Carnegie Mellon University

CARNEGIE INSTITUTE OF TECHNOLOGY

THESIS

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF Doctor of Philosophy

TITLE **Dependable Cyber-Physical Systems**

PRESENTED BY **Junsung Kim**

ACCEPTED BY THE DEPARTMENT OF

Electrical and Computer Engineering

Raj Rajkumar	May 1, 2014
ADVISOR, MAJOR PROFESSOR	DATE
Jelena Kovacevic	May 1, 2014
DEPARTMENT HEAD	DATE

APPROVED BY THE COLLEGE COUNCIL

Vijayakumar Bhagavatula	May 2, 2014
DEAN	DATE

Dependable Cyber-Physical Systems

Submitted in partial fulfillment of the requirements for

the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

Junsung Kim

M.S., Electrical Engineering, Korea Advanced Institute of Science and Technology
B.S., Electrical Engineering, Korea Advanced Institute of Science and Technology

Carnegie Mellon University
Pittsburgh, PA

May 2014

Abstract

CPS (Cyber-Physical Systems) enable a new class of applications that perceive their surroundings using raw data from sensors, monitor the timing of dynamic processes, and control the physical environment. Since failures and misbehaviors in application domains such as cars, medical devices, nuclear power plants, etc., may cause significant damage to life and/or property, CPS need to be safe and dependable. A conventional way of improving dependability is to use redundant hardware to replicate the whole (sub)system. Although hardware replication has been widely deployed in conventional mission-critical systems, it is cost-prohibitive to many emerging CPS application domains. Hardware replication also leads to limited system flexibility.

This dissertation studies the problem of making CPS affordably dependable and develops a system-level framework that manages critical CPS resources including processors, networks, and sensors. Our framework called SAFER (System-level Architecture for Failure Evasion in Real-time applications) incorporates configurable software mechanisms and policies to tolerate failures of critical CPS resources while meeting their timing constraints. It supports adaptive graceful degradation, the effective use of different sensor modalities, and the fault-tolerant schemes of hot standby, cold standby, and re-execution. SAFER reliably and efficiently allocates tasks and their backups to CPU and sensor resources while satisfying network traffic constraints. It also fuses and (re)configures sensor data used by tasks to recover from system failures. The SAFER framework aims to guarantee the timeliness of different types of tasks that fall into one of four categories: (1) tasks with periodic arrivals, (2) tasks with continually varying periods, (3) tasks with parallel threads, and (4) tasks with self-suspensions. We offer the schedulability analyses and runtime support for such tasks with and without resource failures. Finally, the functionality of the proposed system is evaluated on a self-driving car using SAFER. We conclude that the proposed framework analytically satisfies timing constraints and predictably operates systems with and without resource failures, hence making CPS dependable and timely.

Acknowledgments

It has been a long journey to my PhD, but I have been lucky enough to work with talented, thoughtful, and passionate people from CMU (Carnegie Mellon University), RTML (Real-Time Multimedia and Systems Lab), GM-CMU ADCRL (General Motors-Carnegie Mellon Autonomous Driving Collaborative Research Lab), GM (General Motors), and SEI (Software Engineering Institute). I am also grateful to GM, NSF (National Science Foundation), and Goel Graduate Fellowship for funding my research.

My first and foremost thanks go to my adviser Prof. Raj Rajkumar. Since the first time I met him at CMU in 2008, he has been a good mentor and teacher. His endless creative ideas always intrigued me. His passions in his research resonated with me so that I could have countless sleepless nights to accomplish my goals. He always carefully listened to what I said and gave me sincere comments. Also, when I confronted personal problems, he was never hesitant to help me by sharing his experience, providing indirect support, and stepping up for me.

I would like to sincerely thank my thesis committee members: Prof. Dan Siewiorek, Prof. Anthony Rowe, and Dr. Markus Jochim. Prof. Dan Siewiorek always surprised me by providing different research perspectives and encouraged me to focus on the basics of research. Prof. Anthony Rowe's bright ideas always inspired me to be a thrill-seeking researcher. As a former RTML member, he has also been a good friend of mine and helped me in shaping my research ideas and getting used to Pittsburgh. Dr. Markus Jochim always encouraged me to maintain balance between academic and pragmatic thinking. This in turn helped me a lot in identifying exciting research problems.

Without RTML members, I could not have completed my Ph.D. I deeply thank all who spent much time with me: Reza Azimi, Sheryl Benicky, Gaurav Bhatia, Max Buevich, Yong Hoon Choi, Alexei Colin, Vikram Gupta, Dr. Arvind Kandhalu, Prof. Shinpei Kato, Hyoseung Kim, Dr. Karthik Lakshmanan, Jay Reppert, Prof. Anthony Rowe, and Dr. Young-Woo Seo. Reza's cheerfulness spread to me. Gaurav has always been there to have good food, coffee, and discussions together. Vikram and I shared lots of pseudo-masala tea at the CIC (Collaborative Innovation Center) kitchen. The conversations with Arvind and Karthik at 3 a.m. were always productive. I still remember our special breakfast at 6 a.m. after a paper submission, and I hope to have more fun time with them in the near future. Shinpei's boundless energy always kept me awake. I also enjoyed Korean-centric discussions with Hyoseung days and nights. Young-Woo helped me with understanding new research areas. Sheryl always fed us very nice foods for long meetings.

My childhood dream was to make KITT from Knight Rider. I was delighted to be part of GM-CMU ADCRL as I could see that having KITT was not a dream anymore. I appreciate our team members tirelessly working and achieving the impossible: Jason Atwood, Hyunggi Cho, Dr. John Dolan, Tianyu Gu, JongHo Lee, Bruce Li, Dr. Paul Rybski, Alok Sharma, Jarrod Snider, Junqing Wei, and Wenda Xu. I appreciated John's disciplined group management. He always helped me with getting more organized in very complex environment. Being in an autonomous vehicle with Jarrod, JongHo, Junqing, Tianyu, and Wenda was always fun and exciting. Their infinite passion for autonomous driving made me proud of contributing to our automated vehicle. I was grateful that I could work with them.

Thankfully, I was able to work with excellent researchers from GM and SEI: Dr. Markus Jochim, Massimo Osella, Dr. Björn Andersson, and Dr. Dionisio de Niz. I was able to learn about the automotive industry much as Massimo was very inclusive when I was a student intern at GM. The discussions with Björn and Dio were insightful and helped me with understanding the state-of-the-are real-time theories. I also had great opportunities to work with bright students: Dipendra Kumar Misra, Praful Puranik, and Uttkarsh Sarraf. I am thankful for all the support they have given me.

I thank the KDisTech members who made my Pittsburgh life more lively: Sang Kil Cha, Hyeju Jang, Hanbyul Joo, Minhee Jun, Gunhee Kim, Soonho Kong, Jay-Yoon Lee, Seunghak Lee, Soochahn Lee, Shane Moon, Hyun Soo Park, JunBum Shin, Seungmoon Song, Daegun Won, and Sungwook Yang. I deeply thank my friends who have backed me for decades: Fantastic Bundang 4 (Hyungbin Ahn, Minchul Kim, and Insun Park), my long-time roommate Moonseok Lee, (chu-) Tae-Hoon Choi, Sangchul Kim, Cheeta EunJung Li, and Michigan dudes Jay-Yong Lee and Maesoon Im.

My final and deepest thanks go to mom, dad, my sister, and my lovely wife Min Kyung Lee. My dad has taught me how to live. My mom has shown me how to love. Thank you so much for being my parents. My little sister has always been encouraging and supportive. I am grateful that you are always together with mom and dad in Korea. My wife Min has been my closest friend, companion, and advisor. I would never have completed this dissertation without her. Thank you for your support, understanding, and love.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Advances in CPS (Cyber-Physical Systems) have enabled a variety of different applications such as drones, implantable medical devices, smart cars, distributed transportation systems, smart grids, and planetary robots, which are tightly coupled with the physical world. As CPS become part of everyday life, we will have many societal benefits ranging from autonomous driving preventing accidents to smart buildings saving energy to implantable medical devices changing the paradigm for patient treatment. A recent report from NIST (National Institute of Standards and Technology) predicted that the technical CPS innovations could be applicable to areas constituting up to $82 trillion in economic activity by 2025.

The rise of CPS, however, poses new reliability and safety challenges. CPS sense the physical environment, process data in real-time, control actuators, and guarantee the timing of the whole execution chain for ensuring safety. Since CPS are tightly coupled with the physical world, anomalies such as hardware failures and timing errors may cause significant damage to life and/or property. Therefore, CPS need to satisfy strict timing constraints based on operating characteristics, making timing guarantees an essential requirement. System dependability is also of high importance in CPS applications due to the interactions with the physical environment. A typical example of such systems of an autonomous vehicle is depicted in Figure 1.1 [2].

Common practices addressing those anomalies tend to over-provision resources, replicating

Figure 1.1: During a road test of an autonomous vehicle research platform developed at CMU

hardware components and keeping CPU and network loads low. However, many CPS systems are targeted towards large-scale cost-sensitive markets that have stringent space and bill-of-material constraints that cannot afford overprovisioning. For example, the automotive industry has been trying to consolidate in-vehicle CPUs to reduce assembly and maintenance costs, as CPU- and network-hungry autonomous driving features hit the market. More specifically, a recent high-end vehicle has several active safety features such as adaptive cruise control, collision avoidance, lane departure warning, and parking assist. Such a vehicle may not have enough space or it may become too expensive to deploy traditional hardware redundancy for all CPS features to meet reliability requirements. Higher assembly costs and complexity coming from overprovisioning resources may not be desirable either. This trend is expected to continue as these features will be available even in mid-range cars in the near future. We tackle such challenges by devising new computational models reflecting the timing nature of CPS for system timeliness and providing a runtime framework that improves CPS dependability.

1.1 Thesis Statement

A dependable cyber-physical system is achievable using a software framework that enables system analyzability and predictability.

In this dissertation, we study the problem of enabling dependable CPS through a system-level software framework that manages CPS resources including processors, networks, and sensors. The framework analytically satisfies timing constraints and predictably operates systems with and without resource failures. To address these challenges, we develop an analysis engine that supports efficient task allocation for software replication and guarantees timeliness of different types of tasks that fall into one of four categories:

- Tasks with periodic arrivals: the proposed approach leverages the characteristics of software replicas and network traffic for efficient task and/or backup allocation.
- Tasks with continually varying periods: a new task model is proposed to analyze tasks with continually varying periods and workloads. We name this type of tasks *rhythmic tasks*.
- Tasks with parallel threads: task transformations are used to effectively schedule real-time fork-join tasks with global fixed-priority assignment.
- Tasks with self-suspensions: a new scheduling algorithm that assigns different priority per task segment is provided to deal with tasks that suspend themselves.

We design and prototype a runtime framework called SAFER (System-level Architecture for Failure Evasion in Real-time applications). It incorporates configurable software mechanisms and policies to tolerate failures of critical CPS resources while meeting task timing constraints. SAFER supports adaptive graceful degradation, effective use of different sensor modalities, and the fault-tolerance schemes of hot standby, cold standby, and re-execution. SAFER takes outputs of the analysis engine, and it then reliably and efficiently allocates tasks and their backups to CPU and sensor resources with network traffic considerations. It also fuses and (re)configures sensor data used by tasks to recover from system failures. The functionality of the proposed

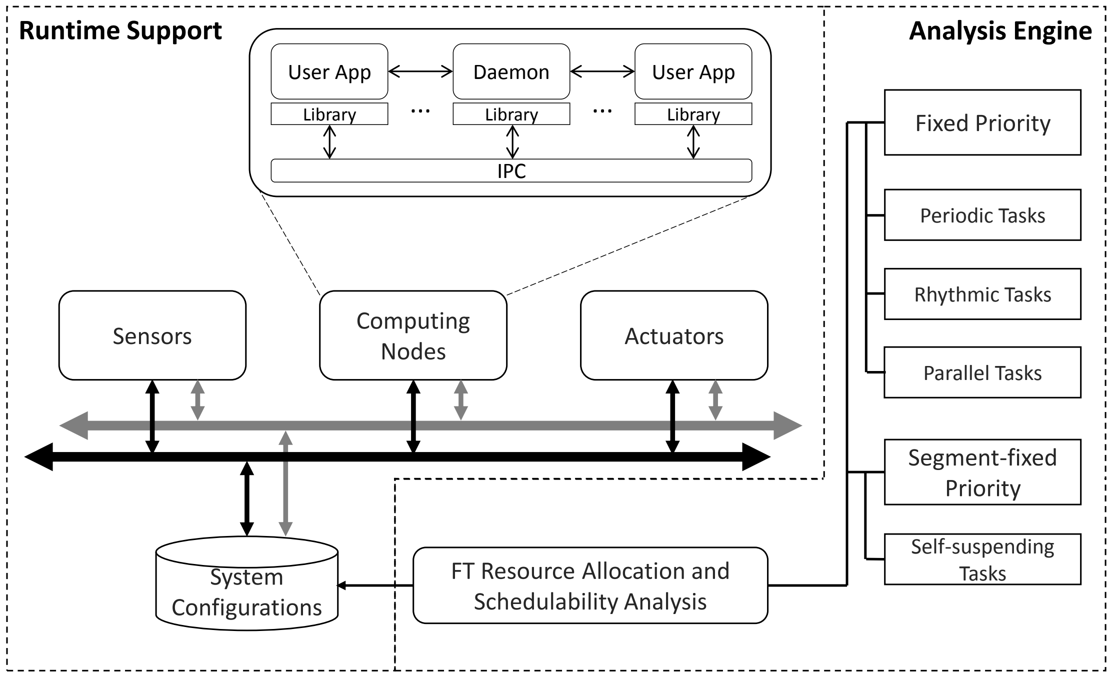


Figure 1.2: Overview of the dissertation.

system is evaluated on a self-driving car using SAFER.

An overview of the proposed framework is depicted in Figure 1.2. The efficient resource allocation algorithms for software replication are described in Chapters 3 and 4. We study schedulability analyses for the different task models: periodic tasks in Chapter 5, rhythmic tasks in Chapter 6, parallel tasks in Chapter 7, and self-suspending tasks in Chapter 7. The results of the resource allocation and schedulability analysis methods are utilized in SAFER to telerate failures of critical CPS resources. The details of how SAFER works can be found in Chapter 8.

1.2 Scope of the Thesis

We assume that the system comprises p nodes communicating via messages over a network, where each node has a multi-core processor executing real-time tasks. Those tasks are scheduled

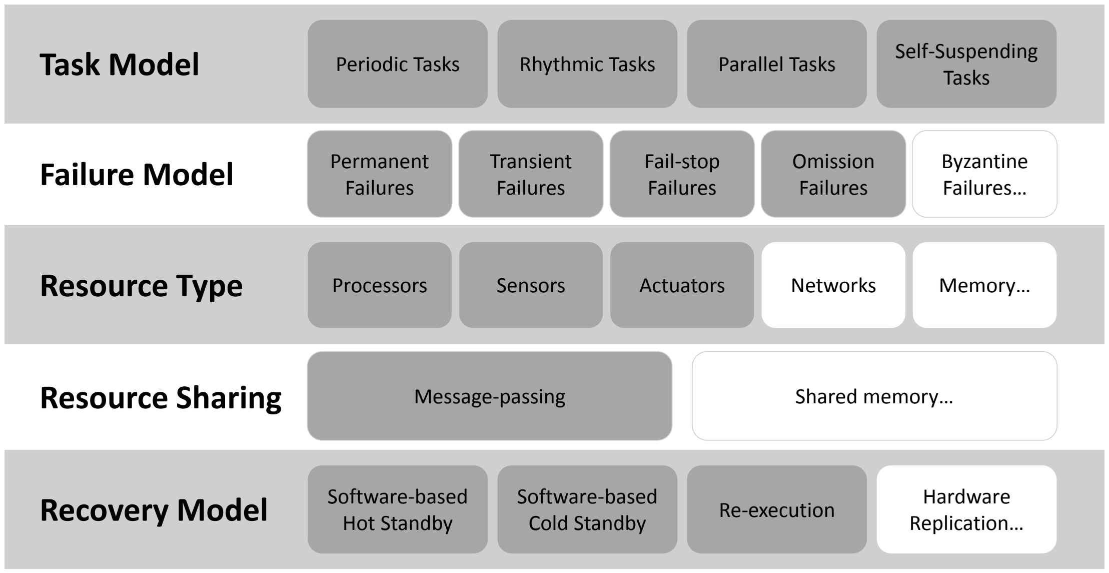


Figure 1.3: Design space of the dissertation.

under the fixed-priority preemptive scheduling policy. Some tasks are independent[1], and the other tasks use a Publish-Subscribe architecture to communicate with each other so that any task on the system can be configured to be recoverable. The network has an upper-bound on message delivery and is completely connected. In other words, a message is eventually delivered within a known delay bound, and the network is assumed not to partition[2]. The design space of this dissertation is depicted in Figure 1.3, where the dark gray rectangles represent the design assumptions considered in this dissertation.

1.2.1 Failure Model

Tasks, processors, and sensors on the system are subject to fail-stop failures, where they fail by crashing and do not generate incorrect outputs. In other words, tasks running on a live pro-

[1]Although tasks that share mutually exclusive resources are beyond the scope of this dissertation, conventional real-time synchronization protocols such as priority inheritance and priority ceiling protocol can be leveraged to incorporate such tasks.

[2]Redundant links can make network partitioning highly unlikely. Such network redundancy is the topic of future study and is beyond the scope of this dissertation.

cessor/node are assumed to always emit correct outputs. Therefore, in order for the system to continue to correctly operate, recovery and restoration processes might be required. These failures may also happen concurrently. In this dissertation, no single-point-of-failure is allowed. We also expect failures to get recovered within a guaranteed time duration.

The system network may experience occasional omission failures, i.e., it may suffer from intermittent packet loss. This implies that the network does not fail completely. This again can be realized (say) by using redundant links.

1.2.2 Task Model

Each task is assumed to generate an infinite series of independent jobs. Each job will have different characteristics based on the task model: periodic task, rhythmic task, parallel task, or self-suspending task. One common property is that all jobs have associated timing deadlines. Although we generally assume a hard real-time system, the effects of a deadline miss may vary. We assume that all jobs are preemptable with negligible cost. We also assume that there is negligible migration cost when a job is migrated from one core to another.

We model a real-time task as a sequence of jobs that are releasing every T units of time. Depending on whether T continually varies or not, we classify tasks into two different classes: periodic tasks or rhythmic tasks. When a task runs with multi-threads, we categorize it as a parallel task. When a task suspends itself and hence consists of multiple execution segments, we treat it as a self-suspending task. Although either a periodic task or a rhythmic task could be a parallel task and/or a self-suspending task, this dissertation assumes that (1) a self-suspending task does not have parallel threads, (2) a parallel task does not suspend itself, and (3) both a self-suspending task and a parallel task are periodic.

We aim to limit the fail-over time on each task to yield a reliable system. Depending on the fail-over time requirement, we classify each task into one of three classes: (1) *Hard Recovery Task*, which should be able to recover and complete within its original deadline, (2) *Soft Recovery*

Task, which has more relaxed fail-over time requirements, and (3) *Best-Effort Recovery Task*, whose recovery is optional depending on the amount of available resources.

1.2.3 Recovery Model

We use passive replication (primary-backup [3]), which has been mostly used for soft real-time systems. To support both hard and soft recovery tasks described above, we provide two different types of backups: *hot standby* and *cold standby*.

A hot standby runs concurrently with its primary on another processor. Depending on the reliability requirements, multiple hot standbys may coexist on different nodes. Only the primary emits its outputs. Its hot standbys simultaneously run on different nodes, receive the same input as the primary, but they do not generate any outputs. When a primary fails, one of the hot standbys is promoted to be the primary and starts generating outputs. Since it has been already running, only interface redirection from a null output device to the active channel needs to happen. This enables the use of hot standbys to recover hard recovery tasks[3].

A cold standby is a dormant task which is triggered to run when a failure is detected. When its primary is running, its binary resides in the system memory, and it does not consume any CPU resources; however, state information from the primary task computations are periodically sent to its cold standby node(s). On failure of the primary or a specified number of hot standbys, the cold standby becomes active and starts from the last check-pointed status. The cold standbys can be used for recovering soft recovery tasks and best-effort recovery tasks.

1.3 Approach Overview

In this section, we describe our proposed approach. Various aspects of our work fall into one of three categories: (1) resource allocation for fault-tolerant computing, (2) schedulability analysis

[3]A time synchronization service is important to support hot standbys.

for cyber-physical systems, and (3) runtime support for fault-tolerance features.

1.3.1 Resource Allocation for Fault-Tolerant Computing

Based on the task classification defined in Section 1.2.2, the recovery-time requirement imposed by hard recovery tasks does not allow much room for re-executing the failed jobs. For such tasks, a practical solution is to use multiple hot standbys that can take over the functionality under the presence of failures. Jobs of these hot standbys are released in parallel with those of the primary task, and they have the same deadline as the primary.

We assume a fail-stop failure model [4], where a working replica can assume control by detecting the lack of output from the primary. The replica can immediately provide the output since it would also have the output by the original deadline. In order to maximize task reliability, a process and its hot standbys should *not* be co-located on the same processor. We refer to this as the *placement constraint*. For this purpose, we develop a task allocation algorithm that optimizes for allocating tasks with hot standbys having such placement constraints.

Cold standbys reduce the resource over-provisioning costs further by getting activated only under failure conditions. The cold standbys use the task state information, which can be stored in shared memory or obtained during subsequent execution, and are used to recover soft recovery or best-effort recovery tasks. The benefit of a cold standby is that it leads to lesser consumption of resources under normal conditions. However, the recovery time bounds under cold standby will be much larger than those guaranteed by a hot standby. Using the system reliability requirement and the maximum number of processors that can fail during system operation, we can reduce the resource over-provisioning required for cold standbys of tasks allocated across different processors. Using this observation, we develop an algorithm that uses *virtual tasks* to consolidate and capture the resource requirements for cold standbys. The details can be found in Chapter 3.

We then consider a set of tasks that communicate each other to achieve the same goal in Chapter 4. For example, in Steer-by-Wire (SBW) systems [5], sensors measure information about steering wheel movement, and computational components in microprocessors compute signals for controlling the wheels with the information from sensors. Actuators receive the control signals for the motors directly, and these signals are handled periodically for timely handling of user operations and reactions to the environment. In order to reflect this nature, we define an *application flow*, which is composed of periodically executing tasks generating information data and events regularly that flow through multiple tasks. An application flow also has an end-to-end delay constraint from input to output. From a dependability perspective, a single failure of a task within an application flow may affect all of its successors such that the overall application flow timing requirement is violated. By extending the two task allocation algorithms above, we also propose an algorithm designed for the application flow model. The algorithm captures communication among tasks and cluster them based on their network bandwidth needs.

1.3.2 Schedulability Analyses for Cyber-Physical Systems

To properly allocate CPS tasks to resources, it is important to understand the characteristics of CPS tasks beyond the conventional periodic task model. In this subsection, we will show three different task types: tasks with continually varying periods, tasks with parallel threads, and tasks with self-suspensions. We then propose how to predict their executions and analyze their properties for satisfying timing constraints on CPS.

Tasks with Continually Varying Periods

CPS require a high level of confidence in system timeliness as a critical task not meeting its timing deadline can lead to system failure. The dynamic nature of CPS is a dominant factor affecting the CPS timeliness. In automotive sub-systems, for example, the engine events activating the fuel injection task come from reference pulses generated by sensors at the engine crankshaft. Therefore, the periods of these tasks vary depending on the speed of the crankshaft. As an analog variable, speed is continuous and hence the period of the task can change both rapidly and continuously. The execution time of these tasks also vary and the worst-case execution time (WCET) arises when the engine speed increases to its maximum [6]. It is known in the automotive community that the engine control performance deteriorates with undersampling, i.e., tasks having a longer period than the required minimum period for a given speed.

Conventional task models such as periodic tasks or aperiodic tasks are not adequate to deal with such dynamic CPS behaviors as they do not incorporate physical attributes. In Chapter 5, therefore, we define a new task model called *Rhythmic Tasks* for characterizing and analyzing tasks that have continually varying periods depending on external physical events. We also propose response-time analyses for rhythmic tasks under three cases: constant engine speed, accelerating engine speed and decelerating engine speed. We provide guidelines well-suited for CPS applications to evaluate schedulable utilization levels for the rhythmic task model.

Tasks with Parallel Threads

Many CPS tasks for perception (tracking) and actuation (planning) must run in real-time; however the CPU-hogging nature of these algorithms poses challenges in guaranteeing their timeliness. The timing challenge can be addressed by the fact that such algorithms are immensely parallelizable. For example, a planning algorithm of a self-driving car can benefit from parallelized tasks composed of numerous threads. The motion-planning algorithm calculates the best path for the vehicle to follow among a myriad of potential paths. Since the candidate paths are independent, this algorithm can be expedited by parallelizing the cost calculation for each path. The more paths the algorithm goes through, the better the driving quality will be. A perception subsystem of a self-driving car can also benefit from parallel tasks. In order for the vehicle to understand its surroundings, the perception subsystem should be able to process massive amounts of data from various types of sensors. The vehicle can classify and track the detected obstacles, whose number has a major impact on how many parallel threads are spawned by the perception subsystem.

In Chapter 6, we extend the fork-join real-time task model proposed in [1] so that an arbitrary number of threads can be scheduled, where the number of threads can vary depending on the physical attributes of the system. To efficiently schedule the proposed task model, we also propose a task transform to schedule the task model on a given number of processing cores. Then, we provide a resource augmentation bound for global Deadline Monotoic (DM) scheduling for fork-join real-time tasks. The proposed scheme is implemented on Linux/RK [7] and ported to the self-driving car Boss [8]. We evaluate our proposed scheme on Boss by showing its driving quality in terms of curvature and velocity profiles of the vehicle with an enhanced motion-planning algorithm [9].

Tasks with Self-Suspensions

An increasing number of special-purpose processors in CPS are added to improve the efficiency of frequently used operations. Unfortunately, the use of such special-purpose processors (a.k.a. hardware accelerators) may introduce suspension delays that must be taken into account in schedulability analyses when a task waits for a shared resource and interacts with an I/O device or communication interface. Offloading complex computations to hardware accelerators such as Digital Signal Processors (DSPs) or Graphics Processing Units (GPUs) can cause suspension delays as well, hence reducing the benefits of using such hardware accelerators.

Although many conventional real-time theories [10] have incorporated the delays in the worst-case execution/response time of a task that suspends itself, the analysis results lead to have significant pessimism. A pessimistic analysis is not desirable in a compute-intensive system such as the self-driving car depicted in Figure 1.1. Such systems run computationally-demanding algorithms ranging from perception [11] to planning [9, 12] on GPUs in real-time. In this case, if we use traditional schedulability analysis, the potential utilization improvement due to the use of GPUs is eliminated by the pessimism in the CPU scheduling.

In Chapter 7, to improve the schedulability of a taskset with tasks with self-suspensions, we propose the segment-fixed priority scheduling that decomposes self-suspending tasks into multiple segments assigning them different priorities if needed. We use phase enforcement to prevent jitters [13, 14], and we develop an exact schedulability analysis.

1.3.3 Runtime Support for Fault-tolerance Features

To support the proposed approach in real-time, it is important to build a robust functional architecture for CPS that allows to perform the repeating sequence of perception, computation and control in the presence of possible system failures. Most importantly, no single point of failure is permitted. In other words, a task/processor/sensor failure should not lead to a system failure. Secondly, failure recovery within a guaranteed duration should be achieved. Since CPS are usually tightly connected to the physical world, failure recovery without predictable timing behavior could yield unpredictable results in the physical world. Apart from these two goals, predictive fault discovery and notification, resource isolation, ease of use of abstraction, ease of application development, and sensor/actuator control are other factors considered.

The goals are achieved using a layer called SAFER (System-level Architecture for Failure Evasion in Real-time applications) that incorporates configurable task-level fault-tolerance features to tolerate fail-stop processor, task, and sensor failures for CPS in a timely manner. To detect failures, SAFER monitors the health status and state information of each task and broadcasts the information. When a failure is detected using either time-based failure detection or event-based failure detection, SAFER reconfigures the system to retain the functionality of the whole system using task-level fault-tolerance techniques. More specifically, SAFER provides the following features: (a) Each task can have zero, one or more backup(s), (b) Each backup can be either a hot standby or a cold standby, (c) Failure detection and recovery latencies can be guaranteed, (d) A primary and each of its backup(s) are always allocated to run on independent processor boards to avoid common failure modes, (e) State transfer is managed for seamless recovery from failures, (f) In case of a sensor failure, tasks using the failed sensor will be notified to apply appropriate sensor recovery schemes. The detailed information is discussed in Chapter 8.

Our fault-tolerant task allocation schemes proposed in 1.3.1 are integrated with SAFER through a model-based development tool, SysWeaver, developed at CMU [15]. Our analysis engine based on the formal timing analysis is added to SysWeaver, and we add a simulation capability of SAFER features under the presence of failures. Specifically, by injecting failures, we are able to simulate the timing behavior of the system and verify its operation with different models and system parameters.

Sensor/Actuator Failure Recovery

The effective use of *different sensor/actuator modalities* in CPS is essential. CPS have various sensor modalities providing 360-degree coverage. Many analog sensors are prone to intermittent faults, so using different sensor modalities is better than duplicating the same type of sensors because different types of sensors typically respond to the same environmental condition in diverse ways. Suppose an autonomous vehicle is equipped with radars for blind spot detection. If a backward-looking radar does not work properly, a vision algorithm detecting obstacles from images obtained through a backward-looking camera can be used. A similar approach is also applicable to actuators. An autonomous vehicle may use a low-grade sensor with complex data-processing algorithms after a high-grade sensor with simple algorithms fails, until the vehicle can safely stop.

Chapter 2

Literature Review

This chapter describes existing work related to this dissertation. Our work can fall into one of three categories: (1) resource allocation for fault-tolerant computing, (2) schedulability analysis for CPS, and (3) runtime support for fault-tolerance features. We will discuss the related work in each domain and explain how our work differs from the literature. Sensor failure recovery will also be discussed as a part of the runtime support for fault-tolerance features.

2.1 Resource Allocation for Fault-tolerant Computing

Real-time scheduling algorithms for uniprocessors have been studied extensively for guaranteeing timeliness. For example, Liu and Layland proposed a static real-time scheduling algorithm, RMS (Rate Monotonic Scheduling), which prioritizes periodic tasks according to their rates since [16]. Emerging demands on computational capability have driven various multiprocessor scheduling algorithms for supporting similar properties on multiprocessor environments. Multiprocessor scheduling is therefore a well-studied problem in real-time systems literature [17, 18, 19, 20]. Existing solutions are broadly classified into global [21] scheduling with unrestricted task migration, partitioned [22] scheduling with strictly no task migration, and hybrid [23, 24] with restricted task migration. Although each of these approaches has its own benefits, in this work we are primarily interested in the partitioned approach on multiprocessor environments due to the high cost associated with migrating tasks across processors. We allow on-demand activation to deal with failed processors, where we explicitly capture the associated timing cost of restoring the task state on a different processor.

A wealth of literature exists on the topic of fault-tolerant computing [25, 26, 27]. *active* and *passive* replication are standard mechanisms to improve system reliability. These replication-based approaches affect on research [28, 29, 30] in real-time domain. The key distinction made in real-time contexts is systems with strict bounds on system recovery time. A successful recovery is one where the system not only resumes its normal operation but does so within a pre-specified recovery time. These recovery time requirements are typically derived from the physical environment in which the system operates. For example, an autonomous vehicle cannot stall significantly in the middle of a highway during system recovery.

Fault-tolerant scheduling in multiprocessor systems has also received attention in [31]. In [32], the FFD (First-Fit Decreasing) is augmented with placement constraints to allocate replicated tasks. Most closely related to our work is [33], where the authors proposed a BFD (Best-Fit Decreasing) with placement constraints as a practical solution, which we call BFD-P (BFD with Placement constraint). Our work differs from theirs in two ways: (i) we propose a new bin-packing heuristic using a cluster of replicated tasks which performs better than BFD-P, and (ii) we deal with the allocation of cold standbys that are passive entities, which can be potentially consolidated across processors.

2.2 Schedulability Analysis for Cyber-Physical Systems

2.2.1 Tasks with Continually Varying Periods

Extensions to the conventional periodic task model [16] such as constrained-deadline sporadic tasks [34] and arbitrary deadline tasks [35] have been explored in the past. Although these task models represent tasks having different relationships between their periods and deadlines, the task parameters themselves are static and/or worst-case in nature. In this dissertation, we explore a model where certain tasks have dynamically changing parameters, which are determined by external or *physical* system attributes such as the engine speed in a PCM.

The importance of task periods on the quality of engine control has been demonstrated in [6]. As the engine speed varies, the system must continuously change the engine control task periods. Given vehicle dynamics [36], maintaining a close relationship between the control task and the engine speed is key for achieving high efficiency. Worst-case execution time analysis of engine tasks was carried out in [37]. At higher speeds, system designers tend to adaptively reduce the task computation times to counteract the shrinking task periods, and try to maintain approximately constant system utilization. We develop the rhythmic task model in detail to represent such types of engine control tasks and study the resulting properties.

Some task models with dynamically changing parameters have been studied in the past. For instance, the elastic task model [38] treats tasks as springs with given elastic co-efficients. More recently, the gravitational task model [39] was introduced by representing tasks as bobs hanging on a pendulum with the objective of preferably executing at a target set point. Although these task models have dynamically changing parameters, their usage is often motivated by the need to provide *quality of service* or to maximize *system utility*. Also, due to the fact that the elastic task model uses dynamic-priority scheduling and the gravitational task model is based on non-preemptive jobs, the previous work is not appropriate for fixed-priority preemptive scheduling. We consider a model where the changes in task parameters are resulting from the physical nature of the system, and changes in the operating environment drive task requirements.

From a schedulability analysis perspective, the analyses of minimum task periods and maximum worst-case execution time are well-known results for the periodic task model [40, 41]. The acyclic task model [42] uses a task model where a task comprises successive invocations but with no constraints between the periods of successive invocations. The utilization bound for acyclic tasks was also derived. Our rhythmic task model is more restricted, is motivated by cyber-physical requirements and should yield better utilization. We provide some bounds and guidelines to find schedulable regions for the generic rhythmic task model. These results are also helpful to understand the utilization bounds when only the task periods are given. We study the properties of acceleration and deceleration, which correspond to the maximum rate at which task periods can be decreased and increased respectively. In this regard, the closest work to ours is that of the mode change protocol [43]. However, we are interested in understanding the effect of a series of *continuous* mode changes on the schedulability of lower-priority tasks, as opposed to one single independent system-level mode change.

Tasks with relationships between task periods and physical attributes can be also found in other cyber-physical subsystems besides the engine control task. For example, in the context of autonomous driving [8], the sensor processing tasks need to execute at a higher rate when the vehicle is moving at a higher speed, since the vehicle would cover a longer distance in a shorter time. Another good example is building energy management [44] where fine-grained management depending on varying environmental parameters will save more energy. Also, most CPS with control algorithms can likely obtain benefits from the rhythmic task model because the quality of control is affected significantly by sampling rates.

2.2.2 Tasks with Parallel Threads

Since Dhall and Liu [17] showed that RMS and Earliest Deadline First (EDF) scheduling could utilize only one processor regardless of how many processors a system had, there has been extensive research on global real-time scheduling [20, 45, 46, 47, 48, 49, 50, 51], where a comprehensive survey can be found in [51]. It is well-known that the anomaly of global scheduling happens when a set of tasks has two types of tasks: tasks with a low ratio of the worst-case execution time to relative deadline and tasks with a high ratio of the worst-case execution time to relative deadline. Many algorithms have been invented to avoid such cases, and corresponding schedulability tests have been proposed. Using our proposed task transformation, any existing global scheduling algorithm can be applied to schedule parallel real-time tasks. In this dissertation, we have used the schedulability bounds for global DM proposed in [46, 50].

There has not been much research on scheduling parallel real-time tasks [1, 52, 53, 54]. Lakshmanan et al. [1] proposed a fork-join real-time task model composed of alternating sequential and parallel segments. They also provided the analysis and resource augmentation bound for the partitioned DM scheduling [22] of parallel real-time tasks using the task *stretch* transformation. The proposed multiprocessor scheduling algorithm is shown to have a resource augmentation bound of 3.42, which implies that any task set that is feasible on m unit-speed processors can be scheduled by the proposed algorithm on m processors that are 3.42 times faster. Our work is a generalization of this model and provides a resource augmentation bound when global scheduling is used.

Saifullah et al. [53] also proposed a parallel synchronization model that is also generalized from the fork-join task model in [1] so that a task can have an arbitrary number of threads per segment. Based on the proposed model, a task decomposition method is used to decompose each parallel task into a set of sequential tasks. The task decomposition achieves a resource augmentation bound of 4 and 5 when the decomposed tasks are scheduled using global EDF and partitioned DM scheduling, respectively. Our work focuses more on global fixed-priority scheduling and shows the evaluation results measured from a real-world implementation.

More recently, Nelissen et al. [54] presented both offline and online algorithms to minimize the number of cores to be used to schedule multi-threaded tasks using a similar model to the model proposed in [53]. By using scheduling algorithms which can guarantee the schedulability of the given tasks as long as the sum of densities of all the given tasks is less than or equal to the number of processing cores, they obtained a resource augmentation bound of 2. Our perspective is different from theirs in a sense that we schedule a set of tasks under a given hardware constraint (the number of processing cores) rather than finding hardware for the given tasks. We also use global DM scheduling algorithm more commonly used in practice and show the evaluation results obtained from a working system.

Apart from work using the *thread* model mentioned above, there has also been research based on *gang scheduling*, where all parallel components of the same task should arrive and complete at the same time. Gang EDF [52] was proposed to address gang scheduling in the real-time context. Our work is different from this in two ways: (1) our model allows the parallel segments to be preempted during the parallel execution, and (2) a different number of parallel threads can be used.

