of remaining objects (to retrieve) at time, t, denoted by $R(t)$, as follows:

Definition 2. Remaining objects: For each task, $\tau_j \in \Gamma(t)$, we define the set of remaining objects of task τ_j (to retrieve) at time t, denoted by $R^j(t)$, as the subset of objects, $R^j(t) \subset O^j$, whose retrieval has not started by time t

(i.e., for which the proposition, $t_i^j < t$, is false). We define the *total* set of remaining objects at time, t, denoted by $R(t)$, as the union of sets, $R^j(t)$, over all current tasks, $\tau_j \in \Gamma(t)$.

We are now ready to re-define the information utility optimization problem in terms of the remaining objects of current tasks. Specifically, the information utility accrued from retrieving the set of remaining objects, $R(t)$, denoted by $Q_\Gamma(t)$, is given by:

$$Q_\Gamma(t) = \sum_{\tau_j \in \Gamma(t)} \sum_{O_i^j \in R^j(t)} Q_i^j[k_i^j] = \sum_{O_i^j \in R(t)} Q_i^j[k_i^j] \tag{8.2}$$

Our algorithms aim at finding an appropriate version, k_i^j (for each remaining object of each current task), such that the above quality metric is maximized, while meeting task timeliness and validity constraints. Note that, the solution is revisited each time a new task arrives.

We use a greedy heuristic to solve this problem. The algorithm checks the schedulability of different retrieval options upon each task arrival and computes an appropriate k_i^j for each object to be retrieved such that total information utility is maximized and constraints are met. Two variations of the greedy approach are compared: one is top-down (Optimistic Algorithm) and the other is bottom-up approach (Pessimistic Algorithm).

The two algorithms differ in the order in which they explore the solution space in search for a solution that satisfies the (timeliness and validity) constraints. Let K_i^j be the top QoI level at which object O_i^j is available. The optimistic algorithm starts from the top quality level $k_i^j = K_i^j$, for all objects, O_i^j, that have not yet been retrieved (i.e., for all $O_i^j \in R(t)$). It explores other lower quality solutions only if earlier solutions were found unschedulable. In contrast, the pessimistic algorithm starts from the least quality solution that can still retrieve all objects (i.e., $k_i^j = 1$ for all $O_i^j \in R(t)$). If that solution violates the constraints then no solution (that retrieves all objects) exists.[3] Otherwise, the algorithm incrementally improves solution quality, while constraints permit. Evaluation compares the optimistic solutions to pessimistic solutions in underloaded versus overloaded systems, showing interesting insights. Below, we describe these solutions in more detail.

8.4.1 Quality Adjustment Algorithms

Optimistic Algorithm

The Optimistic algorithm, denoted by OPTIMISTIC, is top-down approach. It starts from the highest QoI level for each data object, and then moves to lower levels, if needed. Specifically, upon a new task arrival, the algorithm updates the set of current tasks, $\Gamma(t)$, defined above (see Definition 1), and recomputes the total set of remaining objects, $R(t)$ (see Definition 2). If they are schedulable at the top QoI level, it returns the selected version of each

[3] We assume that the penalty for missing an object is high enough that solutions that retrieve subsets of the object set only are not considered.

object and terminates. Otherwise, it chooses an object and lowers its QoI level. We call this operation, *demotion*. Which object to demote is an important issue to consider. We use a greedy approach that picks an object whose demotion from its current level to a new lower level, low, will result in the minimum impact on total information utility, normalized by some notion of cost savings. We call this quantity the *impact factor* of the change in object QoI level. The steps of object selection and demotion are repeated until constraints are met. There are multiple ways the impact factor can be computed. This discussion is deferred to a later section. For now, assume the existence of a function $ComputeImpactFactor(O_i^j, hi, low)$, that can compute the normalized difference in information utility per unit difference in cost between any two QoI levels, hi and low, of any $O_i^j \in R(t)$. The steps of the optimistic algorithm, executed on task arrival, are thus as follows:

1) Update the set of current tasks, $\Gamma(t)$.

2) Update the remaining object set, $R(t)$.

3) Initialize the selected retrieval version of each object $O_i^j \in R(t)$ to the maximum QoI level, $k_i^j = K_i^j$.

4) If the deadline and validity constraints are met, return the selected versions, k_i^j, for all objects. Stop.

5) Otherwise, for all objects, $O_i^j \in R(t)$, and for all possible demotion levels, $new_i^j < k_i^j$,
find $ComputeImpactFactor(O_i^j, k_i^j, new_i^j)$.

6) Pick the demotion (i.e., object O_i^j and new level new_i^j) that offers the minimum impact factor.

7) Demote O_i^j by setting $k_i^j = new_i^j$

8) Go to 4).

In the evaluation, we shall distinguish two versions of the above algorithm; namely INCREMENTAL-OPTIMISTIC and OPTIMISTIC. In the OPTIMISTIC algorithm, retrieval level of *all objects* (of both the new tasks and the existing tasks) are initialized to the highest level. In INCREMENTAL-OPTIMISTIC, we only initialize retrieval level of objects of the newly arrived task(s). For previous tasks, we start with their previously computed levels.

Pessimistic Algorithm

The Pessimistic algorithm, denoted by PESSIMISTIC, is bottom-up approach: it starts from the lowest QoI level for each data object, and then moves up to higher levels. Specifically, upon a new task arrival, it updates the remaining object set, $R(t)$, and checks if constraints are satisfied at the lowest level $k_i^j = 1$ for each $O_i^j \in R(t)$. If not, the task set is infeasible and the new arrival cannot be served. Otherwise, it considers upgrading the QoI levels. To upgrade a QoI level, an object is selected whose upgrade results in the maximum impact on information utility per unit of added

cost. In other words, an object and a new level are picked that maximize $ComputeImpactFactor(O_i^j, hi, low)$ (when the object is upgraded from its current level, low, to the new level, hi). This is repeated until no feasible promotions are possible while satisfying the constraints. The steps of this algorithm, executed on task arrival, are shown below:

1) Update the set of current tasks, $\Gamma(t)$.

2) Update the remaining object set, $R(t)$.

3) Initialize the selected retrieval level of each object $O_i^j \in R(t)$ to the minimum level, $k_i^j = 1$.

4) If the deadline and validity constraints are violated, reject new task. Stop. Otherwise, continue.

5) For all objects, $O_i^j \in R(t)$, and for all possible promotion levels, $new_i^j > k_i^j$,
find $ComputeImpactFactor(O_i^j, new_i^j, k_i^j)$.

6) Tentatively, pick the promotion (i.e., object O_i^j and new level new_i^j) that offers the maximum impact factor.

7) If the deadline and/or validity constraints are violated, undo promotion. Stop.

8) Otherwise, promote O_i^j by setting $k_i^j = new_i^j$

9) Go to 5).

Similarly to the optimistic case, we distinguish two variations of the above; namely, PESSIMISTIC and INCREMENTAL-PESSIMISTIC, where in INCREMENTAL-PESSIMISTIC we only initialize retrieval level of objects of new task(s). For previous tasks, we start with their previously computed levels.

Two more issues must be resolved in order to complete the description of the above algorithms. First, how to determine if a particular choice of QoI levels for retrieved objects meets constraints? This is a schedulability analysis problem. Second, how to implement $ComputeImpactFactor(O_i^j, hi, low)$? These two issues are addressed in the following two subsections, respectively.

8.4.2 The Scheduling Policy

In the optimistic and pessimistic algorithms described above, step (4) checks whether or not constraints are violated. This is a schedulability analysis problem. It is possible to offer further variations of the QoI optimization space by choosing the underlying scheduling policy.

Two sets of constraints govern the design of the scheduling policy. The first is decision task deadlines, d^j. The other is validity intervals, I_i^j, of retrieved objects. For a given decision task, it is easier to meet validity constraints of data objects if a data object, O_i^j, that expires later (i.e., has a larger validity interval, I_i^j) is retrieved earlier. This

is because, once retrieval of a data object starts, its validity interval starts counting towards expiration. Hence, data objects with shorter validity intervals should be retrieved last to make sure they are still fresh at time F_j, when a decision is made based on those objects. We call this data retrieval policy *Least Volatile First* retrieval (LVF).

There are several policies one can conceive of that may attempt to meet both sets of constraints. Policies such as *Earliest Deadline First* (EDF) and *Deadline Monotonic* (DM) are good at meeting deadlines, whereas LVF, as argued above, is good at meeting validity interval constraints. The two can be combined by picking a task first in an order determined by task deadlines, then retrieving the objects pertaining to the selected task in an LVF manner. Another way to combine the policies would be to design a hybrid policy, *Earliest Deadline or Expiration First* (EDEF), which computes the earlier of the absolute deadline of a task and the first expiration time of any of its object validity intervals, then assigns the highest priority to the task with the shortest of the computed resulting values. Accordingly, the following scheduling policies are considered:

1. EDF-LVF: Tasks are first scheduled using EDF. When a task is scheduled, its objects are retrieved in an LVF order.

2. DM-LVF: Tasks are first scheduled using deadline monotonic. When a task is scheduled, its objects are retrieved in an LVF order.

3. EDEF-LVF: Tasks are scheduled using EDEF (described above). When a task is scheduled, its objects are retrieved in an LVF order.

4. NONE-LVF: All objects are scheduled by LVF no matter what task they belong to. If multiple objects have the same validity, an object of a task which arrived earlier has a higher priority.

In the evaluation section (Section 8.5), we compare their performance, success ratio and overall quality.

8.4.3 Computing the Impact Factor

In this work, we compare five options for computing the impact factor, $ComputeImpactFactor(O_i^j, hi, low)$. Informally, the goal of this function is to compute some notion of normalized information utility gain per unit cost, when comparing retrieval of two versions of object, O_i^j, indexed hi and low. There is less ambiguity in how to compute the information utility gain. In a unidimensional optimization problem, it is simply given by the difference, $Q_i^j[hi] - Q_i^j[low]$. The notion of cost, however, is less obvious, leading to multiple opportunities for normalization. For example, cost may be measured by the absolute difference, $C_i^j[hi] - C_i^j[low]$. It can alternatively be measured by difference in utilization. However, it is not immediately clear what to divide retrieval times by in order to compute

utilization. The existence of multiple constraints (namely, deadlines and validity interval constraints) leads to multiple notions of utilization that may be considered. This leads to several options for computing the impact factor, as described below:

Option 1: OVERDEFAULT. Normalize by the absolute difference in object retrieval times.

$$\frac{Q_i^j[hi] - Q_i^j[low]}{C_i^j[hi] - C_i^j[low]}$$

Option 2: OVERI. Normalize by the difference in object utilization, computed by dividing object retrieval times by their validity intervals.

$$\frac{Q_i^j[hi] - Q_i^j[low]}{\frac{C_i^j[hi]}{I_i^j[hi]} - \frac{C_i^j[low]}{I_i^j[low]}}$$

Option 3: OVERD. Normalize by the difference in object utilization, computed by dividing object retrieval times by task relative deadlines.

$$\frac{Q_i^j[hi] - Q_i^j[low]}{\frac{C_i^j[hi]}{D^j} - \frac{C_i^j[low]}{D^j}}$$

Option 4: OVERMINDI. Normalize by the difference in object utilization, computed by dividing object retrieval times by the more stringest of the above two constraints.

$$\frac{Q_i^j[hi] - Q_i^j[low]}{\frac{C_i^j[hi]}{\min(I_i^j[hi], D^j)} - \frac{C_i^j[low]}{\min(I_i^j[low], D^j)}}$$

Option 5: OVERNONE. Do not normalize.

$$Q_i^j[hi] - Q_i^j[low]$$

In the evaluation section, we empirically compare the implications of the above choices for computing the impact factor on the quality of solutions to the original optimization problem.

8.5 Evaluation

In this section, we evaluate and compare the various algorithmic options. The options are as follows:

Quality adjustment

- OPTIMISTIC: this is Optimistic algorithm. For quality adjustment, as a top-down approach, it starts with the

highest level and moves to the lower levels.

• INCREMENTAL-OPTIMISTIC: this is a variation of Optimistic algorithm. The difference lies in that quality level adjustments for the objects in $R(t)$ start from their *current* quality levels, upon a task arrival (when scheduling decision is required).

• PESSIMISTIC: this is Pessimistic algorithm. As a counterpart of OPTIMISTIC, for quality adjustment, it starts with the lowest level and moves to the higher levels as a bottom-up approach.

• INCREMENTAL-PESSIMISTIC: this is a variation of Optimistic algorithm. The difference lies in that quality level adjustments for the objects in $R(t)$ start from their *current* quality levels, upon a task arrival (when scheduling decision is required).

Scheduling policy

• EDF-LVF: it assigns the highest priority to a task which has the earliest absolute deadline when scheduling decision is needed. Object retrievals are scheduled by LVF order.

• DM-LVF: it assigns the highest priority to a task which has the shortest relative deadline, and object retrievals are scheduled by LVF order.

• EDEF-LVF: it assigns the highest priority to a task which has the smallest value either of validity expiration or absolute deadline, and schedules object retrievals in LVF order.

• NONE-LVF: no matter what task an object belongs to, all objects are scheduled by LVF. If one more objects have the same validity, an object of a task which arrived earlier has a higher priority.

Impact factor

Impact factors, OVERDEFUALT, OVERD, OVERI, OVERMINDI and OVERNONE, are calculated as described in Sec. 8.4.3.

We combine one of the options for each quality adjustment, impact factor, and scheduling policy. For instance, if we used PESSIMISTIC for quality adjustment, OVERDEFAULT for impact factor, and EDEF-LVF for scheduling policy, it is denoted by PESSIMISTIC + OVERDEFAULT + EDEF-LVF. Upon a case that only a single kind of option is compared and the other options are the same, the same components can be omitted in an expression.

We generated 2,000 synthetic samples. A task's utilization is also evenly from one of the 10 groups, $(0-1.0], (0.1-0.2], \ldots, (0.9-1.0]$. In order for the overall task load to be dominantly controlled by arrival rate, the number of data items in each task is fixed as 5. We run all samples for 100,000 time units. Each deadline of decision task is randomly drawn from [5-1000]. The validity interval and the retrieval time of each data item are randomly drawn from [5-200]

and [1-30], respectively. From a higher to lower quality level, retrieval time can decrease by 1 or 2. For validity and quality index, a value is picked between the previous value and a value exponentially decreased as much as the retrieval time decreases. To show the result according to arrival rate, we run data samples of average urgency of 0.2-0.3 for 10 arrival rate groups. The inter-arrival times follow a Poisson process with arrival rate evenly from 10 groups, $0.001 * (1.6)^i$ where $0 \leq i \leq 9$. If a generated inter-arrival time is less than the deadline, the invocation is discarded.

We define `average urgency` as

- `average urgency` $= \Big(\prod_j (\frac{\sum_{i=1}^{m_j} C_i^j[k]}{D_j})\Big)^{\frac{1}{N(\tau_{total}(t))}}$.

where $N(\tau_{total}(t))$ is the number of tasks in $\tau_{total}(t)$. This is a geometric mean of $(\frac{\sum_{i=1}^{m_j} C_i^j[k]}{D_j})$ for all tasks. This value reflects that the sum of the all retrieval times of a task is how close to its deadline. If this value is large, for a decision task set, it could be unschedulable in the first place depending on tasks' arrival patterns.

In addition, we define `average utilization` as follows:

- `average utilization` $= \dfrac{\sum_j \sum_{i=1}^{m_j} C_i^j[k]}{\text{simulation duration}}$.

This value reflects how much of data load supposed to execute is present for the simulation duration. Since some tasks can be dropped or some objects' quality are adjusted depending on system's availability, the whole amount of the load could not finally execute.

Lastly, `effective quality` is defined as follows:

- `effective quality` $= \dfrac{Q(t)}{N(\tau_{total}(t))}$.

That is, `effective quality` is an average quality over the all released tasks.

8.5.1 Optimistic vs. Pessimistic Algorithm

Fig. 8.1 and Fig. 8.2 show the effective quality and success rate, respectively, for the PESSIMISTIC, INCREMENTAL-PESSIMISTIC, OPTIMISTIC and INCREMENTAL-OPTIMISTIC algorithms, as a function of average urgency. Fig. 8.3 and Fig. 8.4 show the results according to `average utilization`. Three observations are noted. First, observe that optimistic algorithms tend to perform better. This may be attributed to the fact that they tend to traverse the solution space in smaller steps; steps that make minimum impact, as opposed to pessimistic algorithms that traverse the solution space in steps that make maximum impact. Hence, the latter may be more prone to getting trapped in local optima. Second, the INCREMENTAL-OPTIMISTIC algorithm does better than the plain OPTIMISTIC. This may

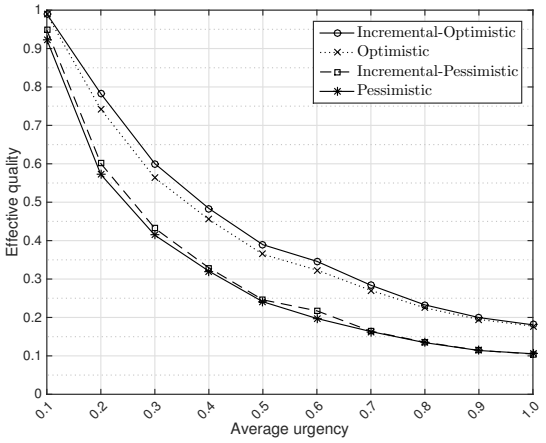


Figure 8.1: Effective quality according to average urgency.