2.2.3 Tasks with Self-Suspensions

Previous work related to task-fixed priority scheduling with suspension includes [14, 55, 56, 57, 58]. Ridouard, et al. [56, 58] proved that the problem of scheduling real-time tasks with self-suspension is NP-Hard in the strong sense. In [55] the authors present a comparison between two multi-processor priority inheritance protocol (MPCP and MSRP), where tasks can suspend waiting for a remote lock. In this work the authors highlight the different approaches to deal with this suspension. In MPCP, a task waiting for a global lock is allowed to suspend, allowing lower-priority tasks to run, and a period-enforcement is used to avoid jitter [13]. In MSRP, on the other hand, a busy wait is used and no lower-priority tasks are allowed to run. In our work, we also use a period enforcement mechanism to avoid jitter in the suspension, but each segment (e.g. before and after the suspension) is given a different priority according to different schemes of segment deadline assignments. In [57] the authors analyze the execution of tasks with segments running in a local processors and segments running on remote co-processors that can be seen as a suspension in the local processor. In this case the authors bound the suspension with a minimum and maximum and provide a recurrence equation to find the worst-case interference that a task can suffer from higher-priority ones with a number of these segments. In contrast, we provide a schedulability bound for taskset with only the highest-priority task with suspension while using a generalized task model with suspensions where each segment is assigned its own priority. The period enforcement of offsets, which [57] do not use, allows us to provide improved schedulability.

In [14] the authors analyze the scheduling of fixed-priority tasksets with self-suspension. Specifically, the authors characterize the critical instant of sporadic self-suspending task under the influence of non-suspending tasks and developed a response time test. In addition, they provide two execution control policies that transform the interference of high-priority suspending tasks into that similar to a non-suspending ones to be able to use their response-time test with these tasks. In contrast, we developed a schedulability bound for a taskset where the higher-priority is a self-suspending task and developed a response-time test for suspending tasks where each segment can be assigned different priorities and have release enforcement.

The schedulability of self-suspending tasks has also been studied for soft-real-time guarantees. In [59] Liu and Anderson presented a technique to analyze the schedulability of soft real-time tasks with suspension with bounded deadline tardiness requirements scheduled under global EDF in multiprocessors. In [60] the authors studied the problem of bounding the tardiness of soft-real-time tasks when using GPU as coprocessors, and model them with two techniques, as a shared resource and as a container. For the shared resource approach they used locking protocols to frame the analysis of GPU execution either as suspension or busy time depending on the locking protocol. For the container approach they use a hierarchical bandwidth reservation (a container) approach grouping all the tasks that use the GPU in a separate container to provide a FIFO scheduling discipline and considering their suspension as busy time.

In [61] the author presents a schedulability analysis for tasks with offsets. These offsets are used to synchronize the release of groups of tasks that synchronized within the group (known as transactions). In [62] the authors extend this work to allow offsets and deadlines to go beyond periods improving the schedulable utilization. The efficiency of the response time analysis in this model is then further improved in [63]. These papers have some similarities with the use of offsets between segments in tasks in our model. However, in our work we start with suspension intervals that separate task segments from where we derive intermediate deadlines that in turn allows us to assign per-segment fixed priorities.

In [64] the authors developed another schedulability analysis for tasks with offsets. However, in this case the analysis assumes EDF scheduling and the results cannot be applicable to fixed-priority tasks.

2.3 Runtime Support for Fault-tolerance Features

Fault-tolerant distributed embedded systems have been extensively studied in the literature. The ISIS system [65] is a well-known software system that supports fault-tolerance services. FT-CORBA (Fault-Tolerant CORBA) [27, 66, 67] has been used in various applications to design and implement a fault-tolerant distributed system, and practical experiences on two different FT-CORBA infrastructures are described in [68]. CORBA-based fault-tolerant middleware services are also surveyed in [69]. There are other replication-based recovery services such as Arjuna [70], REL [71] and IFLOW [72], which are not based on CORBA. One clear distinction between the existing work and SAFER is that SAFER provides the framework to support timely failure recovery in a generic distributed embedded system.

There have also been efforts on building real-time fault-tolerant systems. MEAD [73] provides a proactive fail-over framework using a failure prediction method to overcome the unpredictable nature of failure occurrences and support somewhat predictable timing behavior. FLARe [74] is designed and implemented to support fault-tolerance for distributed soft real-time applications. SAFER differs from the above-mention systems in that SAFER is built on a publish-subscribe model rather than a client-server model. In addition, SAFER provides predictable timing characteristics of failure detection and recovery when real-time systems become SAFER-enabled. SAFER also provides a flexible failure detection and recovery infrastructure. SAFER detects failures using heartbeat signals (*time-based*) as well as OS (Operating Systems) signals (*event-driven*). A primary can use hot standby and/or cold standby as a backup.

2.3.1 Sensor/Actuator Failure Recovery

There have been extensive efforts on detecting sensor failures and recovering from those failures. In general, there are two types of sensor failures: *hard* and *soft* failures [75][1]. When a failure is hard, the failed sensor is stuck at a certain condition so that it outputs only one value including no output and the following measured data become invalid. When a sensor experiences a soft failure, its output becomes less reliable. In other words, the quality of the measurement is degraded, and the data should be cautiously used. To detect such failures, many different approaches have been proposed. In [75, 79], the authors used a bank of Kalman filters for a multiple model adaptive estimator to detect and identify sensor failures. The authors of [80, 81] leveraged a Bayesian belief network model to achieve these goals. Other methods such as fuzzy logic [82], the Nadaraya Watson statistical estimator [83], and subspace model identification [84] were also used. Although our scheme can be used on top of those methods, we choose to use a Kalman filter-based method in this dissertation due to its speed and accuracy on our evaluation platform.

Recovering from sensor failures has been extensively studied in the literature ranging from building a fault-tolerant sensor [85] to making data fusion reliable [78]. In [85], Marzullo proposed a process control program that can tolerate sensor failures. He used replication and voting to mask failures and studied hierarchies of failure models to average sensor values in a fault-tolerant way. Our work is more general in a sense that we do not necessarily use the same type of sensors. Our focus is also more on leveraging different modalities of sensors to improve the system dependability. In [86], the authors proposed an algorithm that used one type of sensors as backups of different types of sensors. They formulated it as a multi-modal sensor allocation problem and provided schemes that could deal with binary and multi-level sensors. In [78], the

[1]There are other papers [76, 77, 78] that classify sensor failures into more detailed categories, but most of them fall into these two categories.

authors proposed a fault-tolerant data fusion technique for the same type of sensors to minimize the mean square error of sensor measurements. Although both papers use multi-sensor data fusion technique [87] to achieve their goals and our work is similar in that sense, our approach takes into account both the same and different types of sensors. There has also been research on building a reliable distributed control system [88, 89, 90], which can be applicable to actuator failure recovery.

Chapter 3

Resource Allocation for Fault-Tolerant Computing

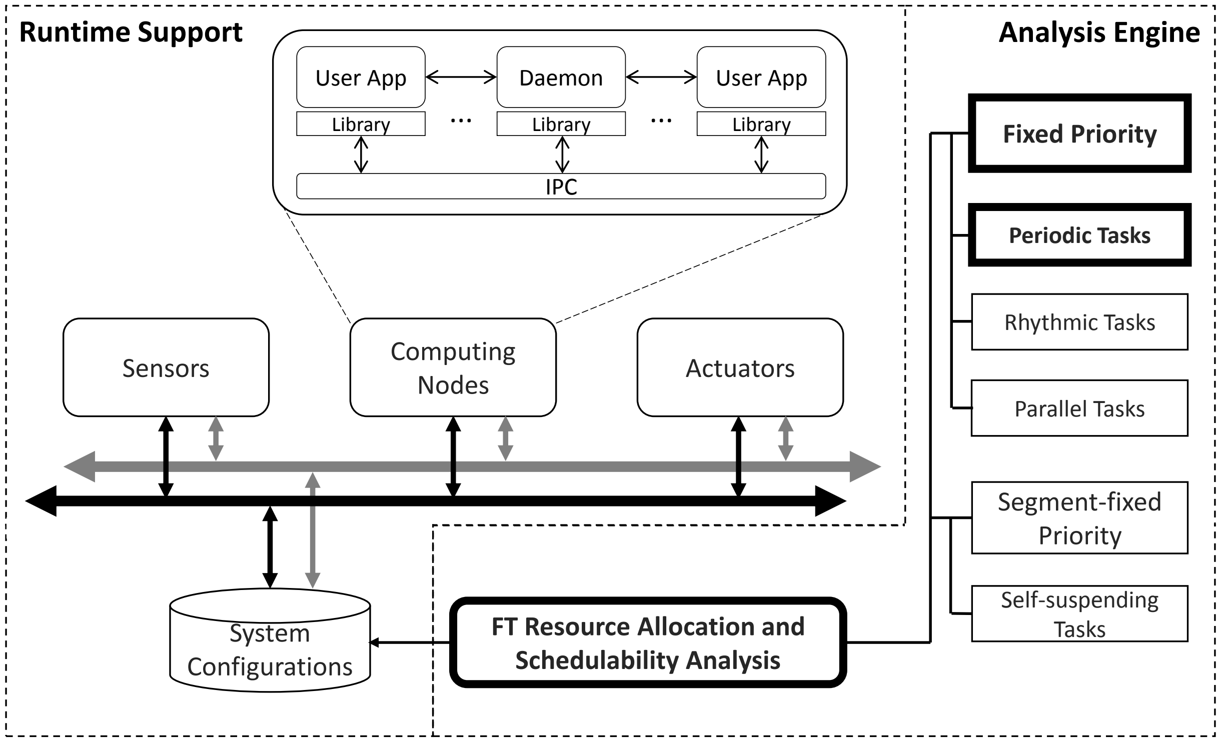


Figure 3.1: Resource allocation for fault-tolerant computing in the dissertation overview.

Cyber-Physical Systems (CPS) are growing in terms of both scale and complexity. An emphasis on scalability, extensibility, and flexibility has led to complex electrical/electronic multi-processor architectures. In a variety of applications such as industrial control, avionics, and automotive systems, such complexity can lead to unavoidable failures in both hardware and software. Furthermore, developers/designers may not be able to predict when and where faults can happen. System-level dependability is therefore a key concern in evolving CPS. An emerging application of such systems is autonomous driving. For example, the Urban Challenge winning autonomous vehicle, Boss [8], used several embedded processors and ten Intel Core2Duo processors due to high computing power requirements. However, CMOS scaling for performance improvements has decreased the reliability of processors [91]. This could potentially have catastrophic effects if not taken into account, for example, unmanned vehicles can lose driving capability due to processor failures.

Two different notions, *fault-tolerant* and *fail-safe*, are applicable for characterizing the system behavior under failures. A fault-tolerant system requires that a system/user does not recognize a failure occurrence during the operation in terms of functionalities. Several conventional replication methods such as hardware redundancy, software redundancy, and re-execution can be used to build fault-tolerant systems. A fail-safe system requires a different type of fault handling. It allows failures, but must not generate an unsafe system state by overriding a proper procedure when a failure occurs. Developing a fail-safe system requires us to consider specific failure scenarios. As described earlier, enumerating all possible failure scenarios is not an easy task for complex systems, therefore, we focus on achieving the more robust property of fault tolerance.

Systems that interact with the physical world such as autonomous vehicles should adhere to the strict timing constraints imposed by their operating environment. In such real-time systems, tasks are conventionally modeled as a periodic sequence of jobs that are releasing every T units of time, where each job needs to finish within a relative deadline of D time-units from its release. Dealing with unpredictable failures in such systems is not a trivial job. Therefore, instead of assuming a priori failure scenarios, dynamically handling failures with bounded recovery time is desirable for real-time systems. By handling failures within a required timing boundary, the Time-To-Recovery is bounded, and the system need not be stopped. Tasks with the requirement that they complete within D units of time from release, even under the presence of failures (i.e. recover and complete within the original deadline), are denoted as *Hard Recovery Tasks*. Tasks with more relaxed recovery-time requirements are denoted as *Soft Recovery Tasks*. Optional tasks that are not critical for system operation and do not require bounded recovery times are denoted as *Best-Effort Recovery Tasks*.

The recovery-time requirement imposed by Hard Recovery Tasks does not allow much room for re-executing the failed jobs. For such tasks, a practical solution is to use multiple *hot standby* replicas that can take over the functionality under the presence of failures. Jobs of these hot standbys are released in parallel with those of the primary task, and they have the same deadline as the primary. It is not required that the hot standbys execute whenever the primary executes, however, they have the same relative deadline as the primary. This relaxed synchronicity between hot standby and its primary enables a more practical solution.

We assume a *fail-stop* failure model [4], where a working replica can assume control by detecting the lack of output from the primary. The replica can immediately provide the output since it would also have the output by the original deadline. In order to maximize task reliability, a process and its hot standbys should not be co-located on the same processor. We refer to this as the *placement constraint*. For this purpose, we develop a task allocation algorithm called R-BFD (Reliable Best-Fit Decreasing) that optimizes for allocating tasks with hot standbys having such placement constraints.

Cold standbys reduce the resource over-provisioning costs further by getting activated only under failure conditions. The cold standbys use the task state information, which can be stored in shared memory or obtained during subsequent execution, and are used to recover soft recovery or best-effort recovery tasks. The benefit of cold standbys is that it leads to lesser consumption of resources under normal conditions. However, the recovery time bounds under cold standby will be much larger than those guaranteed by hot standby. Using the system reliability requirement and the maximum number of processors that can fail during system operation, we can reduce the resource over-provisioning required for cold standby replicas of tasks allocated across different processors. Using this observation, we develop an algorithm called R-BATCH (Reliable Bin-packing Algorithm for Tasks with Cold standby and Hot standby) that uses virtual tasks to consolidate and capture the resource requirements for cold standbys.

In this chapter, we consider the problem of allocating real-time periodic tasks along with their replicas while meeting the reliability and timeliness requirements of a given system as depicted in Figure 3.1. The outcomes of the proposed techniques in this chapter can be used by the runtime support discussed in Chapter 8. The rest of this chapter is organized as follows. Section 3.1 will discuss the timing properties of different task replication mechanisms in a multiprocessor environment, and will summarize our approach. The proposed task partitioning algorithm will be described in Section 3.2, and evaluated in Section 3.3. Finally, we provide our concluding remarks in Section 3.4.

3.1 Design Implications

In this chapter, we will find a reliability-enforced allocation while reducing the processor required. In other words, we will tolerate ρ permanent processor failures while minimizing number of processors which can support the given reliability requirement of a target system. Under the assumption that applications are executed periodically, transient failures can be recovered by exploiting the periodic nature of tasks. For the scheduling policy, we use RMS [16] with harmonic tasks to achieve full processor utilization [92] when required. Regarding processor failures, we use a fail-stop model, in other words, we assume that a failure stops a processor completely, and completed tasks always generate correct results. Selected tasks on those stopped processors will be recovered in different processors. We also impose a *placement constraint*, which requires that replicas must be on different processors. If two or more processors fail simultaneously, the system may be able to recover through multiple hot standbys for hard recovery tasks and a combination of hot standby and cold standbys for soft recovery tasks.

3.1.1 System Assumptions

In order to model the behavior of each task, we assume a set of tasks, Γ. Γ is composed of n tasks, τ_1, τ_2, ..., and τ_n. These tasks are divided into three subsets of Γ, ***Hard Recovery Task*** set, Γ_H, ***Soft Recovery Task*** set, Γ_S, and ***Best-effort Recovery Task*** set, Γ_B, and these categories will be defined in Section 3.1.2.

A task τ_i is represented as (C_i, T_i, D_i, α_i), where C_i is the worst-case execution time, T_i is the period, D_i is the deadline relative to the release time, and α_i represents the ratio of recovery time to deadline. The *recovery time* is defined as the time instant relative to the release time of τ_i, within which jobs of τ_i should be recovered. For example, if $\alpha_i = 1$ or $\tau_i \in \Gamma_H$, the failed job should be recovered within the original deadline, D_i, which is equal to T_i. A task τ_i has the response time denoted as R_i, where $C_i \leq R_i \leq T_i$ is satisfied because R_i is the duration from the instant of job release to the moment of job completion of τ_i.

Every task τ_i can have $\psi(i)$ hot standbys, which can be represented by $\tau^h_{i,1}$, $\tau^h_{i,2}$, ...,$\tau^h_{i,\psi(i)}$. Either τ_i or $\tau^h_{i,0}$ denotes the primary of τ_i. Since our objective is to tolerate ρ processor failures, each task τ_i also can have $\zeta(i)$, which is $\rho + 1 - \psi(i)$ cold standbys, which can be represented as $\tau^c_{i,1}$, $\tau^c_{i,2}$, ...,$\tau^c_{i,\zeta(i)}$. In addition, u_i is the utilization of τ_i, defined as $\frac{C_i}{T_i}$.

A set of processors, P, is used for running a task set, Γ. P is composed of m homogeneous processors, P_1, P_2, ..., P_m. Then, Π_i is the set of processors which have τ_i and its hot standbys. Each element of Π_i, Π_{ij} is the processor allocated to τ_{ij}, j^{th} hot standby of task τ_i. Therefore, the placement constraint is expressed as $\forall i$, $\Pi_{ij} \neq \Pi_{ik}$, where $j \neq k$. Each processor P_k has its own failure rate, f_k, which denotes the probability of a permanent failure. We assume homogeneous processors for convenience of presentation, $f_1 = f_2 = ... = f_m = f$.

3.1.2 Recovery-timing Requirements and Our Approaches

Real-time systems are composed of multiple tasks with strict timing constraints. In such systems, fault recovery entails not only restoring functionality but also doing so within a bounded time. A classical example would be a feedback control system, where failure to recover within a bounded time could cause the physical state of the controlled system to move into an unsafe state. Automotive and avionics systems are examples of such applications. Depending on the timing constraint imposed on recovery time, we classify tasks into three categories:

- *Hard Recovery:* Tasks that need to meet their original deadlines relative to their release time even under failures are denoted as Hard Recovery tasks. In other words, α_i for a task in this category is 1. Meeting the original deadline provides very little room for recovery and re-execution. These tasks are the most difficult ones to provision for failure recovery.
- *Soft Recovery:* Tasks with relaxed recovery deadlines are denoted as Soft Recovery tasks. In this case, α_i for these tasks is greater than 1, but bounded.
- *Best-Effort Recovery:* Tasks that do not have any recovery deadlines and only require recovery from a functional perspective are called as Best-Effort Recovery. For tasks in this class, α_i may not be bounded.

Task replication is a fundamental technique used to improve system reliability. By introducing task replicas on multiple processors, tasks on failed processors can be recovered. In this chapter, we consider two techniques for task replication viz. hot standby and cold standby with different timing characteristics.

- *Hot Standby Approach:* This approach uses two or more on-line copies of a certain task. One or more replicas of the task will be running simultaneously. Each replica must be on a different processor, in order to make sure that at least one of them is working when a processor failure occurs. When a failure occurs, a replica will take over the task on a failed processor. For the hot standby approach, at least one of the backup replicas should meet the original deadline when a failure occurs. In order to support this tight constraint, we use

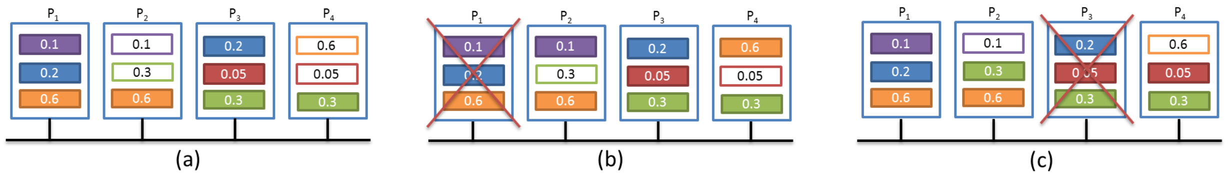


Figure 3.2: Example operation scenario of hot standby and cold standby. (a) normal case (b) P_1 has failed, and the tasks on P_1 have been recovered. (c) P_3has failed, and the tasks on P_3 have been recovered.

hot standby replicas, whose jobs are released synchronously with that of the *Primary* copy and execute in parallel with the same deadline. Any hot standby can be promoted to be the primary after recovery.

- *Cold Standby Approach:* In this case, replicas are not executed until failures occur. This means that they only use up memory, but not processor under normal operation, and they are triggered on demand when failures occur. Because cold standby are not running under normal conditions, only the state information of each primary copy needs to be shared among replicas. When failures occur, they should be detected as soon as possible, and the tasks on failed processors should be recovered using replicas within a predefined time. For the purposes of this dissertation, we consider that faults are detected instantaneously, and all task information is available in memory for fault recovery. This approach can recover both soft recovery tasks and best-effort recovery tasks.

3.1.3 Determination of Standby Type

The benefit of using a hot standby is the ability to meet the original deadline with less timing penalties than the cold standby approach, since all hot standbys are running concurrently with a primary task. The primary copy and the backup copy are released at the same time. However, this does not mean that all the replicas complete at the same time. We use the same deadline for every replica, therefore, all replicas are guaranteed to finish by the deadline. The disadvantage of using this scheme is that it requires more resources than single copy execution. Specifically, for tasks with hot standbys, additional required resource is $\psi(i)$ times u_i for τ_i.

For the cold standby approach, when there are no failures, only the primary copy is executed during the normal operation. Since we assume failures can be detected through properties of fail-stop processors, the failure of a primary copy can trigger the execution of a backup copy. For bounding recovery time, we propose a type of utilizations, *Transient Overload Utilization* (TOU). TOU is the additional utilization while a task is being recovered. In other words, TOU is a required resource for reexecuting a task on a new processor within the remaining time, $\alpha_i T_i - R_i$, after τ_i fails. Let u_i^t denote TOU of τ_i. Then, we have the following *Theorem*.

Theorem 1 *If $\alpha_i \geq 2$, $u_i^t \leq u_i$ for any task.*

Proof In TOU, u_i^t, the backup copy of τ_i should meet the primary's original deadline by using a cold standby, which implies that $\alpha_i T_i - R_i \geq C_i$ should be satisfied. This is because the remaining time for computation after τ_i's execution should be enough for one more execution in the worst case. Therefore, u_i^t is denoted as $\frac{C_i}{\alpha_i T_i - R_i}$. Then, u_i^t is equal to u_i when $\alpha_i = 1 + u_i$ for the best case ($R_i = C_i$) and $\alpha_i = 2$ for the worst case ($R_i = T_i$). Because $2 \geq 1 + u_i$, $u_i^t \leq u_i$ is satisfied when $\alpha_i \geq 2$ for any task.

In most embedded systems, only certain tasks are safety-critical. By using a large α_i, u_i^t can be relaxed. The importance of *Theorem* 1 is seen in the following example. Suppose that a task with $\alpha_i = 1$ uses a cold standby. Then, the backup copy of τ_i should meet the condition $T_i - R_i \geq C_i$. Therefore, $u_i^t = \frac{C_i}{T_i - R_i}$. Since $u_i^t \geq u_i$, another processor which runs the cold standby of the primary task which is using only cold standby should have more utilization value than u_i. Therefore, based on α_i, we can choose the type of Standby. An important observation in the cold standby approach is that the processor running the backup should have enough unused utilization. This is also applicable to other processors, which should take over other tasks that are not running at the same time on the failed processor. The reserved slack for cold standbys can be used by any task but the reserved utilization for hot standby cannot be utilized by other tasks.

In this chapter, we will use both hot standby and cold standby within the same system as necessary. In order to decrease the number of processors required to achieve reliability requirements and tolerate simultaneous failures, cold standby can be promoted to the hot standby if the primary of τ_i fails and the current number of τ_i is less than $\psi(i)$. Since at least one hot standby for certain τ_i can meet its original deadline, D_i, we are ready to tolerate another potential failure if a cold standby can be promoted to a hot standby. By using this, we can tolerate as many as ρ failures for τ_i.

Figure 3.2 has an example scenario. Suppose a task set $\Gamma = \{\tau_1, \tau_2, \tau_3, \tau_4, \tau_5\}$, and tasks have utilizations, 0.6, 0.3, 0.2, 0.1, and 0.05, respectively. With respect to replicas:

- τ_1, τ_2 have both 1 cold standby and 1 hot standby
- τ_3 has 1 hot standby
- τ_4, τ_5 have 1 cold standby each

These tasks are allocated to a set of processors, P, which has four processors. The allocation is shown in Figure 3.2(a). In Figure 3.2(b), P_1 has failed, and P_1 holds for τ_1, τ_3, and τ_4. Because τ_1 has one hot standby on P_2 and one cold standby on P_4, τ_1 is recovered by a hot standby on P_2. Then, the cold standby on P_4 is promoted to a hot standby. τ_3 is recovered by hot standby on P_3, and τ_4 is recovered by the cold standby on P_2. Similar operations happen when P_3 also has failed in Figure 3.2(c).

3.2 Task Allocation with Hot Standby and Cold Standby Replication

Allocating tasks to processors is a well-known bin-packing problem [93]. In this chapter, we consider the design-time allocation of tasks and it is well-known that this allocation is NP-hard [94].

There are several popular heuristics such as BFD, FFD, NFD (Next-Fit Decreasing) , and WFD (Worst-Fit Decreasing). Each of these algorithms regards the utilization value of each task as the object size and allocates each task to a proper processor. We will take BFD as a base-line algorithm.

3.2.1 Fault-tolerant Partitioned Scheduling

The main difference between conventional and fault-tolerant partitioned scheduling is a placement constraint in order to improve system reliability by spreading copies of the primary and the replicas. For task allocation, the original BFD 1) sorts the tasks in descending order of size, 2) fits the next task into the best processor that it can fit into, 3) adds a new processor if a task does not fit into any current processor, and 4) iterates this procedure until no tasks remain. In the current context, BFD should be modified to satisfy the placement constraint of replicas. Therefore, two steps are changed. For step 1), when the tasks are sorted, the replicas are sorted together with their primaries. For step 2), the processor should be determined using placement constraints. This algorithm is denoted as BFD-P [33] and is listed in Algorithm 3.

3.2.2 Task Allocation with Hot Standby using R-BFD

Consider a task set $\Gamma : \{\tau_1, \tau_2, ..., \tau_n\}$, where each task τ_i has a primary copy τ_i and $\psi(i)$ hot standbys $\{\tau^h_{i,1}, ..., \tau^h_{i,\psi(i)}\}$. The placement constraint dictates that none of these copies get co-located on the same processor.

R-PACK is a basic algorithm for allocating a set of n objects $\Omega : \{O_1, ..., O_n\}$ with placement constraints $\Pi : \{\Pi_1, ..., \Pi_n\}$ (see Algorithm 1). It iterates over the objects in the given order, allocating each object O_i to the best-fit processor $\Pi_{ij} \leftarrow P_k$ that satisfies the placement constraint $P_k \notin \Pi_i$. If no existing processor can fit O_i, then a new processor is added for it. The set of processors and updated placement constraints are obtained from R-PACK.

R-BFD sorts the given tasks in decreasing order of sizes (see Algorithm 2). The primary

Algorithm 1 R-PACK($\Omega : \{O_1, ..., O_n\}$,$\Pi : \{\Pi_1, ..., \Pi_n\}, P : \{P_1, ..., P_m\}$)

```
1:  for $i = 1$ to $n$ do
2:      ▷ Only worry about non-empty object list
3:      if $O_i \neq \emptyset$ then
4:          For $O_i$, find a best-fit processor $P_k$, s.t. $P_k \notin \Pi_i$
5:          if $P_k$ exists then
6:              $\Pi_i \leftarrow \Pi_i \cup P_k$ ▷ Allocate to existing processor
7:          else
8:              $\Pi_i \leftarrow \Pi_i \cup P_m$ ▷ Need to add new processor
9:              $P \leftarrow P \cup \Pi_i$
10:             $m \leftarrow m + 1$
11: return $(P, \Pi)$
```

copies are first allocated using R-PACK. The hot standbys are then allocated in batches using R-PACK. The key distinction here is that BFD-P (see Algorithm 3) would allocate the whole set of tasks (including hot standbys) with placement constraints. By operating in batches, R-BFD can better fill up the space left over in the previous processors, whereas BFD-P would lead to wastage of this space. This effect is shown in Figure 3.3. Figure 3.3 shows the result of BFD-P in 3.3(a) and the result of R-BFD in 3.3(b) under the given task set $\Gamma = \{\tau_1, \tau_2, \tau_3\}$ with one hot standby and utilization values, 0.6, 0.3, and 0.2, respectively. In this example, R-BFD saves 1 processor by allocating the hot standby replicas in batches. Since BFD-P assigns τ_3 to a new processor, placement constraints bring one more processor for $\tau_{3,1}^h$.

3.2.3 Analysis

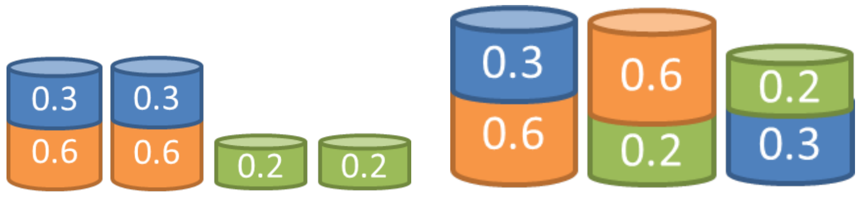
In order to abstract away from the performance of the individual uniprocessor scheduling algorithm used within the processors themselves and focus on the task allocation, we restrict our analysis to an optimal uniprocessor scheduling configuration. Under RMS, we focus on har-

Algorithm 2 R-BFD($\Gamma : \{\tau_1, \tau_2, ..., \tau_n\}$)

```
1:  Sort $\Gamma$ in descending order of utilization
2:  ▷ Allocate the primary copies first
3:  $(P, \Pi) \leftarrow$ R-PACK($\Omega \leftarrow \{\tau_1, \tau_2, ..., \tau_n\}$,
4:          $\Pi \leftarrow \{\Pi_1 \leftarrow \emptyset, ..., \Pi_n \leftarrow \emptyset\}$, $P \leftarrow \emptyset$)
5:  ▷ Allocate the replicas one by one
6:  for $j = 1$ to $\max_{\forall \tau_k \in \Gamma}(\psi(k))$ do
7:      ▷ Ignore tasks that do not need $j$ replicas
8:      $\forall \tau_i$ s.t. $\psi(i) < j$, $\tau^h_{i,j} \leftarrow \emptyset$
9:      $(P, \Pi) \leftarrow$ R-PACK($\Omega \leftarrow \{\tau^h_{1,j}, \tau^h_{2,j}, ..., \tau^h_{n,j}\}$, $\Pi$, $P$)
10: return $(P, \Pi)$
```

monic task sets that can achieve 100% utilization. Generalizing to arbitrary task periods would lead to non-ideal processor utilization due to the relationship between task periods [16], thereby detracting away from the properties of the task allocation algorithm itself. In the following discussion of BFD and R-BFD properties, we focus on harmonic tasks, although the algorithms themselves can be applicable to arbitrary task sets.

Lemma 2 *For harmonic task sets with $\psi(i) = 0\ \forall i$, BFD heuristic requires at most $M_0 = (\frac{11}{9}(OPT_0) + 4)$ processors, where OPT_0 is the number of processors required by an optimal algorithm.*



(a) BFD-P (b) R-BFD

Figure 3.3: Benefit of using R-BFD.

Proof Harmonic task sets result in a schedulable utilization bound of 100% in each processor. The task allocation problem is therefore reduced to the standard bin-packing problem, for which BFD has been established to require no more than $\frac{11}{9}(OPT_0) + 4$ processors [95] when the optimal algorithm requires OPT_0 processors.

Lemma 3 *For harmonic task sets with $\psi(i) = k\ \forall k$, where k is a positive integer constant, BFD requires at most $k(\frac{11}{9}(OPT_0) + 4)$ processors, where OPT_0 is the number of processors that would be required by an optimal algorithm to schedule the same task set with $k = 0$.*

Proof Let the optimal number of processors required to schedule the same task set assuming $\psi(i) = 0$ be OPT_0. The number of processors required by BFD to schedule the same task set under $\psi(i) = 0$ is M_0. We know that $M_0 \leq \frac{11}{9}(OPT_0) + 4$ (by Lemma 2). Due to the sorted order in which BFD considers objects and the placement constraints, when $\psi(i) = k$ the number of processors required by BFD is kM_0. This can be shown from Algorithm 3, which results in k identical processors following every $(k + 1)$th processor if $\psi(i) = k$.

Lemma 4 *For harmonic task sets with $\psi(i) = k\ \forall k$, R-BFD requires no more processors than BFD.*

Proof This follows from the fact that hot standby replicas in Algorithm 2, R-BFD at least consider the same candidate processors for allocation as in Algorithm 3, BFD-P. R-BFD therefore requires no more processors than BFD-P.

Corollary 5 *For harmonic task sets with $\psi(i) = k\ \forall k$, where k is a positive integer constant, R-BFD requires at most $k(\frac{11}{9}(OPT_0) + 4)$ processors, where OPT_0 is the number of processors that would be required by an optimal algorithm to schedule the same task set with $k = 0$.*

Proof Follows from Lemmas 3 and 4.

3.2.4 Dealing with System Reliability Requirements

Consider a uniform multiprocessor system with m processors. Let F denote the system SIL (Safety Integrity Level) [96] requirement specified in terms of the PFD (Probability of Failure

Algorithm 3 BFD-P($\Gamma : \{\tau_1, \tau_2, ..., \tau_n\}$)

```
1:  Sort $\Gamma$ in decreasing order of utilization
2:  $\tau_{i,0} \leftarrow \tau_i$
3:  **for** $i$ in 1 to $n$ **do**
4:      **for** $j$ in 0 to $\psi(i)$ **do**
5:          For $\tau_{i,j}$, find a best-fit processor $P_k$ s.t. $P_k \notin \Pi_i$
6:          **if** $P_k$ exists **then**
7:              $\Pi_{ij} \leftarrow P_k$ ▷ *Allocate to an existing processor*
8:          **else**
9:              $\Pi_{ij} \leftarrow P_m$ ▷ *Need to create a new processor*
10:             $P \leftarrow P \cup \Pi_i$
11:             $m \leftarrow m + 1$
12:         $\Pi_i \leftarrow \Pi_{ij} \cup \Pi_i$ ▷ *Update placement constraints*
13: **return** $(P, \Pi)$
```

on Demand). Let the reliability specification of each individual processor be f, denoting that the processors are designed to have a PFD less than f. Based on F and f, the system designer can estimate ρ, which is the minimum number of additional processors required to satisfy the system reliability. ρ should be greater than the maximum number of processor failures that can be expected in $(m + \rho)$ processors.

$$\rho = \min_p \{p \in Z | F \geq \sum_{j=p}^{m+p} Prob(\text{exactly } j \text{ processor failures})\}$$

$$\rho = \min_p \{p \in Z | F \geq \sum_{j=p}^{m+p} \binom{m+p}{j} f^j (1-f)^{m+p-j}\} \tag{3.1}$$

A system designer may choose a value of ρ greater than the one obtained using Equation (3.1) depending on design margins.

Algorithm 4 R-BATCH: Allocate the given task set Γ to processors P for handling ρ processor failures

```
1: $P \leftarrow \emptyset$
2: $(P, \Pi) \leftarrow$ R-BFD($\Gamma$)
3: $(\Gamma, \Pi) \leftarrow$ generateVirtualTask($\Gamma$, $\Pi$, $P$, $\rho$)
4: $(P, \Pi) \leftarrow$ R-PACK($\Gamma$, $\Pi$, $P$)
5: **return** $(P, \Pi)$
```

3.2.5 Task Allocation with Cold Standby using R-BATCH

Allocating tasks with cold standby replication in addition to hot standby replicas is accomplished by R-BATCH (Reliable-Bin-packing Algorithm for Tasks with Cold standby and Hot standby) given in Algorithm 4. The basic idea behind the algorithm is to estimate $\zeta(i) = \rho + 1 - \psi(i)$, the number of cold standbys needed for task τ_i. We observe that the cold standby replicas for processors other than those hosting τ_i can be consolidated. Suppose a task set $\Gamma = \{\tau_1, \tau_2\}$, and both tasks have 0.6 utilizations. Since they cannot fit into one processor together, two processors are necessary for Γ. In order to tolerate one failure per task by using a hot standby, two more processors are required. A cold standby, however, can reduce one processor compared to exploiting a hot standby by sharing one unused processor.

R-BATCH creates virtual task, $\tau^v_{q,j}$, where q distinguishes each virtual task and j denotes j^{th} cold standby replica of tasks covered by $\tau^v_{q,j}$. Then, $\Gamma^v_{q,j}$ is a set of tasks which can be taken over by $\tau^v_{q,j}$. Each virtual task is responsible for handling the cold standby replica for multiple processors. As it is assumed that no more than ρ distinct processors will fail during the system runtime, the virtual tasks can consolidate the replication and reduce the number of processors required considerably. The procedure for generating virtual tasks is defined in Algorithm 5.

After generating virtual tasks, the placement constraint should be satisfied. This constraint is considered by following three *Lemmas*.