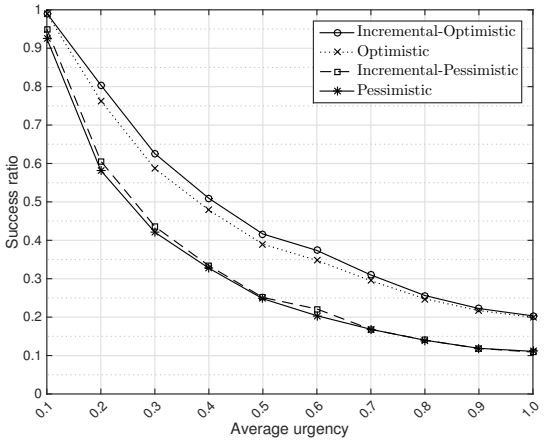


Figure 8.2: Success ratio according to average urgency.

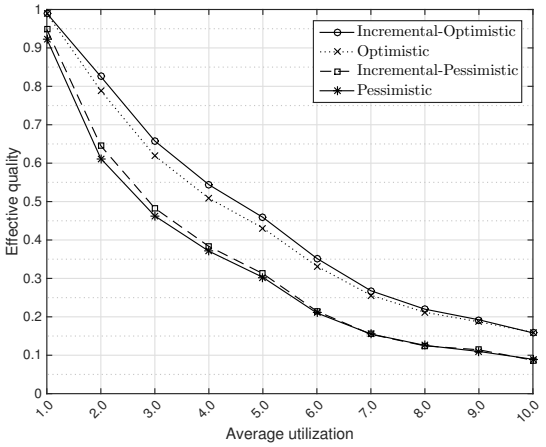


Figure 8.3: Effective quality according to average utilization.

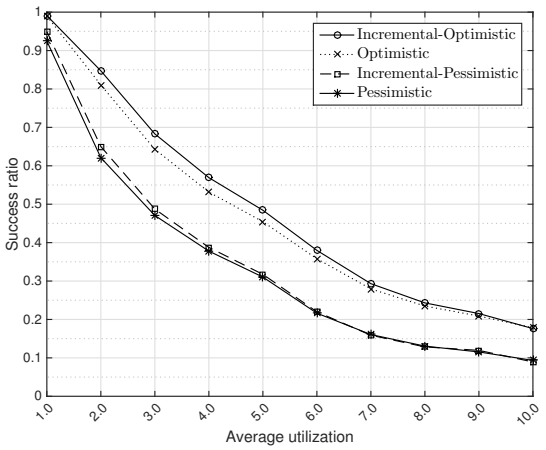


Figure 8.4: Success ratio according to average utilization.

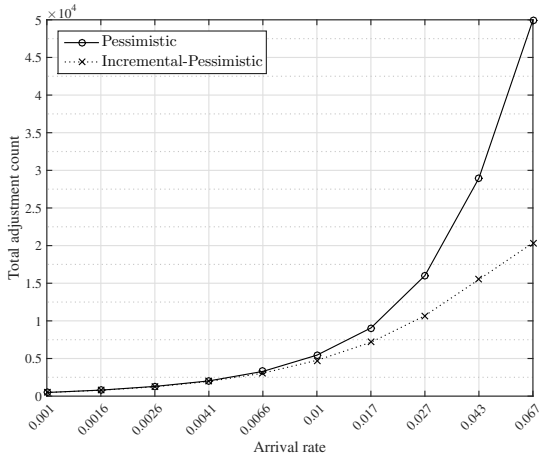


Figure 8.5: Total adjustment count according to arrival rate.

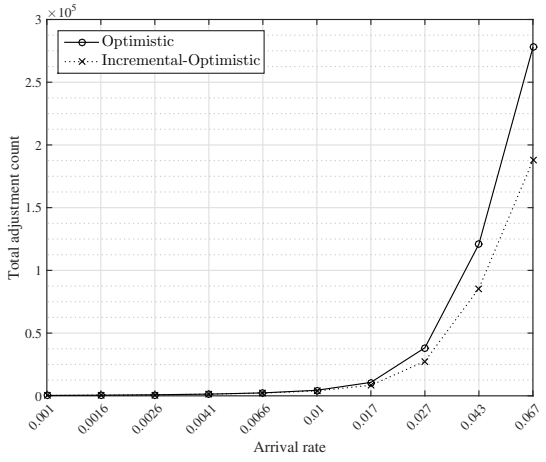


Figure 8.6: Total adjustment count according to arrival rate.

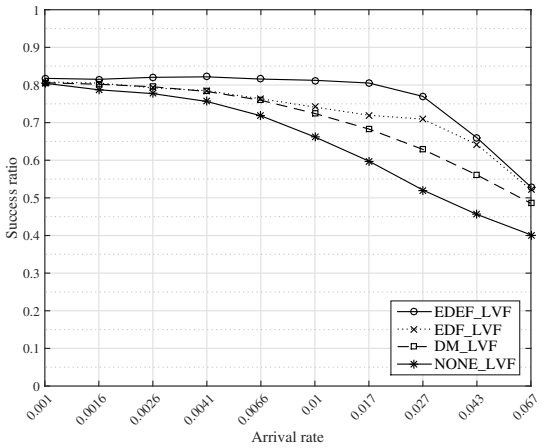


Figure 8.7: Success ratio of various scheduling options for retrieving order according to arrival rate.

be attributed to the fact that it (i) starts from a position that is better adapted to current load (as opposed to starting from top levels of all objects), and (ii) moves in the direction of demotion, which is consistent with its trigger condition (being triggered by new load arriving that one needs to make room for). Third (and in contrast to the above), the INCREMENTAL-PESSIMISTIC does about the same as the plain PESSIMISTIC because it starts with a level adapted to current load and moves in the direction of promotion, which makes it unlikely to find more solutions that accommodate the extra load resulting from a newly arrived task, compared to PESSIMISTIC. The figure sheds light on the complexity of the problem. Many factors are at play; the step size by which levels are adapted, the direction in which they are adapted, and the initial conditions from which the adaptation starts. A more complete exploration of that space may result in better solutions.

Fig. 8.5 shows `total adjustment count` of PESSIMISTIC and INCREMENTAL-PESSIMISTIC while Fig. 8.6 shows the result of OPTIMISTIC and INCREMENTAL-OPTIMISTIC. As expected, non-INCREMENTAL-approach shows much higher counts than INCREMENTAL-approach in each figure.

8.5.2 Scheduling Policy

We compare the four scheduling policies, EDF-LVF, DM-LVF, EDEF-LVF and NONE-LVF. Fig. 8.7, Fig. 8.8 and Fig. 8.9 plot the success ratio of all samples when scheduled by the four scheduling policies according to arrival rate, average utilization, and average urgency, respectively. The most noticeable result is that, EDEF-LVF shows the best performance in terms of success ratio. EDF-LVF follows the next and DM-LVF is the third while NONE-LVF shows the least performance. This trend holds for the all three figures. This likely stems from the fact that EDEF-LVF considers both deadline and validity constraints when picking the next task. In contrast, EDF-LVF and DM-LVF consider only deadlines when picking the task (and consider validity constraints only internally to a task,

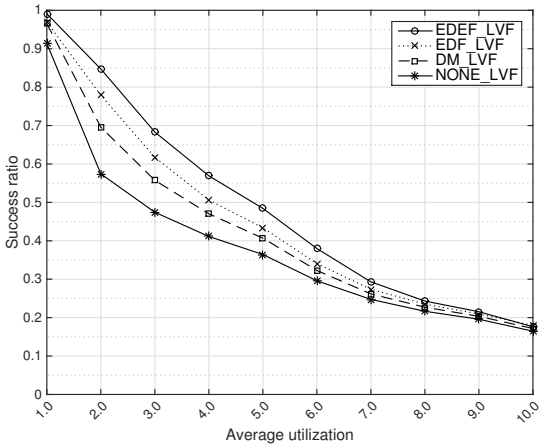


Figure 8.8: Success ratio of various scheduling options for retrieving order according to average utilization.

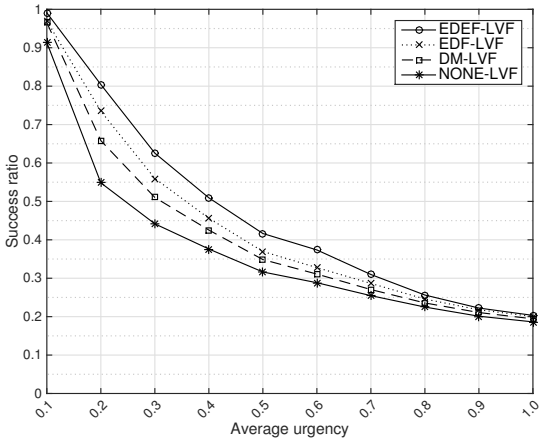


Figure 8.9: Success ratio of various scheduling options for retrieving order according to average urgency.

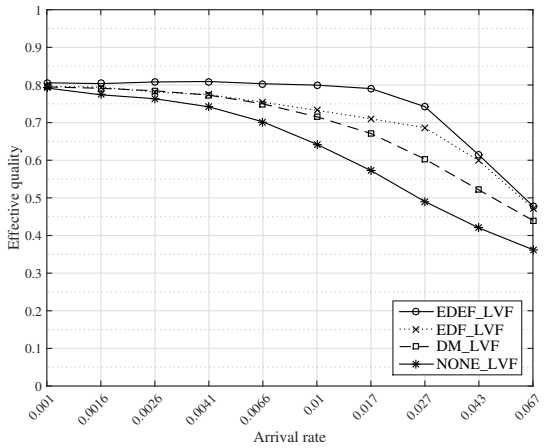


Figure 8.10: Effective quality of various scheduling options for retrieving order according to arrival rate.

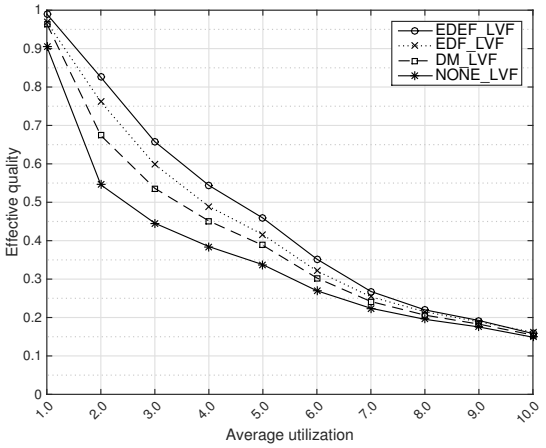


Figure 8.11: Effective quality of various scheduling options for retrieving order according to average utilization.

in retrieving relevant objects for the task, once the task is picked). Finally, NONE-LVF is worst since it does not consider deadlines. Comparing the three figures, note that success ratio is more sensitive to average urgency (see Fig. 8.9) that it is to arrival rate or average utilization.

Next, we compare the scheduling policies in terms of effective quality. Fig. 8.10, Fig. 8.11 and Fig. 8.12 plot effective quality on y-axis versus arrival rate, average utilization and average urgency, respectively. Similar trends are seen to those already described for success ratio.

8.5.3 Impact Factor

Fig. 8.13, Fig. 8.14, and Fig 8.15 plot the adjustment count, effective quality and success ratio, respectively, versus arrival rate, for different choices of impact factor used in the algorithm. Namely, we compare DEFAULT, OVERD,

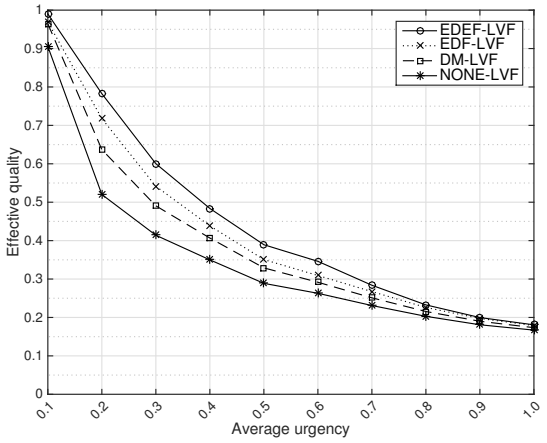


Figure 8.12: Effective quality of various scheduling options for retrieving order according to average urgency.

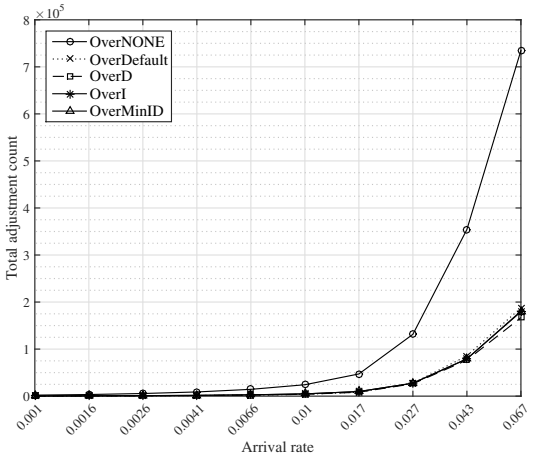


Figure 8.13: Total adjustment count according to arrival rate for various options for impact factor.

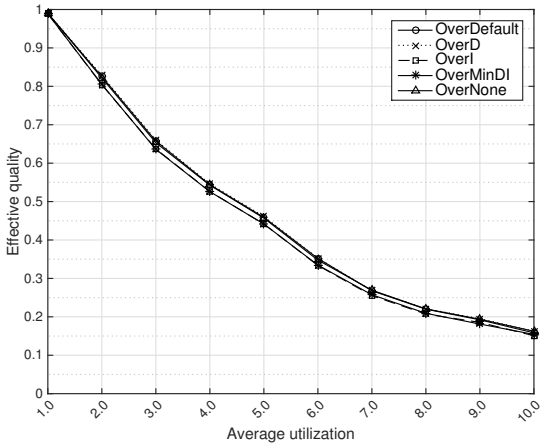


Figure 8.14: Effective quality according to average utilization for various options for impact factor.

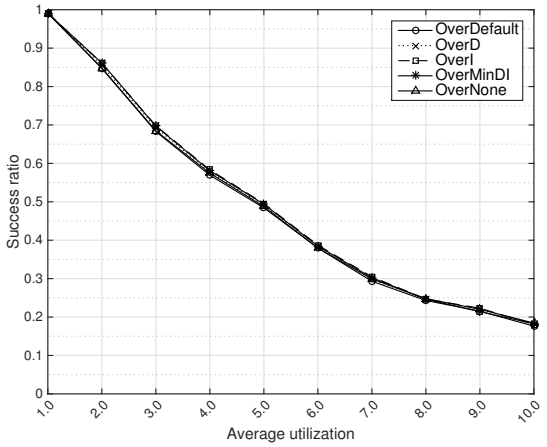


Figure 8.15: Success ratio according to average utilization for various options for impact factor.

OVERI, OVERMINID and OVERNONE. The scheduler used was INCREMENTAL-OPTIMISTIC + EDEF-LVF, since it did best according to results presented so far. Note that, OVERNONE shows very poor performance in terms of `total adjustment count`, while the other approaches and metrics show similar results. More work may be needed to understand the conditions for which the different impact factor choices are best suited.

Chapter 9

Decision-centric Data Scheduling with Normally-off Sensors in Smart City Environment

While in the previous chapter we explored various algorithmic options for maximizing quality of data, in this chapter we develop the *optimal* algorithm to schedule pertinent data items to make multiple decisions when each data item for a decision has a single level of quality. Hence, again, one-off decisions are activated sporadically and thus and also in order to save limited energy and battery life, sensors are normally off and activated by the demand of a decision. Collected data has validity intervals, after which it must be re-sampled, since the measured value may change. Once a decision is made based on the data, sensors are turned off again. We call this model *sporadic decision-centric data scheduling with normally-off sensors*. It gives rise to novel scheduling problems because of the way the timing of activation of different sensors affects load attributed to data sampling; the shorter the interval between activation of a given sensor and the time a corresponding decision is made, the lower the number of samples taken by that sensor to support the decision, and thus decision cost. We define the aforementioned decision-centric data scheduling problem and derive the optimal scheduling policy, called EDEF-LVF, for this task model where data objects are independent and distinctive across multiple decision tasks and each data has a single quality level. Simulation results confirm the superiority of EDEF-LVF compared to several baselines. In addition to that, we investigate the impact of inter-decision data dependency by proposing several heuristic algorithms to schedule dependent data objects that are used by multiple decision tasks. We show that considering such data dependency provides a chance to save more scheduling resource by reusing already sampled data. Most of the work of this chapter is published in [Kim et al., 2016a].