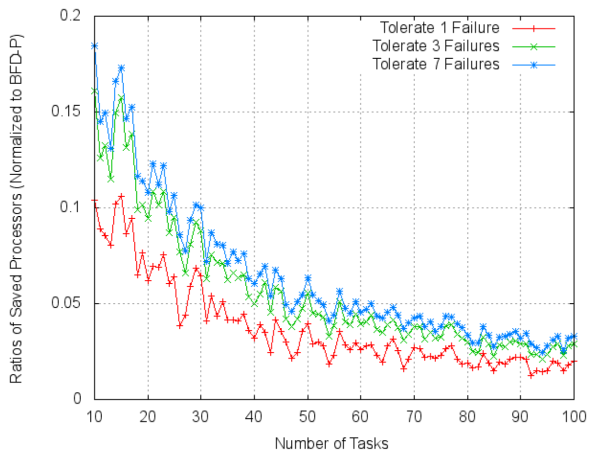
Lemma 6 *Two virtual tasks $\tau^v_{q,j}$ and $\tau^v_{p,j}$, where $q \neq p$, can be located on the same processor.*

Lemma 7 *Two virtual tasks $\tau^v_{q,i}$ and $\tau^v_{p,j}$, where $q \neq p$ and $i \neq j$, can be located on the same processor if $\Gamma^v_{q,i} \cap \Gamma^v_{p,j} = \emptyset$ is satisfied.*

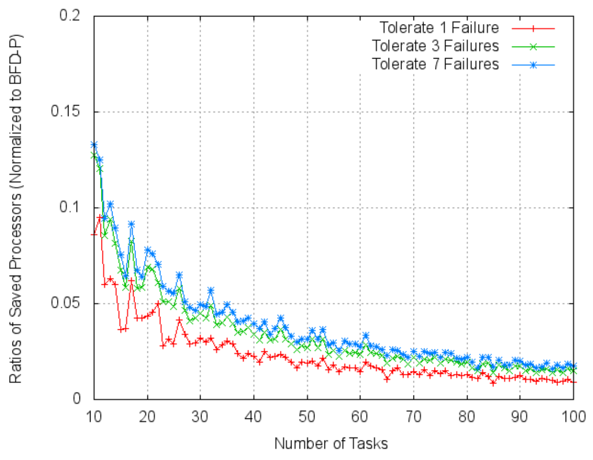
Lemma 8 *$\tau^v_{q,j}$ and τ_i can be located on a same processor if $\tau_i \notin \Gamma^v_{q,j}$ is satisfied.*

Algorithm 5 generateVirtualTask($\Gamma : \{\tau_1, ..., \tau_n\}, \Pi : \{\Pi_1, ..., \Pi_n\}, P, \rho$)

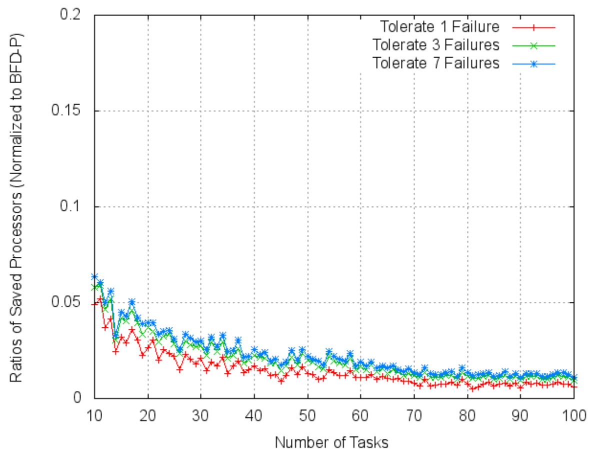
```
1:  $q \leftarrow 0$
2:  for $j = 0$ to $\max_{\forall \tau_i \in \Gamma} (\zeta(i))$ do
3:      for all $\tau_i$ such that $\tau_i \in \Gamma$ do
4:          if $j < \zeta(i)$ and $\tau^c_{i,j} \notin \Gamma^v$ then
5:              $\Gamma^v_{q,j} \leftarrow \{\tau^c_{i,j}\}$ ▷ Generate virtual task $\tau^v_{q,j}$
6:              $q \leftarrow q + 1$
7:              $u^v_{q,j} \leftarrow u_i$ ▷ Set the virtual utilization of $\tau^v_{q,j}$
8:              $\Pi^v_{q,j} \leftarrow \Pi^v_{q,j} \cup \Pi_i$ ▷ Set the placement constraint
9:              ▷ Pick a processor not containing copies of $\tau_i$
10:             for all $P_k$ such that $P_k \notin \Pi_i$ do
11:                 $alloc \leftarrow 0$
12:                 for all $\forall \tau_p$ such that $P_k \in (\Pi_p - \Pi_i)$ do
13:                     ▷ Allocate $\tau_p$ to $\tau^v_{q,j}$ if possible
14:                     if $alloc + u_p \leq u^v_{q,j}$ and $\tau^c_{p,j} \notin \Gamma^v$ then
15:                         $\Gamma^v_{q,j} \leftarrow \Gamma^v_{q,j} \cup \{\tau^c_{p,j}\}$
16:                         $alloc \leftarrow alloc + u_p$
17:                 if $alloc \neq 0$ then
18:                     $\Pi^v_{q,j} \leftarrow \Pi^v_{q,j} \cup P_k$
19:             $\Gamma^v \leftarrow \Gamma^v \cup \Gamma^v_{q,j}$, $\Pi^v \leftarrow \Pi^v \cup \Pi^v_{q,j}$
20: return ($\Gamma^v \cup \Gamma, \Pi^v \cup \Pi$)
```



(a) $u_{max} = 0.3$



(b) $u_{max} = 0.5$

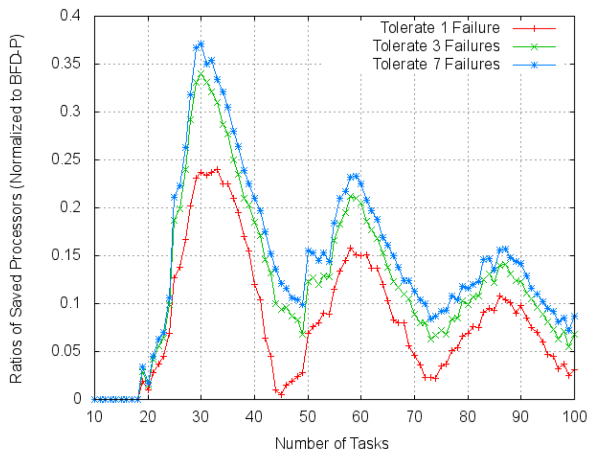


(c) $u_{max} = 0.7$

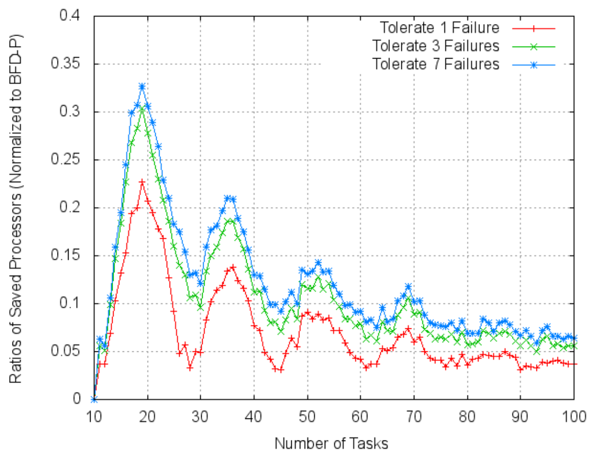
Figure 3.4: Ratios of saved processors when R-BFD is used on a single-node case. Results are normalized to BFD-P.

3.3 Evaluation

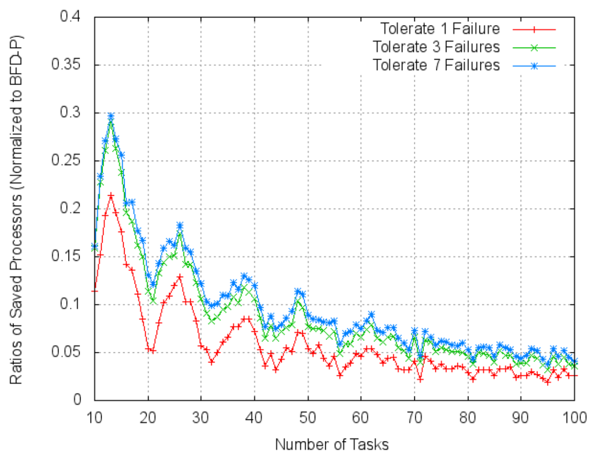
We will now evaluate the performance benefits of using R-BATCH and R-BFD on randomly generated task sets. We analyze task sets with different characteristics by varying the maximum task utilization (u_{max}). We generate random tasks whose utilization is uniformly distributed between 0 and u_{max}. Since we focus on harmonic task sets that can achieve 100% utilization, we do not generate the worst-case execution time and a period for a randomly generated task.



(a) $u_{max} = 0.3$



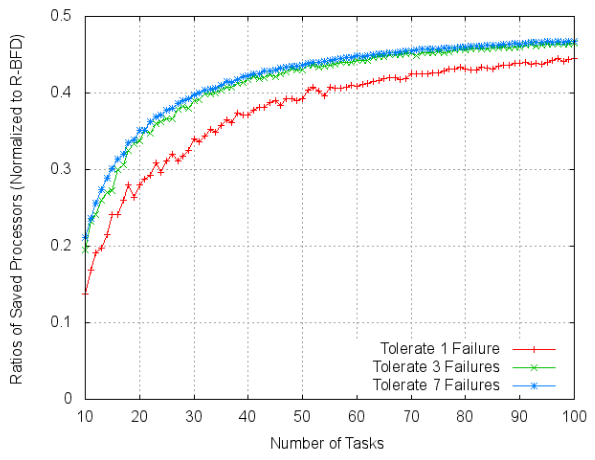
(b) $u_{max} = 0.5$



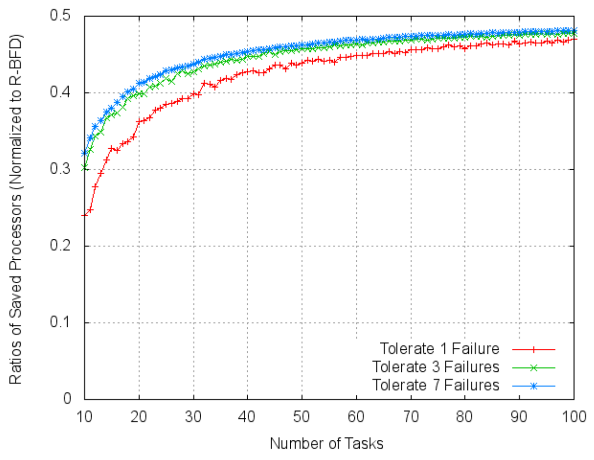
(c) $u_{max} = 0.7$

Figure 3.5: Ratios of saved processors when R-BFD is used on a 4-node case. Results are normalized to BFD-P.

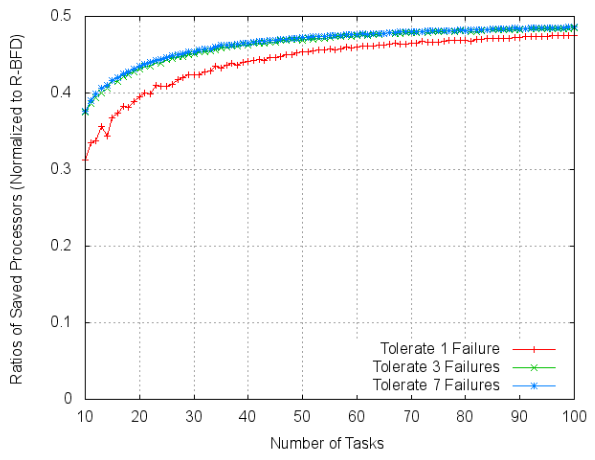
The results presented here are at u_{max} values of 0.3, 0.5, and 0.7. The number of tasks is varied from from 10 to 100. We characterize the performance with respect to tolerating 1, 3, and 7 processor failures by introducing 2, 4, and 8 replicas respectively (including the primary) under R-BFD. With R-BATCH, we set the number of hot standbys for each task to 1, 2, and 4 hot standby replicas including the primary. Any possible remaining failures can be recovered by using a cold standby. Because we are using hot standby and cold standby together, all tasks can



(a) $u_{max} = 0.3$



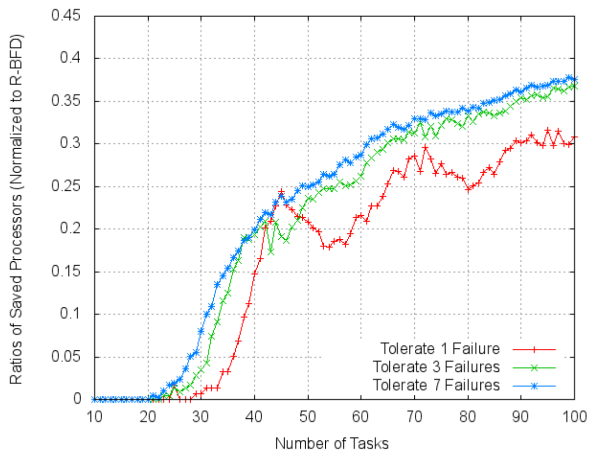
(b) $u_{max} = 0.5$



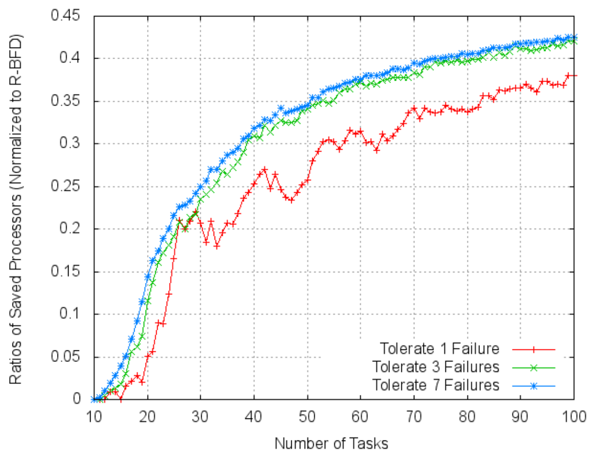
(c) $u_{max} = 0.7$

Figure 3.6: Ratios of saved processors when R-BATCH is used on a single-node case. Results are normalized to R-BFD.

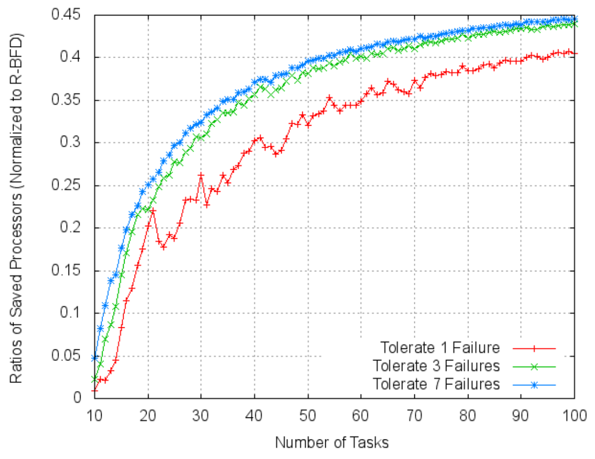
be recovered even if all of them have Hard Recovery requirements. Tasks can be promoted from cold standby to hot standby when failures occur. We provide results from both single-node and 4-node platforms to illustrate the performance benefits, where a 4-node platform has 4 processors per board. Since 4-node platform is augmented at the granularity of boards, it can fail under only a single processor failure out of 4 processors. Each data point is obtained by averaging sum of all results from 50 iterations.



(a) $u_{max} = 0.3$



(b) $u_{max} = 0.5$



(c) $u_{max} = 0.7$

Figure 3.7: Ratios of saved processors when R-BATCH is used on a 4-node case. Results are normalized to R-BFD.

Figure 3.4 and 3.5 show the number of processors saved by R-BFD over BFD-P, normalized to BFD-P on single-node and 4-node cases. Figure 3.4(a), 3.4(b), and 3.4(c) show the results at different u_{max} values. For small u_{max} and task set sizes, R-BFD is most beneficial. R-BFD can save up to *19%* on the single-node case. For the 4-node case, it can save up to *37%* processors compared with BFD-P. Regarding Figure 3.5(a), there are no differences when the number of tasks is changed from 10 to 19. This happens due to the nature of the 4-node case. Because we merge 4 processors into a set of processors, we have more space for allocating tasks. Therefore, until a certain number of tasks fills up 4 processors without replication, there is no difference. This is also a reason for the fluctuations seen in Figure 3.5.

Figure 3.6 and 3.7 show the number of processors saved by R-BATCH over R-BFD, normalized to R-BFD. R-BATCH can save up to *45%* additional processors compared to R-BFD. The interesting observation is that benefits get larger when more tasks are used. This is because we can consolidate more tasks by using virtual tasks. Fluctuations in Figure 3.7(a) are due to the same reasons given above for R-BFD. Another interesting observation is that R-BATCH can save more platforms as u_{max} increases. This is a consequence of the fact that larger virtual tasks can cover more cold standbys.

3.4 Summary

Fault recovery in hard real-time environments requires restoring functionality within pre-specified deadlines. In this work, we have provided a comprehensive solution for guaranteeing reliability requirements with bounded recovery times. We have proposed categorizing tasks based on their recovery time requirement into (i) *Hard Recovery*, (ii) *Soft Recovery*, and (iii) *Best-Effort Recovery*. We then developed a task-partitioning strategy called *R-BFD* for allocating hot standbys to processors in order to improve system reliability. In order to further reduce the resource over-provisioning required for task reliability, we introduced the notion of a cold standby that consumes processing time only when activated. Our consolidated task allocation algorithm called *R-BATCH* can allocate both hot standby and cold standby tasks to meet system-level reliability requirements. Evaluation results suggest that R-BFD saves up to *37%* of the required number of processors, while achieving the same levels of reliability and satisfying the recovery time requirements as the conventional BFD-P heuristic on 4-node platform case. The introduction of *cold standby* can save up to *45%* additional processors using R-BATCH, in comparison to R-BFD with pure hot standbys.

Chapter 4

Fault-Tolerant Computing in the Automotive Context

In this chapter, we exploit our proposed techniques in Chapter 3 to an AUTOSAR-compliant automotive platform to see how applicable they are in the automotive context. To this end, we consider end-to-end delay requirements to capture the nature of automotive platforms that execute multiple *runnables* in sequence to use sensory data to control actuators.

4.1 Motivation

Continuing improvements on embedded systems encourage x-by-wire technology as well as various types of safety and comfort features in future vehicles [97]. In particular, safety features such as lane keeping, lane changing, collision avoidance and driver warning require high dependability because of their safety-critical nature. However, these features require complex hardware and software platforms, and the design and implementation of these features is a challenge. Moreover, because safety features are composed of several subsystems including sensors, processors, and actuators, the whole system needs to be carefully designed to avoid situations where, for example, a single defective sensor can cause an unintended event [98]. Modern multi-core processors can execute several applications in parallel and still the overall system needs to be designed such that single chip failures cannot result in an undesirable situation. Hence, system-level dependability is a key concern in rapidly evolving automotive industries.

Dependable systems can be implemented by using fault-tolerant techniques, and the conventional fault-tolerance technique is to replicate processes, either concurrently or sequentially. Graceful degradation can also contribute to system dependability [99]. However, graceful degradation can involve considering all possible failure scenarios, which can be an exponentially hard problem. Replication, such as Triple Modular Redundancy (TMR) for hardware and N-version programming for software, is a typical approach, but it requires more resources than graceful degradation. Although these techniques have been extensively used in domains such as avionics, space shuttles, and industrial facilities, they may not always be appropriate long-term solutions for automotive architectures due to these exorbitant costs. Our work presents resource-efficient techniques for achieving the required dependability.

Applications in the automotive system are closely connected to the physical environment and use sensor information to obtain current physical information. For example, in Steer-by-Wire (SBW) systems [5], sensors measure information about steering wheel movement, and computational components in microprocessors compute signals for controlling the wheels with the information from sensors. Actuators receive the control signals for the motors directly, and these signals are handled periodically for timely handling of user operations and reactions to the environment.

In order to reflect this nature, we define an *application flow*, which is composed of periodically executing runnables generating information data and events regularly that flow through multiple runnables. An application flow also has an end-to-end delay from input to output. Each runnable is represented by a periodic task [16], ρ_i, which releases a job every T_i units of time, where each job consumes at most C_i units of computation time and should be completed within a relative deadline, $D_i \leq T_i$.

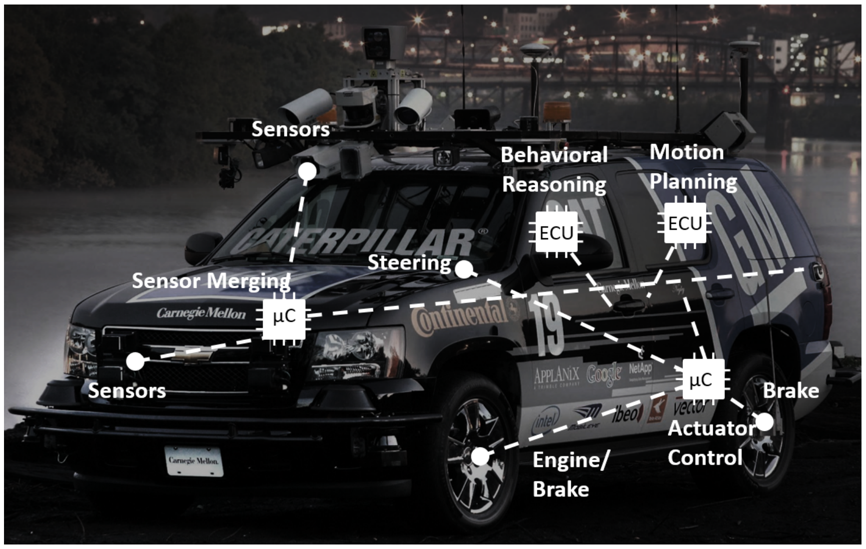


Figure 4.1: The application flow model can be applicable to a driverless car.

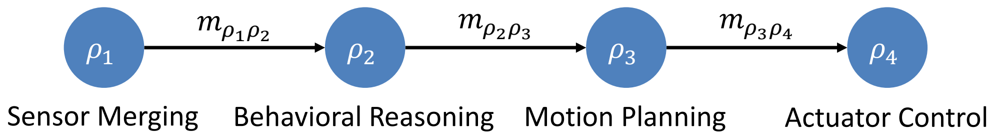


Figure 4.2: An exemplary application flow from Figure 4.1, where ρ represents a runnable, and m denotes a message between two runnables.

Within an application flow, runnables are classified into sensor/actuator runnables and computational runnables. For instance, an actuator runnable controlling the steering wheel motors must run on the Electronic Control Unit (ECU) connected to the motors in an SBW system. Every runnable generates data to be fed to other runnables, except actuator runnables which terminate an application flow. Figure 4.1 and Figure 4.2 show an exemplary diagram of an application flow applied to the autonomous vehicle which won the DARPA Urban Challenge [8].

Handling failures on-demand with bounded recovery time is desirable for real-time fault-tolerant systems. By handling failures within a pre-defined timing boundary, Time-To-Recovery can be bounded, and the system can operate continuously. To limit Time-To-Recovery, our previous research [100] categorized software tasks into three classes: Hard Recovery Tasks, Soft Recovery Tasks, and Best-Effort Recovery Tasks. We apply the same classification to Software-

Components (SW-Cs), where a runnable is a part of an SW-C [101]. An SW-C with the requirement of completing a released job within a D_i units of time, even with the presence of failures, is classified as a Hard Recovery Software-Component (HSC). An SW-C with a more relaxed recovery-time requirement is a Soft Recovery Software-Component (SSC). An optional SW-C that is not critical for system operation and does not require bounded recovery time is classified as a Best-Effort Recovery Software-Component (BSC). The R-BATCH (Reliable Bin-packing Algorithm for Tasks with Cold standby and Hot standby) scheme [100] provides a comprehensive solution that allocates HSC, SSC, and BSC to multi-processors for guaranteeing reliability requirements with bounded recovery times.

From a dependability perspective, a single failure of a runnable within an application flow will affect all of its successors such that the overall application flow requirement is violated. R-BATCH, which uses hot standby and cold standby with stand-alone runnables, cannot be directly utilized for systems with data dependencies. Therefore, we propose a new allocation algorithm, R-FLOW (Reliable application-FLOW-aware SW-C partitioning algorithm), designed for the application flow model. R-FLOW has three properties that distinguish it from R-BATCH:

- A new application flow model that captures communication among SW-Cs,
- Clustering of SW-Cs based on their communication bandwidth needs,
- Controlling the number of hot standbys and cold standbys while guaranteeing the recovery-time requirement of all application flows.

In this chapter, we assume a fail-stop failure model [102], where a failed component is assumed to stop generating any data and a working component can assume control by detecting the lack of output from the failed component. The component that takes over then aims to meet the desired deadline of the failed component.

We also aim at supporting R-FLOW within the AUTOSAR framework [101] for providing dependability. The AUTOSAR framework comprises of application software, a Virtual Functional Bus and a Runtime Environment (RTE). The RTE is responsible for enabling interaction between application software and the operating system along with support for different services within AUTOSAR. Currently, there is no explicit support for recovering from task failures by means of task replications within AUTOSAR. The standard instead assumes that architecture designers will introduce custom extensions to meet such reliability needs. In this chapter, we propose enhancements to the different layers of AUTOSAR to enable fault-tolerance and, therefore, provide support for R-FLOW. This enables fault-tolerance support to be built into the framework by providing an API for fault-tolerance rather than having to rely on custom service modules.

The rest of this chapter is organized as follows. The next section describes the system model with the timing properties of different SW-C replication mechanisms in a multiprocessor environment. Then, R-FLOW, a new SW-C partitioning algorithm, is proposed. Based on the proposed algorithm, R-FLOW, fault-tolerance characteristics within the AUTOSAR framework is summarized. After that, R-FLOW is evaluated by using an AUTOSAR-compliant fault-tolerant platform implementation. Finally, we provide our concluding remarks in the final section.

4.2 System Model and Design

We assume a set of given runnables, Υ, which is composed of n runnables, ρ_1,ρ_2, , and ρ_n. Each runnable ρ_i is a part of an atomic SW-C, ω_j, which may have several runnables. For representing the relationship between a runnable, ρ_i, and an SW-C, ω_j, we define a function Θ such that $\Theta(\rho_i) = \omega_j$ when ω_j contains ρ_i. The inverse function of Θ, Θ^{-1}, returns all runnables contained in an SW-C. The set of SW-Cs, Ω, is also given. For guaranteeing different recovery requirements, we classify Ω into three overlapping sets, Hard Recovery Software Component Set, Ω_H, Soft Recovery Software Component Set, Ω_S, and Best-effort Recovery Software Component Set, Ω_B. Each SW-C, ω_j, is an element of at least one of the three subsets, Ω_H, Ω_S, and Ω_B. The exact definition of these subsets is defined in a later section.

4.2.1 Software Architecture

A subset $A_k \subset \Upsilon$ contains runnables for the kth application flow out of total m application flows. For a certain runnable, $\rho_i \in A_k$, ρ_i also can be an element of A_l, where $k \neq l$. The relationship among tasks in the kth application, A_k, is represented by a directed graph, G_k. Inside the graph G_k, a node u denotes a runnable, $\rho_i \in \Upsilon$, and an edge (u, v) of G_k indicates a data flow from u to v. An edge (u, v) has its own message, m_{uv}, which is generated by the node u, and consumed by the node v in G_k. The allocation of each runnable, ρ_i, to G_k is assumed to be given at design time. Let $u(G_k, \rho_i)$ be the node of G_k allocated to the runnable, ρ_i. If ρ_i is not a member of A_k, the value of $u(G_k, \rho_i)$ is $\emptyset$. This function also works with an SW-C. An application flow, A_k, is represented by a couple (Δ_k, T_k^A) where, Δ_k is an end-to-end delay requirement, and T_k^A is a period of the application A_k. The end-to-end delay is defined as the worst-case delay between the release time of the first executed node in G_k and the completion time of the last executed node in G_k. All runnables in the application A_k share the same period T_k^A.

A runnable, ρ_i, is represented by a quadruple $(C_i, T_i, D_i, \alpha_i)$, where C_i is its worst-case execution time, T_i is its period, D_i is its relative deadline to the release time of each job (instance), and α_i is the ratio of recovery time to relative deadline. The recovery time is defined as the time instant relative to the release time of a failed job, and the failed job must be fully recovered at the recovery time. For instance, if $\alpha_i = 1$, the replica of the runnable ρ_i should recover what ρ_i is supposed to execute within the original relative deadline, D_i. Let R_i denote the response time of a runnable ρ_i, where the response time is the time interval between job release and job completion of ρ_i. Although all runnables in one application have the same period, the deadlines of those runnables will be determined based on the end-to-end delay requirement, Δ_k for A_k.

An SW-C $_j$ is also represented by a quadruple $(C_j, T_j, D_j, \alpha_j)$, but it differs from a runnable in that C_j is set to $\max_{\forall \rho \in \Omega^{-1}(\omega_j)} C_i$, T_j is $\min_{\forall \rho \in \Omega^{-1}(\omega_j)} T_i$, D_j is $\min_{\forall \rho \in \Omega^{-1}(\omega_j)} D_i$, and α_j is $\min_{\forall \rho \in \Omega^{-1}(\omega_j)} \alpha_i$. Let u_j denote the utilization of an SW-C ω_j, and it is defined as $\frac{C_j}{T_j}$. The density of ω_j is defined as $\frac{C_j}{D_j}$ and denoted by d_j. Both u_j and d_j are used for measuring the

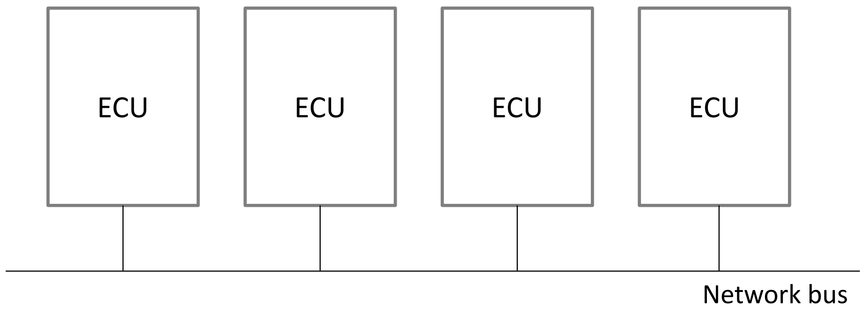


Figure 4.3: Abstracted hardware architecture of an AUTOSAR-compliant platform.

amount of processor resources consumed by ω_j. Every SW-C ω_j can have $\psi(j)$ hot standbys[1], which is represented by $\omega^h_{(j,1)}, \omega^h_{(j,2)}, \ldots, \omega^h_{(j,\psi(j))}$. Either ω_j or $\omega^h_{(j,0)}$ denotes the primary of ω_j. Since our objective is to tolerate π processor failures, each SW-C ω_j also can have $\zeta(j)$, which is $\phi + 1 - \psi(j)$ cold standbys, which are represented by $\omega^c_{(j,1)}, \omega^c_{(j,2)}, \ldots, \omega^c_{(j,\zeta(j))}$.

4.2.2 Hardware Architecture and Fault Hypothesis

We will adopt the abstracted platform architecture shown in Figure 4.3. In this architecture, we will assume that we use a fault-tolerant network such as FlexRay [103] as the underlying in-vehicle network. Leveraging timeliness of such a network can guarantee the bounded delivery of packets generated by each SW-C. In other words, we can focus only on permanent processor failures rather than network failures.

A set of processors (or ECUs), P, is used for running a runnable set, Υ. P is composed of l homogeneous processors, P_1, P_2, $\ldots$, P_l. Then, Π_i is the set of processors utilized by SW-C ω_i and its hot standbys. Each element of Π_i, $\Pi_{(i,j)}$ is the processor allocated to $\omega_{(i,j)}$, the jth hot standby of SW-C ω_i. Two replicas of the same SW-C cannot be allocated to the same processor. We refer to this as a placement constraint. It can be expressed as $\forall i$, $\Pi_{(i,j)} \neq \Pi_{(i,k)}$, where $j \neq k$. Each processor P_k has its own failure rate, f_k, which denotes the probability

[1]In this dissertation, we create a new AUTOSAR task for each replica of SW-C. The reason behind this is that assigning several runnables to one AUTOSAR task can delay the Time-To-Recovery when a failure occurs.

of a permanent failure. We assume homogeneous processors for convenience of presentation, i.e. $f_1 = f_2 = \cdots = f_l = f$. In this chapter, only permanent failures are considered [102]. Occurrences of fail-stop failures of these processors can be detected by using periodic heartbeat signals.

4.2.3 Dealing with End-to-end Delay of an Application Flow

Under the given requirements on end-to-end delay of an application flow and DMS (Deadline Monotonic Scheduling) [15], the deadline of a runnable should be determined. The shorter the deadline of a runnable, the more responsive is the system. However, with a shorter deadline, a runnable has larger density, and the system may need more processors. Hence, our design objective is to pick the longest deadline that meets the end-to-end delay of a given application flow. The end-to-end delay for an application flow is calculated in [104], where a pipeline task model is used on a FlexRay network. For A_k, Δ_k is bounded by the following equation.

$$\begin{aligned}\Delta_k \leq \sum_{i=1}^{n(A_k)-1} \left(R_{u(G_k,\rho_i)} + \left\lceil \frac{T_{m_{u(G_k,\rho_i)u(G_k,\rho_{i+1})}}}{\gamma} \right\rceil \times \gamma + T_{u(G_k,\rho_{i+1})} \right) \\ + (n(A_k)-1) \times \phi + R_{u(G_k,\rho_{n(A_k)})} = \Delta_k^B\end{aligned} \tag{4.1}$$

where, $n(A_k)$ represents the number of elements in A_k, R_ρ is the response time of a runnable ρ, T_m is the period of message m, γ is the communication cycle length of the FlexRay network, T_ρ is the period of runnable ρ, and ϕ is the duration of a static slot in the FlexRay network. Under the assumption that the FlexRay network is synchronized among ECUs, $T_{(i+1)}$ can be omitted because $T_{(i+1)}$ reflects the offset to ρ_i in terms of ρ_{i+1}. The response time of each runnable can be obtained by using the standard response-time test in the AUTOSAR framework as described in [105]. Then, we can obtain the deadlines of runnables by satisfying the following objective function for each application flow.

$$\underset{\forall \rho_i \in A_k}{\text{minimize}} \quad \sum_{i=1}^{n(A_k)} d_i \tag{4.2}$$

$$\text{subject to} \quad \Delta_k \le \Delta_k^B \tag{4.3}$$

$$C_i \le D_i \ \text{ for } \forall i \tag{4.4}$$

In the above objective function, we want to minimize the density d_i, which is defined as $\frac{C_i}{D_i}$, for saving resources while the end-to-end delay requirement is met by Constraint (3). If we find a set of deadlines for each runnable which minimizes the objective function above, those deadlines can be used for allocating runnables to processors. In this chapter, due to the NP-hardness of the given objective function, we use a heuristic which assigns a deadline to each runnable that is proportional to its period such that it follows the RMS (Rate Monotonic Scheduling) priority assignment [16]. Each deadline is assigned by the following equation.

$$D_i = \frac{\left(\Delta_k - \sum_{\forall \rho_j \in A_k} \left\lceil \frac{T_{m_{u(G_k,\rho_i)u(G_k,\rho_{i+1})}}}{\gamma} \right\rceil \gamma \right) \times T_i}{\sum_{\forall \rho_j \in A_k} T_j} \tag{4.5}$$

4.2.4 How to Determine Standby Type

Task replication is a fundamental method for improving the reliability of a target system. In that sense, replicating SW-Cs in the AUTOSAR framework can increase system reliability. SW-Cs on multiple ECUs can recover SW-Cs on failed processors. In this chapter, we consider two techniques for replicating SW-Cs viz. hot standby and cold standby with different timing characteristics [100].

- Hot Standby Approach: This approach uses two or more on-line copies of a certain SW-C. One or more replicas of the SW-C will be active simultaneously. Each replica must be on a different processor, in order to make sure that at least one[2] of them is working when an

[2]If an SW-C is involved in both Ω_H and Ω_S, the SW-C has both of hot standby and cold standby.

ECU failure occurs. When a failure occurs, a replica will take over the task on a failed processor. At least one of the backup replicas should meet the original deadline when a failure occurs. Hot standbys are released synchronously with that of the primary copy and execute in parallel with the same deadline. Any hot standby can be promoted to be the primary after a primary fails.

- Cold Standby Approach: In this case, replicas are not active until triggered due to failures. In other words, they only utilize memory, but not processor cycles under normal operation, and they are activated on demand when failures occur. Only the state information of each primary copy needs to be shared and updated among replicas. When failures occur, they should be detected as soon as possible, and the SW-Cs on failed ECUs should be recovered using replicas within a predefined time. The main benefit of using cold standbys is that the cold standbys for processors other than those hosting a certain SW-C can be consolidated. Suppose an SW-C set $\Omega = \{\omega_1, \omega_2\}$, and both SW-Cs have a utilization of 0.6 each. They cannot t into one processor together. Hence, in order to tolerate one failure per SW-C by using only hot standbys, two more processors will be required. A cold standby, however, can use only one processor since the cold standbys for both ω_1 and ω_2 can be co-resident on a single processor if their primaries are running on different processors.

The benefit of using a hot standby is its ability, when the primary fails, to meet the original deadline with a smaller timing penalty than the cold standby approach, since all hot standbys are running concurrently with the primary component. However, hot standbys require additional resources, $\psi(j)$ times u_j for ω_j. For the cold standby approach, when there are no failures, only the primary copy is executed. Since processor failures can be detected through the properties of fail-stop processors, the failure of a primary copy triggers the execution of a cold standby unless a hot standby is running. For bounding recovery time, we use Transient Overload Density (TOD), which is extended from Transient Overload Utilization defined in [100]. TOD is the additional processor utilization required for an SW-C that is recovered through the cold standby approach. TOD is the required additional resource for re-executing an SW-C on a new processor within the remaining time, $\alpha_j D_j - R_j$, after a processor $\Pi_(j, 0)$ fails. Let d^t_j denote the TOD of ω_j. Then, we can use the result that $d^t_j \leq d_j$, when $_j \geq 2$ from [100] to determine the type of replicas for ω_j. By using a large α_j, d^t_j can be relaxed. The importance of this result is seen in the following example. Suppose that an SW-C ω_j with $\alpha_j = 1$ uses a standby. Then, the backup copy of ω_j should meet the condition $D_j - R_j \geq C_j$. Therefore, $d^t_j = C_j/(D_j - R_j)$. Since $D_j - R_j < D_j$, $d^t_j \geq d_j$, where the hot standby approach might be appropriate. For $\alpha_j > 1$, the utilization of the standby goes down, but the recovery time goes up. Hence, based on the value of α_j, we can choose the type of standby. An important observation regarding the cold standby approach is that the processor running the backup should have enough unused utilization. The reserved slack for cold standby replicas can be used by any SW-C in the presence of failures, but the reserved utilization for hot standby cannot be utilized by other SW-Cs. A cold standby can also be promoted to a hot standby if the primary of ω_j fails and the current number of ω_j is less than $\psi(j)$. Since at least one hot standby for a certain ω_j can meet its original deadline, D_j, the system can be ready to tolerate another potential failure if a cold standby can be promoted to a hot standby. By using this approach, we can tolerate as many as π failures for ω_j.