9.1 Introduction

This chapter presents the problem of scheduling the acquisition of pertinent data items (objects) to support real-time *decision-making*. We assume that making an informed decision (*e.g.*, by a control mechanism) requires the acquisition of a specified set of data items that furnish the requisite information. Since devices on the Internet of Things, such as sensors, may have limited battery power and bandwidth, we assume a normally-off sensor model: no measurements of the environment are made until they are requested for a decision. When a data item is requested (which typically represents a measurement of the environment), sampling is started, and measurements are periodically sent. When

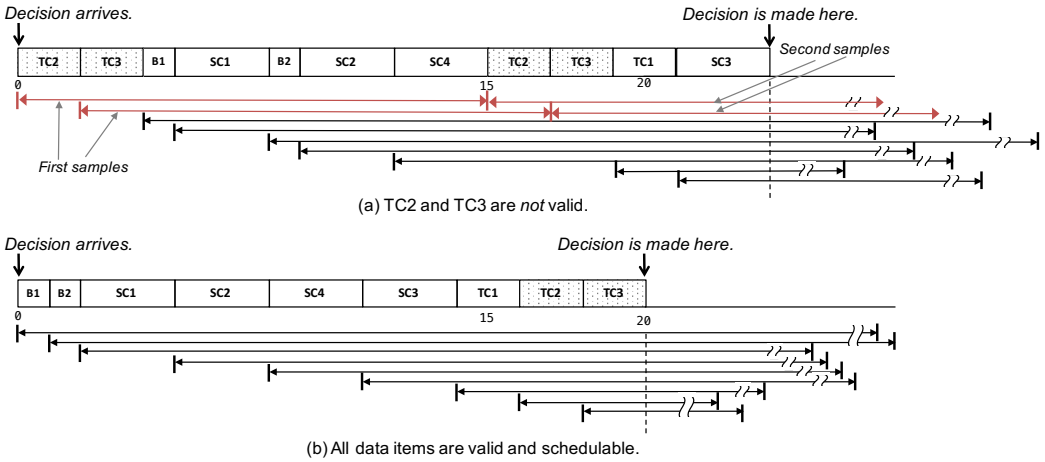


Figure 9.1: In (a), data item TC1 and TC2 are re-sampled since they were retrieved *too early*. Hence a proper ordering of data retrieval is necessary as (b).

a decision is made, all devices that furnished the data are de-activated again. We assume that each data item has a *validity interval* after which it is considered stale. The validity interval determines the sampling period. At the time a decision is made, all data items it is based on must be within their validity intervals. Each decision must also be made by a deadline. Under these two constraints, we derive the optimal scheduling policy in the presence of multiple decision tasks when data objects are independent across decision tasks and each data object has a single quality level.

We show that when only a single decision task is present at a time, the *optimal* data retrieval policy is to retrieve the *Least Volatile item First (LVF)* (where an item is said to be less volatile if it has a *longer* sampling period). We also prove that, if multiple decision tasks are present, the *optimal* policy is a form of hierarchical scheduling policy that retrieves first the items belonging to the task with the earliest constraint (the smallest minimum of a validity interval expiration and a decision deadline) and retrieves the least volatile item first among the items needed for the same decision task. We call it *Earliest Deadline or Expiration First - Least Volatile First (EDEF-LVF)*.

The main contribution of this work over the previous work lies in deriving *the first optimality result for scheduling multiple decision tasks* under the proposed normally-off sensor model. No optimality results have been proven for the general case of multiple decision tasks. We derive an optimal algorithm for this case and compare it to various heuristics demonstrating that it outperforms them in terms of ability to meet both schedulability and validity constraints, while minimizing decision cost. In addition to that, in Section 9.5, we present several heuristic algorithms to schedule dependent data objects that are shared by multiple decision tasks.

9.2 An Illustrative Example

Consider a disaster response infrastructure comprising cameras and other sensors, deployed along key routes, but turned off by default to save battery. An emergency vehicle is equipped with a smart GPS device that can query these sensors for route conditions. Unlike the typical case, where route conditions are fed periodically regardless of demand, we assume a normally-off sensor model, where no information is generated unless explicitly requested.

When prompted by the user, the GPS device must collect information on possible routes to decide on the best route to take for the emergency vehicle. Two resources are involved. A communication channel to the vehicle to get the data (which may include pictures and other measurements), and the local processor to process it. Here, we focus on the former resource.[1]

Our model is agnostic to how decision tasks are invoked. The above scenario suggested that decision tasks (namely, tasks that collect data to decide on routes) are started by the user. Alternatively, they can be started by another sensor. For example, delivery of a fire alarm from a location may automatically trigger a decision task that collects data on routes to that location. Table 9.1 shows an illustrative set of data items that might need to be retrieved, together with the estimated retrieval time of each item and its sampling period (equal to its validity interval), once the sensor is activated.

Table 9.1: An example set of items needed to decide on a route

Item (Acronym)	Retrieval time (sec)	Period (sec)
Bridge 1 health sensor (B1)	1	1500
Bridge 2 health sensor (B2)	1	1500
Security Cam 1 (SC1)	3	120
Security Cam 2 (SC2)	3	120
Security Cam 3 (SC3)	3	120
Security Cam 4 (SC4)	3	120
Traffic Cam 1 - Rural (TC1)	2	30
Traffic Cam 2 - Downtown (TC2)	2	15
Traffic Cam 3 - Downtown (TC3)	2	15

The GPS must decide on the best route to take. Assume that we require a response within 2 minutes. This is the decision deadline. Assume further that data processing takes one minute. This leaves one minute for data retrieval. The sum of retrieval times in Table 9.1 amounts to 20 seconds, which is significantly less than a minute. Hence, the retrieval is schedulable. Nevertheless, the order of retrieval matters. If items TC2 and TC3 are retrieved first, as shown in Figure 9.1 (a), they will be re-sampled by the time the rest of the items are retrieved. Hence, resources (*e.g.*, sensor battery) will be wasted on samples that are not used. On the other hand, if items TC2 and TC3 are retrieved last,

[1]Note that, for a moving vehicle, the communication channel is wireless. In another example, such as collecting data at a command center, the channel may be wired. To keep the discussion simple, we abstract away from the underlying channel technology. We consider a single bottleneck resource.

as shown in Figure 9.1 (b), each sensor will be sampled exactly once by the time the decision is made. Hence, the decision task will minimize resources used. Once a decision is made, the queried sensors are deactivated.

The scenario becomes more complicated if multiple routes need to be computed (for example by a GPS device at a central dispatcher). Deciding on each route will require retrieval of its own data items. Hence, multiple decision tasks are present. In the following section, we formulate the problem more rigorously and present scheduling policies for data retrieval that minimize decision cost used, while meeting deadlines and validity constraints.

9.3 Background

We begin by reviewing the case of a single decision task and introducing some definitions and initial results. While analysis of the single task case is not a contribution of [Kim et al., 2016a], it offers good background pertinent to the main result (presented in the next section), which is an optimal scheduling policy for multiple decision tasks. In our model, a decision task collects data items needed to make a decision. In this model, we assume that to make a decision, one needs to retrieve N data items $O_1, \ldots, O_N$ from corresponding sensors, $S_1, ..., S_N$. The sensors are normally off, but can be activated remotely. Once activated, they sample their environment periodically, at period I_i, equal to the validity interval of the sensor measurement. Delivering a measurement from sensor S_i (*i.e.*, item O_i) takes C_i time units, called the retrieval duration. Let t_i denote the activation time of sensor S_i, which is also the time its data is sampled. Subsequent samples will occur at times $t_{ik}(= t_i + kI_i)$ (where k is an integer). Once all needed data items are retrieved from all N sensors via the communication medium, the decision can proceed. At that time, the sensors are deactivated. Let the time instant at which all decision data has been fetched be denoted by F. We shall henceforth call it the *decision time*. We require that $F \leq D$, where D is the decision deadline. We define the cost of a decision, $Cost$, by the communication resources consumed. Namely:

$$Cost = \sum_{1 \leq i \leq N} C_i \lceil \frac{F - t_i}{I_i} \rceil. \tag{9.1}$$

Let the optimal retrieval policy be one that chooses activation times t_i such that cost is minimized. Clearly, since each item must be retrieved at least once, the optimal cost is:

$$Cost_{opt} = \sum_{1 \leq i \leq N} C_i \tag{9.2}$$

which occurs when $\forall i : F - t_i \leq I_i$. In other words, it occurs when no sensor is sampled twice. Let a feasible retrieval schedule be one that satisfies the decision deadline.

Clearly, *if any feasible retrieval schedule exists, then a feasible retrieval schedule exists with a cost exactly equal*

Table 9.2: Notation for the problem with a single deicison

Description	Notation
Data item i	O_i
Retrieval duration of O_i	C_i
Relative deadline of the decision	D
Validity interval of O_i	I_i
Start time of retrieval of O_i	t_i
Finish time (decision made)	F

to $Cost_{opt}$. This is because at the time the decision is made in any schedule, only one sample from each sensor is within its validity interval. If that sample was obtained at time $t_{ik} > t_i$, other previous samples from the same sensor need not have been retrieved, as they were not used. In other words, the sensor should have been activated at time t_{ik}, thus saving the cost of the extra samples.

The above suggests that we cast our schedulability problem as one of finding sensor activation times t_i and a retrieval order such that decision cost is $Cost_{opt}$. If no solution is found, the problem is unschedulable. Accordingly, we define an *optimal retrieval scheduling policy* as one that finds a retrieval order that meets the two constraints below, whenever any other policy does:

$$\textbf{Data freshness: } t_i + I_i \geq F \ \ (\forall i, 1 \leq i \leq N),$$

$$\textbf{Decision deadline: } t + D \geq F,$$

where the decision task starts at time t. Note that, the freshness constraint above ensures cost minimality. If it is violated, a second sample is taken from the sensor, which makes cost non-optimal. These can also be represented together as

$$\min\left(\min_{1 \leq i \leq N}(t_i + I_i),\ \ t + D\right) \geq F$$

The notation can be found in Table 9.2.

We make the following assumptions: (i) We categorize our work as *online* scheduling in which the upcoming tasks' arrival patterns and their workload are not predictable. (ii) Decision tasks are *not* periodic. (iii) We avoid any re-fetching (second samples) of a data item in order to minimize cost. (iv) Parameters of data validity intervals (*i.e.*, sensor periods) and decision deadlines are given. (v) Retrieval of a single data item is non-preemptible. Hence, if a decision task arrives in the middle of retrieving a data item of another task, the scheduling decision is made when the current retrieval is done.

The optimal solution to the problem of scheduling the acquisition of data items with validity intervals when the items are acquired by a single decision task is the *Least Volatile item First* (LVF). Although the optimality of the LVF

Table 9.3: Notation for the problem with multiple deicisons

Description	Notation
Relative deadline of decision task m	D^m
Arrival time of decision task m	t^m
Finish time (decision made) of decision task m	F^m
Data item i of decision task m	O_i^m
Retrieval duration of O_i^m	C_i^m
Validity interval of O_i^m	I_i^m
Start time of retrieval of O_i^m	t_i^m

scheduling policy has been mentioned and shown before [Hu et al., 2015, Kim et al., 2016b, Kim et al., 2016a], to make this thesis self-contained, we show the proof as follows:

Theorem 8. *If a feasible schedule for data item retrieval that meets freshness and deadline constraints of a decision task exists, then a Least Volatile item First (LVF) schedule is feasible and meets these constraints.*

Proof. Let $O_1, O_2, \ldots, O_N$ be a set of N data items with a certain feasible order that satisfies the validity constraints. Let us consider two items, O_i and O_j where $I_i < I_j$. That is, item O_j has a longer validity interval. Suppose O_i is scheduled first.

Since the schedule is valid, the following inequality should satisfy:

$$\min\left(t_* + I_i, t_* + C_i + I_j\right) \geq F \tag{9.3}$$

when O_i is retrieved at t_*. F is when all data retrievals are complete. Then let us switch the order of O_i and O_j. Then the schedule is feasible if it satisfies the following inequality:

$$\min\left(t_* + I_j, t_* + C_j + I_i\right) \geq F \tag{9.4}$$

when O_j is retrieved at t_*. We can easily see that (9.4) is true if (9.3) is true because $I_i < I_j$. Therefore it is optimal to retrieve an item with longest validity (*i.e.*, least volatile) first. Since the Least Volatile item First schedule can be obtained from any ordering by repeating pairwise reordering as above, we prove the theorem. □

9.4 Optimal Scheduling of Data Item Acquisition for Multiple Decision Tasks

In this section, we present the formal problem description, the optimal algorithm, and the evaluation.

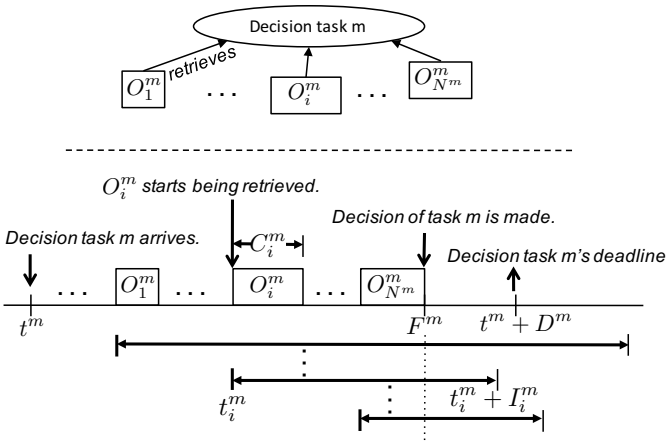


Figure 9.2: Illustrative description of the problem of scheduling data item acquisitions with multiple decision tasks.

9.4.1 Problem Description

In the previous section, we discussed the problem of scheduling pertinent data item acquisitions in the presence of a single decision task. In this section, we solve the problem with *multiple* decision tasks. We denote decision task m's deadline as D^m ($D^m \geq \sum_{i=1}^{N^m} C_i^m$). It retrieves N^m data items, $O_1^m, \ldots, O_{N^m}^m$ with relative validity intervals $I_1^m, \ldots, I_{N^m}^m$ and retrieval durations $C_1^m, \ldots, C_{N^m}^m$, respectively. We find the optimal on-line retrieval order of the data items from decision tasks.

A single data item is retrieved only by a single decision task. In other words, no data items are shared across multiple decision tasks. The time that decision task m arrives is denoted by t^m, and the time that data item O_i^m starts being retrieved is denoted by t_i^m. The decision is made when all needed data items have been retrieved. We denote this time instant as F^m. For each decision task, all retrieved data items should be valid (*i.e.*, within their validity interval) at the time the decision is made, and the decision should be made by the decision deadline, that is:

$$\textbf{Data Validity: } t_i^m + I_i^m \geq F^m \ \ (\forall i, 1 \leq i \leq N^m), \tag{9.5}$$

$$\textbf{Decision deadline: } t^m + D^m \geq F^m. \tag{9.6}$$

These can also be represented together as:

$$\min\left(\min_{1 \leq i \leq N^m}(t_i^m + I_i^m),\ t^m + D^m\right) \geq F^m \tag{9.7}$$

The notation can be found in Fig. 9.2 and Table 9.3.

9.4.2 Overview

Our optimal algorithm for scheduling data item retrieval for multiple decision tasks is presented in Section 9.4.4. We first summarize the key properties of the optimal algorithm:

- A scheduling decision is made only when a decision task arrives or completes (*i.e.*, all data items for a task have been retrieved). By scheduling decision, we mean the decision to select a (new) task to retrieve data items for. Note, in particular, that this scheduling decision is not made upon retrieval of *each individual data item*. We prove this property in Lemma 12 in Section 9.4.3.
- Items pertaining to a single decision task are retrieved in a Least Volatile item First order. We prove this property in Theorem 10 in Section 9.4.5.

The first property (that scheduling decisions occur at task arrival/completion times) leads to a hierarchical scheduling algorithm in which any data retrieval for a lower priority task cannot proceed ahead of any data retrieval for a higher priority task. Accordingly, tasks are *prioritized* (at the task level, not the individual data item level). Once a task is chosen to execute, retrieval of its data items follows the LVF order. Next, we prove the above properties.

9.4.3 Decision Task Level Prioritization

We shall first prove that the optimal policy falls in the category of *hierarchical scheduling*, where a task is selected first (upon completion or arrival of some task), then items needed for the selected task are retrieved in some order. We call such a policy *per-decision prioritization*. This is in contrast to a policy where individual data items have retrieval priorities. Retrieval of items pertaining to different tasks is then globally ordered by item priority. We call the latter policy, *per-retrieval prioritization*.

The following lemma proves that if decision tasks meet their deadlines and validity constraints by some per-retrieval prioritization policy, they are always schedulable (*i.e.*, meet the same constraints) by per-decision prioritization as well. Hence, the optimal policy belongs to the latter category.

Lemma 12. *Per-decision prioritization is no worse than per-retrieval prioritization in terms of meeting both validity and deadline constraints.*

Proof. Suppose that a new decision task Y arrives while another decision task, X, is running. Then, we can consider two scenarios, which are also illustrated in Fig. 9.3:

- Per-retrieval prioritization: scheduling decisions are made at individual data item retrieval boundaries according to some global priority order of individual data items regardless of which task they pertain to. Therefore, some data retrievals for task X may be interleaved with data retrievals for task Y, as shown in Fig. 9.3 (a).

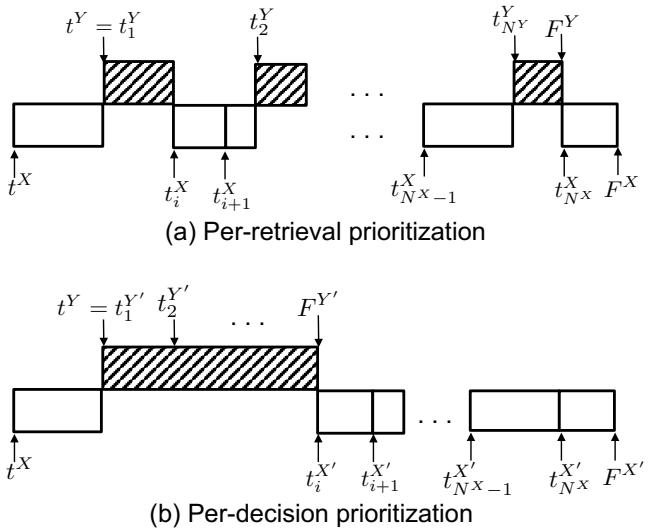


Figure 9.3: Scheduling decision occurs in decision task level not data retrieval level.

- Per-decision prioritization: scheduling decisions are made at the task level (when a task arrives or terminates). In between successive scheduling decisions, all retrieved data items are for the same task. An example where task X is preempted by task Y (which retrieves its data items until it completes) is shown in Fig. 9.3 (b).

In what follows, we show that if decision tasks meet both the deadline and validity constraints with per-retrieval prioritization, they can do so with per-decision prioritization. For notational simplicity, let us denote the two prioritization schemes as Case (a) and Case (b), respectively (see Fig. 9.3). Let F^Y and $F^{Y'}$ respectively denote the finish time of task Y in Case (a) and (b).

(i) Deadline constraint of task Y: Suppose task Y met its deadline in Case (a). Then, it is easy to see that it meets the deadline in Case (b) simply because the extra delay incurred due to retrieval of items for task X (that is interleaved with retrieval of items for task Y) is removed. Thus, $F^{Y'} \leq F^Y \leq t^Y + D^Y$.

(ii) Validity constraint of task Y: Let us denote the start time of retrieval of O_i^Y $(1 \leq i \leq N^Y)$ as t_i^Y and $t_i^{Y'}$ in Case (a) and (b), respectively. Then, we can see that

$$(F^{Y'} - t_i^{Y'}) \leq (F^Y - t_i^Y), \tag{9.8}$$

for all $1 \leq i \leq N^Y$, because again the delays incurred due to retrieval of items for task X disappear in Case (b) and thus the retrievals are "packed" (back to back). Now, suppose O_i^Y for all i met their validity constraints in Case (a):

$$F^Y \leq t_i^Y + I_i^Y. \tag{9.9}$$

By (9.8) and (9.9), the validity constraints in Case (b) are still met:

$$F^{Y'} \leq t_i^{Y'} + I_i^Y. \tag{9.10}$$

(iii) Deadline constraint of task X: Note that, the finish time of task X does not change as the total delay by task Y remains same. Hence, if task X met its deadline constraint in Case (a), it is still satisfied in Case (b).
(iv) Validity constraint of task X: This can be proved similar to (ii) – "packing" data retrievals helps meeting the validity constraints. Suppose O_i^X for all i met their validity constraints in Case (a):

$$F^X \leq t_i^X + I_i^X, \tag{9.11}$$

Now, because

$$(F^{X'} - t_i^{X'}) \leq (F^X - t_i^X),$$

$$F^{X'} \leq t_i^{X'} + I_i^X. \tag{9.12}$$

Hence, O_i^X for all i meet their validity constraints in Case (b).

Therefore, we can say that the per-decision prioritization (*i.e.*, Case (b)) is superior to the per-retrieval prioritization (*i.e.*, Case (a)) in terms of the scheduling optimality because any instance that can be feasibly schedulable by the per-retrieval prioritization can be still schedulable by the per-decision prioritization. □

With per-decision prioritization, scheduling is hierarchical. A task is selected upon arrival or completion of some (other) task. All items retrieved are for the same task, until a new one is selected. Hence, once a decision task starts retrieving its data items, it runs to completion or until a higher priority decision task arrives. In the next section, we discuss how to prioritize decision tasks.

9.4.4 Priorities of Decision Tasks

We showed above that scheduling decisions are made at the level of tasks. Thus, now the question is how to prioritize tasks. The optimal algorithm, as we prove in this section, assigns the highest priority to the task with *the smallest value of the minimum of its item validity expiration times and its decision deadline*. For this, let us first define the earliest expiration for decision task m:

$$\text{minExp}(m, t, \mathcal{S}).$$

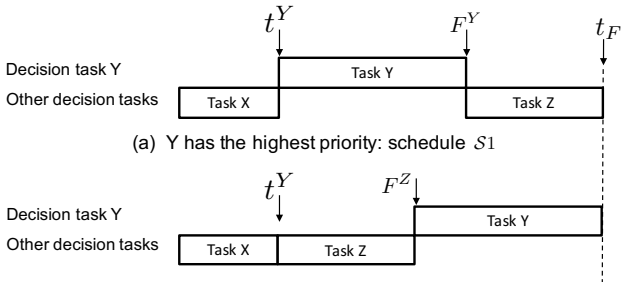


Figure 9.4: The optimal algorithms selects a task with the earliest deadline or expiration when a scheduling decision is needed.

It is defined as the earliest time that the validities of the data items (of task m) that have been retrieved until time t expire in a particular schedule $\mathcal{S}$, *i.e.*, $\min_{k=1}^{i}(t_k^m + I_k^m)$ for data items $O_1^m \ldots O_i^m$ with $t_k^m < t$. If no data item has been retrieved yet (*e.g.*, when a decision task has just arrived), it is defined to be infinity. In the following, we omit the parameter $\mathcal{S}$ when no ambiguity arises. Note that $\mathrm{minExp}(m,t)$ is monotonically decreasing with time, since the value is updated only to a smaller value as time proceeds. Hence we have,

$$\mathrm{minExp}(m,t) \geq \mathrm{minExp}(m,t^{'}) \text{ , if } t \leq t^{'}. \tag{9.13}$$

Now, suppose a scheduling decision is to be made among N decision tasks present at time t in the system. We shall show that the optimal scheduling algorithm selects the decision task with the smallest minimum of the validity expiration and the deadline, *i.e.*:

$$\min_{m=1,\ldots,N} \big(\mathrm{minExp}(m,t), t^m + D^m\big), \tag{9.14}$$

where t^m and D^m are the arrival time and the relative deadline of decision task m, respectively.

Theorem 9. *If a feasible order exists for a decision task set, the scheduling scheme that assigns highest priority to a task with a smaller minimum of item validity expiration times and deadline can always schedule the task set.*

Proof. Suppose a new task Y arrives at time t^Y while a set of $N-1$ decision tasks are present in the system. Suppose task X was running when task Y arrives. A decision needs to be made on whether to let Y execute first or not.

Suppose task Y has the smallest value of the minimum of the item validity expiration times and task deadline as of time t^Y:

$$\begin{aligned} &\min_{Z=1,\ldots,N-1}(\mathrm{minExp}(Z,t^Y), t^Z + D^Z) \\ &\quad \geq \min(\mathrm{minExp}(Y,t^Y), t^Y + D^Y). \end{aligned} \tag{9.15}$$

In what follows, we show that giving the highest priority to task Y is optimal, which is described in Fig. 9.4 (a). The opposite case is generalized by considering picking any other task (denoted by task Z) than Y, which is described in Fig. 9.4 (b). Note here that task Z can be either task X (that was running) or any one of the queued (suspended) tasks. Hence $t^Z \leq t^Y$ for all $Z = 1, \ldots, N-1$.

• **Case 1 – Select task Z:** Task Y is not chosen to execute first as illustrated in Fig. 9.4 (b) (schedule $\mathcal{S}2$). Suppose that task Y and task Z are schedulable. Let the finish time of task Y be t_F. Since task Y is schedulable,

$$t^Y + D^Y \geq t_F \tag{9.16}$$

$$\mathrm{minExp}(Y, t_F, \mathcal{S}2) \geq t_F. \tag{9.17}$$

Note that (9.16) is the deadline constraint and (9.17) is the data freshness (*i.e.*, validity) constraint.

Now, because as of t^Y, $\mathrm{minExp}(Y, t^Y) = \infty$ as task Y has just arrived and thus no data retrieval has been started. Hence, by (9.15),

$$t^Z + D^Z \geq t^Y + D^Y \tag{9.18}$$

for any task Z. That is, task Z's deadline is later than task Y's deadline. Also by (9.15), for the data items that have been retrieved until t^Y,

$$\mathrm{minExp}(Z, t^Y, \mathcal{S}2) \geq t^Y + D^Y. \tag{9.19}$$

That is, the validity interval of any data item of any task Z is later than task Y's deadline.

• **Case 2 – Select task Y:** task Y is chosen as shown in Fig. 9.4 (a) (schedule $\mathcal{S}1$). We will show that task Y and Z are schedulable by using the properties observed in Case 1.

First of all, it is easy to show that task Y is schedulable as it finishes earlier than t_F, the finish time of task Y in Case 1. This is simply because task Y is not delayed by any task Z. Let the finish time of task Y be $F^Y = t^Y + \sum_{k=1}^{N^Y} C_k^Y$. Then, by this and (9.16),

$$F^Y \leq t_F \leq t^Y + D^Y. \tag{9.20}$$

Note that, task Y's validity constraints are met in schedule $\mathcal{S}1$ if they were in schedule $\mathcal{S}2$ because all the data retrievals of task Y simply shift and validity intervals are relative to the beginning of retrieval. Therefore, task Y satisfies both the deadline and validity requirements.

Now, task Z is delayed by task Y in schedule $\mathcal{S}1$. Its finish time is t_F (as task Y and Z are swapped). Recall that

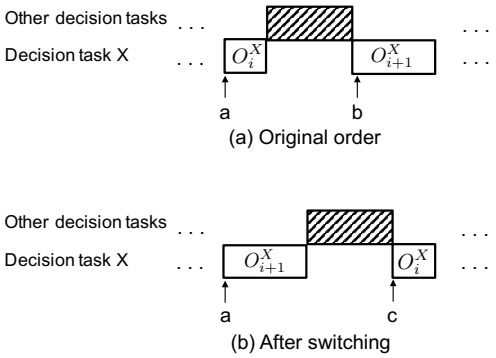


Figure 9.5: Even when multiple decision tasks are present, the scheduling order of data retrievals within a decision task still follows LVF.

task Z's deadline is later than task Y's and that the latter is later than t_F (see (9.18) and (9.16), respectively).

$$t_F \leq t^Y + D^Y \leq t^Z + D^Z. \tag{9.21}$$

Hence, task Z meets its deadline. Now, it becomes harder for the data items of task Z retrieved until t^Y to meet their validity constraints because of the delay by task Y. Nevertheless, they still meet the validity constraints due to (9.19) and (9.20):

$$\text{minExp}(Z, t^Y, \mathcal{S}1) = \text{minExp}(Z, t^Y, \mathcal{S}2) \geq t^Y + D^Y \geq t_F. \tag{9.22}$$

For the data items retrieved on or after t^Y, the validity constraints are still satisfied because the validity intervals are relative to the time instant at which they start being retrieved. Hence, task Z meets both the deadline and validity requirements in schedule $\mathcal{S}1$. Recall that task Z can be any task other than Y. Therefore, schedule $\mathcal{S}1$ is always schedulable if schedule $\mathcal{S}2$ is so.

Now, a similar reasoning can be applied to the opposite case of (9.15). Therefore, the scheduling policy that prioritizes the task by choosing the one with the smallest value of the minimum of its item validity expiration times and deadline is optimal. □

9.4.5 Order of Data Retrievals within a Decision Task

For the problem of scheduling data retrievals within a single decision task, explained in Section 9.3, we have seen that LVF is the optimal scheduling policy. We will show that this order is still optimal for retrieving items of each individual task, even when multiple tasks are present.

Theorem 10. *In the presence of multiple tasks, the Least Volatile item First (LVF) schedule is a feasible policy for retrieval of items of each task whenever a feasible order exists.*

Proof. Suppose that decision task X with N^X data retrievals is running and that a feasible order of the retrievals exists even when the task itself is preempted by other decision tasks. Suppose O_i^X and O_{i+1}^X $(1 \leq i \leq N^X - 1)$, in such an order that O_i^X precedes, and possible preemption by other higher-priority decision tasks in between them as shown in Fig. 9.5 (a).

Now, suppose $I_i^X < I_{i+1}^X$. That is, data item O_i^X has a shorter validity interval. Let a and b respectively denote the start time of retrieval of O_i^X and that of O_{i+1}^X. Then, since both satisfy the validity requirements,

$$a + I_i^X \geq F^X, \tag{9.23}$$

$$b + I_{i+1}^X \geq F^X, \tag{9.24}$$

where F^X is the finish time of the decision task.

Now, let us switch the order of O_i^X and O_{i+1}^X, and denote the start time of retrieval of O_i^X as c as shown in Fig. 9.5 (b). We see that $a < c$. Then, for validity requirements, we have

$$a + I_{i+1}^X \geq F^X \tag{9.25}$$

$$c + I_i^X \geq F^X. \tag{9.26}$$

By the initial assumption of $I_i^X < I_{i+1}^X$, (9.23) implies (9.25). Similarly, because $a < c$, (9.23) implies (9.26).

We can see that the resultant order (*i.e.*, let a data item with longer validity interval, O_{i+1}^X, be retrieved first) is still feasible. This is the Least Volatile item First schedule. Since this LVF schedule can be obtained from any ordering by repeating pairwise reordering as above, we prove the theorem. □

9.4.6 Summary of the Scheduling Algorithm

The scheduling algorithm for multiple decision tasks is summarized in Algorithms 4 and 5. Suppose a set of decision tasks is present in the system. A scheduling decision is made when (a) a task finishes or (b) a new task arrives, as explained in Section 9.4.3 and Section 9.4.4. Recall that a single data retrieval is non-preemptive, as explained in Section 9.3. Hence, when a new task arrives, a scheduling decision is postponed until the current retrieval completes.

As can be seen from Algorithm 4, the task-level prioritization is done by comparing a per-task value called SED_m (Lines 3,8,17). It holds the current value of $\min\left(\mathrm{minExp}(m,t), t^m + D^m\right)$ (*i.e.*, the least value of the minimum of item validity expiration times and task deadline, as described in Section 9.4.3). Instead of calculating it every time a comparison is needed, we can update the value only when a data retrieval begins, as shown at Line 2 in Algorithm 5.

Suppose a decision task is selected by the scheduler. Its data retrievals are scheduled according to the Least Volatile item First order by the LVF procedure in Algorithm 5. By Theorem 10, the LVF order is preserved even when the task is preempted by others.