4.3 Fault-Tolerant SW-C Allocation with Application Flows

This chapter proposes a comprehensive method for allocating SW-Cs[3] to processors while meeting the requirements on reliability and end-to-end delay. The proposed scheme is composed of two complementary phases: flow-aware allocation and reliability-aware allocation. The flow-aware allocation tries to co-locate SW-Cs which have dependencies on each other such that the end-to-end delay can be reduced. After all primary SW-Cs are allocated, their replicas can be

[3]Allocating a runnable in the AUTOSAR framework implies that the corresponding SW-C is also allocated. In other words, all runnables of a SW-C should be assigned to one processor. The replication of a runnable therefore implies a SW-C replication.

placed on appropriate processors while satisfying the placement constraint. Hot standbys and cold standbys will be allocated differently because cold standbys only use up memory, but not processor utilization.

4.3.1 SW-C Allocation with Application Flows

Using fewer processors than conventional allocation methods is a major goal in this chapter. The allocation of SW-Cs to processors during design time is a well-known bin-packing problem [95]. Each SW-C ω_j is treated as an item to be packed with a size, utilization value u_j, and these items will fill up one or more processors, each having a total capacity of 1 under the assumption that the SW-C periods are harmonic. If an item does not fit into the remaining space of any available processors, one more processor is added. Since the bin-packing problem is known to be NP-hard [95], there are various types of heuristics such as BFD (Best-Fit Decreasing), FFD (First-Fit Decreasing), WFD (Worst-Fit Decreasing), and NFD (Next-Fit Decreasing). However, none of these heuristics considers dependencies among SW-Cs for using a fewer number of processors.

In the previous section, we stated that co-locating SW-Cs that have dependencies on each other can reduce the amount of required resources. Suppose that we are given an application A, which is composed of a set of runnables, $\Upsilon : \{\rho_1, \rho_2, \rho_3\}$. The graph G for A depicted in Figure 4.4, shows that A has a pipeline task model. Each runnable has the period of $100ms$, $100ms$, and $100ms$, and the worst-case execution time of $10ms$, $10ms$, and $10ms$, respectively. A set of SW-Cs corresponding to Υ is given in $\Omega : \{\omega_1, \omega_2, \omega_3\}$, where each SW-C has only one runnable. The length of the communication cycle in FlexRay is assumed to be $20ms$, and the slot duration is $10\mu s$, which is negligibly small. The end-to-end delay Δ for A should be

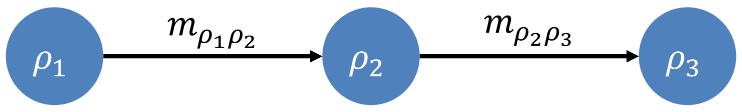

$$\begin{tikzcd} \rho_1 \arrow[r, "m_{\rho_1\rho_2}"] & \rho_2 \arrow[r, "m_{\rho_2\rho_3}"] & \rho_3 \end{tikzcd}$$

Figure 4.4: The graph G for the application A.

equal to or less than $300ms$. Suppose that all three SW-Cs communicate via a FlexRay network. Then, if all job instances are completed by their deadlines, the given end-to-end delay cannot be satisfied due to the communication delay. For example, if a job instant of ω_1 which is released at $0ms$ finishes at $100ms$, a generated message by ω_1 will spend one FlexRay slot at least for transmission and may arrive at $120ms$ due to the communication cycle, where all these effects are reflected in Equation 4.5. If the same behavior happens on the second processor running ω_2, the end-to-end delay will not be satisfied. Therefore, in order to meet the requirement, the relative deadline of each SW-C should be $80ms$, $80ms$, and $80ms$, which gives a large density value, $0.375 \left(\frac{1}{8} + \frac{1}{8} + \frac{1}{8}\right)$. If we assume, however, that all SW-Cs are allocated to the same processor, a deadline of $100ms$ would be enough for each SW-C giving 0.3 as the total density of the given SW-C set, representing a utilization savings of 25%. This example illustrates that co-locating SW-Cs communicating with each other on a FlexRay network can save a substantial amount of resources.

We propose a technique called FBFD (Flow-BFD), a variant of BFD which considers dependencies among application flows. We use BFD as a base-line algorithm rather than other heuristics such as WFD and NFD due to its well-known worst-case behavior [95]. For SW-C allocation, the original BFD (1) sorts the SW-Cs in descending order of their densities, (2) allocates the next SW-C into the processor that it best ts into, (3) adds a new processor if an SW-C does not t into any current processor, and (4) iterates this procedure until no SW-Cs remain. Here, we used SW-Cs instead of using runnables because $\forall \rho_i \in \omega_j$ should be allocated to a processor together.

FBFD uses a flexible definition of items to be packed. It tries to allocate all corresponding SW-Cs of an application flow as a single item when possible. Else, it splits these consolidated items when necessary. FBFD starts by combining SW-Cs as part of the same application flow, where these composite-SW-Cs can include several application flows because one SW-C can be used by several application flows. These consolidated SW-Cs are sorted in descending order of

Algorithm 6 FBFD($\Omega : \{\omega_1, \omega_2, ..., \omega_n\}, \Pi : \{\Pi_1, \Pi_2, ..., \Pi_n\}, P$)

```
1:  $\Omega^C \leftarrow \emptyset$
2:  ▷ *Consolidate SW-Cs based on application flows*
3:  **for** $i = 1$ to $n$ **do**
4:      **if** $\omega_i \notin \Omega^C$ **then**
5:          Find a composite-SW-C $\omega_j^c$ communicating with $\omega_i$
6:          **if** $\omega_j^c$ exists **then**
7:              $\Omega_j^C \leftarrow \Omega_j^C \cup \{\omega_i\}$
8:          **else**
9:              ▷ *Generate a new composite-SW-C*
10:             $\Omega^C_{n(\Omega^C)} \leftarrow \{\omega_i\}$
11:             $\Omega^C \leftarrow \Omega^C \cup \Omega^C_{1+n(\Omega^C)}$
12: Sort $\Omega^C$ in descending order of density
13: $i \leftarrow 1$
14: **while** $\Omega^C \neq \emptyset$ **do**
15:     **if** $i = 0$ **then**
16:         ▷ *Split the biggest composite into two pieces such that one piece can fit into the proces-
            sor which has the largest remaining spce and satisfies the placement constraint*
17:         $\Omega^C \leftarrow \Omega^C - \Omega_j^C + \Omega^C_{1+n(\Omega^C)}$
18:         Update $\Pi_j$ such that $\forall \omega_j \in \Omega_j^C - \Omega^C_{1+n(\Omega^C)}$
19:         Add a new processor $P_{|P|}$
20:     ▷ *Satisfying the placement constraint*
21:     For $\Omega_i^C$, find a best processor, $P_k$ such that $P_k \notin \Pi_j$ and $\forall \omega_j \in \Omega_i^C$
22:     **if** $P_k$ exists **then**
23:         ▷ *Allocate $\omega_j^c$ to $P_k$*
24:         Update $\Pi_j$ such that $\forall \omega_j \in \Omega_i^C$
25:         $\Omega^C \leftarrow \Omega^C - \Omega_i^C$
26:     **else**
27:         Continue
28:     $n \leftarrow n + 1$
29:     $n \leftarrow n \mod |\Omega^C|$
30: **return** $(P, \Pi)$
```

size in terms of their total densities. Then, FBFD fits the next SW-C into the best processor. If there is no processor into which an SW-C fits into, this SW-C is set aside and FBFD searches for any unallocated SW-Cs in the list which can fit into the current processors. If FBFD cannot find any such SW-C, it picks the biggest unallocated composite-SW-C among remaining composite-SW-Cs, and splits it into two pieces such that at least one piece can be allocated to the remaining space. In this case, the sum of densities of two pieces will be greater than the size of the original combined SW-Cs due to the communication delay. A new processor is added if necessary, and the remaining piece will be sorted again in descending order of sizes with other remaining unallocated composite-SW-Cs. These steps will be iterated until no SW-Cs remain. This procedure is also described in Algorithm 6.

4.3.2 Fault-Tolerant SW-C Allocation

We now extend FBFD to make it into a reliability-aware allocation scheme, R-FLOW. For achieving this goal, we (1) allocate replicas of SW-Cs and (2) spread those replications across different processors while satisfying the placement constraint. Different forms of replicas, hot standbys and cold standbys, will be allocated depending on the application flow properties. The number of replicas meeting the system reliability requirements can be obtained using Equations 3.1 derived in Chapter 3.

Allocating Hot Standby/Cold Standby with FBFD

Suppose we have the same exemplary set of SW-Cs as given in the previous subsection. The only difference is that ω_3 has one hot standby, $\omega^h_{3,1}$. Even if all the primary SW-Cs are allocated together by FBFD, the placement constraint brings a new processor for allocating $\omega^h_{3,1}$. The deadline of $\omega^h_{3,1}$ should be recalculated in this case because the pipeline, $\omega_1 \to \omega_2 \to \omega^h_{3,1}$, from Figure 4.4 should meet the end-to-end delay requirement. Since the periods of T_1 and T_2 are already known to be $100ms$ and $100ms$, respectively, Equation 4.5 can be used for determining D_3 as $75ms$, which generates a big item in terms of density. As this example shows, the deadline of each hot standby should be recalculated for allocating it. This deadline recalculation also affects the sorting which happens at the beginning of the allocation because an SW-C with low density does not necessarily mean a hot standby with low density. Therefore, after all primaries are allocated, the SW-Cs are sorted again in decreasing order, in accordance with BFD. This procedure of deadline recalculation and resorting is also executed whenever all jth replicas are

Algorithm 7 R-FLOW$(\Omega : \{\omega_1, \omega_2, ..., \omega_n\})$

```
1: ▷ Allocate the primaries first
2: $(P, \Pi) \leftarrow FBFD(\Omega \leftarrow \{\omega_1, \omega_2, ..., \omega_n\}, \Pi \leftarrow \emptyset, P \leftarrow \emptyset)$
3: ▷ Allocate the replicas one by one
4: for $j = 1$ to $\max_{\forall \omega_k \in \Omega}(\psi(k))$ do
5:     Recalculate the deadlines
6:     Recalculate the number of hot standbys
7:     ▷ Ignore SW-Cs that do not need the $j^{th}$ replica
8:     $\forall \omega_i$ such that $\psi(i) < j$, $\omega^h_{i,j} \leftarrow \emptyset$
9:     $(P, \Pi) \leftarrow FBFD(\Omega \leftarrow \{\omega^h_{1,j}, \omega^h_{2,j}, ..., \omega^h_{n,j}\}, \Pi, P)$
10: $(\Omega, \Pi) \leftarrow$ generateVirtualTask$(\Omega, \Pi, P, \pi)$
11: $(P, \Pi) \leftarrow FBFD(\Omega, \Pi, P)$
12: return $(P, \Pi)$
```

allocated. Due to the recalculation of deadlines, the TOD of each SW-C is also affected. If the TOD of an SW-C becomes negative and it does not have any hot standby, the SW-C cannot be recovered within its required recovery time. In this case, the number of hot standby should be adjusted accordingly. Reflecting these properties, we propose a new allocation method, R-FLOW, which allocates primaries, hot standbys, and cold standbys with application flows. The pseudo-code for R-FLOW is described in Algorithm 7, where `generateVirtualTask()` is described in [100].

4.4 Fault-Tolerance with AUTOSAR

We now describe how an implementation of our approach can be integrated into the AUTOSAR framework. We made modifications to various modules within AUTOSAR to enable our fault-tolerance algorithm. This section gives a description of the changes made as well as a description of services introduced as part of the implementation.

To support the replication of SW-Cs, the AUTOSAR Software Component Template was modified to introduce new properties. An Automotive Safety Integrity Level (ASIL) value is assigned to every SW-C to represent the level of safety required for that particular component. This enables R-FLOW to pick the replication scheme to apply, assuming that the replication of an SW-C results in the replication of all of its runnables. A new structure was added to the Software Component Template which provides a description of the internal data representing the current state of the runnable. This is required to enable cold standbys to remain synchronized with the primary component to ensure that the cold standby has the most current state available when it is activated. To this end, a property describing the maximum initialization time of the SW-C is added which describes the amount of time needed for the cold standby to produce valid data.

A health status module was added to the AUTOSAR ECU Specification and it is responsible for sending the ECU status to all other ECUs. This is relayed through the AUTOSAR Communication Service (COM) which is responsible for communicating the health status to all ECUs. The appropriate mechanisms for COM are put in place by introducing a health message which is broadcasted using the existing COM API. Several callbacks were added to the AUTOSAR Runtime Environment (RTE). One of them is a callback function from the COM module regarding the health status of all ECUs. This callback function is responsible for activating any replicas that reside on this ECU depending on the status of other ECUs and as part of the on-line procedures described in the previous section.

There are several assumptions regarding offline analysis and synthesis. First of all, it is assumed that all runnables within SW-Cs are executed periodically within the context of an AUTOSAR Task. Secondly, the runnable to Task Mapping exists before R-FLOW is used for replication and allocation. Lastly, the ECU description is assumed to be already available as part of the RTE Generation process. This provides R-FLOW with the description of available resources for task allocation. An example depicting how replication is done within the System

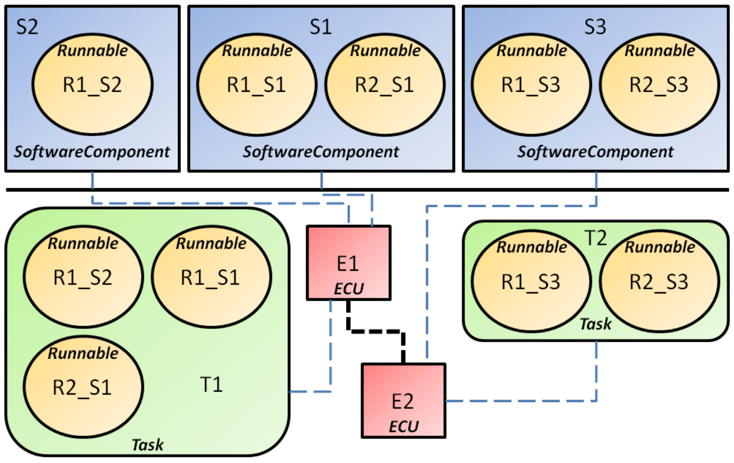


Figure 4.5: An example configuration without replication.

Model is shown in Figure 4.5. Here, we have three SW-Cs S1, S2, and S3, two AUTOSAR Tasks T1 and T2, and two ECUs E1 and E2 which are connected through a network. Suppose that R-FLOW decides to replicate S1, giving component S1. This results in the replication of its runnables R1_S1 and R2_S1, producing R1_S1 and R2_S1 respectively. Given that S1 is mapped to E2 as per R-FLOW, we now have to reschedule the Tasks that need to run on the ECUs. There are two approaches to deal with mapping of the replicated runnables to a Task. New Tasks can be created or the runnables can be assigned to existing Tasks. This is left up to the System Designer to decide or can be automated by a tool using predefined rules such as creating a new task for each replicated runnable. Tasks can contain multiple runnables to capture any explicit ordering present as part of a task schedule on the original ECU. In this example, a new Task T3 is created that contains the replicated runnables. The new configuration is depicted in Figure 4.6.

4.5 Evaluation

We developed an experimental platform to evaluate our approach and look at overheads and costs associated with fault-tolerance. A real automotive platform was necessary to show the complexities involved in adding fault-tolerance to the system. This section describes the system

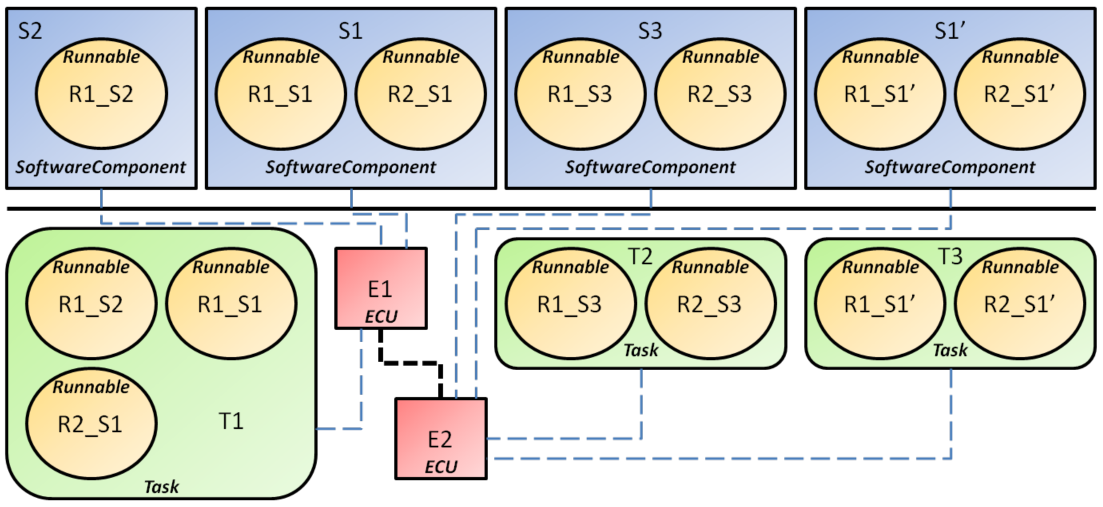


Figure 4.6: An example configuration with replication.

architecture in detail.

4.5.1 Hardware

The hardware architecture was built as part of a testbench as shown in Figure 4.7. The computational architecture is comprised of five Softec HCS12X development kits using the MC9S12XDP512 processor from Freescale [106]. The ECUs run at 50Mhz and use a 12V supply. Daughter boards from Freescale are used for FlexRay connectivity. The ECUs are connected using a dual-channel FlexRay bus and two Full-Speed CAN networks. One of the five ECUs acts as the system gateway and is connected to a PC using two RS232 channels. This ECU acts as the fault injection module and is also used to collect data from the system for diagnosis and analysis. The types of faults that can be injected include communication shutdown, shorting of bus channels, shutdown and restart of ECUs, and injection of faulty network bus messages. These faults can be controlled using a PC interface and data collected by the system are analyzed at the PC as well.

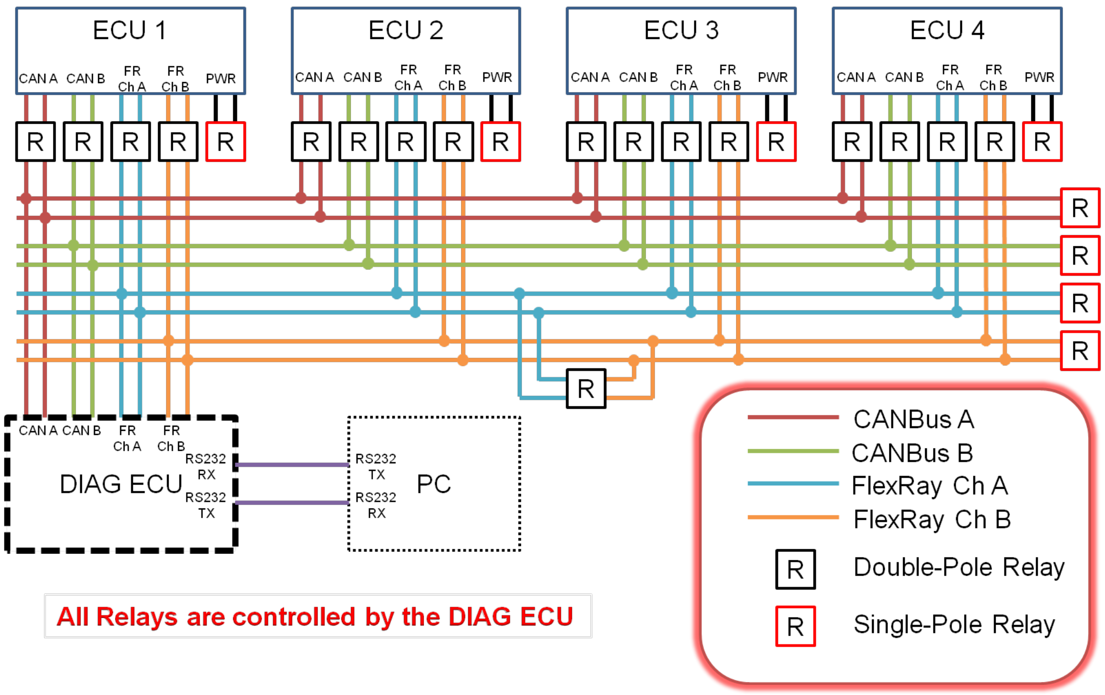


Figure 4.7: Testbench architecture.

4.5.2 Software

The basic software running on the hardware includes the RTA-OSEK [107] operating system and generated code from SysWeaver [15] using an AUTOSAR Code Generation module that was added to SysWeaver. The code generated conforms to the AUTOSAR 4.0 specifications and produces the minimum required implementation to produce a working system. All the relevant modules including the RTE, COM, PDU Router and SW-C supplementary headers are generated. An OIL file for configuration of the RTA-OSEK OS is also generated for each ECU. This generated code includes the necessary code modifications required as part of the fault-tolerance support described in the previous section. Integration was added for R-FLOW within the Fault-Tolerance View of SysWeaver to produce the necessary replicas and SW-C allocation. Code is then generated for every ECU along with a configuration file for the Gateway ECU. The Freescale CodeWarrior compiler for the HCS12X is then used along with the RTA-OSEK configuration tool to produce an executable for each ECU. This is done automatically by SysWeaver.

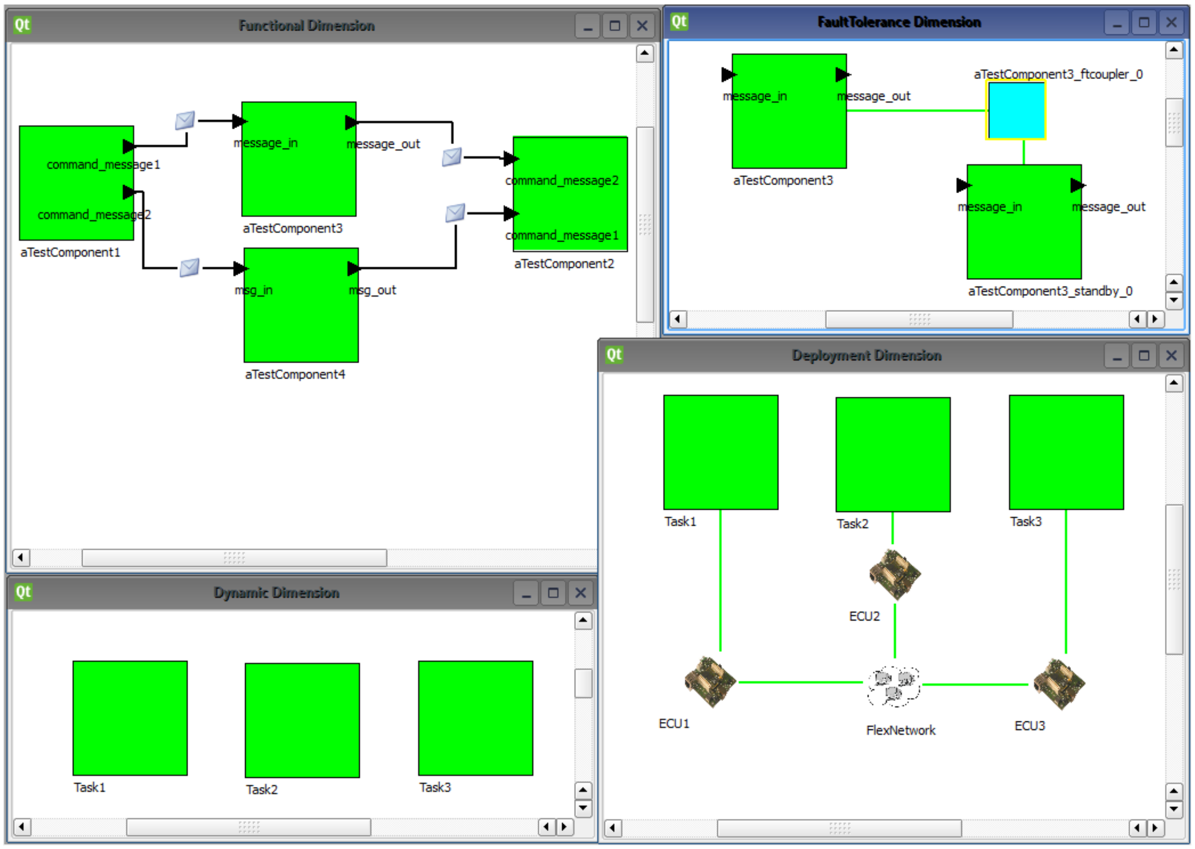


Figure 4.8: SysWeaver system model.

The system model is created within SysWeaver and comprises of SW-Cs that were designed in the tool as well. The Functional View within SysWeaver comprises of communicating SW-Cs. Each SW-C has the relevant AUTOSAR properties associated with it, including the information required by R-FLOW. For this chapter, we concentrate on Sender-Receiver Communication as the only communication mechanism between SW-Cs since a chain of communicating SW-Cs constitutes an application flow. The Deployment View within SysWeaver consists of the hardware configuration including the ECUs and any network buses within the system. Each ECU has properties associated with it as described in the AUTOSAR ECU Description. A Dynamic View exists which contains AUTOSAR Tasks, where each AUTOSAR Task contains a Schedule Table of runnables with their respective Task offset and periodic intervals. Given these properties, R-FLOW is then invoked to produce the required Replicas and Task Allocation. The Fault

Tolerance View in SysWeaver shows the replicas produced along with the type of replication, and the SW-C allocation can be seen in the Deployment View. Figure 4.8 shows an example system model in SysWeaver. The figure shows 2 application flows, A and B.

4.5.3 Results

In this section, we evaluate the performance of FBFD relative to BFD, which does not consider the effects of application flows. Then, we will show the benefits of using R-FLOW on randomly chosen application flows. We analyze the characteristics of these schemes by varying the number of application flows and the number of SW-Cs in an application flow. Our experiments pick different end-to-end delays: 500ms, 1000ms, 1500ms, or 2000ms. In order to get the period value for each SW-C within an application flow, we divide the end-to-end delays by the number of SW-Cs in the flow. Then, the worst-case execution time of each SW-C is randomly chosen such that the utilization of each SW-C is uniformly distributed between 0% and 30%. The number of application flows is itself varied from 10 to 20, and the number of SW-Cs in an application flow is varied from 10 to 15 in each experiment. In all our experiments, the communication cycle length is set at 20ms and each data point is averaged after 500 iterations.

The performance metric used for comparison is the ratio of saved processors which is defined as (n(BFD)-n(FBFD))(n(BFD)) , where n(A) means the number of required processors when scheme A is used. Higher the value of this metric, better is the performance of the scheme under consideration.

For R-FLOW, while the same parameter variations above are applied, we also conduct experiments on using only hot standbys or only cold standbys for guaranteeing system reliability. We vary the number of tolerated failures from 1 to 4, yielding a total of 2 to 5 copies due to the inclusion of the primary.

Figure 4.9 and Figure 4.10 show the number of saved processors when FBFD is used, normalized to the number of processors required by BFD. Figure 4.9 depicts the ratio of processors

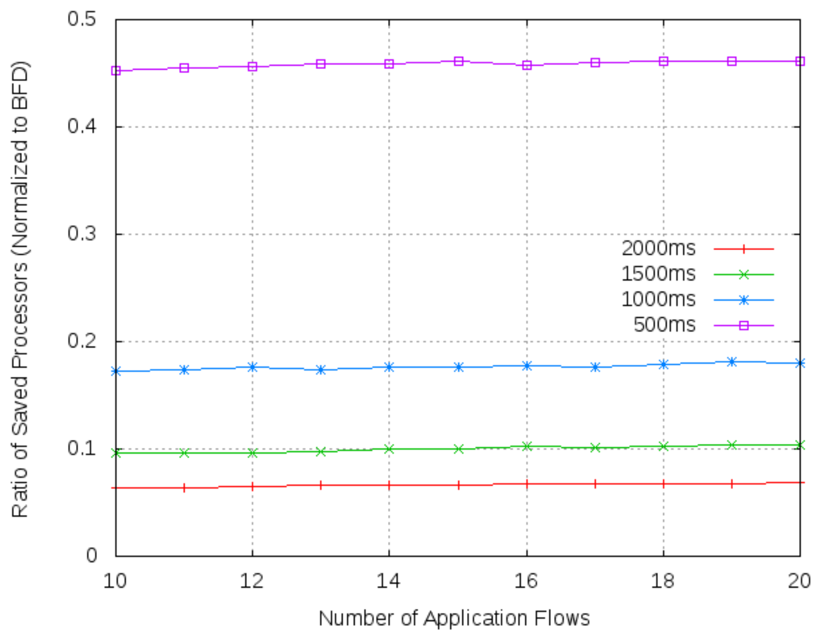


Figure 4.9: The ratio of saved processors when we use FBFD under varying number of application flows and fixed number of SW-Cs per application flow.

saved when the number of application flows and the end-to-end delay are varied. The number of SW-Cs in each application is fixed at 10. As seen in Figure 4.9, FBFD can save a substantial number of processors (up to 45% processors) when the end-to-end delay is 500ms. FBFD can save more processors when the end-to-end delay is shorter because the overhead of communication on an application flow represents a larger ratio of the delay when the end-to-end delay is shorter. It can also be seen that the number of application flows does not affect the performance. This means that the size of an SW-C set does not play a major role. Figure 4.10 presents the results of an experiment where the number of SW-Cs in an application flow is varied from 10 to 15. Again, a large number of processors (up to 56% processors) can be saved when the end-to-end delay is 500ms. As shown in Figure 4.10, a larger number of stages in an application flow have a greater impact on the performance of this algorithm because of the bigger impact of communication delays on shorter end-to-end delays.

Figure 4.11 and Figure 4.12 show the percentage of saved processors when R-FLOW is used,

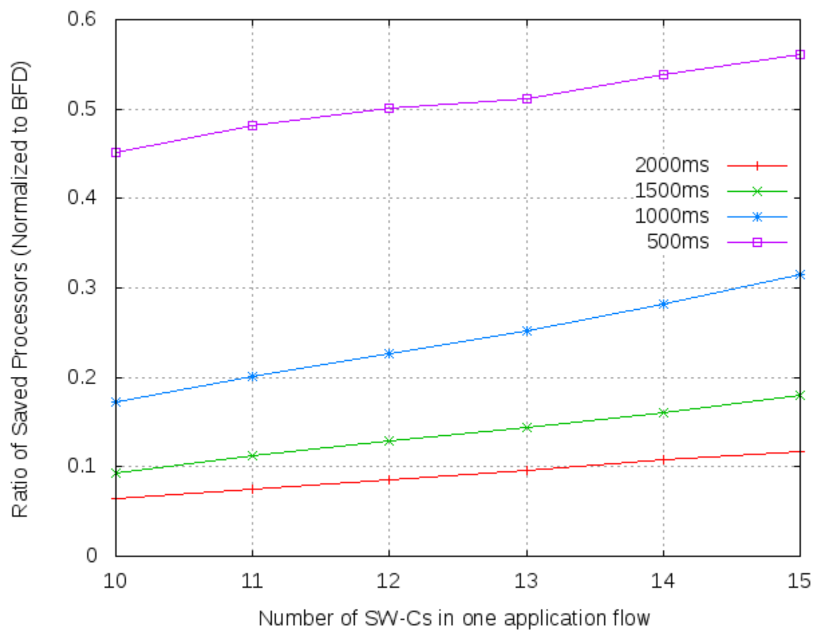


Figure 4.10: The ratio of saved processors when we use FBFD under fixed number of application flows and varying number of SW-Cs per application flow

normalized to R-BATCH. In each of these experiments, the number of tolerated processor failures is varied from 0 to 4, where tolerating 0 processor failures is equivalent to FBFD. The number of applications and the number of SW-Cs in an application flow are both fixed at 10. Figure 4.11 captures the results for the experiment when only hot standbys are used. As seen in the figure, R-FLOW can save up to about 60% of processors when the end-to-end delay is 500ms. The end-to-end delay is also the dominant performance factor when hot standbys are used. The rate of improved savings growth is slow because the density (the ratio of computation time to deadline) of a hot standby is different from that of the primary when R-FLOW is utilized. This is not true when R-BATCH is used. Since the size of a hot standby for R-FLOW is larger due to additional communication delays between the primary and the hot standby, the number of saved processors is not increased as more replicas are introduced to tolerate more failures. Figure 4.12 represents the case where, only cold standbys are used for recovering from processor failures, and has a different trend. The ratio of saved processors does not vary much as the end-to-end delays of

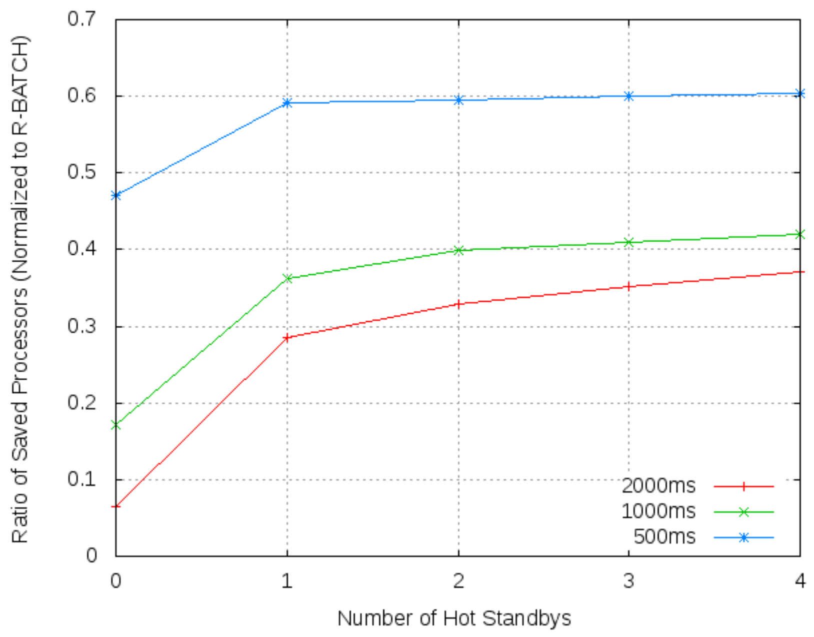


Figure 4.11: The ratio of saved processors when we use R-FLOW in order to tolerate varying number of processor failures with hot standby.

application flows decrease. The reason behind this is that virtual SW-Cs recover several SW-Cs simultaneously and an SW-C with a higher density can save more. Therefore, the effect of end-to-end delay on the ratio of saved processors is negligible.

In summary, R-FLOW can save a substantial number of processors (up to 60% processors) relative to the required processors by the R-BATCH scheme.

4.6 Summary

In this chapter, we have proposed a processor assignment methodology called R-FLOW for allocating SW-Cs while the end-to-end delay of application flow and the given reliability requirement are guaranteed. We have defined a new model using an abstraction called an application flow, which enables timing analysis. We have also described the classification of SW-Cs based on their fault-tolerance requirements. The classification and application flow models are used within R-

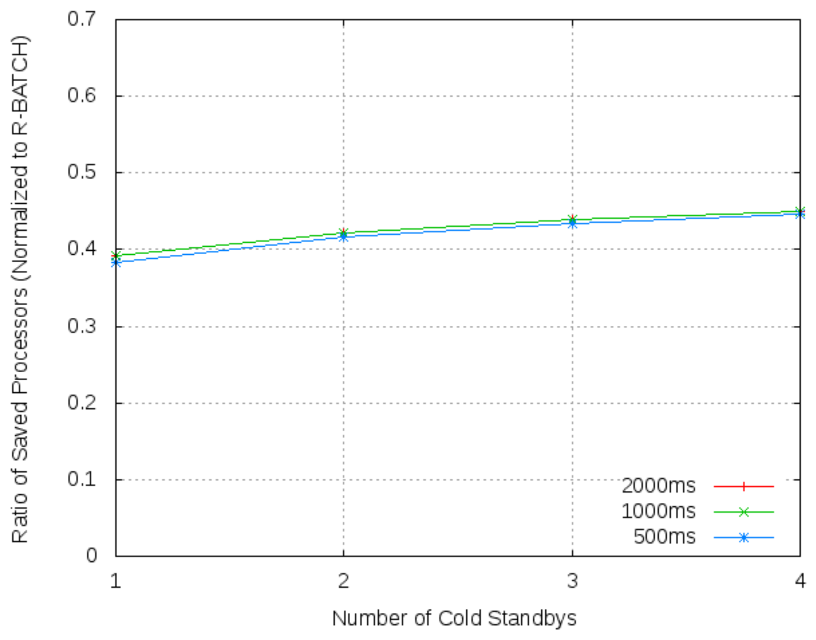


Figure 4.12: The ratio of saved processors when we use R-FLOW in order to tolerate varying number of processor failures with cold standby.

FLOW, an application flow-aware SW-C partitioning algorithm for improving system reliability. Our results have shown that R-FLOW results in a savings of up to 45% of processors when only primary components are allocated. If replicas are used to enhance reliability, savings of more than 60% of processors can be achieved as compared to our earlier scheme called R-BATCH, while satisfying the same level of reliability requirements. Finally, we have described how R-FLOW can be used within the AUTOSAR framework, and have implemented this algorithm within the SysWeaver tool from Carnegie Mellon resulting in automatic code generation for an AUTOSAR-compliant system.

As our next steps, we will focus more on improving the on-line performance of R-FLOW by introducing a new protocol, which is responsible for managing the primary, hot standbys, and cold standbys of each SW-C. We will also compare our approach to the methods defined as part of the ISO26262 standard [108] on Functional Safety, and investigate compliance with the requirements and implementation aspects of the standard.