Algorithm 4 The *optimal* algorithm, EDEF-LVF, for scheduling data item acquisitions with multiple decision tasks

t: current time
$\mathcal{R}$: ready queue of decision tasks
1: **if** a new decision task arrives or a task finishes **then**
2: **if** new task Y is arriving **then**
3: $\mathrm{SED}_Y \leftarrow t + D^m$
4: **if** no task is running **then**
5: LVF(task Y)
6: **else**
7: task X: the running task
8: **if** $\mathrm{SED}_X \leq \mathrm{SED}_Y$ **then**
9: Enqueue task Y to $\mathcal{R}$
10: LVF(task X) ▷ Continue
11: **else**
12: Enqueue task X to $\mathcal{R}$
13: LVF(task Y)
14: **end if**
15: **end if**
16: **else** ▷ A task is finishing
17: Task $X \leftarrow \mathrm{argmin}_{\text{Task } m \in \mathcal{R}}(\mathrm{SED}_m)$
18: Dequeue task X from $\mathcal{R}$
19: LVF(task X)
20: **end if**
21: **end if**

Algorithm 5 LVF(task m)

$O^m_{(1)}, \ldots, O^m_{(N^m)}$: data items sorted in decreasing order of I^m_i
k: number of items that have been retrieved so far
1: Retrieve $O^m_{(k+1)}$ if $k < N^m$
2: $\mathrm{SED}_m \leftarrow \min(\mathrm{SED}_m, t^m_k + I^m_k)$
3: $k \leftarrow k + 1$
4: Repeats until $k == N^m$ or a new decision task arrives

The complexity of the algorithm is $\mathcal{O}(N)$ where N is the number of decision tasks present in the system. This is due to searching for the decision task that has the smallest value of SED_m at Line 17 in Algorithm 4 (assuming a linear search).

We call the optimal algorithm EDEF-LVF which stands for ***E**arliest **D**eadline or **E**xpiration **F**irst - **L**east **V**olatile **F**irst*. This represents that the optimal algorithm schedules first a task which has the earliest (smallest minimum of) deadline and expiration, and then the retrieval of data items for each task is scheduled by LVF. We show EDEF-LVF is the optimal algorithm for scheduling of data item acquisition for multiple decision tasks in Theorem 11.

Theorem 11. EDEF-LVF *is the optimal algorithm for scheduling of data item acquisitions for multiple decision tasks (with no data overlap).*

Proof. Since (i) assigning the highest priority to a task with the smallest minimum of deadline and expiration is optimal for task prioritization by Theorem 9, and (ii) scheduling retrieval of data items according to LVF for each task

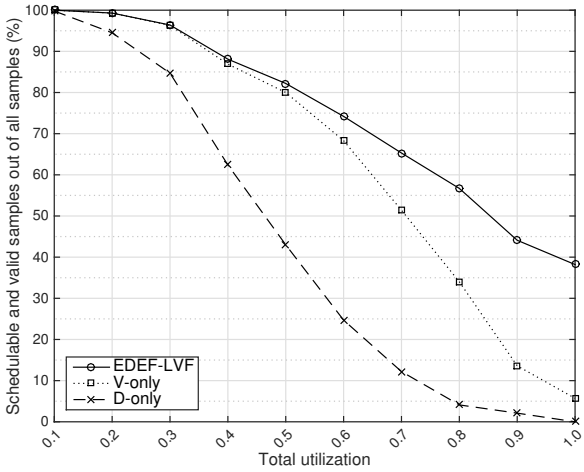


Figure 9.6: The ratio of schedulable and valid sample count out of all samples by EDEF-LVF, D-ONLY, and V-ONLY.

is optimal by Theorem 10, EDEF-LVF is optimal. Therefore, the theorem follows. □

9.4.7 Evaluation

In this section, we evaluate EDEF-LVF by comparing against various heuristics. The objective is to demonstrate that EDEF-LVF is superior to the heuristics in terms of meeting deadlines and validity constraints. The algorithm and heuristics we compare are as follows:

• EDEF-LVF: This is our optimal algorithm that assigns the highest priority to the task that has the smallest value of the minimum of its data items' validity expirations and task deadline, and it schedules data item retrievals for each task in LVF order.

• D-ONLY: This is a heuristic algorithm that assigns the highest priority to the task that has the smallest value of task deadline, then it schedules data item retrievals for each task in LVF order.

• V-ONLY: This is a heuristic algorithm that assigns the highest priority to the task that has the smallest value of data items' expirations, then it schedules data item retrievals for each task in LVF order.

• RATEFIRST: Inspired by Rate Monotonic Scheduling (RMS) [Liu and Layland, 1973], it assigns higher priority to a decision task with higher arrival rate. Once a task is determined to be scheduled, the retrieval of its data items is done by LVF.

• RTDB: This is a model adapted from real-time database systems. The details are in Section 9.4.7.

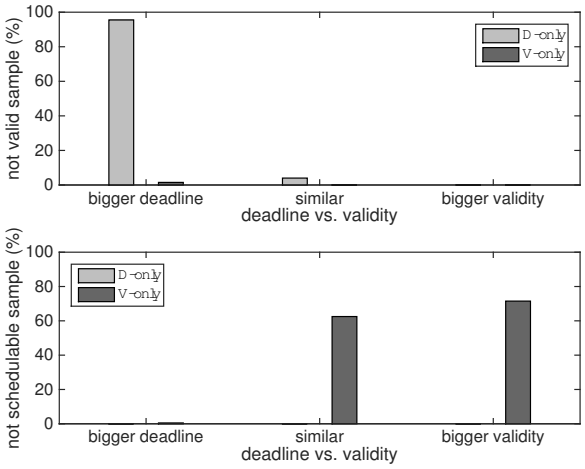


Figure 9.7: Impact of relative value of deadline and validity interval on the ratio of invalid samples (top) and unschedulable samples (bottom) when scheduled by D-ONLY and V-ONLY.

We define the total utilization as

$$\sum_{m} \frac{\sum_{1 \le i \le N^m} C_i^m}{D^m}.$$

If utilization is larger than 1.0 for a decision task set, it could be unschedulable depending on the tasks' arrival pattern. Since we generate random samples, some task sets could already be infeasible in the first place.

We generated 10,000 synthetic samples evenly from ten utilization groups, (0-0.1], (0.1-0.2], . . ., (0.9-1.0]. (On the x axis, in the graphs, only the upper limit of each interval is shown.) Each sample can have up to 100 data items and 10 decision tasks. Each deadline of a decision task is randomly drawn from [10-100]. The validity interval and the retrieval time of each data item are randomly drawn from [5-50] and [1-10], respectively. Decision tasks' inter-arrival times follow a Poisson process with arrival rate of 0.04.

EDEF-LVF vs. D-ONLY vs. V-ONLY

Fig. 9.6 plots the ratio of schedulable and valid sample count out of all samples, when scheduled by EDEF-LVF, D-ONLY, and V-ONLY. Remember that each sample represents a different task set. A sample is schedulable and valid if the policy finds a schedule that meets both deadlines and data validity constraints. First of all, the results show that fewer samples are valid and schedulable as the total utilization increases. As expected, EDEF-LVF, which considers both the deadline and the validity constraints, is superior to D-ONLY and V-ONLY. The comparison between D-ONLY and V-ONLY depends on the relative values of deadlines and validity constraints. For example, if deadlines are very long while validity tends to be short, the schedule will probably meet the deadline constraints relatively easily, while

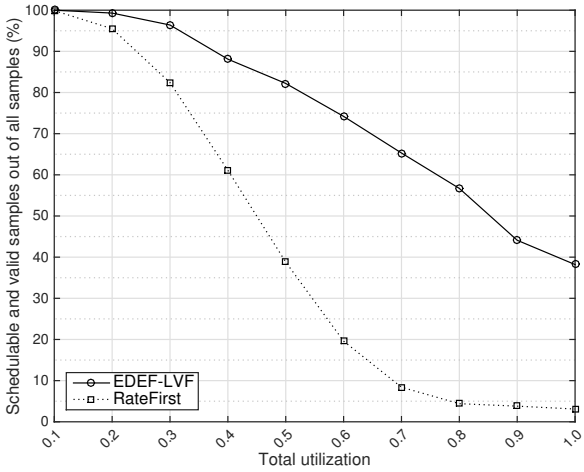


Figure 9.8: EDEF-LVF vs. RATEFIRST: the ratio of schedulable and valid sample count out of all samples for different total utilization.

it will unlikely meet the validity constraints.

Hence, we experimented to show the impact of the relative values of deadlines and validity intervals. The results in Fig. 9.7 show the impact on the ratio of invalid (top) and unschedulable (bottom) sets, when scheduled by D-ONLY and V-ONLY. In the case of 'bigger deadlines', we generate deadlines in the range [40-60], and validity constraints in the range [5-20]. For the case of 'bigger validity' intervals, we generate deadlines in the range [5-20], and validity intervals in the range [40-60]. In the case of 'similar' deadlines and validity constraints, both deadlines and validity intervals are in the [20-40] range. The arrival rate is set to 0.05 in this experiment, and the total utilization is maintained between 0.6 and 1.0. The results in Fig. 9.7 underscore the need for considering both deadlines and validity constraints. For example, when the deadlines are long compared to the validity intervals (hence, it is harder to meet the validity constraints), D-ONLY can easily violate the validity constraints (top-left). The same reasoning is applied to the opposite case.

EDEF-LVF vs. RATEFIRST

One can think of a heuristic that gives higher priority to tasks with higher arrival rates (which is similar with Rate Monotonic Scheduling). Hence, we compare our algorithm with this heuristic called RATEFIRST. For this experiment, we statically assign arrival rate to each task and then assign higher priority to a task with a higher arrival rate.[2] The result in Fig. 9.8 shows that EDEF-LVF outperforms RATEFIRST by far. This is because RATEFIRST does not consider the deadline and validity constraints. This result underscores again the need for prioritization based on the two requirements.

[2]This does not mean that the inter-arrival times are fixed. The arrivals of decision tasks still follow the Poisson process with the specified arrival rate.

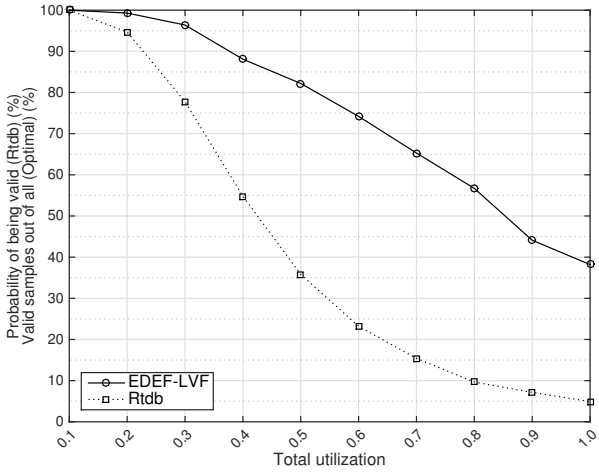


Figure 9.9: The ratio of valid samples out of all for EDEF-LVF and the average probability of meeting validity constraints for RTDB.

Comparison with the Real-Time Database Model, RTDB

A direct comparison of our algorithm with the approaches used in real-time database literature is not trivial because of differences in models and assumptions – namely, data items are passively updated periodically in real-time databases. Hence, we performed an indirect comparison as explained in what follows.

First of all, the Database Freshness (also called Quality of Data, QoD) is defined as the ratio of fresh data to the entire temporal data in a database [Kang et al., 2002b, Kang et al., 2004, Adelberg et al., 1995]. For this, we define `DbFreshness` that represents the average probability that data is fresh when a decision task accesses it. For M decision tasks, the average probability that they use fresh data is

$$\sum_{m=1}^{M} \frac{\texttt{DbFreshness}^{N^m}}{M}. \tag{9.27}$$

Now, suppose a data item whose validity interval is I_i^m. If it is updated every $P_i^m = I_i^m$, it is fresh whenever accessed. For all data in the system to be fresh anytime, they should be updated with the periods that are same as their validity intervals. Now, the system utilization required to achieve a complete freshness is

$$\sum_{m=1}^{M} \sum_{i=1}^{N^m} \frac{C_i^m}{P_i^m}.$$

If it is not greater than 1, we can have such sensor update transactions that ensure freshness. If it is over 1, say α, the periods should be scaled as much as α (in order to make the system utilization 1). Then, the scaled period is now $\alpha \cdot P_i^m$, and accordingly the `DbFreshness` is $\frac{1}{\alpha}$ because of the longer update period. That is, the probability of

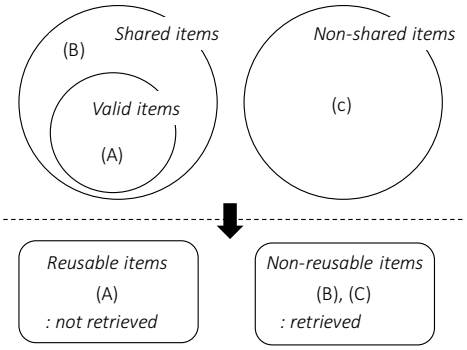


Figure 9.10: Shared items and reusable items

using fresh data is no longer 1, and is rather determined by the validity intervals.

We call this method RTDB and compare it with EDEF-LVF in Figure 9.9. [3] For EDEF-LVF, the y-axis is the ratio of valid samples to all samples. For RTDB, we use (9.27), the average probability that data are valid when accessed at arbitrary time, as explained above. The result is that active data retrievals (as done by our decision-task model) can ensure data freshness more easily than passive updates (as done in real-time databases).

9.5 Inter-Decision Data Dependency

In the previous sections of this chapter, we assumed that data objects are not shared among different decision tasks. In this section, we relax the assumption - we consider dependent data items that are shared by multiple decision tasks. Once we assume such a model, the $\texttt{Cost}_{\texttt{opt}}$ is not the optimal cost any more. That is because, if an item is shared by multiple tasks, there is a possibility that the same data item can be reused by multiple tasks. Then the reusing (retrieved data items) can reduce the cost. In specific, if a data item is shared by other tasks as well, and if the item has been already retrieved by one of the other tasks and the item is still valid, it is eligible for a task to make a decision without retrieving the item again as regarding to reuse the item. Hence, non-shared items are not reusable at all thus always should be retrieved. Among shared items, some items are reusable while the others are not depending on the validity (see Figure 9.10).

Suppose an item of task m, O_i^m, and another item of task n, O_j^n, is the same data item, that is, task m and task n share the same data item O_i^m $(= O_j^n)$. Let r_i^m be a flag indicating data item O_i^m is retrieved or not. If the item is retrieved $r_i^m = 1$ otherwise, $r_i^m = 0$.

$$r_i^m = \begin{cases} 1, & \text{if } O_i^m \text{ is retrieved,} \\ 0, & \text{otherwise} \end{cases}$$

If O_i^m has been already retrieved, *i.e.*, $r_i^m = 1$, and later if task n is determined to reuse the item, $O_j^n = O_i^m$, $r_j^n = 0$,

[3] Figure 9.9 is shown separately from Figure 9.8 since RATEFIRST is irrelevant to Rtdb.

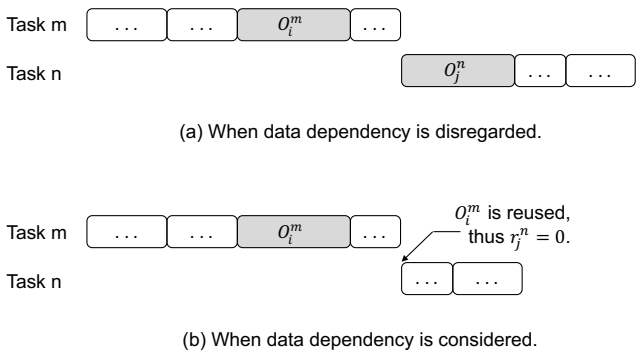


Figure 9.11: Cases when data dependency is regarded and not.

as shown in Figure 9.11. In Section 9.5.2, how to determine whether to reuse a shared item or not is explained.

9.5.1 Problem Description

Here are the assumptions we make for the problem: (i) We categorize the problem as *online* scheduling in which the upcoming tasks' arrival patterns and their workload are not predictable. (ii) Decision tasks are *not* periodic but sporadic. (iii) We avoid any re-collecting (second samples) of a data item in order to minimize cost. (iv) Parameters of data validity intervals, data processing time for retrieval and decision deadlines are given. (v) Retrieval of a single data item is non-preemptible. Hence, if a decision task arrives in the middle of retrieving a data item of another task, the scheduling decision is made when the current retrieval is complete. (vi) If a data item is shared by other tasks as well, and if the item has been already retrieved by one of the other tasks and the item is still valid, it is eligible for a task to make a decision without retrieving the item again as regarding to reuse the item.

The problem is to schedule pertinent data item acquisitions in the presence of multiple decision tasks and shared data items across tasks as minimizing the net $\texttt{Cost}_{\texttt{dep}}$ as follows:

$$\textit{minimize} \quad \texttt{Cost}_{\texttt{dep}} = \sum_{1 \le i \le N^m} C_i^m \cdot r_i^m. \tag{9.28}$$

9.5.2 Solution Algorithms

Heuristic Algorithm: DBAR

This algorithm basically assigns the highest priority to the task that has the smallest value of the minimum of its (already retrieved) data items' validity expirations and task absolute deadline (EDEF), and once a task is scheduled, it schedules data item retrievals for each task in LVF order.

When a shared data item is requested to be retrieved by a decision task m, an algorithm checks if the requested item has already been retrieved and a data item would not expire until task m's decision is made. However, it is costly

and not easy to absolutely predict all future data items will be reused or retrieved, and the exact finish time when a decision is made. One heuristic is to check if any validity expiration is later than task m's absolute deadline. This is relatively simple since a task's absolute deadline is known as soon as a task arrives. Then, its feasibility is simply guaranteed by (9.6) if only (9.6) is met. That is,

$$\text{if } F^m \leq t^m + D^m,$$
$$\text{and } t^m + D^m \leq Exp_i^m,$$
$$\text{then } F^m \leq Exp_i^m,$$

where the decision task arrives at time t^m and expiration of O_i^m is denoted as Exp_i^m. When a data item is shared across multiple tasks, a data item can be reusable - if a data item is shared and is still valid. Otherwise, the others are not reusable - non-shared items and shared but not valid items are not reusable. Accordingly, once a data item is determined to reusable, the data is not retrieved, $r_x^h = 0$, otherwise, $r_x^h = 1$.

For a shared item to be reusable for future tasks, the expiration of a shared item needs to be updated when a shared item is retrieved. For that purpose, we keep a list of SharedQ which records the expiration of the latest retrieval of each shared data item. If a shared item, O_i^m, is retrieved at t_i^m and its validity is I_i^m, $Exp_i^m = t_i^m + I_i^m$ and the value of Exp_i^m is updated to the corresponding list of SharedQ.

Heuristic Algorithm: FBAR

This algorithm schedules tasks according to EDEF, and once a task is scheduled, it retrieves data items of each task according to LVF order. When a shared data item is requested by a decision task m, this algorithm, let us call FBAR, conservatively estimates the finish time as if all future data items will be retrieved. This provides a sufficiently *later* finish time than an actual finish time. That is because, in an actual execution, if some of the data items are reused so not retrieved, the actual finish time should be earlier than or equal to the estimated one. Let us denote the estimated finish time which is sufficiently late as F'^m. Also, here we define readyQ as,

- readyQ: a queue for a set of tasks that have arrived or have been preempted thus are ready to execute.

Then,

$$F'^m = t^m + \sum_{\forall O_x^h} C_x^h \cdot 1$$

where O_x^h is a data item of which task is in readyQ and of which task's priority is higher than task m. However, an actual finish time, F'^m, is

$$F'^m = t^m + \sum_{\forall O_x^h} C_x^h \cdot r_x^h.$$

Then $F^m \leq F'^m$. This algorithm checks if

$$F'^m \leq Exp_i^m. \tag{9.29}$$

$$\text{then } F^m \leq Exp_i^m,$$

$$\text{since } F^m \leq F'^m.$$

Hence, if it satisfies (9.29), those shared items automatically satisfy validity constraints.

Minimum $\texttt{Cost}_{\texttt{dep}}$ Algorithm

The algorithm schedules tasks according to EDEF, and once a task is selected to execute, it follows the order of retrievals for data items of each task in LVF order. To decide whether a data item will be retrieved or not, this algorithm exactly estimates the finish time of a decision task by testing all options of whether all data items (of which tasks are in readyQ) are retrieved or not, *i.e.* $r_i^m = 1$ or 0. Then selects an option which provides the minimum value of,

$$\sum_{\forall m} \sum_{1 \leq i \leq N^m} C_i^m \cdot r_i^m, \tag{9.30}$$

where all task m are in readyQ. However, when assigning such r_i^m for every item, the all data items and tasks must satisfy validity and deadline constraints, (9.5) and (9.6). That is tested by comparing an accurately estimated finish time of task m, F_{est}^m, with deadline and estimated expirations, ${Exp_{est}}_i^m$. F_{est}^m is estimated as follows:

$$F_{est}^m = t^m + \sum_{\forall h} \sum_{1 \leq i \leq N^h} C_i^h \cdot r_i^h + \sum_{1 \leq i \leq N^m} C_i^m \cdot r_i^m.$$

where task h is higher priority tasks than task m in readyQ. And similarly, ${Exp_{est}}_i^m$ is estimated as,

$${Exp_{est}}_i^m = t^m + \sum_{\forall h} \sum_{1 \leq i \leq N^h} C_i^h \cdot r_i^h + \sum_{1 \leq h \leq i-1} C_h^m \cdot r_h^m.$$

where O_h^m has higher priority than O_i^m (in LVF order). Thus, if

$$F_{est}^m \leq {Exp_{est}}_i^m, \text{ and}$$

$$F_{est}^m \leq t^m + D^m$$

are satisfied, the solution is qualified to be fielded. Then, again, among those eligible solutions, the one which reveals the minimum value of (9.30). We call this algorithm as MINCOST.

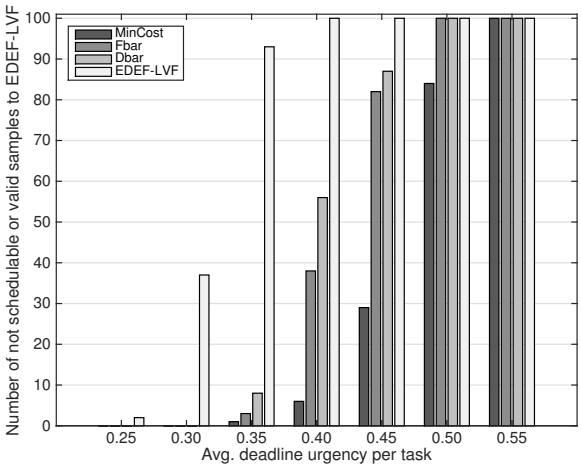


Figure 9.12: Not Schedulable or not valid samples according to average `deadline urgency` per task

Spent Resource

Suppose a scheduling algorithm disregards dependency of data items and thus retrieves every data item, but still meets both validity and deadline constraints. On the other hand, suppose other scheduling algorithm, such as DBAR, FBAR or MINCOST, reuses some eligible shared data items, and meets both validity and deadline constraints. However, those two algorithms should not be considered to show the same performance. That is because the latter saves more communication resource. To count usage or saving of the communication resource, we define `Spent resource` as

- $\texttt{Spent resource} = \dfrac{\text{executed time units}}{\text{total time units}}$.

That is, `Spent resource` counts the time units which are not idle throughout the entire elapsed time. Thus, it is a metric to show the overall usage of communication resource.

9.5.3 Evaluation

In this section, we evaluate various algorithmic options which assume dependent data items and compare them against an algorithm which assumes independent data items. The objective is to demonstrate that, once data dependency is designated, reusing the same, already-retrieved (by a preceding task) and still-valid data item helps a decision task (i) save more (communication) resource utilization and thus (ii) more probably meet deadline and validity constraints. The algorithm and heuristics are as follows:

[Not considering data dependency]

- EDEF-LVF: this is an optimal algorithm when data items are assumed to be *independent* across tasks. Thus

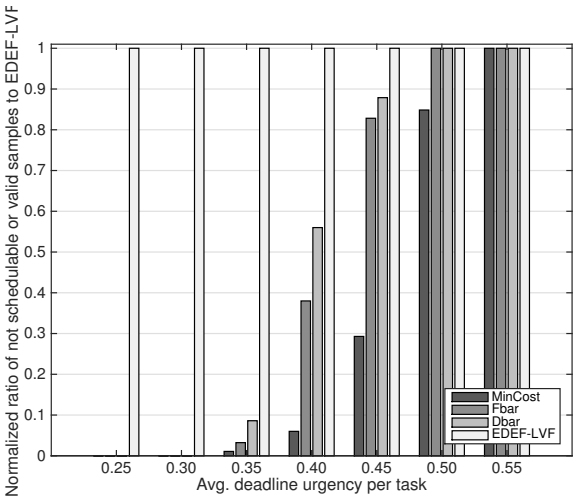


Figure 9.13: Normalized (to EDEF-LVF) ratio of not Schedulable or not valid samples according to average `deadline urgency` per task

every data item is retrieved at each time. This algorithm assigns the highest priority to the task that has the smallest value of the minimum of its data items' validity expirations and task absolute deadline, and it schedules data item retrievals for each task in LVF order.

[Considering data dependency]

- DBAR: this is an algorithm prioritizes tasks according to EDEF and once a task is scheduled, its belonging data items are basically scheduled by LVF. However, when a shared data item is requested, it checks if the requested data item's expiration is later than its task's absolute deadline. If it is, the item is not retrieved, otherwise, the item is retrieved.

- FBAR: this is an algorithm prioritizes tasks according to EDEF and once a task is scheduled, its belonging data items are basically scheduled by LVF. However, when a shared data item is requested, it checks if the requested data item's expiration is later than its task's estimated finish time. The estimated finish time is calculated as if all data items of which tasks are in readyQ are all retrieved. Then if the data item's expiration is later than the estimated finish time, the item is not retrieved, otherwise the item is retrieved.

- MINCOST: before tasks going to be executing, it previously determines whether each shared data item could be reused or not. Upon a task's arrival, this algorithm considers all combinations of being reused/not for every data item. Then it selects a combination which provides the minimum $\texttt{Cost}_{\texttt{dep}}$ for all tasks in readyQ.

We define average urgency for task j as,

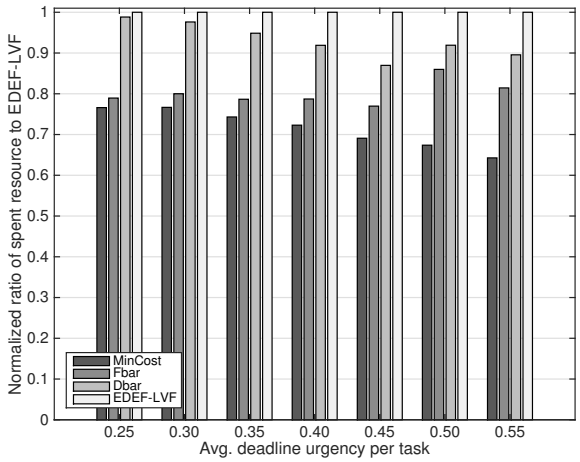


Figure 9.14: Normalized `Spent resource` to EDEF-LVF according to average `deadline urgency` per task

$$\bullet \texttt{ deadline urgency } = \frac{\sum_{i=1}^{N^m} C_i^m}{D_m}.$$

This shows how much the sum of the processing of retrievals is close to the deadline.

In addition, we define `Shared load ratio` as follows:

$$\bullet \texttt{ Shared load ratio} = \frac{\sum_m \left(\sum_{\substack{1 \le i \le N^m \\ O_i^m \text{ is shared}}} \frac{C_i^m}{I_i^m} \right)}{\sum_m \left(\sum_{1 \le i \le N^m} \frac{C_i^m}{I_i^m} \right)}.$$

This measure shows the ratio of shared load (relative to validity) over the total. And again, `Spent resource` is defined as

$$\bullet \texttt{ Spent resource} = \frac{\text{executed time units}}{\text{total time units}}.$$

Note that, at each running, the arrival pattern thus the combination of all running and active tasks are different even in the same sample. Hence, even though just a few samples are less successful in one approach to the other, the result should not be read as saying the one is inferior to the other, and vice versa. More dominant overall trend should be weighed more.

We generated 900 synthetic samples evenly from 9 'shared load / total load' groups, (0-0.1], (0.1-0.2], (0.2-0.3], (0.3-0.4], (0.4-0.5], (0.5-0.6], (0.6-0.7], (0.7-0.8], (0.8-0.9]. In addition to that, 700 synthetic samples are generated evenly from 7 `deadline urgency` groups, (0.2-0.25], (0.25-0.3], (0.3-0.35], (0.35-0.4], (0.4-0.45], (0.45-0.5], (0.5-0.55]. (On the x axis, in the graphs, only the upper limit of each interval is shown.) Each sample can have up

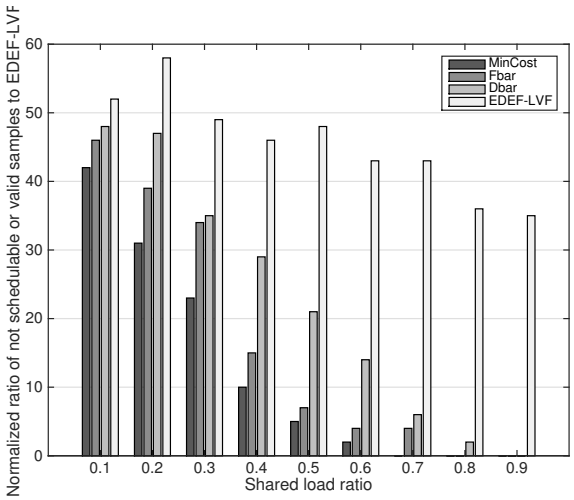


Figure 9.15: Not Schedulable or not valid samples according to `shared load` ratio

to 50 data items and 10 decision tasks. Each deadline of a decision task and the validity interval of each data item are randomly drawn from [15-50]. The retrieval time of each data item is randomly drawn from [1-10]. We ran all experiments for 100,000 time ticks. Each decision task's inter-arrival time follows a Poisson process with arrival rate 0.02. If a generated inter-arrival time is less than the deadline, the invocation is discarded.

Figure 9.12 shows the number of samples that are not schedulable or valid according to average `deadline urgency` per task while Figure 9.13 shows normalized values of the data in Figure 9.12 against EDEF-LVF. We vary average `deadline urgency` per task but fix `shared load ratio` to be 0.7-0.9. If any task in the sample task set fails to meet validity and deadline constraints, the task set is considered to be fail, and it is counted in the result. The values are normalized to the value of EDEF-LVF. Basically MINCOST shows the best performance while EDEF-LVF does the least. Although DBAR and FBAR are in between, but FBAR outperforms DBAR. One reason is that, to decide whether to retrieve a shared data item, FBAR checks if $F'^m \leq Exp_i^m$ while DBAR checks if $t^m + D^m \leq Exp_i^m$. If it was the same task set and arrival pattern, FBAR could check it with a probably tighter bound than DBAR since $F'^m \leq t^m + D^m$ as long as task m meets the deadline constraint. `Deadline urgency` under 3.0, all algorithmic options except MINCOST can schedule all tasks in all task sets. Beyond 0.55 of `deadline urgency`, all samples are not schedulable or valid - at least one task fails in every sample set.

Figure 9.14 shows `Spent resource` of the algorithmic options according to average `deadline urgency` per task. The values are normalized to the value of EDEF-LVF. The performance order is same with the one in Figure 9.13. One point to observe is that as `deadline urgency` gets higher the performance gap between MINCOST and EDEF-LVF gets larger. That is, as `deadline urgency` gets higher, MINCOST takes more benefits by reusing already retrieved shared data items.

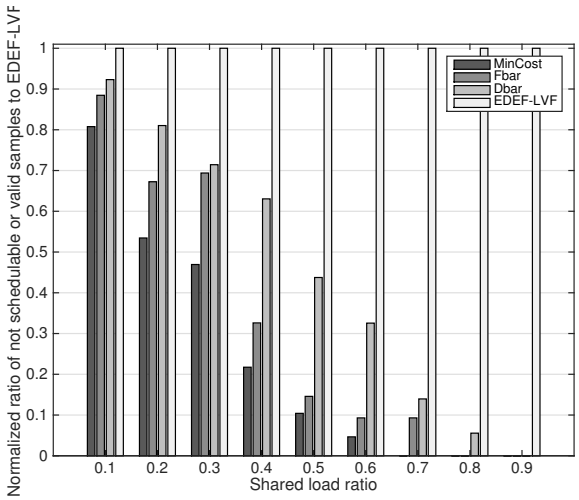


Figure 9.16: Normalized (to EDEF-LVF) ratio of not Schedulable or not valid samples according to `shared load` ratio

Figure 9.16 shows the number of samples that are not schedulable or valid according to `shared load ratio` and Figure 9.16 shows the same data but what is normalized to the values of EDEF-LVF. In fact, no matter how much data items are shared that does not impact on EDEF-LVF since EDEF-LVF disregards the data dependency. Hence, the trend of EDEF-LVF along with x-axis does not mean anything. For that reason, it is more informative and easier to look at Figure 9.16 to compare the performance difference between the algorithmic options according to `Shared load ratio`. `Shared load ratio` is varied while the average of `deadline urgency` per task is fixed at 0.2-0.3. The values are normalized to the value of EDEF-LVF. As `shared load ratio` is getting higher, much more samples are schedulable and valid in the proposed algorithms not EDEF-LVF. That is, as more data items are shared, the algorithms reusing available shared items get benefits from that. In cases that `shared load ratio` is over 0.8, all algorithms consider data dependency across tasks can schedule all tasks in all task sets.

Figure 9.17 shows `spent resource` of the algorithmic options according to `shared load ratio`. The values are normalized to the value of EDEF-LVF. As `shared load ratio` gets higher the gap between MINCOST and EDEF-LVF gets larger. That is, as `shared load ratio` gets higher, MINCOST takes more benefits by reusing already retrieved shared data items.