Chapter 5

Tasks with Continually Varying Periods

Cyber-Physical Systems (CPS) embed computing and communication capabilities in all types of physical objects. Embedded and real-time systems are now essential to control the physical environment, to monitor the timing of dynamic processes taking place in it, to efficiently coordinate

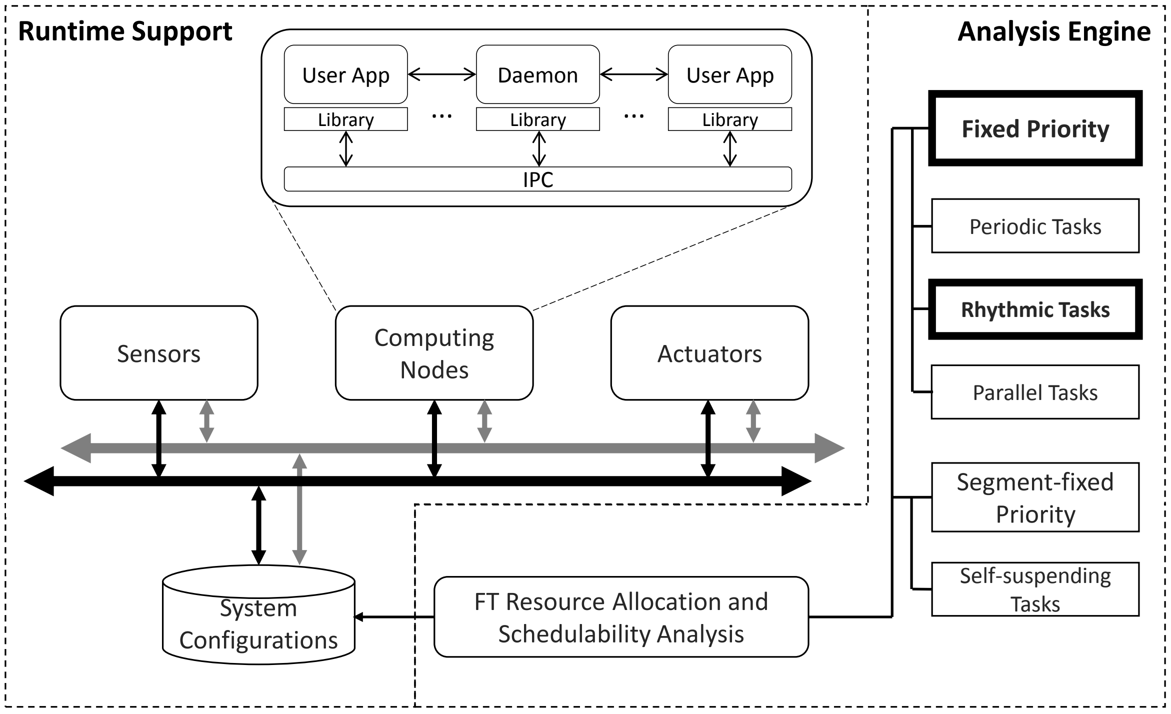


Figure 5.1: Tasks with continually varying periods in the dissertation overview.

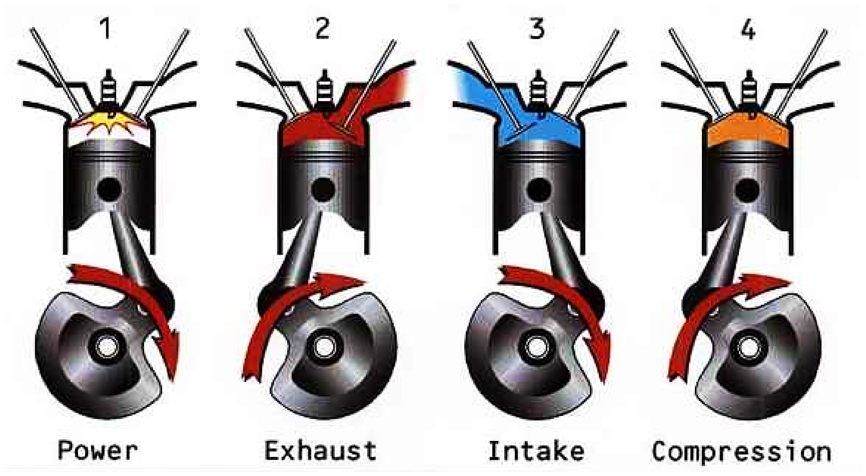


Figure 5.2: Four-stroke cycle in gasoline engines [110].

CPS operations and most importantly to ensure safety. This trend will only continue and in fact is expected to accelerate [109].

Among many applications of CPS such as aerospace systems, building and industrial infrastructure control, medical devices, robotic systems and transportation vehicles, automotive sub-systems such as engine control and chassis control must operate in real-time. The embedded *control* system in a car is also safety-critical and requires a high level of confidence in system correctness. In such systems, a critical task not meeting its timing deadline can lead to system failure.

Specifically, the engine and transmission in a car together form the car's powertrain, which is controlled by Powertrain Control Module (PCM) software using closed-loop control. At *periodic* intervals, software calculates the engine speed and position, determines the next time to fire a spark signal, and based on speed-change commands from the driver, adjusts settings for fuel flow. The software then senses the exhaust system to determine the effectiveness of the combustion process as depicted in Figure 5.2. As the engine runs faster, the fuel intake cycle gets shorter, and the frequency of calculating the injected fuel volume goes up. Incorrect fuel volumes or mistimed fuel injection can even damage the engine. Therefore, control algorithms of engine events require significant signal conditioning, and place stringent response-time requirements. In order to meet these requirements, a real-time operating system such as OSEK [107] and AUTOSAR [101] is used.

Although real-time operating systems are widely used in cars and PCM applications contain many periodic tasks, the engine events activating the fuel injection task come from reference pulses generated by sensors at the engine crankshaft. That is, the periods of these tasks *vary* depending on the speed of the crankshaft. As an analog variable, speed is continuous and hence the period of the task also can change both rapidly and continuously. Also, the execution time of these tasks vary and the worst-case execution time (WCET) arises when the engine speed increases to its maximum [6]. It is known in the automotive community that the engine control performance deteriorates with under-sampling, i.e., tasks having a longer period than the required minimum period for a given speed. In particular, the controller task period is known to have a greater impact on control performance than execution time.

In this chapter, we will focus on Rate-Monotonic Scheduling (RMS) [16], the optimal fixed-priority preemptive scheduling policy, which automotive OS standards such as OSEK and AUTOSAR support and other general-purpose OSes like Linux also do. Under RMS, the shorter the period of a task, the higher is its priority. A common assumption of using RMS is that the task periods do *not* change during run-time. A *Utilization Bound (UB) test* is often used to check if the given tasks are *schedulable*, which means that each of the given tasks meets its timing deadlines.

Conventional task models such as periodic tasks or aperiodic tasks are not adequate to handle engine tasks with varying periods. Consider a periodic task with $60ms$ execution time and a $140ms$ period, along with an engine task that has a period varying from $10ms$ to $120ms$, which is depicted in Figure 5.3. The UB test [16] gives $4ms$ as the maximum computation time of the engine task to guarantee the schedulability of the given two tasks, which is 46% utilization in the worst case. However, the engine task at lower speeds can have up to $60ms$ as its computation time, yielding 93% utilization. This 47% difference comes from the worst-case assumption for

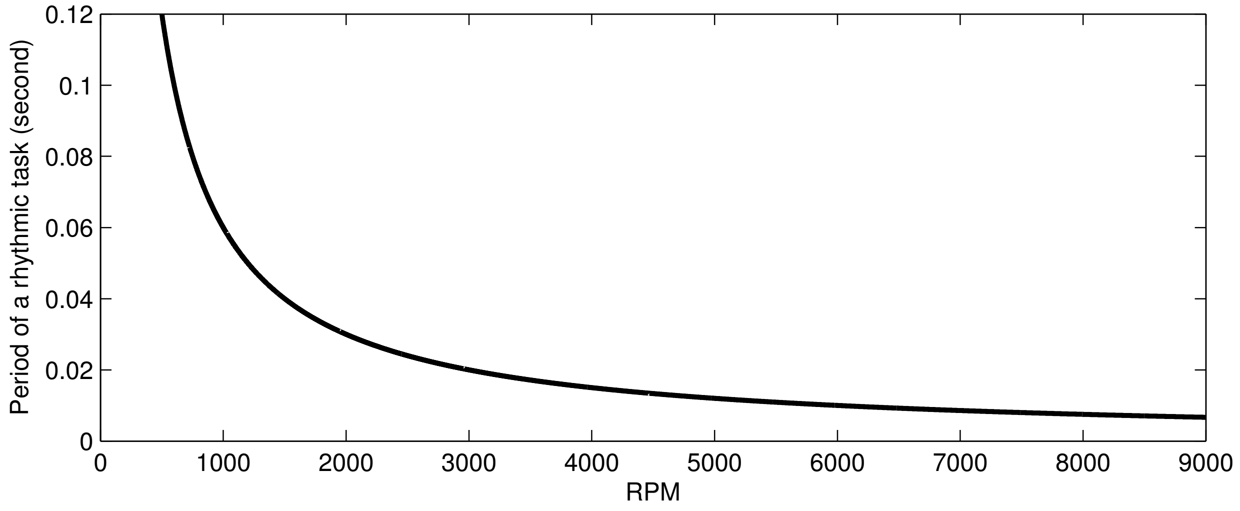


Figure 5.3: The variation of period according to engine RPM.

using a single offline UB test, where the shortest period and no period change are taken into account. In case we have more than one periodic task, this analysis could require the addition of more hardware, which is undesirable in a mass production industry such as car manufacturing.

In this chapter, we define a new task model called *Rhythmic Tasks* for characterizing and analyzing tasks that have continually varying periods depending on external physical events. We provide response-time analyses for rhythmic tasks under three cases: constant engine speed, accelerating engine speed and decelerating engine speed. We provide guidelines to evaluate schedulable utilization levels for the rhythmic task model by introducing *harmonic points* and *flexion points*. An integrated rhythmic task analysis framework with periodic tasks is also provided. We finally provide a case study of the rhythmic task model for PCM to show the applicability of the rhythmic task model to a CPS. To the best of our knowledge, this is the first model considering both continually varying periods and WCET for cyber-physical systems.

From this chapter, we relax the assumption of periodic tasks that are made in Chapters 3 and 4. Given that CPS have certain computational patterns, new task models for CPS are devised to incorporate such characteristics. In this chapter, we focus on the dynamic nature of CPS tasks as depicted in Figure 5.1. The rest of this chapter is organized as follows. In Section 5.1, we define the rhythmic task model. In Section 5.2, we provide an analysis of one rhythmic task and one periodic task. In Section 5.3, we provide an integrated analysis framework for one rhythmic task with multiple periodic tasks. In Section 5.4, a case study on an automotive PCM is presented. Finally, in Section 5.5, we provide our concluding remarks and discuss future work.

5.1 The Rhythmic Task Model

5.1.1 Definitions

A *rhythmic task* is a task with a (potentially) continually varying period and varying WCET. The change in the period value of a rhythmic task can depend on the current physical attributes of the system. The physical attributes of the given system are represented by a state vector, $v_s \in \mathbb{R}^k$, where k is the number of dimensions that capture the current system status. The WCET, period and deadline of a rhythmic task are a function of v_s and are denoted as $C(v_s)$, $T(v_s)$ and $D(v_s)$ respectively. Hence, the utilization of a rhythmic task is also a function of v_s and it is represented as $U(v_s)$.

Let J_i denote the i^{th} job of the rhythmic task and $T(v_s, J_i)$ denote the period of J_i. We define the acceleration $\alpha(v_s)$ of the rhythmic task as $1 - \frac{T(v_s,J_{i+1})}{T(v_s,J_i)}$. If $T(v_s, J_{i+1}) < T(v_s, J_i)$, acceleration is positive and the engine speed is increasing. The duration of acceleration is limited by $n_\alpha(v_s)$ in terms of the number of job releases. In other words, the rhythmic task can be positively accelerated by a factor of $\alpha(v_s)$ for $n_\alpha(v_s)$ job releases. When $T(v_s, J_{i+1}) > T(v_s, J_i)$, $\alpha(v_s)$ becomes negative and represent the deceleration of the rhythmic task. To avoid ambiguity, we use $n_\beta(v_s)$ when $\alpha(v_s)$ is negative. For ease of readability, we denote $C(v_s), T(v_s), D(v_s), U(v_s), \alpha(v_s), n_\alpha(v_s)$ and $n_\beta(v_s)$ as C^*, T^*, D^*, u^*, α, n_α, and n_β respectively. A summary of the notation is given in Table 5.1.

Table 5.1: Rhythmic task model notation.

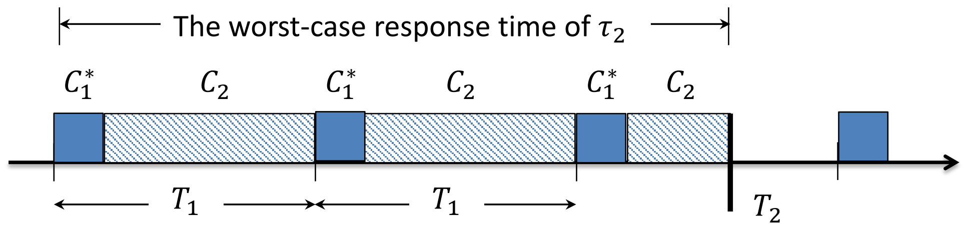
τ^*	Rhythmic task
v_s	State vector that represents physical environmental attributes
$C(v_s)$	Varying worst-case execution time of τ^*
$T(v_s)$	Continually varying period of τ^*
$D(v_s)$	Relative deadline of τ^*
$u(v_s)$	Utilization of τ^*
α	Acceleration rate of τ^*
n_α	Maximum acceleration duration of τ^*
n_β	Maximum deceleration duration of τ^*

5.1.2 System Assumptions

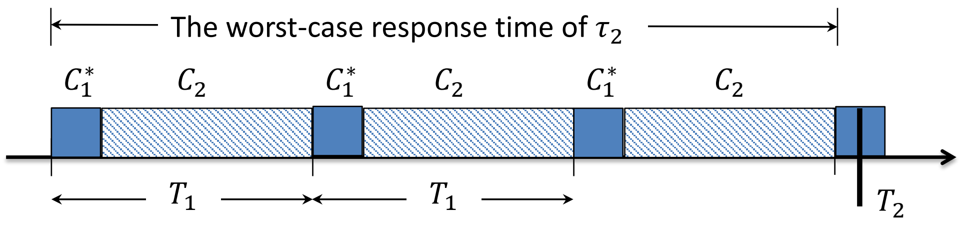
We consider a set of hard real-time tasks $\Gamma = \{\tau_1, \tau_2, ..., \tau_n\}$, where n is the number of tasks. The tasks in Γ are classified into two subsets: *Periodic Task Set*, Γ^P, and *Rhythmic Task Set*, Γ^R. We assume that Γ^R consists of m tasks ($m \leq n$). In other words, $\Gamma = \Gamma^P \cup \Gamma^R$ and $\Gamma^P \cap \Gamma^R = \emptyset$. For the sake of convenience, if a task τ_i is in Γ^R, the task may be denoted as τ_i^*. If a task τ_i is in Γ^P, the task will be represented as τ_i without the asterisk symbol (*).

A periodic task τ_i is specified as (C_i, T_i, D_i), where C_i is its WCET, T_i is its period, and D_i is the deadline of each of the task's jobs relative to the release time of each job. A rhythmic task τ_i^* is denoted by (C_i^*, T_i^*, D_i^*), where the WCET, period, and deadline are functions of v_s. In addition, u_i is the utilization of τ_i, defined as $\frac{C_i}{T_i}$. In this chapter, we assume that $T_i = D_i$ and $T_i^* = D_i^*$.

A rhythmic task τ_i^* is classified into three categories according to how C_i^* varies. A rhythmic task in the first category has a constant value of C_i^*. We refer to a rhythmic task with constant C_i^* as a *Constant Computation Rhythmic Task* (CCRT). In the second category, the utilization of a rhythmic task is maintained constant, and C_i^* varies accordingly. We name a rhythmic task with



(a) Case I: The worst-case response time of τ_2 is less than or equal to its deadline.



(b) Case II: The $\left\lceil \frac{T_2}{T_1^*} \right\rceil$-th instance of τ_1^* is executed at T_2.

Figure 5.4: Two different cases to consider to prove *Lemma* 9.

a constant utilization u_i^* as a *Constant Utilization Rhythmic Task* (CURT). In the third category, the WCET C_i^* of a rhythmic task is defined by a function $C_i^* = f_i(v_s)$, where f represents a general behavior of rhythmic tasks. One example of rhythmic tasks in this category is a task having a step function for C_i^* to maintain approximately constant utilization with discrete steps. We refer to a task in this category as a *General Computation-time Rhythmic Task* (GCRT).

We assume the use of fixed-priority scheduling, specifically RMS on a uniprocessor. Therefore, for RMS, the priority of a rhythmic task τ^* at time *t* will be determined based on its period T^* depending on the instantaneous system state v_s at time *t*. In this chapter, we assume that $m = 1$ and that the rhythmic task has the highest priority among all the n tasks. Therefore, T_1^* and C_1^* represent the period and the WCET of the rhythmic task τ_1^*. Hence, the inequality, $\forall v_s, C_1(v_s) \leq T_1(v_s) \leq T_2$, also holds. In other words, the only rhythmic task in the system always has the shortest period and, by RMS, is assigned the highest priority of all tasks. This chapter defines the rhythmic task model and studies the single rhythmic task scenario with many periodic tasks. This work in itself is useful for general CPS applications. However, there could be a need for multiple rhythmic tasks, and analyzing such systems is key future work.

5.1.3 Application of Rhythmic-Task Definition to Engine Control

The engine shown in Figure 5.2 has two revolutions every engine cycle, and the speed of revolutions is affected by four primary parameters ($k = 4$): RPM, number of active cylinders, the amount of fuel injected into cylinders, and gear ratio. Hence, v_s := <RPM, Number of active cylinders, Fuel amount, Gear ratio>. The period of the rhythmic task driven by the engine cycle is directly related to the duration of each revolution. If the rhythmic task is triggered every revolution, its period varies as illustrated in Figure 5.3. As can be seen, the period of the rhythmic task can vary over a wide range. Using the parameters of v_s, the acceleration rate α and the maximum acceleration duration n_α can be determined. Most modern cars are equipped with a *rev limiter* to prevent engines from being *redlined*. We can treat this redline as the maximum RPM for calculating α. By accelerating the engine at a particular gear level till the engine hits the redline from the minimum engine RPM, we can measure the acceleration duration at that gear level. Using the acceleration duration and the RPM changes, the acceleration rate α is determined. The measured duration is converted to n_α. Similarly, the maximum deceleration duration n_β also can be determined.

5.1.4 Problem Formulation

Our objective is to determine whether a given taskset with a rhythmic task τ_1^* running at the highest priority is schedulable under (a) steady-state conditions (b) positive acceleration, and (c) deceleration. We will provide a schedulability test for each of these cases. For steady-state conditions, we will propose an algorithm to determine schedulability given the current (C_1^*, T_1^*) values along with the periodic tasks. For accelerating and decelerating conditions, we will provide a range of period change ratios for which schedulability holds. These outcomes can be used by CPS developers to determine when C_1^* needs to be decreased in order to maintain schedulability.

5.2 One Rhythmic Task and One Periodic Task

We first consider a simple taskset Γ with one rhythmic task τ_1^* and one periodic task τ_2.

5.2.1 Steady-State Analysis

Lemma 9 *Given one rhythmic task, represented by* (C_1^*, T_1^*)*, and one periodic task, represented by* (C_2, T_2)*, both tasks are schedulable if the following inequality is satisfied.*

$$C_1^* \leq max\left(\frac{T_2 - C_2}{\left\lceil \frac{T_2}{T_1^*} \right\rceil}, T_1^* - \frac{C_2}{\left\lfloor \frac{T_2}{T_1^*} \right\rfloor}\right) \tag{5.1}$$

Proof In order to prove this statement, we should find the maximum value of C_1^*, which does not cause τ_2 to miss its deadline. By assumption (see Section 5.1.2), τ_1^* has higher priority than τ_2. Hence, the two tasks are schedulable if τ_2 is schedulable. In order to obtain the maximum schedulable value of C_1^*, we should consider two different cases. The first case is when the response time of τ_2 is less than or equal to its relative deadline, which is illustrated in Figure 5.4(a). In this case, $C_1^* \left\lceil \frac{T_2}{T_1^*} \right\rceil + C_2 \leq T_2$ should be satisfied. Then, in this case, the maximum value of C_1^* is given by

$$C_1^* = \frac{T_2 - C_2}{\left\lceil \frac{T_2}{T_1^*} \right\rceil} \tag{5.2}$$

The second case is depicted in Figure 5.4(b). Since τ_1^* can preempt τ_2, the $\left\lceil \frac{T_2}{T_1^*} \right\rceil$-th instance of τ_1^* can overlap the period of task τ_2. Therefore, $C_1^* \left\lfloor \frac{T_2}{T_1^*} \right\rfloor + C_2 \leq T_1^* \left\lfloor \frac{T_2}{T_1^*} \right\rfloor$ should hold. Hence, in this case, the maximum value of C_1^* is given by

$$C_1^* = T_1^* - \frac{C_2}{\left\lfloor \frac{T_2}{T_1^*} \right\rfloor} \tag{5.3}$$

The maximum value of Equation (5.2) and Equation (5.3) will provide the bound of C_1^*, given τ_2. Therefore, if Inequality (5.1) is satisfied, both tasks are schedulable.

Inequality (5.1) allows us to visualize the schedulability of one rhythmic task and one periodic task. Given τ_2: $(6, 14)$, a well-known worst-case task for the least-upper bound on schedulable utilization as an example [16], Figure 5.5 plots the maximum value of C_1^* as T_1^* varies from 0 to 14. If (C_1^*, T_1^*) lies under the curve in this figure, the taskset with τ_2: $(6, 14)$ is schedulable.

Accordingly, we can see the utilization change of the taskset. Figure 5.6 shows the variation in the total utilization as T_1^* changes. In this figure, we observe two types of interesting points: local maxima and local minima. We call local maxima as *harmonic points*, since the task periods are "compatible" at these points, and local minima as *flexion points*, since the slope changes from negative to positive here. Let $U(\Gamma)$ denote the total utilization and $U_{lub}(\Gamma)$ denote the least-upper bound on schedulable utilization of the given taskset Γ having one rhythmic task and one periodic task.

Lemma 10 *At $T_1^* = \frac{T_2}{i}$, where $i \in \mathbb{Z}^+$, harmonic points occur, where $U(\Gamma)$ is 1.*

Proof Substituting $\frac{T_2}{i}$ for T_1^* in the right-hand side of Equation (5.2) returns $\frac{T_2 - C_2}{i}$ since $\frac{T_2}{T_1^*}$ becomes an integer i. We also obtain the same value from the right-hand side of Equation (5.3). Then, Inequality (5.1) is equivalent to $C_1^* \leq \frac{T_2 - C_2}{i}$. The utilization of two tasks is given by $U(\Gamma) = \frac{C_1^*}{T_1^*} + \frac{C_2}{T_2}$. By substituting $\frac{T_2 - C_2}{i}$ and $\frac{T_2}{i}$ for C_1^* and T_1^* respectively, we obtain 1 as the total utilization.

Lemma 11 *Flexion points, local minima of $U(\Gamma)$, happen at $T_1^* = \frac{C_2}{i(i-1)} + \frac{T_2}{i}$, where $i \geq 2$ and $i \in \mathbb{Z}^+$. $C_1^* = \frac{T_2 - C_2}{i}$ also holds.*

Proof From Lemma 10, the harmonic points happen when $T_1^* = \frac{T_2}{i}$, where $i \in \mathbb{Z}^+$. Let's assume that the flexion points occur when $T_1^* = \frac{T_2}{i} + x$, where x is the value we want to find. The flexion points correspond to the intersections of Equation (5.2) and Equation (5.3) as shown in [16]. We can substitute $\frac{T_2}{i} + x$ for T_1^* in both equations. Suppose $i \geq 2$. Then, because $\frac{T_2}{i} \leq \frac{T_2}{i} + x \leq \frac{T_2}{i-1}$, $\left\lceil \frac{T_2}{T_1^*} \right\rceil = \left\lceil \frac{T_2}{\frac{T_2}{i} + x} \right\rceil = i - 1$ and $\left\lfloor \frac{T_2}{T_1^*} \right\rfloor = \left\lfloor \frac{T_2}{\frac{T_2}{i} + x} \right\rfloor = i$. We substitute these in both equations, and we obtain $C_1^* = \frac{T_2 - C_2}{i-1} = x + \frac{T_2}{i} - \frac{C_2}{i}$. Solving for x, we get $x = \frac{T_2 - C_2}{(i-1)i}$. When $i \geq 2$ and $i \in \mathbb{Z}^+$, $T_1^* = \frac{T_2}{i} + x = \frac{T_2}{i} + \frac{T_2 - C_2}{(i-1)i} = \frac{C_2}{i(i-1)} + \frac{T_2}{i}$

Lemma 12 *The minimum flexion point, U_{lub}, occurs at $i = 2$.*

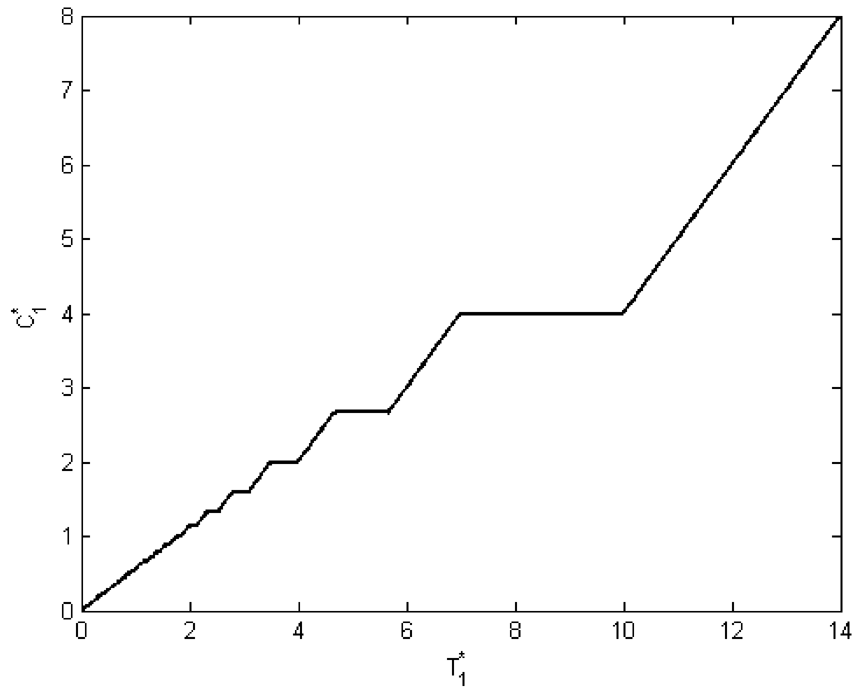


Figure 5.5: Schedulable region of the taskset including τ_1^* and τ_2 as (6,14).

Proof Substituting the results from *Lemma* 11 gives us $U(\Gamma) = \frac{T_2 - C_2}{\frac{C_2}{i-1} + T_2} + \frac{C_2}{T_2}$. Because $T_2 - C_2 \geq 0$, U_{lub} can be found when i has the smallest allowable value, which is 2.

We can see that the results from Lemma 11 and Lemma 12 are consistent with the curve shown in Figure 5.6. One interesting observation of Lemma 11 is that only the parameters of the periodic task affect the locations of the flexion points. A similar property will be shown in Theorem 17.

Lemma 13 *The flexion points of one rhythmic task and one periodic task occur at* $C_2 = T_2(\sqrt{i(i-1)} - (i-1))$*, where* $i \in \mathbb{Z}^+$ *and* $i \geq 2$.

Proof From the proof of Lemma 12, $U(\Gamma) = \frac{T_2 - C_2}{\frac{C_2}{i-1} + T_2} + \frac{C_2}{T_2}$. Differentiating with respect to C_2, we obtain $\frac{\partial U(\Gamma)}{\partial C_2} = \frac{1}{T_2} + \frac{i(i-1)T_2}{((i-1)T_2 + C_2)^2}$. We solve the equation $\frac{\partial U(\Gamma)}{\partial C_2} = 0$, and the solution is given by $C_2 = T_2\left(\sqrt{i(i-1)} - (i-1)\right)$, where the flexion points happen.

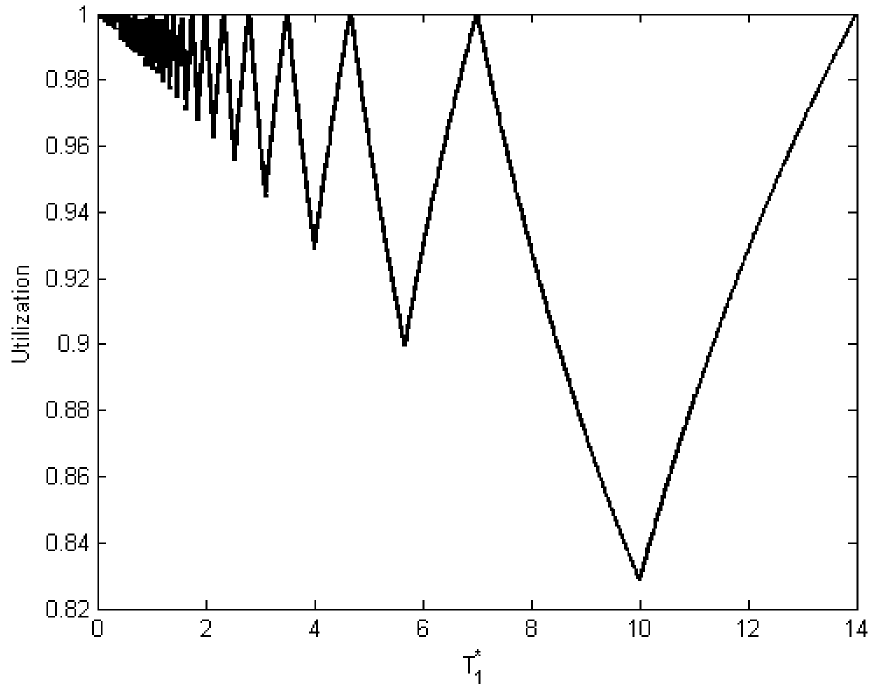


Figure 5.6: Corresponding utilization value where there are one rhythmic task and one periodic task, (6,14).

5.2.2 Acceleration Analysis

The positive acceleration of an engine is a significant event from a schedulability perspective. Car manufacturers always provide two types of information on engine specifications, horse power and torque. The horse power of an engine is related to the maximum speed analogous to the steady-state discussed in Section 5.2.1, and the torque of an engine has a strong relationship with the acceleration of a vehicle. The acceleration of a car is immediately followed by the change of task periods controlling the engine. As shown in [6], the quality of engine control decreases significantly if the periods of engine tasks decrease. In this subsection, we will discuss how much the engine can accelerate by finding the maximum rate of period changes under the given taskset Γ.

As shown in Figure 5.3, the duration for one revolution becomes shorter as positive acceleration occurs. Suppose that the i^{th} revolution occurs at time t. Then, as the engine accelerates, the $(i+1)^{th}$ revolution will have a shorter period. Let α denote the rate of period change, where $0 \leq \alpha \leq 1$ and $\alpha \in \mathbb{R}$. Let n_α denote the maximum positive acceleration duration in terms of the number of job releases of the rhythmic task. Suppose that $T^{*,i}$ is the period of the i^{th} revolution of the rhythmic task τ^*. Then, if the period of the rhythmic task reduces after the first job release, we can express the period of the $(i+1)^{th}$ revolution as $T^{*,i+1} = T^{*,i}\,(1-\alpha)$ by using α, when $i \leq n_\alpha$. Otherwise, $T^{*,i+1} = T^{*,i}$. Hence, we can define $T^{*,i}$ as $T^{*,i} = T_1^*(1-\alpha)^{\min(i,n_\alpha)}$ for a non-negative integer i. We will find if the given taskset is schedulable under the given α and n_α.

Suppose that there are two tasks, one rhythmic task τ_1^* and one periodic task τ_2. Under the given (C_1^*, T_1^*) at a certain time, let n_p^a denote the number of preemptions which τ_2 experiences while T_1^* is decreasing. Then, if T_1^* starts decreasing at the first job release, n_p^a is defined as $n_p^a = \max\{n | \sum_{i=0}^{n-1} T_1^{*,i} \leq T_2 \text{ and } n \in \mathbb{Z}^+\}$.

Given the definition of n_p^a, the following inequality should be satisfied.

$$\sum_{i=0}^{n_p^a-1} T_1^{*,i} \leq T_2 \tag{5.4}$$

Let f_C^* denote the function of $T_1^{*,i}$ which returns the computation time of the rhythmic task, where f_C^* has a different type of function depending on which category among CCRT, CURT and GCRT the rhythmic task is classified into. Then, n_p^a and α should meet one of the following two inequalities:

$$\sum_{i=0}^{n_p^a-1} \left\{ f_C^*\left(T_1^{*,i}\right) \right\} + C_2 \leq T_2 \tag{5.5}$$

$$\sum_{i=0}^{n_p^a-2} \left\{ f_C^*\left(T_1^{*,i}\right) \right\} + C_2 \leq \sum_{i=0}^{n_p^a-2} T_1^{*,i} \tag{5.6}$$

where Inequality (5.5) refers to the case when the n_p^a-th instance of τ_1^* completes before T_2 under the assumption that the acceleration is started at the first instance. Inequality (5.6) represents the case when the n_p^a-th instance of τ_1^* overlaps T_2. These two cases are also explained in the proof of Lemma 9. In order to decide if α is possible, it should be substituted in Inequality (5.4).

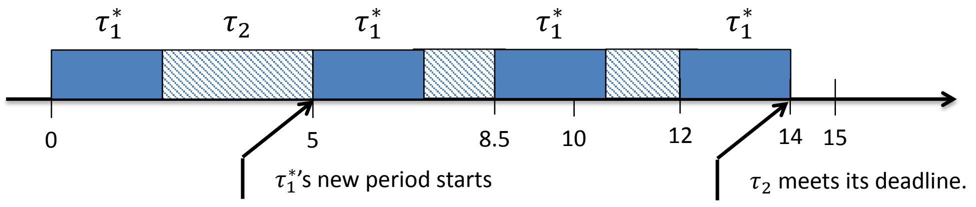


Figure 5.7: The example scenario when a rhythmic task accelerates.

According to the α value, we determine n_p^a, which should be substituted in Inequality (5.5) and Inequality (5.6) to check if one of those inequalities is satisfied.

As an example of a type of f_C^*, we consider CCRT. As C_1^* does not change, Inequalities (5.5) and (5.6) become $n_p^a C_1^* + C_2 \leq T_2$ and $(n_p^a - 1)C_1^* + C_2 \leq \sum_{i=0}^{n_p^a-2} T_1^{*,i}$, respectively.

Acceleration Example: Suppose that there is one CCRT rhythmic task τ_1^* and one periodic task τ_2: $(6, 14)$. Given (2,5) as (C_1^*, T_1^*), acceleration is possible when α is 0.3 and n_α is 1. As shown in Figure 5.7, the period of the rhythmic task will become 3.5 at time 5, and τ_2 meets its deadline at time 14. However, any α greater than 0.3 will make the taskset unschedulable. In this case, therefore, we can say that the maximum possible α is 0.3 when $n_\alpha = 1$.

5.2.3 Deceleration Analysis

Engine deceleration happens generally when the amount of fuel injected into the cylinders decreases. Engine deceleration is also significant since it is related to shifting of gears. This occurs very frequently in urban areas, and careless system design could affect the system schedulability whenever the gear shifting occurs. When the engine decelerates, the engine task periods increase.

For the deceleration analysis, α becomes a negative value to represent the rate of period increase. Under the given (C_1^*, T_1^*) at a certain time, let n_p^d denote the number of preemptions which τ_2 experiences while T_1^* is increasing. Then, if T_1^* starts increasing at the first job release, n_p^d is defined as $n_p^d = \max\{n | \sum_{i=0}^{n-1} T_1^{*,i} \leq T_2 \text{ and } n \in \mathbb{Z}^+\}$.

Given the definition of n_p^d, the following inequality should be satisfied, and the period does

Algorithm 8 Rhythm-Max-C(Γ)

```
Require: Γ: a given taskset including a rhythmic task
Ensure: The WCET of the rhythmic task
1: for $i = 1$ to $n$ do
2:     ▷ Build a set of points to check
3:     $S_i = \{kT_j | j = 1...i, k$ is an integer satisfying
4:         $kT_j \leq T_i\}$
5:     ▷ From Inequalities (5.11) and (5.12)
6:     for For each element $s_i^m \in S_i$, do
7:         Calculate $C_{1,i}^m = \frac{s_i^m - \sum_{j=2}^{i} \left\lceil \frac{s_i^m}{T_j} \right\rceil C_j}{\left\lceil \frac{s_i^m}{T_1} \right\rceil}$
8:     Maintain the largest value of $C_{1,i}^m$ as $C_{1,i}$
9: return $min\{C_{1,i}, 1 \leq i \leq n\}$
```

not increase after the n_β-th job of the rhythmic task if $n_p^d > n_\beta$.

$$\sum_{i=0}^{n_p^d - 1} T_1^{*,i} \leq T_2 \tag{5.7}$$

Then, n_p^d should meet one of the following two inequalities:

$$\sum_{i=0}^{n_p^d - 1} \left\{ f_C^*(T_1^{*,i}) \right\} + C_2 \leq T_2 \tag{5.8}$$

$$\sum_{i=0}^{n_p^d - 2} \left\{ f_C^*(T_1^{*,i}) \right\} + C_2 \leq \sum_{i=0}^{n_p^d - 2} T_1^{*,i} \tag{5.9}$$

The reason behind these two inequalities is already mentioned in Section 5.2.2. Based on α, we can find the value of n_p^d which satisfies Inequalities (5.8) and (5.9) to check if one of those inequalities is satisfied. This process applies to both CURTs and GCRTs. For rhythmic tasks having fixed C_1^* (CCRT defined in Section 5.1.2), they will be always schedulable because periods are *sustainable* as they are increased [111].

5.3 One Rhythmic Task and Many Periodic Tasks

In this section, we consider a taskset Γ with one rhythmic task τ_1^* and $n-1$ periodic tasks $\tau_2, ..., \tau_n$. In Section 5.2, we analyzed the case of having one rhythmic task and one periodic task. The results from the previous section will be extended to support several periodic tasks for constant speed, positive acceleration, and deceleration cases. A real-world example will be analyzed using this model in Section 5.4.

5.3.1 Steady-state Analysis

In order to determine the schedulability of Γ, we find the maximum value of C_1^* which does not make any periodic task miss its deadline. Let $f^*_{C_{max}}(T_1^*)$ denote the function which returns the maximum possible value of WCET for C_1^* which makes Γ schedulable when received T_1^* as an input.