The results show the importance of considering data dependency across tasks, which is close to a practical model and also fits better in such a disaster response scenario which is a resource-poor situation.

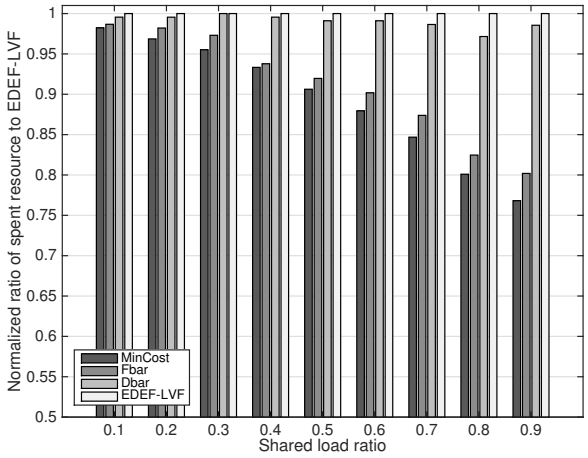


Figure 9.17: Normalized `Spent resource` to EDEF-LVF according to `shared load` ratio

Chapter 10

Related Work

The idea of data freshness was firstly introduced in [Ramamritham, 1993] and then later appeared in a stream of real-time database literature [Adelberg et al., 1995, Adelberg et al., 1996, Lee et al., 1996, Kang et al., 2002b, Kang et al., 2002a, Kao et al., 2003, Gustafsson and Hansson, 2004a, Gustafsson and Hansson, 2004b, Xiong and Ramamritham, 2004, Kang et al., 2004, Xiong et al., 2005, Xiong et al., 2008]. In [Adelberg et al., 1995], it was shown that freshness and timeliness requirements can conflict each other. In real-time databases, data items are periodically updated at predefined time instants (in [Adelberg et al., 1995, Adelberg et al., 1996, Gustafsson and Hansson, 2004a, Gustafsson and Hansson, 2004b, Xiong and Ramamritham, 2004] aperiodic updates are also discussed), from which their freshness start decreasing no matter when they are used [Song and Liu, 1995, Adelberg et al., 1996, Kang et al., 2002a, Kang et al., 2004]. In real-time database literature, normally-off sensors did not receive much attention. Some consideration is found in [Adelberg et al., 1995], where two queueing schemes are discussed – LIFO (Last-In, First-Out) and FIFO (First-In, First-Out). LIFO is found to be better than FIFO in that FIFO installs the oldest updates first, which are close to expiration. On the other hand, FIFO is better than LIFO in that updates arriving earlier should wait for a significant amount of time and thus can easily become stale.

While most real-time database papers assumed that data items are periodically updated at predefined time instants, aperiodic data updates were also discussed [Adelberg et al., 1995, Adelberg et al., 1996, Gustafsson and Hansson, 2004a, Gustafsson and Hansson, 2004b, Xiong and Ramamritham, 2004]. A key common assumption in real-time database literature was that data access and update times are not linked together. Freshness decreased from the time a data object was updated, no matter when the data items were used [Song and Liu, 1995, Adelberg et al., 1996, Kang et al., 2002a, Kang et al., 2004]. In contrast, in our on-demand model, data access and update tims are linked in that sensors are activated by the data access itself, resulting in a novel coupling that removes a degree of freedom available in prior work (where access and update transactions could be scheduled separately). This new coupling makes it harder to simultaneously satisfy the conflicting constraints.

Besides that, in much real-time database work, data validity was further defined summarily for the entire system. An example metric was the percentage of objects that are fresh at a given time. Most of the exsiting work is broken into two general categories; (i) deadline and period assignment for periodic sensor transactions [Xiong and Ramamritham,

2004, Xiong et al., 2005, Xiong et al., 2008], and (ii) scheduling of sensor and user transactions [Kao et al., 2003, Kang et al., 2004]. Dynamic feedback-based schemes were also proposed: offering dynamic adjustments to sensor update periods [Kang et al., 2002b, Kang et al., 2004], or feedback-based miss ratio control - the authors in [Kang et al., 2004] proposed QMF which is a real-time main memory database architecture that applies a feedback-based miss ratio control. In [Gustafsson and Hansson, 2004a], an algorithm (ODTB) is proposed, which skips unnecessary data item updates and thus allows for better utilization of the CPU. In addition, data items in the form of a directed acyclic graph were considered in [Gustafsson and Hansson, 2004b].

As addressed in [Adelberg et al., 1995], data freshness and timeliness can conflict with each other. Hence, there are trade-offs between them as discussed in [Cipar et al., 2012, Qu and Labrinidis, 2007, Röhm et al., 2002]. The high-level idea of FAS (Freshness-Aware Scheduling) in [Röhm et al., 2002] is similar to V-ONLY, which is used in our evaluation. Trade-offs between response times and quality of data have been investigated [Qu and Labrinidis, 2007]. However, the work did not consider time constraints (deadlines) of user requests (similar to decision tasks). Recently, in [Hu et al., 2015], a similar problem to ours was considered. In that model, a decision has a deadline, and the data items have freshness requirements. The objective of a task is to decide on the best course of action among multiple options, where analysis of each course of action entails retrieval of some data items. However, they considered only a single decision task but did not consider multiple decision tasks. Also, they did not consider multiple quality levels of a data object. Note that, it is easy to find the optimal policy for retrieval of data objects pertaining to a single decision task. This is because all such objects share the same decision deadline. The analysis for multiple decision tasks is not trivial because it is not immediately obvious how to optimally combine the different data validity requirements and *different* task deadlines. This is because to meet deadline requirements, scheduling objects as early as possible helps the schedulability while to meet validity requirements, scheduling objects as late as possible (by LVF) helps their freshness. An optimal scheduling policy for multiple decision tasks is therefore our main contribution.

Chapter 11

Conclusion

In cyber-physical systems, timing analysis is significant for ensuring functional correctness and the safety of the entire system. Moreover, for safety-critical systems, one cannot stress enough the importance of timing correctness. The research in this dissertation has been motivated by industry trends and the pressing need to transition from single-core platforms to multicore ones. This need stems from the fact that safety-critical systems have been certified only on single-core chips, while single-core chips are disappearing from the market due to the superiority of multicore chips in cost, size and performance. With this motivation, we proposed a scheduling mechanism for conflict-free I/O on multicore systems. Besides that, we considered the issue that, in the course of the transition and migration platform-to-platform, the tasks' exact (worst-case) execution times are not known or verified. In addition, we tackled a priority inversion issue in existing resource partitioning (hierarchical scheduling) mechanisms. In order to resolve and support these two issues, we developed a new scheduling paradigm that can test the system's schedulability with no task execution time information while globally scheduling tasks across cores irrespective of their assignments to applications. Then, we included conflict-free I/O in this novel framework by considering I/O transactions as the highest priority in the schedulability analysis. By synchronizing I/O sections globally so as to be conflict-free across cores, I/O-level isolation was achieved between cores. Once the existence of a global I/O schedule was assured, periodic tasks were scheduled on each core subject to application budgets that provide temporal modularity. As a result, our schedulability test and task execution model support reducing the cost of migrating software from single-core to multicore systems, improve upon classical schedulability bounds by taking advantage of known information on task periods and deadlines, and show considerable advantage compared to resource partitioning approaches.

Interestingly, but not surprisingly, emerging cyber-physical systems such as Internet-of-Things and Smart City systems also have significant needs for timing analysis. We considered a disaster response infrastructure in a Smart City environment where large numbers of heterogeneous sensing devices are deployed. In such an environment, since the network is supposed to be resource-poor, the sensing devices are desired to be turned off by default, and information is fed only when it is requested. We proposed this "normally-off sensors" model, and developed the optimal algorithm for the order of collecting data items that have validity requirements, to support time-sensitive decision making. Also, we evaluated new heuristics solving more general versions of the problem - when data items

have dependency across multiple decisions. In addition, we developed multiple heuristic options for maximizing quality of data items when they have multiple quality levels.

As mentioned earlier, in traditional real-time systems, an instance of a program execution (a process) is described as a scheduling entity, while, in the emerging applications, the fundamental schedulable units are chunks of data transported over communication media. Another transformation is that, in IoT and Smart City applications, there are multiple options and combinations to utilize and schedule since there are massive numbers of deployed heterogeneous sensing devices. This is contrary to the situation with existing real-time systems, which have a fixed task set to be analyzed. For those differences, we are seeing an exciting transformation and new opportunities. That is, the emerging cyber-physical systems are suggesting new opportunities to enrich the existing research, and the existing cyber-physical systems are inspiring the emerging research. I expect this dissertation could contribute to such interaction and inspiration between the existing and emerging cyber-physical systems.

References

[air, 1991] (1991). Arinc specification 651: Design guidance for integrated modular avionics.

[ARI, 2010] (2010). Avionics application software standard interface: Arinc specification 653p1-3.

[Abdelzaher et al., 2000] Abdelzaher, T. F., Atkins, E. M., and Shin, K. G. (2000). Qos negotiation in real-time systems and its application to automated flight control. *IEEE Trans. Comput.*, 49(11):1170–1183.

[Adelberg et al., 1995] Adelberg, B., Garcia-Molina, H., and Kao, B. (1995). Applying update streams in a soft real-time database system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD.

[Adelberg et al., 1996] Adelberg, B., Kao, B., and Garcia-Molina, H. (1996). Database support for efficiently maintaining derived data. In *Proceedings of 5th International Conference on Extending Database Technology*, volume 1057. Springer.

[Al Sheikh et al., 2011] Al Sheikh, A., Brun, O., Hladik, P.-E., and Prabhu, B. J. (2011). A best-response algorithm for multiprocessor periodic scheduling. In *Proc. of the 23rd Euromicro Conference on Real-Time Systems*, pages 228–237.

[Al Sheikh et al., 2012] Al Sheikh, A., Brun, O., Hladik, P.-E., and Prabhu, B. J. (2012). Strictly periodic scheduling in IMA-based architectures. *Real-Time Syst.*, 48(4):359–386.

[Almeida and Pedreiras, 2004] Almeida, L. and Pedreiras, P. (2004). Scheduling within temporal partitions: response-time analysis and server design. In *Proceedings of the 4th ACM international conference on Embedded software*, pages 95–103.

[Andrei et al., 2008] Andrei, A., Eles, P., Peng, Z., and Rosén, J. (2008). Predictable Implementation of Real-Time Applications on Multiprocessor Systems-on-Chip. In *Proc. of Int'l Conference on VLSI Design*, pages 103–110.

[Bieber et al., 2012] Bieber, P., Boniol, F., Boyer, M., Noulard, E., and Pagetti, C. (2012). New challenges for future avionic architectures. *Aerospace Lab*, 488(4).

[Bini and Buttazzo, 2004] Bini, E. and Buttazzo, G. C. (2004). Schedulability analysis of periodic fixed priority systems. *IEEE Trans. Comput.*, 53(11).

[Bisschop, 2011] Bisschop, J. (2011). AIMMS Optimization Modeling. *Paragon Decision Technology*.

[Bui et al., 2008] Bui, B. D., Caccamo, M., Sha, L., and Martinez, J. (2008). Impact of Cache Partitioning on Multitasking Real Time Embedded Systems. In *Proc. of IEEE Int'l Conference on Embedded and Real-Time Computing Systems and Applications*.

[Cai and Kong, 1996] Cai, Y. and Kong, M. C. (1996). Nonpreemptive scheduling of periodic tasks in uni- and multiprocessor systems. *Algorithmica*, 15(6):572–599.

[Chen et al., 2003] Chen, D., Mok, A., and Kuo, T.-W. (2003). Utilization bound revisited. *IEEE Transactions on Computers*, 52:351–361.

[Chisholm et al., 2015] Chisholm, M., Ward, B. C., Kim, N., and Anderson, J. H. (2015). Cache sharing and isolation tradeoffs in multicore mixed-criticality systems. In *Proceedings of the IEEE Real-Time Systems Symposium*.

[Cipar et al., 2012] Cipar, J., Ganger, G., Keeton, K., Morrey, III, C. B., Soules, C. A., and Veitch, A. (2012). Lazybase: Trading freshness for performance in a scalable database. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys.

[Davis and Burns, 2005] Davis, R. I. and Burns, A. (2005). Hierarchical fixed priority pre-emptive scheduling. In *Proceedings of the 24th IEEE Real-Time Systems Symposium*, pages 389–398.

[Davis and Burns, 2008] Davis, R. I. and Burns, A. (2008). An investigation into server parameter selection for hierarchical fixed priority pre-emptive systems. In *Proceedings of Real-Time and Network Systems, RTNS*.

[Dewan and Fisher, 2010] Dewan, F. and Fisher, N. (2010). Approximate bandwidth allocation for fixed-priority-scheduled periodic resources. In *Proceedings of the 2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS '10, pages 247–256.

[Easwaran, 2007] Easwaran, A. (2007). Compositional schedulability analysis supporting associativity, optimality, dependency and concurrency. *PhD thesis, Computer and Information Science, University of Pennsylvania*.

[Fedorova et al., 2010] Fedorova, A., Blagodurov, S., and Zhuravlev, S. (2010). Managing Contention for Shared Resources on Multicore Processors. *Communications of the ACM*, 53(2):49–57.

[Feo and Resende, 1995] Feo, T. A. and Resende, M. G. (1995). Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6:109–133.

[Fisher, 2009] Fisher, N. (2009). An FPTAS for interface selection in the periodic resource model. In *Proceedings of the 17th International Conference on Real-Time and Network Systems*, pages 127–136.

[Garey and Johnson, 1975] Garey, M. R. and Johnson, D. S. (1975). Complexity results for multiprocessor scheduling under resource constraints. *SIAM J. Comput.*, 4(4):397–411.

[Gustafsson and Hansson, 2004a] Gustafsson, T. and Hansson, J. (2004a). Data management in real-time systems: a case of on-demand updates in vehicle control systems. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS.

[Gustafsson and Hansson, 2004b] Gustafsson, T. and Hansson, J. (2004b). Dynamic on-demand updating of data in real-time database systems. In *Proceedings of the ACM Symposium on Applied Computing*, SAC.

[Han et al., 1996] Han, C.-C., Lin, K.-J., and Hou, C.-J. (1996). Distance-constrained scheduling and its applications to real-time systems. *IEEE Transactions on Computers*, 45:814–826.

[Han and Tyan, 1997] Han, C.-C. and Tyan, H.-Y. (1997). A better polynomial-time schedulability test for real-time fixed-priority scheduling algorithm. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 36–45. IEEE Computer Society.

[Herman et al., 2012] Herman, J. L., Kenna, C. J., Mollison, M. S., Anderson, J. H., and Johnson, D. M. (2012). Rtos support for multicore mixed-criticality systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 197–208. IEEE.

[Holte et al., 1989] Holte, R., Mok, A., Rosier, L., Tulchinsky, I., and Varvel, D. (1989). The pinwheel: a real-time scheduling problem. In *Proc. of the 22nd Annual Hawaii International Conference on System Sciences*.

[Hsueh and Lin, 1996] Hsueh, C.-W. and Lin, K.-J. (1996). An optimal pinwheel scheduler using the single-number reduction technique. In *RTSS*, pages 196–205.

[Hu et al., 2015] Hu, S., Yao, S., Jin, H., Zhao, Y., Hu, Y., Liu, X., Naghibolhosseini, N., Li, S., Kapoor, A., Dron, W., Su, L., Bar-Noy, A., Szekely, P., Govindan, R., Hobbs, R., and Abdelzaher, T. F. (2015). Data acquisition for real-time decision-making under freshness constraints. In *RTSS*.

[Huyck, 2012] Huyck, P. (2012). Arinc 653 and multi-core microprocessors - considerations and potential impacts. In *Proc. of the 31st IEEE/AIAA Digital Avionics Systems Conference (DASC 2012)*, pages 6.B.4–1 – 6.B.4–7.

[Jeffay and Stanat, 1991] Jeffay, K. and Stanat, D. F. (1991). On non-preemptive scheduling of periodic and sporadic tasks. In *Proc. of the 12th IEEE Real-Time Systems Symposium (RTSS)*, pages 129–139.

[Joseph and Pandya, 1986] Joseph, M. and Pandya, P. K. (1986). Finding response times in a real-time system. *The Computer Journal*, 29(5):390–395.

[Kang et al., 2002a] Kang, K.-D., Son, S., and Stankovic, J. (2002a). Star: secure real-time transaction processing with timeliness guarantees. In *Proceedings of the IEEE Real-Time Systems Symposium*, RTSS.

[Kang et al., 2002b] Kang, K.-D., Son, S., Stankovic, J., and Abdelzaher, T. (2002b). A qos-sensitive approach for timeliness and freshness guarantees in real-time databases. In *Proceedings of the Euromicro Conference on Real-Time Systems*, ECRTS.

[Kang et al., 2004] Kang, K.-D., Son, S. H., and Stankovic, J. A. (2004). Managing deadline miss ratio and sensor data freshness in real-time databases. *IEEE Transactions on Knowledge and Data Engineering*, 16(7):2004.

[Kao et al., 2003] Kao, B., Lam, K.-Y., Adelberg, B., Cheng, R., and Lee, T. (2003). Maintaining temporal consistency of discrete objects in soft real-time database systems. *IEEE Transactions on Computers*, 52(3):373–389.

[Kermia and Sorel, 2007] Kermia, O. and Sorel, Y. (2007). A rapid heuristic for scheduling non-preemptive dependent periodic tasks onto multiprocessor. In *Proc. of ISCA 20th International Conference on Parallel and Distributed Computing Systems, PDCS'07*, pages 1–6.

[Kim et al., 2015a] Kim, J.-E., Abdelzaher, T., and Sha, L. (2015a). Budgeted generalized rate monotonic analysis for the partitioned, yet globally scheduled uniprocessor model. In *Proceedings of the 21st IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*.

[Kim et al., 2015b] Kim, J.-E., Abdelzaher, T., and Sha, L. (2015b). Schedulability bound for integrated modular avionics partitions. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '15.

[Kim et al., 2016a] Kim, J.-E., Abdelzaher, T., Sha, L., Bar-Noy, A., and Hobbs, R. (2016a). Sporadic Decision-centric Data Scheduling with Normally-off Sensors. In *Proceedings of the 37th IEEE Real-Time Systems Symposium (RTSS)*.

[Kim et al., 2016b] Kim, J.-E., Abdelzaher, T., Sha, L., Bar-Noy, A., Hobbs, R., and Dron, W. (2016b). On Maximizing Quality of Information for the Internet of Things: A Real-time Scheduling Perspective (Invited). In *Proc. of IEEE Int'l Conference on Embedded and Real-Time Computing Systems and Applications*.

[Kim et al., 2017] Kim, J.-E., Bradford, R., Abdelzaher, T., and Sha, L. (2017). A schedulability test for software migration on multicore systems. In *Proceedings of the 20th ACM/IEEE Conference on Design, Automation and Test in Europe (DATE)*.

[Kim et al., 2014] Kim, J.-E., Yoon, M.-K., Bradford, R., and Sha, L. (2014). Integrated modular avionics (IMA) partition scheduling with conflict-free I/O for multicore avionics systems. *Proceedings of the 38th IEEE Computer Software and Applications Conference (COMPSAC 2014)*.

[Kim et al., 2013] Kim, J.-E., Yoon, M.-K., Im, S., Bradford, R., and Sha, L. (2013). Optimized scheduling of multi-ima partitions with exclusive region for synchronized real-time multi-core systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, pages 970–975.

[Kinnan, 2009] Kinnan, L. (2009). Use of multicore processors in avionics systems and its potential impact on implementation and certification. In *Proc. of the 28th IEEE/AIAA Digital Avionics Systems Conference*.

[Korst et al., 1994] Korst, J., Aarts, E., Lenstra, J. K., and Wessels, J. (1994). Periodic assignment and graph colouring. *Discrete Appl. Math.*, 51(3):291–305.

[Korst et al., 1996] Korst, J. H. M., Aarts, E. H. L., and Lenstra, J. K. (1996). Scheduling periodic tasks. *INFORMS Journal on Computing*, 8(4):428–435.

[Krodel, 2004] Krodel, J. (2004). Commercial off-the-shelf real-time operating system and architectural considerations. *Federal Aviation Administration*.

[Kuo et al., 2003] Kuo, T.-W., Chang, L.-P., Liu, Y.-H., and Lin, K.-J. (2003). Efficient online schedulability tests for real-time systems. *IEEE Trans. Software Eng.*, 29(8):734–751.

[Kuo and Mok, 1991] Kuo, T.-W. and Mok, A. K. (1991). Load adjustment in adaptive real-time systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 160–170. IEEE Computer Society.

[Kuo and Mok, 1997] Kuo, T.-W. and Mok, A. K. (1997). Incremental reconfiguration and load adjustment in adaptive real-time systems. *IEEE Trans. Comput.*, 46(12):1313–1324.

[Lauzac et al., 2003] Lauzac, S., Melhem, R. G., and Moss, D. (2003). An efficient rms admission control and its application to multiprocessor scheduling. In *IPPS/SPDP*, pages 511–518.

[Lee et al., 1999] Lee, C., Lehoczky, J., Siewiorek, D., Rajkumar, R., and Hansen, J. (1999). A scalable solution to the multi-resource qos problem. *In Proceedings of the IEEE Real-Time Systems Symposium*.

[Lee et al., 1996] Lee, C.-G., Kim, Y.-K., Son, S. H., Min, S. L., and Kim, C. S. (1996). Efficiently supporting hard/soft deadline transactions in real-time database systems. In *Proceedings of the Third International Workshop on Real-Time Computing Systems Application*, RTCSA.

[Lee et al., 2004] Lee, C.-G., Sha, L., and Peddi, A. (2004). Enhanced utilization bounds for qos management. *IEEE Trans. Comput.*, 53(2):187–200.

[Lee et al., 2000a] Lee, Y.-H., Kim, D., Younis, M., and Zhou, J. (2000a). Scheduling tool and algorithm for integrated modular avionics systems. In *Proc. of the 19th Digital Avionics Systems Conference (DASC)*.

[Lee et al., 2000b] Lee, Y.-H., Kim, D., Younis, M., Zhou, J., and Mcelroy, J. (2000b). Resource scheduling in dependable integrated modular avionics. In *Proc. of the International Conference on Dependable Systems and Networks (FTCS-30 and DCCA-8)*, pages 14–23.

[Lehoczky et al., 1989] Lehoczky, J., Sha, L., and Ding, Y. (1989). The rate monotonic scheduling algorithm: exact characterization and average case behavior. In *Real Time Systems Symposium, 1989., Proceedings.*, pages 166–171.

[Leung and Whitehead, 1982] Leung, J. Y.-T. and Whitehead, J. (1982). On the complexity of fixed-priority scheduling of periodic real-time tasks. *Performance Evaluation*, 2(4):237–250.

[Liebeherr et al., 1995] Liebeherr, J., Burchard, A., Oh, Y., and Son, S. H. (1995). New strategies for assigning real-time tasks to multiprocessor systems. *IEEE Trans. Comput.*, 44(12):1429–1442.

[Lipari and Bini, 2003] Lipari, G. and Bini, E. (2003). Resource partitioning among real-time applications. In *Proceedings of the 15th Euromicro Conference on Real-Time Systems*, pages 151–158.

[Lipari and Bini, 2005] Lipari, G. and Bini, E. (2005). A methodology for designing hierarchical scheduling systems. *J. Embedded Comput.*, 1(2):257–269.

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

[Locke et al., 1990] Locke, D., Lucas, L., and Goodenough, J. (1990). Generic avionics software specification. technical report CMU/SEI-90-TR-008, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania.

[Miller, 1996] Miller, H. (1996). The multiple dimensions of information quality. *Information Systems Management*, 2(13):79–82.

[Mok, 1983] Mok, A. K.-L. (1983). *Fundamental Design Problems of Distributed Systems for the Hard–Real–Time Environment*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology. Ph.D. Thesis.

[Mutlu and Moscibroda, 2007] Mutlu, O. and Moscibroda, T. (2007). Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. In *Proc. of IEEE/ACM Int'l Symposium on Microarchitecture*.

[Nahrstedt and Smith, 1995] Nahrstedt, K. and Smith, J. M. (1995). The qos broker. *IEEE Multimedia*, 2:53–67.

[Nemati et al., 2011] Nemati, F., Behnam, M., and Nolte, T. (2011). Independently-developed real-time systems on multi-cores with shared resources. In *Proc. of the 23rd Euromicro Conference on Real-Time Systems (ECRTS)*.

[Paolieri et al., 2009a] Paolieri, M., Quiñones, E., Cazorla, F. J., Bernat, G., and Valero, M. (2009a). Hardware Support for WCET Analysis of Hard Real-Time Multicore Systems. In *Proc. of IEEE/ACM Int'l Symposium on Computer Architecture*, pages 57–68.

[Paolieri et al., 2009b] Paolieri, M., Quiñones, E., Cazorla, F. J., and Valero, M. (2009b). An Analyzable Memory Controller for Hard Real-Time CMPs. *IEEE Embedded Systems Letters*, 1(4).

[Park et al., 1996] Park, D.-W., Natarajan, S., and Kanevsky, A. (1996). Fixed-priority scheduling of real-time systems using utilization bounds. *Journal of Systems and Software*, 33(1):57 – 63.

[Park et al., 1995] Park, D.-W., Natarajan, S., Kanevsky, A., and Kim, M. J. (1995). A generalized utilization bound test for fixed-priority real-time scheduling. In *Proceedings of the 2nd International Workshop on Real-Time Computing Systems and Applications*.

[Parkinson and Kinnan, 2007] Parkinson, P. and Kinnan, L. (2007). Safety-critical software development for integrated modular avionics. *White Paper, Wind River Systems*.

[Perry et al., 2004] Perry, W. L., Signori, D., and Boon, J. E. (2004). *Exploring Information Superiority: A Methodology for Measuring the Quality of Information and Its Impact on Shared Awareness*. RAND Corporation.

[Qu and Labrinidis, 2007] Qu, H. and Labrinidis, A. (2007). Preference-aware query and update scheduling in web-databases. In *Proceedings of the 23rd IEEE International Conference on Data Engineering*.

[Ramamritham, 1993] Ramamritham, K. (1993). Real-time databases. *Distributed and Parallel Databases*, 1(2):199–226.

[Ripoll and Ballester-Ripoll, 2013] Ripoll, I. and Ballester-Ripoll, R. (2013). Period selection for minimal hyperperiod in periodic task systems. *Computers, IEEE Transactions on*, 62(9):1813–1822.

[Röhm et al., 2002] Röhm, U., Böhm, K., Schek, H.-J., and Schuldt, H. (2002). Fas: A freshness-sensitive coordination middleware for a cluster of olap components. In *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB '02.

[Roman et al., 2011] Roman, R., Najera, P., and Lopez, J. (2011). Securing the internet of things. *Computer*, 44(9):51–58.

[Rosén et al., 2007] Rosén, J., Andrei, A., Eles, P., and Peng, Z. (2007). Bus Access Optimization for Predictable Implementation of Real-Time Applications on Multiprocessor Systems-on-Chip. In *Proc. of IEEE Int'l Real-Time Systems Symposium*, pages 49–60.

[Rosén et al., 2011] Rosén, J., Neikter, C.-F., Eles, P., Peng, Z., Burgio, P., and Benini, L. (2011). Bus Access Design for Combined Worst and Average Case Execution Time Optimization of Predictable Real-Time Applications on Multiprocessor Systems-on-Chip. In *Proc. of IEEE Real-Time and Embedded Technology and Applications Symposium*.

[Rufino et al., 2010] Rufino, J., Craveiro, J., and Verissimo, P. (2010). Architecting robustness and timeliness in a new generation of aerospace systems. *Architecting dependable systems VII, LNCS*, 6420:146–170.

[Rushby, 1999] Rushby, J. (1999). Partitioning in avionics architectures: Requirements, mechanisms, and assurance. *NASA Langley Technical Report*.

[Saewong et al., 2002] Saewong, S., Rajkumar, R. R., Lehoczky, J. P., and Klein, M. H. (2002). Analysis of hierarchical fixed-priority scheduling. In *Proceedings of the 14th Euromicro Conference on Real-Time Systems*, pages 152–160.

[Schliecker and Ernst, 2011] Schliecker, S. and Ernst, R. (2011). Real-time performance analysis of multiprocessor systems with shared memory. *ACM Trans. Embed. Comput. Syst.*, 10(2):22:1–22:27.

[Schranzhofer et al., 2010] Schranzhofer, A., Chen, J.-J., and Thiele, L. (2010). Timing Analysis for TDMA Arbitration in Resource Sharing Systems. In *Proc. of IEEE Real-Time and Embedded Technology and Applications Symposium*.

[Sha, 2003] Sha, L. (2003). Real-time virtual machines for avionics software porting and development. In Chen, J. and Hong, S., editors, *Real-Time and Embedded Computing Systems and Applications, 9th International Conference, RTCSA 2003, February 18-20, 2003.*, pages 123–135.

[Sha et al., 2016] Sha, L., Caccamo, M., Mancuso, R., Kim, J.-E., Yoon, M.-K., Pellizzoni, R., Yun, H., Kegley, R., Perlman, D., Arundale, G., and Bradford, R. (2016). Real-time computing on multicore processors. *IEEE Computer*, 49(9):69–77.

[Sha and Goodenough, 1990] Sha, L. and Goodenough, J. B. (1990). Real-time scheduling theory and ada. *IEEE Computer*, 23(4):53–62.

[Shin and Lee, 2003] Shin, I. and Lee, I. (2003). Periodic resource model for compositional real-time guarantees. In *Proceedings of the 24th IEEE International Real-Time Systems Symposium*, RTSS '03, pages 2–13.

[Shin and Lee, 2010] Shin, I. and Lee, I. (2003 (rev. 2010)). Periodic resource model for compositional real-time guarantees. Tech. rep., Dep. of Computer & Information Science, University of Pennsylvania.

[Shin and Lee, 2008] Shin, I. and Lee, I. (2008). Compositional real-time scheduling framework with periodic model. *ACM Transactions on Embedded Computing Systems*, 7(3):30:1–30:39.

[Song and Liu, 1995] Song, X. and Liu, J. W.-S. (1995). Maintaining temporal consistency: Pessimistic vs. optimitic concurrency control. *IEEE Trans. Knowl. Data Eng.*, 7(5):786–796.

[Suhendra and Mitra, 2008] Suhendra, V. and Mitra, T. (2008). Exploring Locking & Partitioning for Predictable Shared Caches on Multi-Cores. In *Proc. of Design Automation Conference*.

[Suhendra et al., 2006] Suhendra, V., Raghavan, C., and Mitra, T. (2006). Integrated Scratchpad Memory Optimization and Task Scheduling for MPSoC Architectures. In *Proc. of Int'l Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 401–410.

[Tămaş-Selicean and Pop, 2011a] Tămaş-Selicean, D. and Pop, P. (2011a). Design optimization of mixed-criticality real-time applications on cost-constrained partitioned architectures. In *Proc. of the 32nd IEEE Real-Time Systems Symposium*, pages 24–33.

[Tămaş-Selicean and Pop, 2011b] Tămaş-Selicean, D. and Pop, P. (2011b). Optimization of time-partitions for mixed-criticality real-time distributed embedded systems. In *Proc. of the 2011 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops*, pages 1–10.

[Ward et al., 2013] Ward, B. C., Herman, J. L., Kenna, C. J., and Anderson, J. H. (2013). Making shared caches more predictable on multicore platforms. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems*.

[Weber, 2010] Weber, R. H. (2010). Internet of things new security and privacy challenges. *Computer Law & Security Review*, 26(1):23–30.

[Xiong et al., 2005] Xiong, M., Han, S., and Lam, K.-Y. (2005). A deferrable scheduling algorithm for real-time transactions maintaining data freshness. In *Proceedings of the IEEE International Real-Time Systems Symposium*, RTSS.

[Xiong and Ramamritham, 2004] Xiong, M. and Ramamritham, K. (2004). Deriving deadlines and periods for real-time update transactions. *Computers, IEEE Transactions on*, 53(5):567–583.

[Xiong et al., 2008] Xiong, M., Wang, Q., and Ramamritham, K. (2008). On earliest deadline first scheduling for temporal consistency maintenance. *Real-Time Systems*, 40(2):208–237.

[Yoon, 2011] Yoon, M.-K. (2011). Tunable WCET for hard real-time multicore system. Master's thesis, University of Illinois at Urbana-Champaign, Urbana, IL.

[Yoon et al., 2013] Yoon, M.-K., Kim, J.-E., Bradford, R., and Sha, L. (2013). Holistic design parameter optimization of multiple periodic resources in hierarchical scheduling. In *Proceedings of the 16th ACM/IEEE Design, Automation, and Test in Europe (DATE 2013)*, pages 1313 – 1318.

[Yoon et al., 2011] Yoon, M.-K., Kim, J.-E., and Sha, L. (2011). Optimizing tunable WCET with shared resource allocation and arbitration in hard real-time multicore systems. In *Proceedings of the 32nd IEEE Real-Time Systems Symposium*, pages 227–238.

[Zuhily and Burns, 2007] Zuhily, A. and Burns, A. (2007). Optimal (d-j)-monotonic priority assignment. *Information Processing Letters*, 103(6):247–250.