Theorem 14 *$f^*_{C_{max}}(T_1^*)$ is given by*

$$\min_{\forall \tau_i \in \Gamma} \left\{ \max \left(T_1^* - \frac{\sum_{j=2}^{i} \left\lceil \frac{T_i}{T_j} \right\rceil C_j}{\left\lfloor \frac{T_i}{T_1^*} \right\rfloor}, \frac{T_i - \sum_{j=2}^{i} \left\lceil \frac{T_i}{T_j} \right\rceil C_j}{\left\lceil \frac{T_i}{T_1^*} \right\rceil} \right) \right\} \tag{5.10}$$

Proof In order to check if Γ is schedulable, the worst-case response time of each periodic task τ_i should not exceed its deadline. Under the assumption of critical instant from [16], we should compare the worst-case response time of each periodic task to its deadline [112]. Then, as described in Lemma 9, two different cases should be considered.

The first case is that the execution time of the $\left\lceil \frac{T_i}{T_1^*} \right\rceil$-th instance of the rhythmic task is long enough to overlap T_i, where T_i is the deadline of the i^{th} periodic task τ_i.

$$\left\lfloor \frac{T_i}{T_1^*} \right\rfloor C_1^* + \sum_{j=2}^{i} \left\lceil \frac{T_i}{T_j} \right\rceil C_j \leq \left\lfloor \frac{T_i}{T_1} \right\rfloor T_1$$

As long as the following inequality is satisfied,

$$C_1^* \leq T_1^* - \frac{\sum_{j=2}^{i} \left\lceil \frac{T_i}{T_j} \right\rceil C_j}{\left\lfloor \frac{T_i}{T_1^*} \right\rfloor} \tag{5.11}$$

the given taskset is schedulable.

In the second case, the execution time of the $\left\lceil \frac{T_i}{T_1^*} \right\rceil$-th instance of the rhythmic task does not overlap T_i. Hence, the following inequality must be satisfied:

$$\left\lceil \frac{T_i}{T_1^*} \right\rceil C_1^* + \sum_{j=2}^{i} \left\lceil \frac{T_i}{T_j} \right\rceil C_j \leq T_i$$

Solving for C_1^* returns the following inequality.

$$C_1^* \leq \frac{T_i - \sum_{j=2}^{i} \left\lceil \frac{T_i}{T_j} \right\rceil C_j}{\left\lceil \frac{T_i}{T_1^*} \right\rceil} \tag{5.12}$$

Use the maximum value from Inequalities (5.11) and (5.12) as we are looking for the maximum allowable C_1^*. At this point, we have $n-1$ candidates for C_1^*. Since no periodic task must miss its deadline, we use the minimum value among those candidates. Then, $f_{C_{max}}^*(T_1^*)$ is given by Equation (5.10).

Theorem 15 *The slope of $f_{C_{max}}^*(T_1^*)$ is either 1 or 0.*

Proof We consider two different cases as the proof of Theorem 14. In the first case, Inequality (5.11) indicates that the slope of $f_{C_{max}}^*(T_1^*)$ is 1. In the second case, due to the fact that we consider the execution time of the $\left\lceil \frac{T_i}{T_1^*} \right\rceil$-th instance of the rhythmic task, the right hand side of Inequality (5.12) does not change as T_1^* changes. Hence, the slope of $f_{C_{max}}^*(T_1^*)$ is 0 with respect to T_1^*.

Based on Theorems 14 and 15, we have designed an algorithm for finding the maximum possible value of WCET in Algorithm 8. Figure 5.8 shows an outcome of Algorithm 8 for a rhythmic task with 3 periodic tasks: τ_2: $(1,7)$, τ_3: $(1,10)$ and τ_4: $(1,23)$. As shown, the slope of the curve in Figure 5.8a is either 1 or 0, where the taskset is schedulable if (C_1^*, T_1^*) lies under the curve. Also, the minimum flexion point in Figure 5.8b is at 4 which satisfies Theorem 17 when $T_1^* = \frac{T_2}{2} + \frac{C_2}{2}$.

Corollary 16 *If a taskset is schedulable with (C_1^*, T_1^*), the taskset is schedulable with $\left(\frac{C_1^*}{k}, \frac{T_1^*}{k}\right)$, where k is a positive integer.*

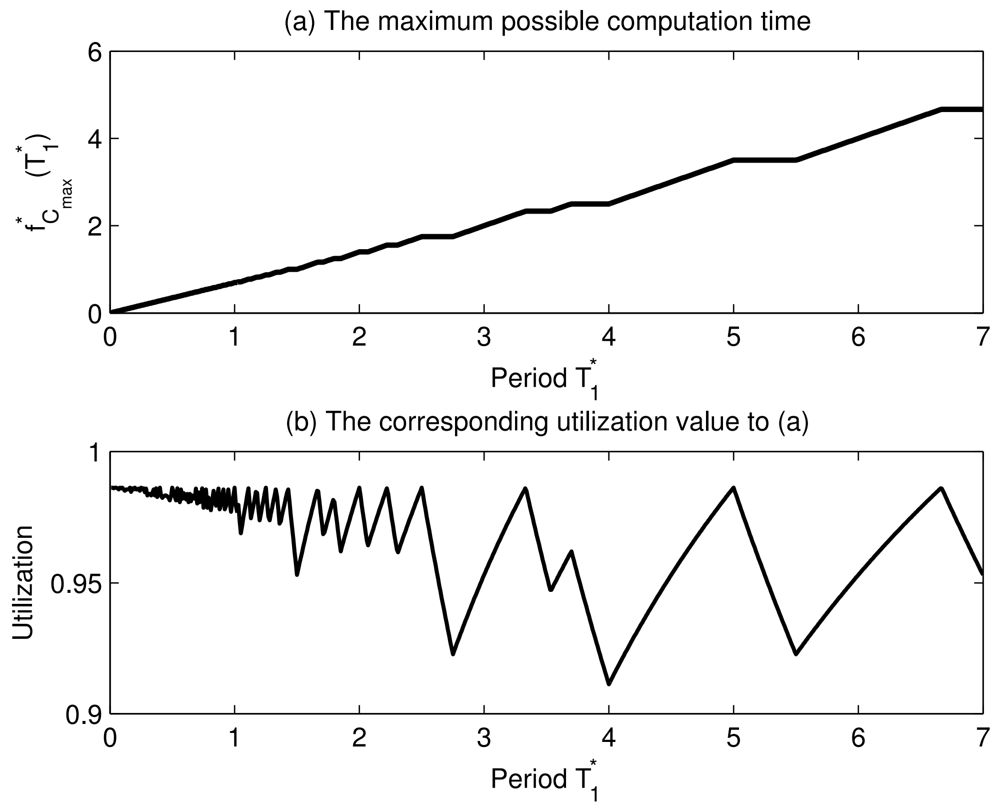


Figure 5.8: An example for a rhythmic task with 3 periodic tasks: τ_2: $(1,7)$, τ_3: $(1,10)$ and τ_4: $(1,23)$.

Proof Compute $f^*_{C_{max}}$. The parameters of (C_1^*, T_1^*) make the system schedulable. Then, $\left(\frac{C_1^*}{k}, \frac{T_1^*}{k}\right)$ will always be below $f^*_{C_{max}}$ from Theorem 15.

Theorem 17 *The minimum flexion point lies in the range,* $\frac{T_2}{2} + \frac{C_2}{2} \leq T_1^* \leq T_2$.

Proof The worst-case utilization happens when $1 \leq \frac{T_2}{T_1^*} \leq 2$ from [16]. Hence, $\frac{T_2}{2} \leq T_1^* \leq T_2$. From [16], since $\frac{T_2 - C_2}{2} \geq C_1^*$ holds good, $T_1^* + \frac{T_2 - C_2}{2} \geq T_2$ is satisfied. Then, by solving for T_1^*, $\frac{T_2}{2} + \frac{C_2}{2} \leq T_1^* \leq T_2$.

Theorem 17 is critical to find the least-upper bound utilization of the given taskset regardless of the other tasks except τ_2, the second highest priority task. This theorem could also be a hint to find the least-upper bound utilization when only the task periods are given, a problem that appears to be unsolved yet.

5.3.2 Acceleration Analysis

Let α denote the rate of period decrease of τ_1^* and n_α denote the maximum positive acceleration duration in terms of the number of job releases of the rhythmic task. Then, the following theorem is satisfied.

Theorem 18 *Given a taskset Γ with one rhythmic task τ_1^* and $n-1$ periodic tasks, Γ is schedulable when the rhythmic task is accelerating if the following inequality is satisfied when $n_\alpha = 1$.*

$$\alpha \leq 1 - \frac{T_1^*}{C_1^*}\left(UB(n) - \sum_{i=2}^{n} \frac{C_i}{T_i}\right)$$

where $UB(n)$ returns the utilization bound [16] of n tasks.

Proof While the acceleration of a rhythmic task continues, the period of the i^{th} instance affects the WCET of $(i+1)^{th}$ instance. Hence, $\frac{C_1^*}{T_1^*(1-\alpha)} + \sum_{i=2}^{n} \frac{C_i}{T_i} \leq UB(n)$ is satisfied from [16] when n_α is 1. Solving for α, $\alpha \leq 1 - \frac{T_1^*}{C_1^*}\left(UB(n) - \sum_{i=2}^{n} \frac{C_i}{T_i}\right)$ holds.

The bound of Theorem 18 is not tight because it uses the utilization bound [16]. A tighter bound can be found by extending the results from Section 5.2. We need to extend the definition of n_p^a first. Let $n_{p,i}^a(t)$ denote the number of preemptions of the periodic task τ_i caused by the rhythmic task τ_1^* during t units of time. Then, $n_{p,i}^a(t)$ is defined as $n_{p,i}^a(t) = \max\{n | \sum_{j=0}^{n-1} T_1^{*,j} \leq t, n \in \mathbb{Z}^+\}$. Therefore, the following inequality should be satisfied.

$$\forall i, \sum_{j=0}^{n_{p,i}^a(T_i)-1} T_1^{*,j} \leq T_i, i \in \{k | k \in \mathbb{Z}^+ \text{ and } k \geq 2\} \tag{5.13}$$

Then, by using the value found above, the response-time test [113] has to be modified as

$$W_i^{k+1} = C_i + C_1^* + \sum_{j=0}^{n_{p,i}^a(W_i^k)} f_C^*(T_1^{*,j}) + \sum_{h=2}^{i-1} \left\lceil \frac{W_i^{k+1}}{T_h} \right\rceil C_h \tag{5.14}$$

where $W_i^0 = C_1^* + \sum_{j=2}^{i} C_j$ and the test terminates when $W_i^{k+1} = W_i^k$.

Based on Inequality (5.13) and Equation (5.14), the generalized algorithm is given in Algorithm 9 which checks if a taskset is schedulable under the given α and n_α. Algorithm 10 is used for obtaining the number of preemptions caused by the rhythmic task, and Algorithm 11

Algorithm 9 Rhythmic-Acc-$\alpha(\Gamma, \alpha, n_\alpha)$

```
Require: Γ: a taskset including a rhythmic task, α: the acceleration ratio and n_α: the duration
    of rhythmic task acceleration in terms of the number of job releases
Ensure: Schedulability of Γ
 1: for $i = 2$ to $n$ do
 2:     ▷ Calculate the initial condition for each task $\tau_i$
 3:     $W_i^0 = C_1^* + \sum_{j=2}^{i} C_j$ and $W_i^1 = 0$
 4:     $k = 0$
 5:     while $W_i^{k+1} \neq W_i^k$ do
 6:         ▷ Pick the maximum number of preemptions for each iteration
 7:         $n_{p,i}^a(W_i^k)$ = Num-Preemptions$(T_1^*, \alpha, n_\alpha, W_i^k)$
 8:         $E_1^*$ = Execution-Time$(n_{p,i}^a(W_i^k), C_1^*, \alpha, n_\alpha)$
 9:         $W_i^{k+1} = C_i + E_1^* + \sum_{h=2}^{i-1} \left\lceil \frac{W_i^{k+1}}{T_h} \right\rceil C_h$
10:         Update necessary parameters
11:     if $W_i^k \leq D_i$ then
12:         Mark $\tau_i$ schedulable
13: if all tasks schedulable then
14:     return TRUE
15: else
16:     return FALSE
```

calculates the preemption duration during the execution for CURT. The maximum value can be found by using this function for the range of α.

For analyzing engine deceleration, the definition of n_p^d also needs to be extended, and a similar modified response-time test can be used.

Algorithm 10 Num-Preemptions($T_1^*, \alpha, n_\alpha, W_i^k$)

```
1: ▷ The time duration of rhythmic task acceleration
2: $d_{acc} = \sum_{j=0}^{n_\alpha - 1}\{T_1^*(1-\alpha)^j\}$
3: if $d_{acc} > W_i^k$ then
4:     $n_{p,i}^a(W_i^k) = \max\{l | \sum_{j=0}^{l-1}\{T_1^*(1-\alpha)^j\} \leq W_i^k,$
5:     where $l \in \mathbb{Z}^+\}$
6: else
7:     $n_{p,i}^a(W_i^k) = n_\alpha + \left\lceil \frac{W_i^k - d_{acc}}{T_1^*(1-\alpha)^{n_\alpha - 1}} \right\rceil$
8: return $n_{p,i}^a(W_i^k)$
```

Algorithm 11 Execution-Time($n_{p,i}^a(W_i^k), C_1^*, \alpha, n_\alpha$)

```
1: if $n_{p,i}^a(W_i^k) < n_\alpha$ then
2:     $E_1^* = \sum_{j=0}^{n_{p,i}^a(W_i^k)}\{C_1^*(1-\alpha)^j\}$
3: else
4:     $E_1^* = \sum_{j=0}^{n_\alpha - 1}\{C_1^*(1-\alpha)^j\} + (n_{p,i}^a(W_i^k) - n_\alpha) \times C_1^*(1-\alpha)^{n_\alpha - 1}$
5: return $E_1^*$
```

5.3.3 Guidelines for CPS Application Developers

In this subsection, we will provide some guidelines that help CPS application developers to apply the rhythmic task analysis results and guarantee the schedulability of the system. The developers should ensure that:

1. The application is categorized into one of the three categories: CCRT, CURT and GCRT.
2. The computation time of a rhythmic task lies under the schedulable region as depicted in Figures 5.5 and 5.8.
    - The minimum flexion point of the total utilization can be used. By not exceeding this bound, the system schedulability is guaranteed. However, it should be noted that this bound could be pessimistic.

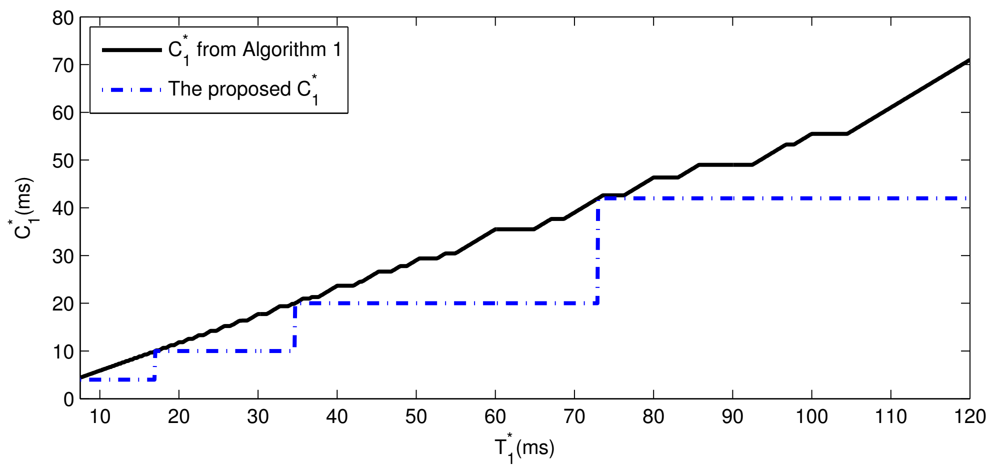


Figure 5.9: The maximum possible computation time for a rhythmic task that has varying period from 7.5ms to 120ms with 9 periodic tasks.

- Algorithm 8 and Theorem 14 can be used together to find the exact schedulable region.

3. The application can be in the form of modules for different speeds. The application can have all the modules or a subset of the modules depending upon the execution time to meet the system schedulability.

4. The difference between the maximum allowable worst-case execution time and the actual computation time should be enough to tolerate the acceleration for the values α, n_α and n_β. Theses values are computed by using Algorithm 9.

Once the parameters of the schedulable rhythmic task (C^*, T^*) are found, this information can be used for finding other schedulable regions. For CCRT, the rhythmic task with a longer period will be schedulable. For CURT, the rhythmic task with $(\frac{C^*}{k}, \frac{T^*}{k})$, where k is a positive integer, will be also schedulable.

5.4 Case Study of the Rhythmic Task Model

In this section, we provide a case study to show how to apply the rhythmic task model to an existing CPS. Our model is applicable to a generic CPS having tasks with varying periods. In this section, we investigate the automotive PCM.

Figure 5.10 illustrates a process for injecting and delivering fuel to each cylinder at every revolution [114]. Since the depicted task is triggered by timing signals from engine events, increasing/decreasing the duration of revolution will change the period of the given task. Suppose the RPM varies from 500 to 9000, so the period of the corresponding task varies from $7.5ms$ to $120ms$. The operations illustrated in Figure 5.10 are also executed. The execution path is composed of a service routine, sensor reads, air calculation, fuel calculation and fuel delivery. We model this task as a rhythmic task. Each block has its own WCET: Service Routine and Fuel Delivery: $4ms$, Sensor Reads: $6ms$, Air Calculation: $10ms$ and Fuel Calculation: $22ms$. In addition, a typical Engine Control Module (ECM) has other features [61] such as monitoring the processor that runs the control algorithms, reporting the current status to a diagnosis module, and managing sensors which measure the amount of fuel injected. We picked nine tasks to show the typical behaviors of the ECM representing the periodic engine operations [61] and study the impact of having a rhythmic task. Specifically, we use the following periodic tasks: τ_2:$(5ms, 120ms)$, τ_3:$(20ms, 120ms)$, τ_4:$(5ms, 180ms)$, τ_5:$(6ms, 200ms)$, τ_6:$(8ms, 240ms)$, τ_7:$(10ms, 240ms)$, τ_8:$(3ms, 300ms)$, τ_9:$(1ms, 360ms)$ and τ_{10}:$(7ms, 400ms)$.

The solid line in Figure 5.9 shows the maximum possible computation time of the rhythmic task using Algorithm 8. As mentioned earlier, if the value of (C_1^*, T_1^*) is below the given solid

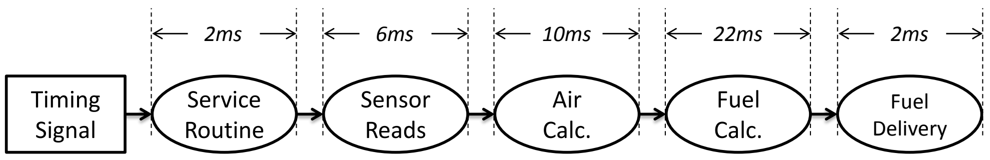


Figure 5.10: Flow diagram for the start of injection in PCM software.

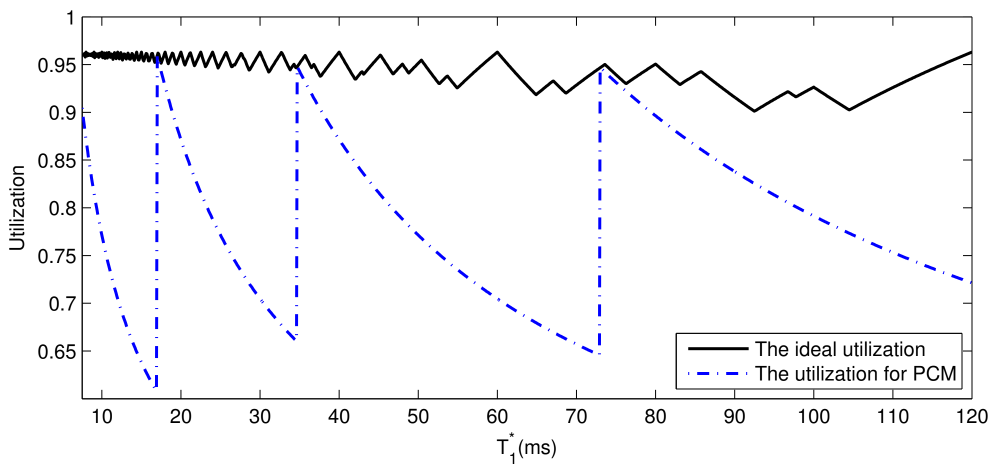
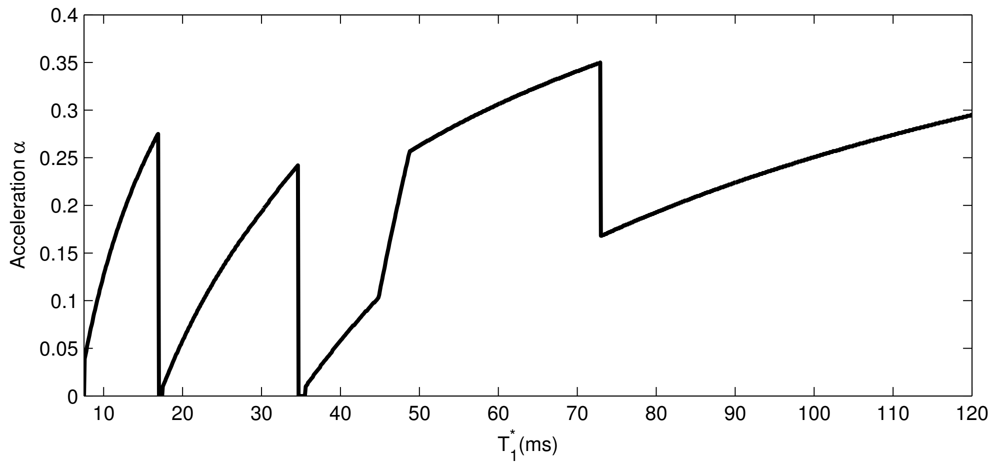


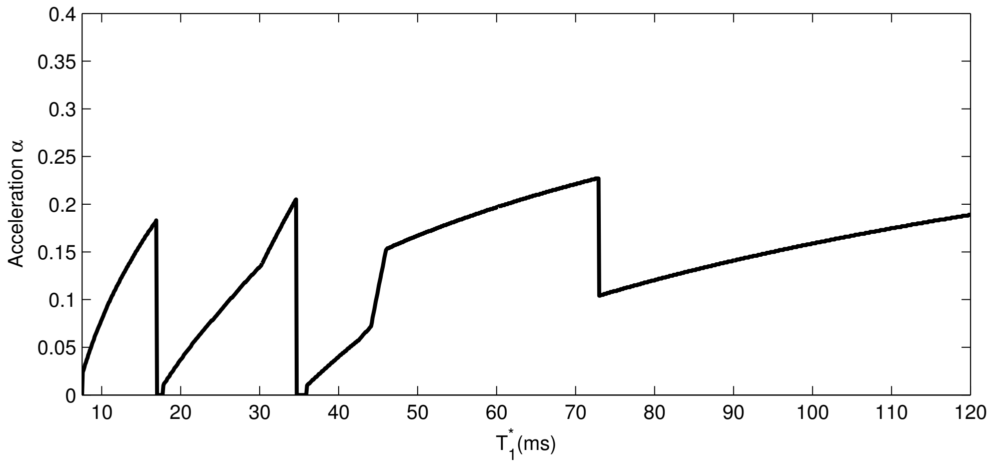
Figure 5.11: The corresponding utilization value to Figure 5.9 for a rhythmic task that has varying period from $7.5ms$ to $120ms$ with 9 periodic tasks.

curve, the taskset is schedulable. This offers design-time guidelines to determine the feasible WCET of the engine control task. The dotted line is a recommendation for when a particular software block from Figure 5.10 can be executed. The Service Routine and Fuel Delivery functions will be executed by every job of the rhythmic task regardless of the value of T_1^*. The Sensor Reads function will be executed in addition to Service Routine and Fuel Delivery if T_1^* is greater than $17ms$. Since the WCET of Sensor Reads is $6ms$, it can make the system unschedulable if T_1^* is smaller than $17ms$. In this case, however, instead of not running the whole instructions of Sensor Reads, the previous sensor values can be used for the operation. Similarly, the function Air Calculation will be executed if T_1^* is greater than $34.7ms$; Fuel Calculation will be executed if T_1^* is greater than $73ms$. These recommendations correspond to making the rhythmic task a GCRT with a discrete step function.

Figure 5.11 illustrates the corresponding utilization based on the WCET given in Figure 5.9. It shows the maximum allowable utilization of the rhythmic task, where the minimum flexion point is at $T_1^* = 92.5ms$, where $\frac{T_2}{2} + \frac{C_2}{2} \leq T_1^* = 92.5ms \leq T_2$, which is computed using Algorithm 8 and Theorem 17. An alternative way of determining the engine control task behavior is to use this information to ensure that the total utilization does not exceed this bound. The dotted



(a) Maximum acceleration duration n_α: 3



(b) Maximum acceleration duration n_α: 5

Figure 5.12: Plots of acceleration values for a rhythmic task that has varying period from $7.5ms$ to $120ms$ with 9 periodic tasks.

line in Figure 5.11 is the utilization curve corresponding to our recommendation above.

Figure 5.12 depicts the bound for Acceleration α considering the maximum allowable rate of period change with different acceleration durations. These curves are generated by using Algorithms 9, 10 and 11, where Algorithm 11 is modified such that it can handle GCRT. The plots use the recommended WCET corresponding to the dotted line from Figure 5.9. As shown in the figures, the acceleration bound plays a negative role. For example, we cannot accelerate the engine at all when T_1^* is $17ms$ or $34.7ms$ because the processor is fully utilized already. This can be avoided by delaying the timing of changing the execution mode from $17ms$/$34.7ms$ to later values. The effect of the acceleration duration is also shown in Figures 5.12(a) and 5.12(b). Figure 5.12(a) is when the maximum acceleration duration n_α is 3, and n_α is 5 in Figure 5.12(b). When the acceleration duration is longer, the acceleration bound becomes significantly low. This happens because a longer acceleration duration increases the number of preemptions of periodic tasks.

5.5 Summary

In automotive systems, safety-critical mechanical systems are being replaced by electronically controlled systems. A critical task not meeting its deadline can be catastrophic. In order to meet these stringent requirements, real-time scheduling techniques such as Rate Monotonic Scheduling (RMS) are used to guarantee the schedulability of the periodic tasks. However, the parameters of certain critical control tasks in cyber-physical systems depend on physical attributes of the system such as the speed of the engine in a car. The periods of these engine tasks vary dramatically depending on the engine speed. Conventional periodic task analysis is too conservative for handling such tasks. In this chapter, we have defined a new task model called *Rhythmic Tasks* for modeling tasks having continually varying periods depending on external physical events. To the best of our knowledge, this is the first model considering continually varying periods. We provide response-time analysis techniques for rhythmic tasks under constant engine speed, accelerating engine speed and decelerating engine speed along with schedulability tests. We have also provided guidelines to find schedulable utilization levels for the rhythmic task model. We apply our analyses and guidelines to a case-study desired from a real environment. The case study shows how the rhythmic task model is applicable to an existing CPS.

Dealing with multiple rhythmic tasks is an important and needed extension for cyber-physical systems. For example, the periods of planning and perception tasks in an autonomous vehicle [8] are functions of the vehicle speed, and the rates of their period changes depend on various environmental factors.

Future applications of rhythmic tasks include fault-tolerance support and autonomous vehicles. Rhythmic tasks can be replicated for fault-tolerance, and our techniques need to be extended. This rhythmic task model can also be used in autonomous vehicle systems. For example, perception algorithms for vision-based obstacle detection can be analyzed using the rhythmic task model.

Chapter 6

Tasks with Parallel Threads

With cyber-physical systems (CPS), such as medical devices, aerospace systems, smart grids, nuclear power plants, robots and transportation vehicles, becoming more popular, demands for new functionality features multiply [109]. For example, active safety options such as adaptive cruise

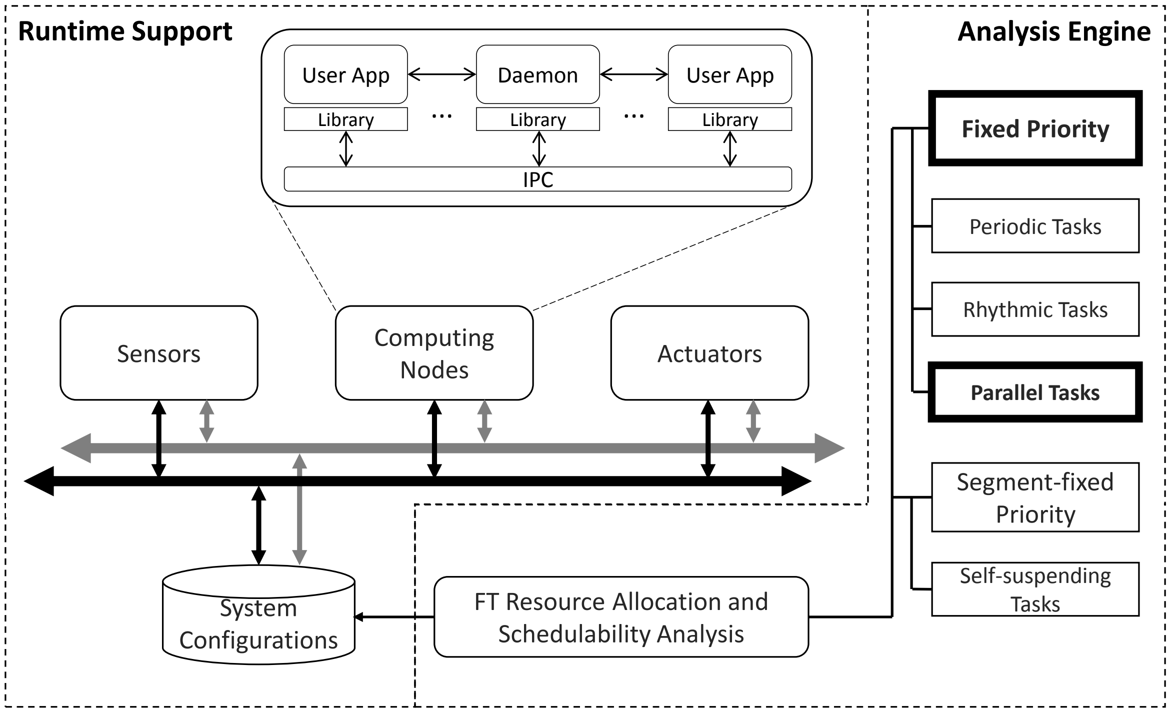


Figure 6.1: Tasks with parallel threads in the dissertation overview.

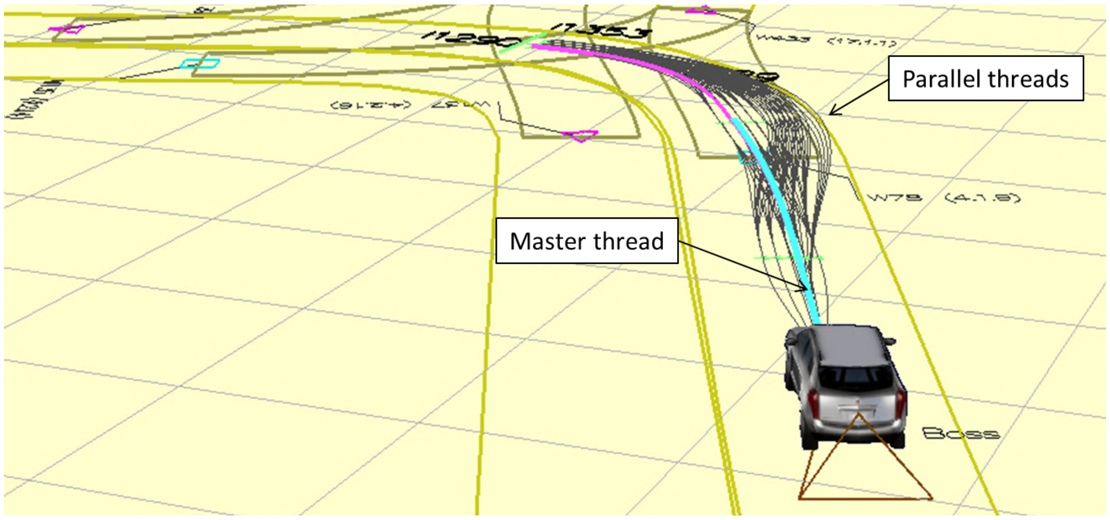


Figure 6.2: A motion planning algorithm for autonomous driving.

control, brake assist, collision avoidance, lane departure warning, sign detection and traction control are not rare anymore in recently built vehicles. We, in fact, expect these CPS functionalities to be readily available even in mid-range cars. With this trend, embedded real-time systems are indispensable in order to sense the physical environment, process data in real-time, control the actuators in a desirable manner and monitor the timing of the whole execution chain for ensuring safety.

Autonomous driving [8, 115, 116, 117, 118] is an appealing emerging CPS technology. In an autonomous car, motion planning, sensor fusion, computer vision and other artificial intelligence algorithms must run in real-time; however, the CPU-hogging nature of those algorithms poses challenges in guaranteeing their timeliness.

The timing challenge can be addressed by the fact that most algorithms for autonomous driving are *parallelizable*. A planning algorithm of a self-driving car can profit from parallelized tasks composed of numerous *threads*. The motion planning algorithm calculates the best path for the vehicle to follow among a myriad of potential paths. This algorithm can be expedited by parallelizing the cost calculation for each path. The more paths the algorithm goes through, the better driving quality will be. Figure 6.2 is a screenshot of the operator interface for Boss, which won the 2007 DARPA Urban Challenge [8] showing a motion planning algorithm in operation. In the figure, the multiple lines coming out from Boss represent possible paths which Boss may follow, where each line is generated by a parallel thread of the motion planning algorithm. When all threads are completed, they merge into a *master thread* that selects the best path. It should be noted that the number of threads can vary depending on the physical conditions such as the shape of the road, the number of detected obstacles and the speed of the vehicle.

A perception subsystem of a self-driving car can also benefit from parallel tasks. In order for the vehicle to understand its surroundings, the perception subsystem should be able to process massive amounts of data from various types of sensors. Boss, for example, manages 36000 independent segments from its Velodyne HDL-64 LIDAR before fusing them with other sensor data. Then, the vehicle can classify and track the detected obstacles, whose number has a major impact on how many parallel threads are spawned by the perception subsystem.

The automotive industry has already started moving towards the multi-core processors for higher performance [106, 119]. AUTOSAR, a widely used automotive software infrastructure, supports multi-core processors [101]. In addition, parallel programming models like `OpenMP` [120] utilize multiple processing cores to guarantee concurrent execution. We believe that other CPS application domains will follow this trend sooner rather than later.

There has been relatively little research on tackling challenges in scheduling parallel real-time tasks. In [1], Lakshmanan et al. proposed a parallel task model and a partitioned fixed-priority scheduling algorithm on a multi-core processor, but the number of threads could not exceed the number of given processing cores. In [53], Saifullah et al. proposed a more generalized parallel real-time task model which allows different fork-join segments of a task to have a different number of threads.

In this chapter, we extend the fork-join real-time task model proposed in [1] so that an arbitrary number of threads can be scheduled, where the number of threads can vary depending on the physical attributes of the system. To efficiently schedule the proposed task model, we also propose a task *stretch** transform to schedule the task model on a given number of processing cores. Then, we prove that a resource augmentation bound of 3.73 is achieved when we use the task *stretch** transform for global Deadline Monotoic (DM) scheduling for fork-join real-time tasks. The proposed scheme is implemented on Linux/RK [121] and ported to the self-driving car Boss [8]. We evaluate our proposed scheme on Boss by showing its driving quality in terms of curvature and velocity profiles of the vehicle with an enhanced motion planning algorithm [9].

In this chapter, we relax the assumption of sequential tasks that are made in Chapters 3 and 4 to incorporate parallel threads for real-time periodic tasks. We focus on the benefit of using parallelism in CPS as depicted in Figure 6.1. The rest of this chapter is organized as follows. In Section 6.1, we define our fork-join real-time task model and describe the system assumptions. We provide a scheduling algorithm to handle parallel real-time tasks in Section 6.2. The analysis using resource augmentation bound for global DM scheduling follows in Section 6.3. We, then, briefly explain in Section 6.4 the modifications made to Linux/RK to support the proposed scheme on a Linux-based system. Section 6.5 shows the curvature and velocity profiles of a self-driving car when our proposed scheme is used. In Section 6.6, we summarize our chapter and discuss future work.

6.1 System Model and Assumptions

Definition: We consider a set of tasks $\boldsymbol{\tau}$ composed of n multi-threaded real-time tasks, and the given set $\boldsymbol{\tau}$ runs on a system with m processing cores. $\boldsymbol{\tau}$ is represented as $\boldsymbol{\tau}$: $\{\tau_1, \tau_2, \ldots, \tau_n\}$, and the tasks in $\boldsymbol{\tau}$ are sorted in non-decreasing order of task periods (deadlines). Each task τ_i begins with a single thread spawning parallel threads, which join with another sequential thread of τ_i. τ_i interchanges this pattern between parallel and sequential segments. The number of parallel threads depends on the physical attributes of the given system $v_s \in \mathbb{R}^p$, where p is the number of dimensions that capture aspects of the operating environment. Then, as depicted in Figure 6.3,

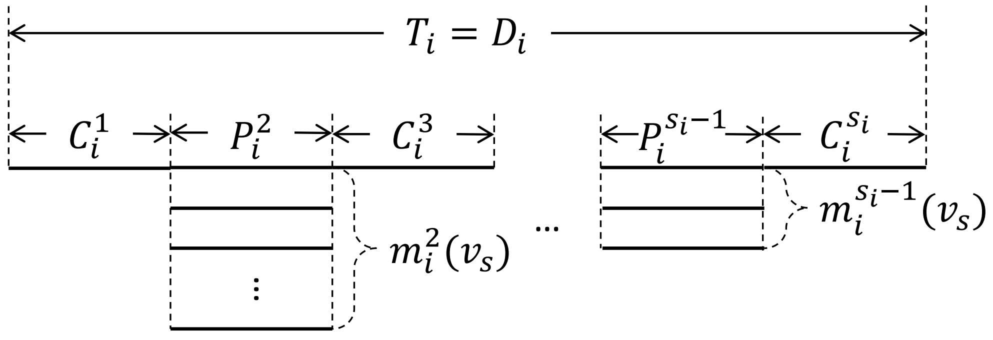


Figure 6.3: A fork-join real-time task model.

each task τ_i is represented as τ_i: $((C_i^1, (P_i^2, m_i^2(v_s)), C_i^3, \ldots, (P_i^{s_i-1}, m_i^{s_i-1}(v_s)), C_i^{s_i}), T_i, D_i)$, where

- s_i is the number of computation segments of τ_i. Since τ_i starts with a sequential segment and ends with a sequential segment while having parallel segments in the middle, s_i is a positive odd integer. For $1 \leq j \leq s_i$, the j^{th} element is a parallel segment if j is an even number. Similarly, the j^{th} element is a sequential segment if j is an odd number.
- $m_i^j(v_s)$ is the number of parallel threads for the j^{th} segment when $1 \leq j \leq s_i$. When j is an odd integer, $m_i^j(v_s)$ is 1 and omitted from the representation of τ_i above for ease of presentation. When j is an even integer, $m_i^j(v_s)$ is equal to or greater than 1 and represents the number of parallel threads spawned by the previous segment.
- C_i^j is the worst-case execution time of the j^{th} segment in task τ_i on a unit-speed processor when the j^{th} element is a sequential segment. Also, let $\tau_i^{j,1}$ denote the j^{th} sequential segment of τ_i.
- P_i^j is the worst-case execution time of each thread run in the j^{th} segment of task τ_i on a unit-speed processor when the j^{th} element is a parallel segment. For parallel segments of τ_i, each thread of parallel threads is represented as $\tau_i^{j,k}$, where k varies from 1 to $m_i^j(v_s)$.
- D_i is the relative deadline to its release time.
- T_i is the period of τ_i. An implicit deadline is assumed, i.e., $T_i = D_i$.

Application Examples to Autonomous Driving: The motion planning algorithm of Boss uses `OpenMP` to parallelize its cost calculations to find the best path. Since the algorithm takes its inputs: the road rules, the road shape, the vehicle speed, the list of static obstacles and the list of dynamic obstacles, we define v_s as $< RoadRule, RoadShape,$ $VehicleInfo, StaticObstacles, DynamicObstacles>$. This vector v_s is then used to decide the number of parallel threads accordingly. The perception algorithm of Boss leverages `pthread` to expedite its executions of processing perceived objects. We therefore define v_s for the perception algorithm as $<SensorList, SensorPose, RawSensorDataList,$ $VehiclePose>$. In this chapter, we consider the number of threads within each parallel segment not to exceed the maximum value of $m_i^j(v_s)$ for $\forall v_s \in \mathbb{R}^p$. For ease of presentation, therefore, we use m_i^j instead of $m_i^j(v_s)$.

Assumptions: Each task τ_i is assumed to generate an infinite series of independent jobs. The release time of the j^{th} segment of each job of τ_i should be after the completion time of the $(j-1)^{th}$ segment[1]. Therefore, if the j^{th} element of τ_i is a sequential segment, all parallel threads of $(j-1)^{th}$ segment of τ_i should complete before the j^{th} element of τ_i starts. We assume that all jobs are preemptable with negligible cost. We also assume that there is negligible migration cost when a job is migrated from a core to another.

Terminology: Using this model, we define the *maximum number of threads* of τ_i, which is the maximum value among m_i^j of τ_i. Formally,

$$m_i = \max_{j=1}^{s_i} m_i^j$$

The *maximum execution length* of a task τ_i on a unit-speed processor is defined as:

$$C_i = \sum_{j=0}^{\frac{s_i-1}{2}} C_i^{2j+1} + \sum_{j=1}^{\frac{s_i-1}{2}} m_i^{2j} P_i^{2j}$$

where, C_i represents the response time on a unit-speed single core processor when run alone. The first term corresponds to sequential task segments and the second term corresponds to fork-join

[1]We will use the terms 'jobs' and 'tasks' interchangeably where the distinction is not of importance.

segments.

To define the *minimum execution length* of a task τ_i, we have to consider two different cases: (1) $m_i \leq m$ and (2) $m_i > m$. For the first case, the minimum execution length is defined as $\eta_i = \sum_{j=0}^{\frac{s_i-1}{2}} C_i^{2j+1} + \sum_{j=1}^{\frac{s_i-1}{2}} P_i^{2j}$, where η_i is the response time when each single thread of τ_i can use a core exclusively. When $m_i > m$, the definition above does not hold good because some threads must be serialized. When $m_i > m$, therefore, we define the minimum execution length η_i as:

$$\eta_i = \sum_{j=0}^{\frac{s_i-1}{2}} C_i^{2j+1} + \sum_{j=1}^{\frac{s_i-1}{2}} \left\lceil \frac{m_i^{2j}}{m} \right\rceil P_i^{2j} \tag{6.1}$$

The definition above can also be used when $m_i \leq m$ because $\lceil \frac{m_i^j}{m} \rceil = 1$ when $m_i \leq m$. Hence, it holds good for both cases. For ease of presentation, we also let $P_i = \sum_{j=1}^{\frac{s_i-1}{2}} \left\lceil \frac{m_i^{2j}}{m} \right\rceil P_i^{2j}$, which is the execution requirement of the parallel segments contributing to η_i.

The task model in this chapter is extended from the fork-join task model[2] proposed in [1]. The two main differences between the previous one and this model are that (1) our model places no limitation on the number of threads, and (2) our model allows different number of threads per parallel segment. Hence, this model is more practical.

6.2 Scheduling Fork-Join Real-Time Tasks

It was shown in [1] that there are unschedulable task sets where the total utilization of the taskset is slightly greater and very close to 1 even though there are m processing cores. In other words, deadlines can be missed even though only $\frac{1}{m}$ of available cycles is used. Although m approaches infinity, the schedulability does not change [1]. This worst-case behavior continues to hold good for the proposed model because it is an extended form of the task model proposed in [1]. In this section, we first consider a scheduling method to handle fork-join real-time tasks on a processor with a given number of cores. Then, we propose the task *stretch** transform to deal with our

[2]We also call our proposed model a fork-join task model unless stated otherwise.

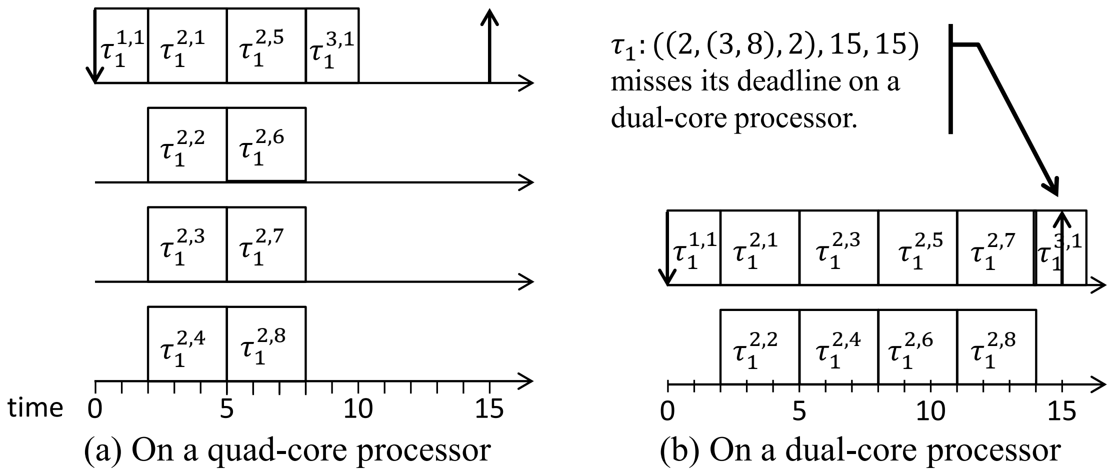


Figure 6.4: $\tau_1 : ((2,(3,8),2),15,15)$ misses its deadline on a dual-core processor, but not on a quad-core processor.

enhanced task model.

6.2.1 Running Fork-Join Real-Time Tasks on m CPU Cores

Consider a task $\tau_i \in \boldsymbol{\tau}$ running on m processing cores. If the maximum number m_i of parallel threads among all parallel segments in τ_i is less than the number of processing cores m, we can directly apply the task transformation algorithm described in [1]. If m_i exceeds the number of processing cores m, then the serialization of some parallel threads must happen as depicted in Figure 6.4, where a task meets its deadline on a quad-core processor, but not on a dual-core processor.

Proposition 19 *A fork-join real-time task τ_i requires at least the minimum execution length η_i units of time on m CPU cores to meet its deadline.*

We obtain the minimum execution length η_i of τ_i depicted in Figure 6.4 as 10 on a quad-core processor and 16 on a dual-core processor from Equation 6.1. From Proposition 19, we can show that the given task is infeasible on a dual-core processor because η_i on a dual-core processor is greater than its deadline.

6.2.2 The Task Stretch* Transform

We propose a task transformation algorithm *stretch** in Algorithm 12. It breaks down a fork-join real-time task into a set of tasks. This set is composed of a long task called a *master string* and a bunch of constrained-deadline tasks with $D < T$. This set can be scheduled using any scheduling algorithm supporting conventional single-threaded tasks such as global DM, global EDF [17] and FBB-FFD [22].

In Algorithm 12, when a new constrained-deadline task is created, it is represented as $\tau : (C, D, \phi)$, where C is the worst-case execution time, D is the relative deadline, and ϕ is the release offset. When a parallel thread is merged into an existing task, we use $\oplus$ as a symbol and $\tau : (C)$ as the thread added to the existing task. Merging a thread does not change either the deadline or the offset of the existing task. In this algorithm, we made a small change on how to use the modulo function. $k \mod q_i$ returns q_i if $k \mod q_i = 0$.

We use two parameters f_i and q_i in Algorithm 12. f_i is the ratio of the parallel execution requirements P_i to the slack of the task $T_i - \eta_i$. We use this value to evenly distribute the slack to each parallel segment. q_i is the number of parallel threads after a task is processed by Algorithm 12. In other words, at any point of time t, $\tau_i^{stretch^*}$ will have at most q_i concurrent running threads on m cores. It should be noted that the deadline assignment for the q_i^{th} thread is different from others because we split the thread so that we can avoid the worst-case scenario explained in [1].

The algorithm is an extension of the task *stretch* transformation proposed in [1]. The *stretch** transformation can handle more general cases: (1) when the number of parallel threads exceeds the number of cores, and (2) when the number of parallel threads of each segment is different. The improvements can be described as follows:

- If the number of parallel threads within a fork-join segment exceeds the number of CPU cores m, all parallel threads $\tau_i^{2j,k}$ with the same value of $(k \mod q_i)$, where $1 \leq k \leq m_i^{2j}$, coalesce into the thread $\tau_i^{2j,k \mod q_i}$. This step guarantees that the number of parallel

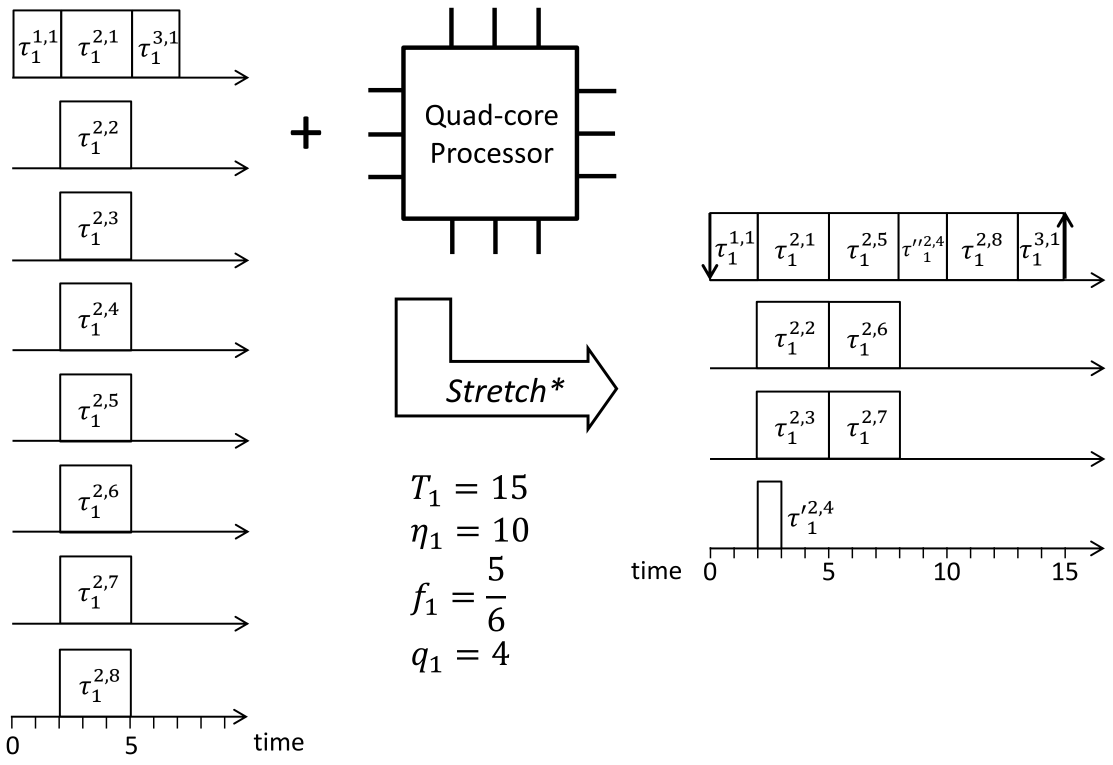


Figure 6.5: The task *stretch** transformation example with $\tau_1 : ((2, (3, 8), 2), 15, 15)$.

threads does not exceed the number of processing cores after the task transformation.

- Based on the new worst-case execution time of the merged threads of each parallel segment, a constrained deadline proportional to $(1 + f_i)$ is assigned to each parallel segment by the algorithm. Accordingly, an offset is also determined so that parallel threads are released at the right time instants.

Figure 6.5 shows an example of the task *stretch** transformation with a task τ_1: $((2, (3, 8), 2), 15, 15)$. The task has 8 parallel threads, and it has a slack of 5 because the minimum execution length η_1 is 10. Using the slack, a portion of $\tau_1^{2,4}$ and $\tau_1^{2,8}$ are scheduled with the master string.

6.3 Resource Augmentation Bound Analysis for Global Deadline Monotonic Scheduling

In this section, we derive the resource augmentation bound of global DM scheduling for the task model described in Section 6.2. To the best of our knowledge, this is the first result of resource augmentation bound of global DM scheduling for parallel real-time tasks. For this approach, we use a density-based schedulability test proposed in [50] given below.

Theorem 20 (from [50]) *A set of periodic or sporadic tasks with constrained deadlines is schedulable with Deadline-Monotonic priority assignment on $m \geq 2$ processors if:*

$$\lambda_{sum} \leq \frac{m}{2}(1 - \lambda_{max}) + \lambda_{max} \tag{6.2}$$

where, λ_{sum} is the sum of the density of each task in the taskset, λ_{max} is the maximum value of task densities, and a density λ is a ratio of the deadline of a task to its worst-case execution time.

Let $\lambda_i^{stretch^*}$ denote the sum of the density of each task in the stretched taskset $\tau_i^{stretch^*}$. As specified in Algorithm 12, two cases, (1) $C_i \leq T_i$ and (2) $C_i > T_i$ should be considered to understand the properties of $\lambda_i^{stretch^*}$. Two corresponding lemmas are presented next.

Lemma 21 *For a fork-join real-time task τ_i, the density of the resulting stretched task $\tau_i^{stretch^*}$ is bounded by $\frac{C_i}{T_i}$ if $C_i \leq T_i$.*

Proof For the case of $C_i \leq T_i$, we use the fact that the execution requirement and $T_i (= D_i)$ of both τ_i and $\tau_i^{stretch^*}$ are equal. Then, from the definition of density, $\frac{C_i}{T_i}$.

Before investigating a fork-join real-time task τ_i with $C_i > T_i$, we assume that τ_i is provided with a level of parallelism so that $\frac{C_i}{min(m,m_i)} \geq P_i$ is satisfied. In the ideal case, based on Amdahl's law [122], $\frac{C_i}{min(m,m_i)} = P_i$ holds good because all the segments are running in parallel. Since we assume a fork-join model that has non-zero serial segments, the ideal case cannot be achieved. However, approaching P_i to $\frac{C_i}{min(m,m_i)}$ is desirable to fully utilize parallelism.

Lemma 22 *For a fork-join real-time task τ_i, the sum of the density of the resulting stretched $\tau_i^{stretch^*}$ is bounded by $\frac{C_i}{T_i - \eta_i}$ if $C_i > T_i$.*

Proof For the case of $C_i > T_i$, it should be noted that the output of the algorithm is a set of tasks composed of a master thread τ_i^{master} and several constrained deadline tasks $\{\tau_i^{cd}\}$. Hence, the following inequality holds good:

$$\lambda_i^{stretch^*} \leq \lambda_i^{master} + \sum_{\tau_i \in \{\tau_i^{cd}\}} \lambda_i$$

Since the worst-case execution time of τ_i^{master} is less than T_i, $\lambda_i^{master} \leq 1$ from the implicit deadline assumption. It is known that there will be at most q_i concurrent running threads including the master thread at any point of time t. We ensure this by assigning an offset whenever a new parallel thread is created in Algorithm 12. The offset also guarantees that only one segment is active at a time. Thus, the density of τ_i can be substituted with the density of a segment that has the largest value among the densities of the segments of τ_i.

Let $P_i^{max} = \max_{j=1}^{\frac{s_i-1}{2}} \lceil \frac{m_i^{2j}}{m} \rceil P_i^{2j}$. We first consider the case of $q_i > 2$. When q_i threads are simultaneously running, for the $q_i - 1$ constrained tasks, there will be $q_i - 2$ parallel threads with the execution time of P_i^{max} and the relative deadline of $(1+f_i)P_i^{max}$. There will also be a parallel thread with the execution time of $(1+\lfloor f_i \rfloor - f_i)P_i^{max}$ and the relative deadline of $(1+\lfloor f_i \rfloor)P_i^{max}$. Therefore, if we let $P_i = \sum_{j=1}^{\frac{s_i-1}{2}} \left\lceil \frac{m_i^{2j}}{m} \right\rceil P_i^{2j}$, the following inequalities are satisfied:

$$\begin{aligned}\sum_{\tau_i \in \{\tau_i^{cd}\}} \lambda_i &\leq \frac{(q_i - 2)P_i^{max}}{(1+f_i)P_i^{max}} + \frac{(1+\lfloor f_i \rfloor - f_i)P_i^{max}}{(1+\lfloor f_i \rfloor)P_i^{max}} \\ &\leq \frac{(q_i - 1)}{(1+f_i)} = \frac{(q_i - 1)P_i}{(P_i + T_i - \eta_i)}\end{aligned}$$

We then consider the case of $0 < q_i \leq 2$. When q_i is 1, it means that τ_i can run on a single core. Therefore, we focus on the case of $q_i = 2$, which means that $\sum_{\tau_i \in \{\tau_i^{cd}\}} \lambda_i$ will have only the task which is split. Therefore,

$$\begin{aligned}\sum_{\tau_i \in \{\tau_i^{cd}\}} \lambda_i \leq \frac{(1+\lfloor f_i \rfloor - f_i)P_i^{max}}{(1+\lfloor f_i \rfloor)P_i^{max}} &\leq \frac{1}{(1+f_i)} \\ = \frac{P_i}{(P_i+T_i-\eta_i)} &= \frac{(q_i-1)P_i}{(P_i+T_i-\eta_i)}\end{aligned}$$

Now, we consider both τ_i^{master} and $\{\tau_i^{cd}\}$.

$$\begin{aligned}\lambda_i^{stretch^*} &\leq 1 + \frac{(q_i-1)P_i}{(P_i+T_i-\eta_i)} = \frac{P_i+T_i-\eta_i+(q_i-1)P_i}{(P_i+T_i-\eta_i)} \\ &= \frac{(f_i+q_i)P_i}{(P_i+T_i-\eta_i)} = \frac{(f_i+min(m,m_i)-\lfloor f_i \rfloor)P_i}{(P_i+T_i-\eta_i)} \\ &\leq \frac{min(m,m_i)P_i}{(P_i+T_i-\eta_i)} \leq \frac{C_i}{T_i-\eta_i}\end{aligned}$$

From the inequality above, the lemma is proved.

We define a task called a *heavy task* that has a density greater than or equal to $\frac{1}{\nu}$ on a ν-speed processing core.

Theorem 23 *Global Deadline Monotonic scheduling of the fork-join real-time task model has a resource augmentation bound of 3.73 when each heavy task is assigned to its own processing core.*

Proof Consider a set of n fork-join real-time tasks $\boldsymbol{\tau}$. We assume that the given taskset is feasible on m identical unit-speed processors, which implies $\sum_{i=1}^{n} \frac{C_i}{T_i} \leq m$. Otherwise, the given taskset is not feasible.

Let there be k heavy tasks on a ν-speed processor. Under the task *stretch** transform described in Algorithm 12, these are either *fully stretched* tasks ($C_i \leq T_i$) or master threads ($C_i > T_i$). Both types of tasks have a deadline equal to their period, and their density is at least 1 on a unit-speed processor by the definition of a heavy task.

Therefore, for the remaining n' tasks:

$$\sum_{i=1}^{n'} \frac{C_i}{T_i} = \sum_{i=1}^{n'} \frac{C_i}{D_i} = \sum_{i=1}^{n'} \lambda_i = \lambda_{sum} \leq (m-k) \tag{6.3}$$

We need to show that these remaining tasks are schedulable on $m'(= m - k)$ processors of speed ν, where $\nu \geq 3.73$.

On a processor that is ν times faster, the minimum execution length η_i^ν on a ν-speed processor is given by

$$\eta_i^\nu = \sum_{j=0}^{\frac{s_i-1}{2}} \frac{C_i^{2j+1}}{\nu} + \sum_{j=1}^{\frac{s_i-1}{2}} \left\lceil \frac{m_i^{2j}}{m} \right\rceil \frac{P_i^{2j}}{\nu} \leq \frac{\eta_i}{\nu} \leq \frac{T_i}{\nu} \tag{6.4}$$

where, $\forall 1 \leq i \leq n$. Also, the maximum execution length of τ_i on a ν-speed processor is

$$C_i^\nu = \sum_{j=0}^{\frac{s_i-1}{2}} \frac{C_i^{2j+1}}{\nu} + \sum_{j=1}^{\frac{s_i-1}{2}} m_i^{2j} \frac{P_i^{2j}}{\nu} = \frac{C_i}{\nu} \tag{6.5}$$

where, $\forall 1 \leq i \leq n$.

Case (1): For each fully stretched task τ_i that is non-heavy on ν-speed processors, the density is $\frac{\frac{C_i}{\nu}}{T_i} \leq \frac{1}{\nu}\frac{C_i}{T_i} \leq \frac{1}{\nu-1}\frac{C_i}{T_i}$ from Lemma 21 and Equation 6.5.

Case (2): Consider the constrained-deadline taskset generated by *stretch** on ν-speed processors for task τ_i. From the perspective of load, the total density on ν-speed processors is bounded by $\frac{C_i^\nu}{T_i - \eta_i^\nu} \leq \frac{C_i/\nu}{T_i - \frac{T_i}{\nu}} = \frac{1}{\nu-1}\frac{C_i}{T_i}$ from Lemma 22, Inequality 6.4 and Equation 6.5.

λ_{sum} on ν-speed processors, therefore, is bounded by $\frac{m'}{\nu-1}$ because $\lambda_{sum} \leq \sum_{i=1}^{n'} \frac{1}{\nu-1}\frac{C_i}{T_i} = \frac{1}{\nu-1}\sum_{i=1}^{n'} \frac{C_i}{T_i} \leq \frac{m'}{\nu-1}$ from Inequality 6.3. The master threads for tasks that cannot be fully stretched are always heavy tasks since they use up the entire T_i on the ν-speed processor. By the definition of heavy tasks, λ_{max} is always upper bounded by $\frac{1}{\nu}$ on ν-speed processors. Then, for $m' \geq 2$ using Inequalities 6.2 and 6.3 and the cases considered above,

$$\begin{aligned}
&\frac{m'}{2}\left(1 - \frac{1}{\nu}\right) + \frac{1}{\nu} \geq \frac{m'}{\nu - 1} \\
\Leftrightarrow &\frac{m'}{2} - 1 \leq \nu \left(\frac{m'}{2} - \frac{m'}{\nu - 1}\right) \\
\Leftrightarrow &m' \frac{4\nu - \nu^2 - 1}{2\nu(\nu - 1)} \leq \frac{1}{\nu} \\
\Leftrightarrow &\frac{4\nu - \nu^2 - 1}{2(\nu - 1)} \leq \frac{1}{m'}
\end{aligned}$$

As $m' \to \infty$, we get,

$$\frac{4\nu - \nu^2 - 1}{2(\nu - 1)} \leq \frac{1}{m'} \Leftarrow \nu \geq 2 + \sqrt{3}$$

This holds good for all $m' \geq 2$ processors using $\nu \geq 2 + \sqrt{3} \approx 3.73$.

6.4 Global Scheduling on Linux/RK

We have designed an operating system abstraction for managing our parallel real-time task model using the resource-reservation paradigm. A parallel task in our model is composed of multiple threads. A thread called *master string* executes all sequential segments and a portion[3] of parallel segments. Parallel threads are spawned by the master thread and execute the remaining portion of parallel segments. In order to represent the multiple threads and their precedence constraints, our abstraction employs the resource management entities, *resource set* and *reserve*, introduced in resource kernels [123], where

- *Resource set*: A resource set corresponds to a parallel task. It is a container of multiple reserves.
- *Reserve*: A reserve represents the amount of CPU budget to be reserved on a single core or multiple cores. A reserve is specified with (C, T, D, ϕ): C is a worst-case execution time; T is a period; D is a relative deadline; ϕ is a release offset.

Figure 6.6 shows the scheduling of a parallel real-time task on four cores with the *stretch** transformation. The parallel task τ_1 has one parallel segment comprising four threads. The *stretch** transformation splits the last thread of the parallel segment, $\tau_1^{2,4}$, into $\tau'^{2,4}_1$ and $\tau''^{2,4}_1$. Hence, $\tau'^{2,4}_1$ is assigned a relative deadline of 8 that is equal to the release offset of $\tau''^{2,4}_1$. The CPU usage and its offset on each core can be represented as a reserve. Since a reserve is equivalent to an individual sequential periodic task, the global DM scheduling algorithm can determine the scheduling priorities for reserves. Then, we assign reserves to threads so that each thread

[3]This portion is obtained by running Algorithm 12.

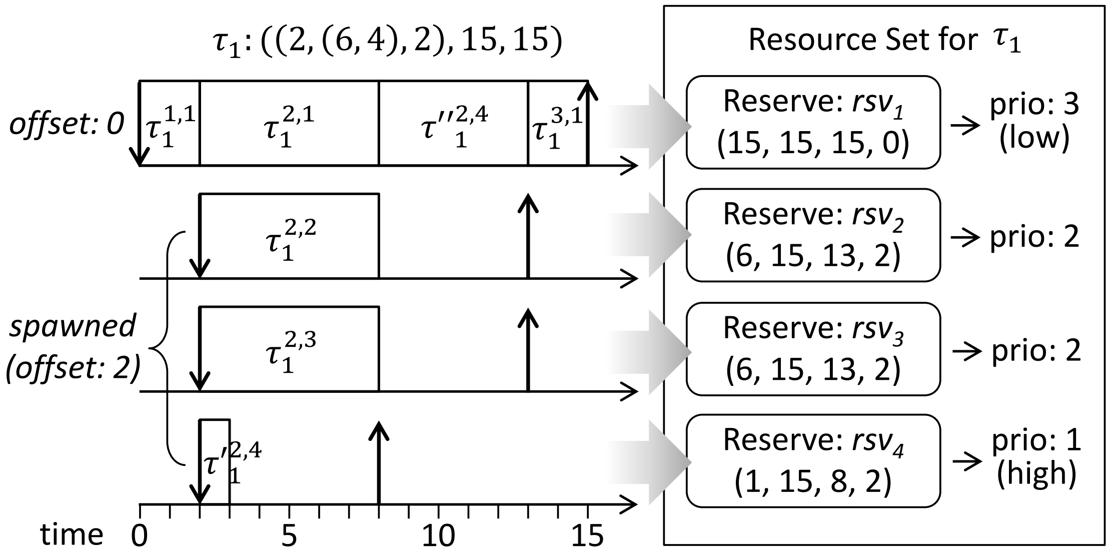

τ_1: ((2, (6, 4), 2), 15, 15)
offset: 0
$\tau_1^{1,1}$
$\tau_1^{2,1}$
$\tau''^{2,4}_1$
$\tau_1^{3,1}$
spawned (offset: 2)
$\tau_1^{2,2}$
$\tau_1^{2,3}$
$\tau'^{2,4}_1$
time
0
5
10
15
Resource Set for τ_1
Reserve: rsv_1 (15, 15, 15, 0)
prio: 3 (low)
Reserve: rsv_2 (6, 15, 13, 2)
prio: 2
Reserve: rsv_3 (6, 15, 13, 2)
prio: 2
Reserve: rsv_4 (1, 15, 8, 2)
prio: 1 (high)

Figure 6.6: CPU resource abstraction for a parallel task with global DM scheduling.

is scheduled with the priority and the release offset of the assigned reserve and consumes the reserve's CPU budget. The master string thread, $(\tau_1^{1,1} \rightarrow \tau_1^{2,1} \rightarrow \tau_1^{3,1})$, is assigned a reserve (rsv_1). The second and the third thread in the parallel segment, $\tau_1^{2,2}$ and $\tau_1^{2,3}$, are assigned (rsv_2) and (rsv_3), respectively. The last thread $\tau_1^{2,4}$ is assigned an ordered list of reserves, $(rsv_4 \rightarrow rsv_1)$. This means that $\tau_1^{2,4}$ first uses rsv_4's priority and CPU budget, and when it uses up rsv_4's budget, it continues its execution with rsv_1's priority and remaining CPU budget.

We implemented the abstraction for parallel tasks on Linux/ RK [121], which is based on the Linux 2.6.38.8 kernel. We used `hrtimers` to release threads at specified offsets and to account the CPU usage of threads. When a thread uses up all reserves assigned to it, the abstraction enforces the CPU usage of the thread by suspending it. The accounting and the enforcement of our abstraction can also be used for the measurement-based worst-case-execution-time estimation of threads in a parallel task, by checking an occurrence of the enforcement with a tentative execution time.

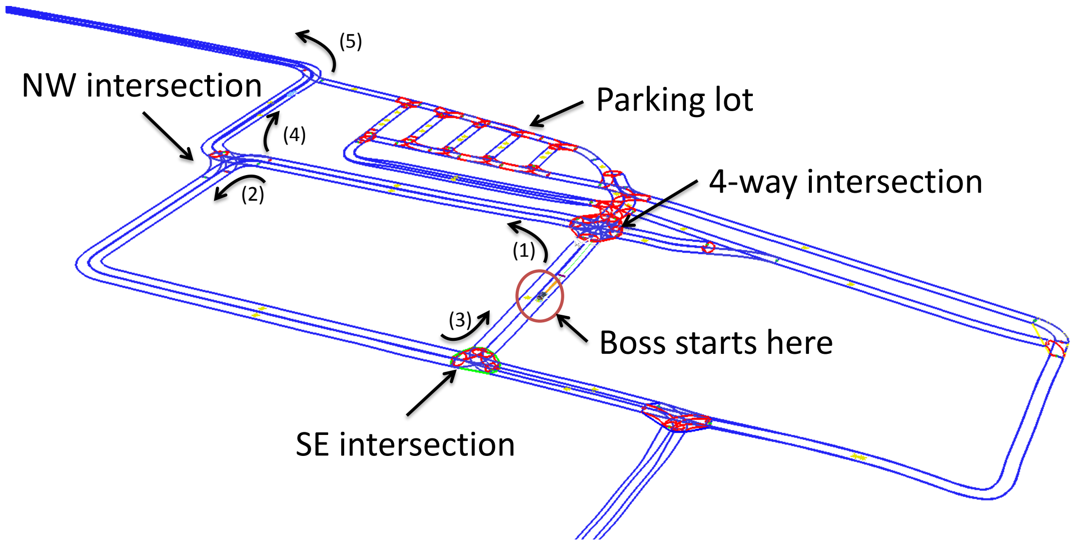


Figure 6.7: The map followed by Boss.

6.5 Case Study on Self-Driving Car

We studied the efficacy of our proposed scheme using a self-driving car platform Boss. The latest motion planning algorithm running on Boss [9] is used for our evaluation. The algorithm considers the distance to the next destination, the lateral offset of the car to the center of the lane, the longitudinal velocity, the longitudinal acceleration, the lateral acceleration and a list of static/dynamic obstacles on the road where the vehicle is driving. With the given information based on which the number of parallel threads varies, the algorithm generates curvature and velocity profiles for the path which the vehicle should follow. The planning algorithm is implemented using `OpenMP`, and we evaluate the quality of autonomous driving by analyzing curvature and velocity profiles of Boss (1) when the conventional reservation approach with Linux/RK [121] is used, (2) when the previous task model [1] is used, and (3) when our proposed task model and algorithm are used.

We ran the planning algorithm on a simulation cluster [124, 125] equipped with an Intel Core i7 quad-core processor. Although we run the exact same algorithm on the vehicle, we measure the results on the simulation cluster due to testing, convenience and safety considerations. We ran a scenario with the layout of our test track located at Robot City in Hazelwood, Pittsburgh, PA, where we test the vehicle at straight multi-lane roads, curvy roads, intersections, U-turns and parking lots. The exact same scenario file is also used during the field test, but the tasks for receiving raw sensor data are replaced with simulation tasks. In Figure 6.7, the test track for the scenario is illustrated. Boss will depart at the point circled in the middle of Figure 6.7. Boss will follow the road, (1) cross a 4-way intersection governed by stop signs, follow the straight road and (2) make a left turn at NW intersection. Then, Boss will (3) make a left turn at SE intersection, proceed to NW intersection and (4) turn right towards the curve marked with (5) connecting to the long straight road.

The scenario is composed of eight tasks: `BehaviorTask`, `MissionPlannerTask`, `OnRoadMotionPlannerTask`, `PrePlannerTask`, `RobotClient`, `ServerTask`, `RoadBlockageDetector`, and `SimpleControllerTask`. The `BehaviorTask` decides what to do such as turning, intersection handling and lane changing. The `MissionPlannerTask` interacts with the stored map to decide where to go. The `OnRoadMotionPlannerTask` and the `PrePlannerTask` send trajectories to the vehicle controller. The `RoadBlockageDetector` works with the `BehaviorTask` so that the vehicle detects the blocked road and finds an alternate route when needed. The `SimpleControllerTask` receives the actuator commands and directly interfaces with the vehicle hardware such as the accelerator, the brake and the steering wheel. On the simulation cluster, this task operates in simulation mode, and the `ServerTask` and the `RobotClient` behave as the vehicle hardware. In this chapter, our focus is on the `OnRoadPlannerTask` running the motion planning algorithm [9] with `OpenMP` enabled. The task generates curvature and velocity profiles for the vehicle hardware, so the lack of resources will affect the control algorithm, making the car drive in an unstable manner. If the planning algorithm does not meet the deadline, the steering wheel, for example, jerks and the car goes to an unexpected place, which can cause an accident.

Figure 6.8 shows the autonomous driving performance, i.e., the curvature and velocity pro-

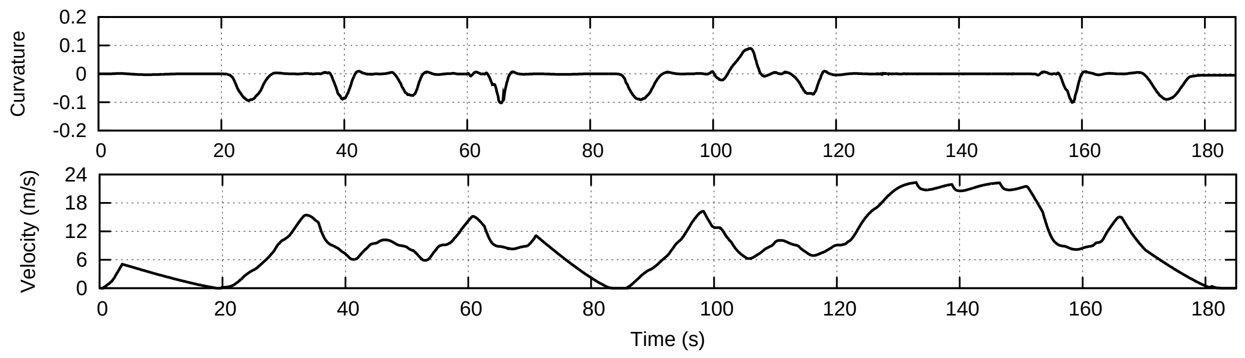


Figure 6.8: Curvature and velocity profiles during the entire journey of Boss illustrated in Figure 6.7.

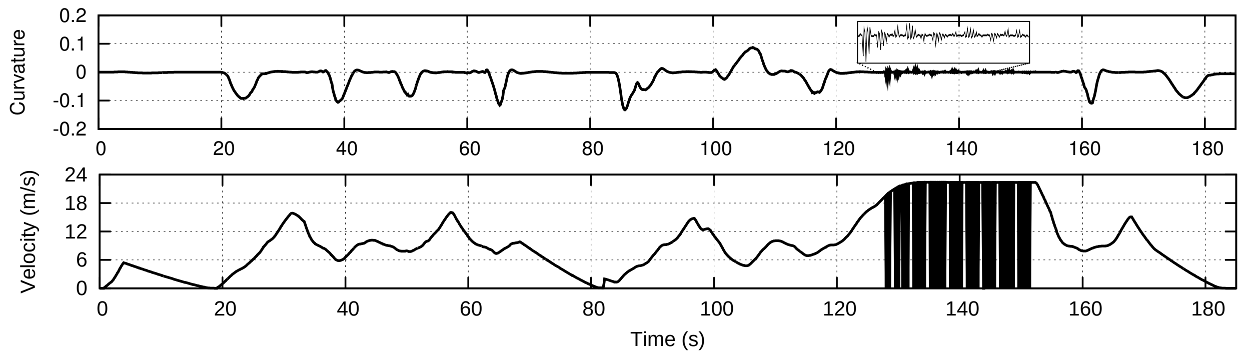


Figure 6.9: Curvature and velocity profiles of Boss when conventional resource reservation is used.

files collected from the output of `OnRoadMotionPlannerTask` when the proposed task model and algorithm are used with a varying number of threads. We limit the maximum number of threads to 50. The curvature graph shows when Boss makes turns; a negative value means a left rotation of steering wheel, and vice versa. For example, Boss arrives at the SE intersection in Figure 6.8 around $t = 65s$, and that is the fourth valley in the curvature graph. Accordingly, we can see the velocity of Boss decreases to turn left. The bigger the absolute value of curvature, the steeper will be the turn made by Boss. From the perspective of autonomous driving quality,

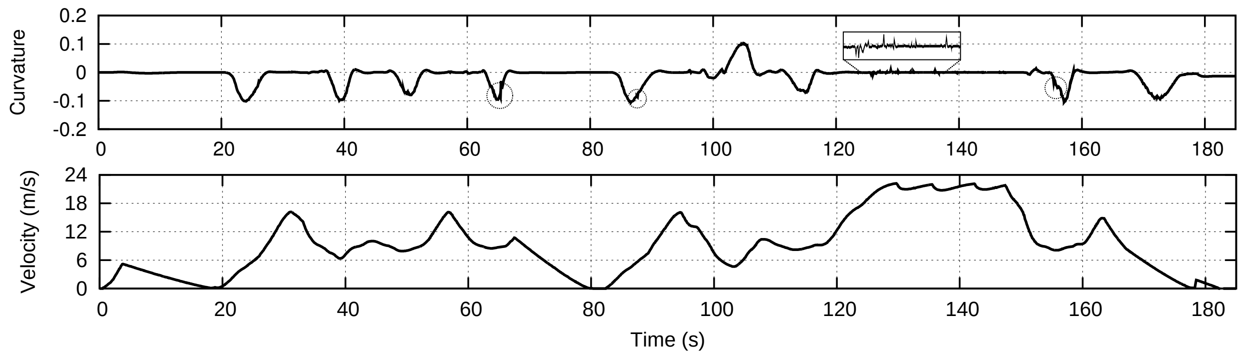


Figure 6.10: Curvature and velocity profiles of Boss when previously known techniques [1] are used.

a sudden control change on an actuator is not desirable.

Figure 6.9 shows an undesirable case when the *conventional* resource reservation approach with Linux/RK is used. Since the traditional Linux/RK does not consider a parallel task model, it assigns all child threads into a reserve allocated to a processing core. Since this may prevent the planning algorithm from running in parallel, the planner may not be able to meet its deadline, which is shown from $130s$ to $150s$ in Figure 6.9. The planning algorithm requires more threads when a car is moving faster and/or when a car is making a sharp turn. The results shown, therefore, are consistent with the property of the planning algorithm. Figure 6.10 also shows the result when the model of [1] is used, where only four threads can run in parallel because the simulation cluster has a quad-core processor. For this case, the velocity profiles are fine, but the curvatures show some jitters that can make the vehicle unstable and also uncomfortable for passengers. The results shown in Figure 6.9 and 6.10 could be potentially dangerous on the real vehicle because the vehicle in the real-world may slip, drift and crash.

6.6 Summary

To meet rapidly increasing demands for complex cyber-physical systems, we motivated the necessity of using multi-core processors and corresponding parallel programming models such as `OpenMP` [120]. In particular, emerging CPS such as a self-driving vehicle can benefit significantly from parallel real-time tasks allowing multiple compute-intensive real-time tasks to support demanding requirements. Thus, a self-driving vehicle can model its physical surroundings in parallel and react to them in real-time. In this chapter, we proposed a fork-join parallel real-time task model, where the amount of parallel executions can vary depending on the physical attributes of the system. The proposed task model is transformed using our *stretch** algorithm. With global deadline-monotonic scheduling, we obtained a resource augmentation bound of 3.73, which means that any task set that is feasible on m unit-speed processors can be scheduled by the proposed algorithm on m processors that are 3.73 times faster. The proposed scheme was implemented on Linux/RK [121] as a proof of concept, and ported to Boss, the self-driving car that won the 2007 DARPA Urban Challenge [8]. On Boss, we evaluated our proposed scheme that improved its autonomous driving quality. Future work to be done includes supporting dynamic changes of periods and execution times of parallel real-time tasks. We already have early work on varying periods [126], and the dynamic nature of CPS will be addressed using this model combined with parallel tasks.

Algorithm 12 Stretch$^*(\tau)$

```
Require: $\tau$: a fork-join real-time task
Ensure: $\tau^{stretch^*}$: a stretch*ed task set
1: $\tau_i^{master} \leftarrow ()$
2: $\{\tau_i^{cd}\} \leftarrow \{\}$
3: if $C_i \leq T_i$ then
4:     ▷ The task can run on a single core
5:     for $j = 1$ to $\frac{s_i-1}{2}$ do
6:         $\tau_i^{master} \leftarrow \tau_i^{master} \oplus \tau_i^{2j-1,1} : (C_i^{2j-1})$
7:         for $k = 1$ to $m_i^{2j}$ do
8:             $\tau_i^{master} \leftarrow \tau_i^{master} \oplus \tau_i^{2j,k} : (P_i^{2j})$
9:     $\tau_i^{master} \leftarrow \tau_i^{master} \oplus \tau_i^{s_i,1} : (C_i^{s_i})$
10: else
11:     ▷ Stretch* the task to its deadline
12:     $f_i \leftarrow \frac{T_i-\eta_i}{\sum_{j=1}^{\frac{s_i-1}{2}} \lceil \frac{m_i^{2j}}{m} \rceil P_i^{2j}}$
13:     $q_i \leftarrow \min(m, m_i) - \lfloor f_i \rfloor$
14:     for $j = 1$ to $\frac{s_i-1}{2}$ do
15:         $\tau_i^{master} \leftarrow \tau_i^{master} \oplus \tau_i^{2j-1,1} : (C_i^{2j-1})$
16:         ▷ 1) Coalesce threads so that the total number of parallel threads is less than $q_i$
17:         for $k = 1$ to $m_i^{2j}$ do
18:             if $k \mod q_i = 1$ then
19:                 ▷ Part of the master string
20:                 $\tau_i^{master} \leftarrow \tau_i^{master} \oplus \tau_i^{2j,k} : (P_i^{2j})$
21:             else if $\tau_i^{2j,k \mod q_i} \notin \{\tau_i^{cd}\}$ then
22:                 ▷ Create a new parallel thread
23:                 $D_i^{2j} \leftarrow (1 + f_i)\lceil \frac{m_i^{2j}}{m} \rceil P_i^{2j}$
24:                 $\phi_i^{2j} \leftarrow \sum_{l=0}^{j-1} C_i^{2l+1} + \sum_{l=1}^{j-1} D_i^{2l}$
25:                 $\{\tau_i^{cd}\} \leftarrow \{\tau_i^{cd}\} \cup \tau_i^{2j,k \mod q_i} : (P_i^{2j}, D_i^{2j}, \phi_i^{2j})$
26:             else if $\tau_i^{2j,k \mod q_i} \in \{\tau_i^{cd}\}$ then
27:                 ▷ Part of the existing threads
28:                 $\tau_i^{2j,k \mod q_i} \leftarrow \tau_i^{2j,k \mod q_i} \oplus \tau_i^{2j,k} : (P_i^{2j})$
29:         ▷ 2) Split among the $q_i$-th thread and the master string
30:         if $\tau_i^{2j,q_i} \in \{\tau_i^{cd}\}$ then
31:             $\{\tau_i^{cd}\} \leftarrow \{\tau_i^{cd}\} - \tau_i^{2j,q_i}$
32:             $\tau_i^{master} \leftarrow \tau_i^{master} \oplus \tau_i'^{2j,q_i} : ((f_i - \lfloor f_i \rfloor)\lceil \frac{m_i^{2j}}{m} \rceil P_i^{2j})$
33:             ▷ Create a new parallel thread
34:             $D_i^{2j,q_i} \leftarrow (1 + \lfloor f_i \rfloor)\lceil \frac{m_i^{2j}}{m} \rceil P_i^{2j}$
35:             $\phi_i^{2j} \leftarrow \sum_{l=0}^{j-1} C_i^{2l+1} + \sum_{l=1}^{j-1} D_i^{2l}$
36:             $\{\tau_i^{cd}\} \leftarrow \{\tau_i^{cd}\} \cup \tau_i''^{2j,k} : ((1 + \lfloor f_i \rfloor - f_i)\lceil \frac{m_i^{2j}}{m} \rceil P_i^{2j}, D_i^{2j,q_i}, \phi_i^{2j})$
37:     $\tau_i^{master} \leftarrow \tau_i^{master} \oplus \tau_i^{s_i,1} : (C_i^{s_i})$
38: return $\tau_i^{stretch^*} := \left(\tau_i^{master}, \{\tau_i^{cd}\}\right)$
```

Chapter 7

Tasks with Self-Suspensions

Recent trends in System-on-a-Chip (SoC) show that an increasing number of special-purpose processors in these systems are added to improve the efficiency of frequently-used operations [127]. Figure 7.2 illustrates a high-level diagram of a modern SoC composed of various subsystems

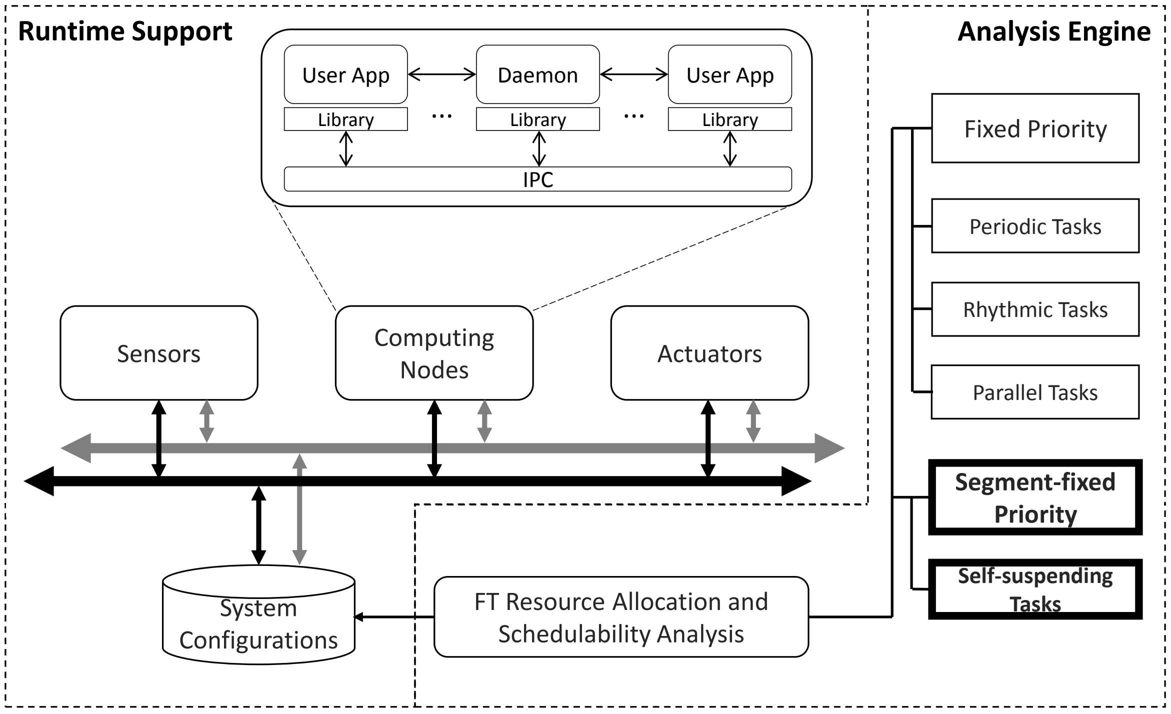


Figure 7.1: Tasks with self-suspensions in the dissertation overview.

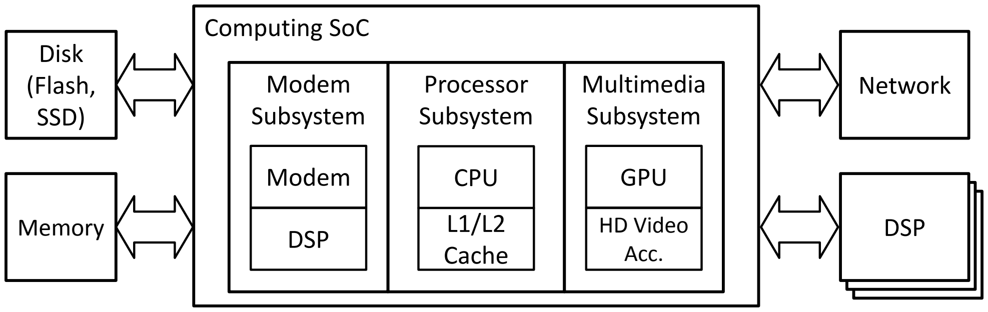


Figure 7.2: Modern SoC architecture.

such as application processor and multimedia and communication subsystems. Unfortunately, the use of such special-purpose processors (a.k.a. hardware accelerators) may introduce suspension delays that must be taken into account in the schedulability when a task waits for a shared resource and interact with an I/O device or communication interface. Offloading complex computations to hardware accelerators such as Digital Signal Processors (DSPs) or Graphics Processing Units (GPUs) can cause suspension delays as well. Many conventional real-time theories [10] have incorporated the delays in the worst-case execution/response time of a task that suspends itself. Even though the analyses can guarantee the timeliness of systems, the analysis results may have significant pessimism.

A pessimistic analysis is not desirable in a compute-intensive system such as the self-driving car that we have recently developed [2]. Such systems run computation-demanding algorithms ranging from perception [11] to planning [9, 12] on GPUs in real-time. In this case, if we use traditional schedulability analysis, the potential utilization improvement due to the use of GPUs is eliminated by the pessimism in the CPU scheduling.

In this chapter, we present a new scheme to schedule self-suspending tasks to improve their schedulable utilization. To derive our new scheme we first study the schedulability of these tasks under RMS [16] that is widely used in embedded real-time OSes like OSEK and general-purpose OSes such as Linux. RMS is also known to be the optimal fixed-priority scheduling policy. Explicitly modeling self-suspending real-time tasks is desirable to remove the pessimism described above, but it breaks a common assumption of RMS that tasks do not suspend themselves during run-time, making RMS not directly applicable. Since such self-suspending behaviors can cause unexpected jitters, the critical scheduling instant and utilization bound test defined and proved in [16] do not always hold for self-suspending tasks. Therefore, RMS is not an optimal scheduling algorithm for this type of tasks. In other words, there exist other scheduling algorithms that can schedule tasksets that cannot be scheduled under RMS.

Research on self-suspending tasks is limited. In [56] the authors proved that the problem of scheduling self-suspending tasks is *NP-hard* in the strong sense. There has also been recent work on scheduling self-suspending tasks for soft real-time systems [59].

In this chapter, we provide schedulability analyses for segment-fixed priority scheduling for self-suspending tasks. We provide response-time analyses for self-suspending tasks with RMS [16] and identify the conditions when RMS can be used without modifications. We then derive a utilization bound as a function of the ratio of suspension time to the task period when RMS is *compatible*.

To improve the schedulability of a taskset that is not compatible with RMS, we propose the segment-fixed priority scheduling that decomposes self-suspending tasks into multiple segments assigning them different priorities if needed. We use phase enforcement to prevent jitters [13, 14]. Finally, we developed an exact schedulability analysis and evaluate it with randomly generated tasksets.

In this chapter, we relax the assumption of sequential tasks that are made in Chapters 3 and 4 to incorporate self-suspensions for real-time periodic tasks as depicted in Figure 7.1. The rest of this chapter is organized as follows. In Section 7.1, we define the self-suspending task model and represent the overall system assumptions. Section 7.2 provides schedulability analyses for self-suspending tasks when a task-fixed priority scheduling is used. Then, in Section 7.3, we propose a new scheme called segment-fixed priority scheduling to overcome the drawbacks of task-fixed priority scheduling. Section 7.4 shows evaluation results of the proposed schemes.

Finally, we conclude our chapter and discuss future work in Section 7.5.

7.1 System Model and Assumptions

A system is modeled as a taskset $\mathbf{\Gamma}$: $\{\tau_1, \tau_2, \ldots, \tau_n\}$ running on a single processor. Each task τ_i generates an infinite number of jobs. Each job consists of multiple computing segments with a minimum suspension interval between each pair of segments. In other words, each job alternates between a computing segment and a suspending stage[1]. A job finishes when the last computing segment of the job is completed. As illustrated in Figure 7.3, each task τ_i is represented as τ_i: $((C_{i,1}, G_{i,1}, C_{i,2}, \ldots, C_{i,s_i-1}, G_{i,s_i-1}, C_{i,s_i}), T_i, D_i)$, where

- $C_{i,j}$ is the worst-case execution time of the j^{th} segment of τ_i on a unit-speed processor. We also let $\tau_{i,j}$ denote the j^{th} segment of τ_i.
- $G_{i,j}$ is a time gap between $\tau_{i,j}$ and $\tau_{i,j+1}$, where $1 \leq j < s_i$. In other words, $\tau_{i,j+1}$ can start its execution after $G_{i,j}$ units of time after $\tau_{i,j}$ completes. $G_{i,j}$ lies between $G_{i,j}^{Min}$ and $G_{i,j}^{Max}$.
- s_i is the number of computing segments of τ_i. Each task τ_i also has $s_i - 1$ suspending stages as $\tau_{i,j}$ and $\tau_{i,j+1}$ are separated by one suspending stage when $1 \leq j < s_i$.
- C_i is the worst-case execution time of τ_i. $C_i = \sum_{j=1}^{s_i} C_{i,j}$.
- G_i is the whole self-suspension time of τ_i. $G_i = \sum_{j=1}^{s_i-1} G_{i,j}$.
- T_i describes the job arrivals of task τ_i. We consider two models (1) periodic model and (2) sporadic model. In the periodic model, the first job of task τ_i can arrive at any time but subsequent jobs of task τ_i arrives T_i time units apart. In the sporadic model, jobs of task τ_i can arrive at any time but two consecutive jobs of task τ_i have arrival times separated by at least T_i time units.

[1]We will use the terms 'segments' and 'stages' interchangeably where the distinction is not of importance.

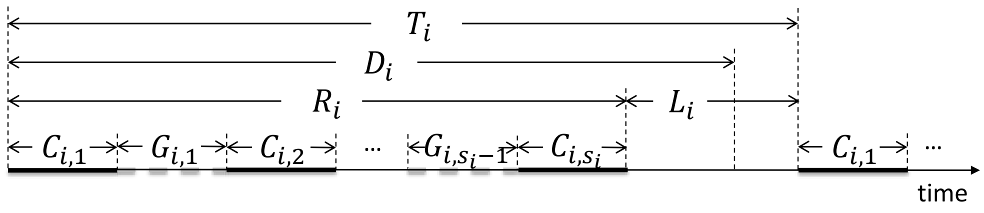


Figure 7.3: A multi-segment self-suspending real-time task model.

- D_i is the relative deadline of each job to its release time. We assume a constrained deadline, i.e., $T_i \geq D_i$.
- R_i is the worst-case response-time of τ_i.
- L_i is the slack from the completion of the last segment of τ_i to the beginning of the next job of τ_i

The tasks in $\mathbf{\Gamma}$ are sorted in non-decreasing order of T_i parameters, that is, $T_1 \leq T_2 \leq \ldots \leq T_n$. We assume that all computing segments are preemptable with insignificant cost. We also assume that the cost of state transitions between computing and suspending stages is negligible on a processor.

Application of multi-segment self-suspending real-time task model: A task leveraging GPU can be modeled using a multi-segment self-suspending real-time task model. For example, a planning algorithm for autonomous driving can benefit from using GPU by calculating numerous potential paths in parallel [12]. The motion planning algorithm receives its inputs such as the current vehicle status, the road map data, and the list of obstacles that are static or moving. The preprocessing for motion planning ($\tau_{plan,1}$) occurs on CPU, and the processed data are transferred to the GPU to generate the best trajectory. While the algorithm runs on the GPU ($G_{plan,1}$), the CPU will let other algorithms run. Once the best trajectory is found, the output is extrapolated ($\tau_{plan,2}$) to be used by the embedded controller. This happens repeatedly every T_{plan} units of time, and this algorithm can be represented as $\tau_{plan} : ((C_{plan,1}, G_{plan,1}, C_{plan,2}), T_{plan})$.

7.2 Fixed Priority Scheduling for Self-Suspending Tasks

In this section we investigate the schedulability of tasksets composed of self-suspending tasks under RMS. We first consider a simple taskset composed of one self-suspending task and one non-suspending tasks[2]. Under the assumption that the self-suspending task is the highest priority task, we provide a response-time test and derive a utilization bound with rate-monotonic policy. We then look at the case of having n self-suspending tasks. To simplify our discussion, we assume a constant gap $G_{i,j} = G_{i,j}^{Min} = G_{i,j}^{Max}$.

7.2.1 One Self-Suspending Task and One Non-Suspending Task

Consider a taskset Γ_{1s1n} with one self-suspending task and one non-suspending task. Let τ_{1ss} denote the self-suspending task, and τ_2 is a non-suspending task. We assume that the self-suspending task has the highest priority. Then, the following properties are satisfied.

Theorem 24 *For Γ_{1s1n}, a critical scheduling instant happens when τ_2 arrives at the same time with one of the segments of τ_{1ss}.*

Proof A critical instant for τ_2 is when the response time of τ_2 is maximized. Since τ_2 is a non-suspending task, a processor will be busy during the execution of τ_2 including preemptions incurred by τ_{1ss}. Let R_2^1 denote the response-time of the first job of τ_2. We assume that the first job of τ_{1ss} arrives at the time origin, and ϕ_2 denotes the release time offset of τ_2 to the time origin. We limit the range of ϕ_2 between 0 to T_1 because τ_{1ss} is periodic and the time origin can be transformed to any of the time instant when a job of τ_{1ss} is released. Then, R_2^1 can be found

[2]'Periodic tasks' are interchangeably used with 'non-suspending tasks' in this dissertation.

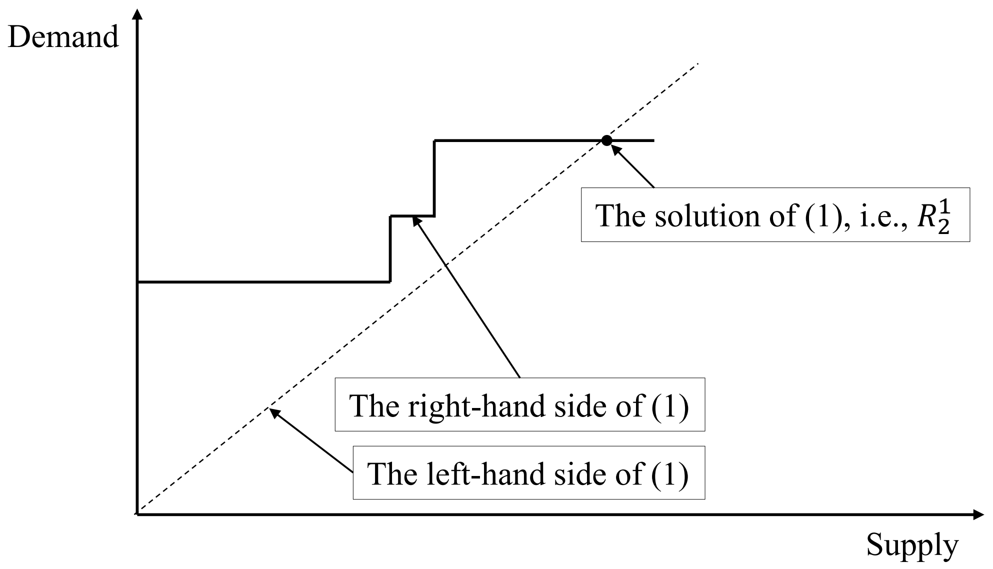

Demand
The solution of (1), i.e., R_2^1
The right-hand side of (1)
The left-hand side of (1)
Supply

Figure 7.4: The illustration of Equation (7.1) to find the response time of τ_2.

by solving the following equation.

$$\begin{aligned}R_2^1 = &-\phi_2 + C_2 + \left\lceil \frac{R_2^1 + \phi_2}{T_1} \right\rceil C_{1,1} - \left\lceil \frac{\phi_2}{T_1} \right\rceil C_{1,1} \\ &+ \sum_{i=2}^{s_1} \left\lceil \frac{R_2^1 + \phi_2 - \sum_{j=1}^{i} (C_{1,j} + G_{1,j})}{T_1} \right\rceil C_{1,i} \\ &- \sum_{i=2}^{s_1} \left\lceil \frac{\phi_2 - \sum_{j=1}^{i} (C_{1,j} + G_{1,j})}{T_1} \right\rceil C_{1,i} + \phi_2 \end{aligned} \tag{7.1}$$

Equation (7.1) calculates the length of busy-period while τ_2 is being executed from time ϕ_2 to $\phi_2 + R_2^1$. We do not start from the time origin because the processor could be idle while τ_{1ss} suspends itself. That's why we subtract the executions of τ_{1ss} happening from the time origin to ϕ_2.

The solution will be the first intersection of a $45°$-line (the left-hand side of Equation (7.1)) and a step function (the right-hand side of Equation (7.1)) as illustrated in Figure 7.4. Although the solution cannot be obtained easily because there are two unknowns with one equation, we can find a useful property of the equation. Since the terms that subtract in Equation (7.1) increase only when ϕ_2 or $\phi_2 - \sum_{j=1}^{i} (C_{1,j} + G_{1,j})$ is an integer multiple of T_1, ϕ_2 can be selected from 0, $C_{1,1} + G_{1,1}$, $\sum_{k=1}^{2} (C_{1,k} + G_{1,k})$, $\ldots$, or $\sum_{k=1}^{s_1-1} (C_{1,k} + G_{1,k})$ when τ_{1ss} has s_1 segments. Those values are aligned with the release time of each segment of τ_{1ss}. Then, let Φ_2 denote a set of possible values of ϕ_2 as described above.

With the given ϕ_2, R_2^1 can be found from the equation. Let $R_2^1(\phi)$ denote the value of the response-time of τ_2 according to ϕ. $\max_{\phi \in \Phi_2} R_2^1(\phi)$ is the worst-case response time of τ_2 because going through all elements from Φ_2 gives all the possible values of the response-time of τ_2. Therefore, for Γ_{1s1n}, a critical scheduling instant happens when τ_2 arrives at the same time with one of the segments of τ_{1ss}.

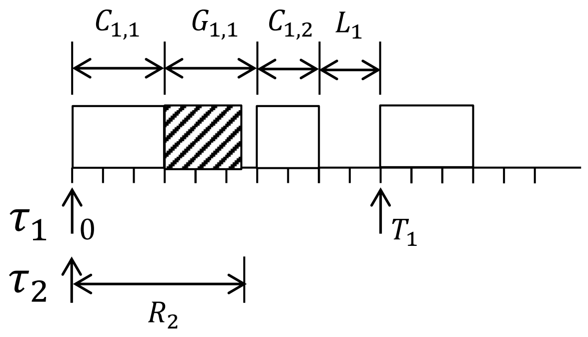
From Theorem 24, we can derive the following corollary.

Corollary 25 *For Γ_{1sns}, the worst-case response time of τ_2 is given as $R_2 = \max_{\phi \in \Phi_2} R_2(\phi)$, where Φ_2 is a set that has each segment release offset of the first job of τ_{1ss} and $R_2(\phi)$ returns the response time of τ_2 under the given release offset ϕ.*

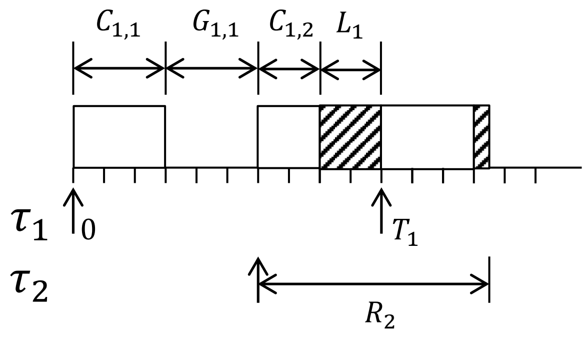
Proof It follows from the proof of Theorem 24.

The following lemma is useful because the worst-case phasing can be obtained by just checking the given task parameters.

Lemma 26 *Consider a taskset having a non-suspending task $\tau_2 : (C_2, T_2)$ and a self-suspending task with two segments $\tau_{1ss} : ((C_{1,1}, G_{1,1}, C_{1,2}), T_1)$. Let L_1 denote the slack from the completion of the second segment of τ_{1ss} to the beginning of the next job of τ_{1ss}. In other words, $T_1 =$*



(a) $\phi_{2,1}$



(b) $\phi_{2,2}$

Figure 7.5: R_2 in the case of $(C_{1,1} \geq C_{1,2}) \wedge (G_{1,1} \geq L_1) \wedge (G_{1,1} \geq C_2 > L_1)$.

$C_{1,1} + G_{1,1} + C_{1,2} + L_1$. *Then, the worst-case response time R_2 of τ_2 can be defined as follows:*

$$R_2 = \begin{cases} & [(C_{1,1} \geq C_{1,2}) \wedge (G_{1,1} < L_1)] \vee \\ R_2(\phi_{2,1}), & [(C_{1,1} \geq C_{1,2}) \wedge (G_{1,1} \geq L_1) \wedge (C_2 \leq L_1)] \vee \\ & [(C_{1,1} < C_{1,2}) \wedge (G_{1,1} < L_1) \wedge (G_{1,1} < C_2 \leq L_1)] \\ & \\ & [(C_{1,1} < C_{1,2}) \wedge (G_{1,1} \geq L_1)] \vee \\ R_2(\phi_{2,2}), & [(C_{1,1} < C_{1,2}) \wedge (G_{1,1} < L_1) \wedge (C_2 \leq G_{1,1})] \vee \\ & [(C_{1,1} \geq C_{1,2}) \wedge (G_{1,1} \geq L_1) \wedge (G_{1,1} \geq C_2 > L_1)] \end{cases}$$

, where $\phi_{2,1}$ is the offset of τ_2 when τ_2 is released with the first segment of τ_{1ss}, and $\phi_{2,2}$ is the offset of τ_2 when τ_2 is released with the second segment of τ_{1ss}.

Proof For this particular case, Equation (7.1) can be rewritten as follows.

$$\begin{aligned} R_2(\phi) = & \left\lceil \tfrac{R_2(\phi)+\phi}{T_1} \right\rceil C_{1,1} + \left\lceil \tfrac{R_2(\phi)+\phi-C_{1,1}-G_{1,1}}{T_1} \right\rceil C_{1,2} \\ & - \left\lceil \tfrac{\phi}{T_1} \right\rceil C_{1,1} - \left\lceil \tfrac{\phi-C_{1,1}-G_{1,1}}{T_1} \right\rceil C_{1,2} + C_2 \end{aligned}$$

Since we assume that τ_{1ss} is released at the time origin in the proof of Theorem 24, $\phi_{2,1}$ is 0. When ϕ is 0, $\left\lceil \frac{\phi}{T_1} \right\rceil C_{1,1}$ and $\left\lceil \frac{\phi-C_{1,1}-G_{1,1}}{T_1} \right\rceil C_{1,2}$ become 0. Similarly, $\phi_{2,2}$ is $C_{1,1} + G_{1,1}$, and

$\left\lceil \frac{\phi - C_{1,1} - G_{1,1}}{T_1} \right\rceil C_{1,2}$ becomes 0. Then, we have the following two equations.

$$\begin{aligned} R_2(\phi_{2,1}) &= \left\lceil \frac{R_2(\phi_{2,1})}{T_1} \right\rceil C_{1,1} + \left\lceil \frac{R_2(\phi_{2,1}) - C_{1,1} - G_{1,1}}{T_1} \right\rceil C_{1,2} + C_2 \\ &\geq \left\lceil \frac{R_2(\phi_{2,1})}{T_1} \right\rceil C_{1,1} + \left(\left\lceil \frac{R_2(\phi_{2,1})}{T_1} \right\rceil - 1 \right) C_{1,2} + C_2 \end{aligned} \tag{7.2}$$

$$\begin{aligned} R_2(\phi_{2,2}) &= \left(\left\lceil \frac{R_2(\phi_{2,2}) + C_{1,1} + G_{1,1}}{T_1} \right\rceil - 1 \right) C_{1,1} + \left\lceil \frac{R_2(\phi_{2,2})}{T_1} \right\rceil C_{1,2} + C_2 \\ &= \left\lceil \frac{R_2(\phi_{2,2}) - C_{1,2} - L_1}{T_1} \right\rceil C_{1,1} + \left\lceil \frac{R_2(\phi_{2,2})}{T_1} \right\rceil C_{1,2} + C_2 \\ &\geq \left(\left\lceil \frac{R_2(\phi_{2,2})}{T_1} \right\rceil - 1 \right) C_{1,1} + \left\lceil \frac{R_2(\phi_{2,2})}{T_1} \right\rceil C_{1,2} + C_2 \end{aligned} \tag{7.3}$$

We identify different conditions that will cause either $R_2(\phi_{2,1}) < R_2(\phi_{2,2})$ or $R_2(\phi_{2,1}) \geq R_2(\phi_{2,2})$. From Equations (7.2) and (7.3), it is obvious that the lengths of $C_{1,1}$ and $C_{1,2}$ are dominant factors because the offsets $\phi_{2,1}$ and $\phi_{2,2}$ decide which segment of τ_{1ss} preempts τ_2 first. It should also be noted that R_2 does not depend on T_2 from the equations. Although $C_{1,1}$ and $C_{1,2}$ are dominant, there are exceptions found below.

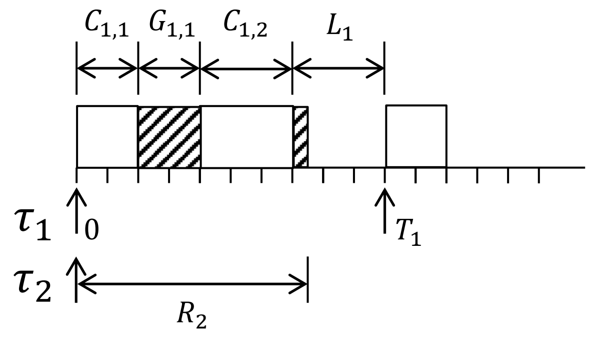
Case 1 ($C_{1,1} \geq C_{1,2}$): When $(G_{1,1} \geq L_1) \wedge (G_{1,1} \geq C_2 > L_1)$, τ_2 will be preempted more when τ_2 is aligned with the second segment of τ_{1ss}. τ_2 will be preempted by both segments of τ_{1ss}, but τ_2 will be preempted only once if it is aligned with the first segment as illustrated in Figure 7.5.

Case 2 ($C_{1,1} < C_{1,2}$): When $(G_{1,1} < L_1) \wedge (G_{1,1} \leq C_2 < L_1)$, τ_2 will be preempted more when τ_2 is aligned with the first segment of τ_{1ss}. τ_2 will be preempted by both segments of τ_{1ss}, but τ_2 will be preempted only once if it is aligned with the second segment as illustrated in Figure 7.6.

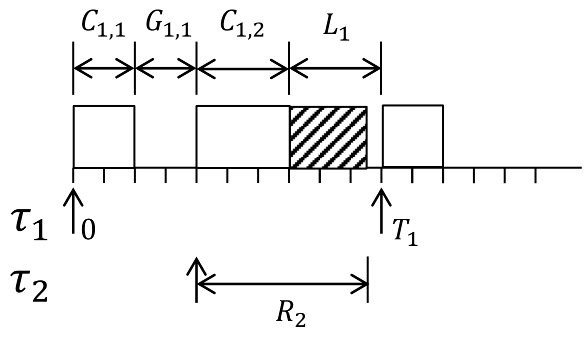
By rearranging the conditions found above, we can obtain the results given in Lemma 26.

7.2.2 One Self-Suspending Task and Many Periodic Tasks

Although we extend the results described in the previous section to understand a case when there are one self-suspending task and many non-suspending tasks, finding a critical scheduling instant



(a) $\phi_{2,1}$



(b) $\phi_{2,2}$

Figure 7.6: R_2 in the case of $(C_{1,1} < C_{1,2}) \wedge (G_{1,1} < L_1) \wedge (G_{1,1} < C_2 \leq L_1)$.

is not trivial. Suppose a taskset Γ that is composed of three tasks: $\tau_1 : ((1, 2\epsilon, 2), 5)$, $\tau_2 : (\epsilon, 5+\epsilon)$, and $\tau_3 : (3\epsilon, 5 + 2\epsilon)$. From Lemma 26, the worst-case response time of τ_2 occurs when τ_2 is released with the second segment of τ_1. However, the worst-case response time of τ_3 does not happen when τ_3 is aligned with the second segment of τ_1. Instead, the worst-case phasing occurs when τ_3 is aligned with the first segment of τ_1 as depicted in Figure 7.7. Therefore, we can claim the following proposition.

Proposition 27 *Consider a taskset Γ_{1s} that has one self-suspending task and $n-1$ non-suspending tasks. Let τ_{1ss} denote the self-suspending task, and τ_i a non-suspending task when $1 < i \leq n$. We assume that the self-suspending task has the highest priority. If $i < j$, τ_i has a higher priority than τ_j. We let ϕ_i^* denote the phasing of τ_i and τ_{1ss} that causes the worst-case response time of τ_i. Then, ϕ_i^* may not be the same as ϕ_j^* when $i < j$.*

We assume that the first job of τ_{1ss} arrives at the time origin. Let Φ_{1ss} denote a set of arrival times, where each arrival time is a time instant when a segment of the first job of τ_{1ss} is released. In other words, $\Phi_{1ss} = \{0, C_{1,1} + G_{1,1}, \ldots, \sum_{j=1}^{s_1-1} C_{1,j} + G_{1,j}\}$. We also define a function $R_i(\vec{\phi_i})$ that returns the response time of τ_i, where $\vec{\phi_i}$ is an $i - 1$ dimensional vector. $\vec{\phi_i}$ consists of τ_i's offset (ϕ_i) to τ_{1ss} and the offsets ($\phi_2, \phi_3, \ldots, \phi_{i-1}$) of the tasks that have higher priorities than τ_i. Each element of $\vec{\phi_i}$ is one of the elements in Φ_{1ss}. When $i > 2$, the actual value of $R_i(\vec{\phi_i})$ can be

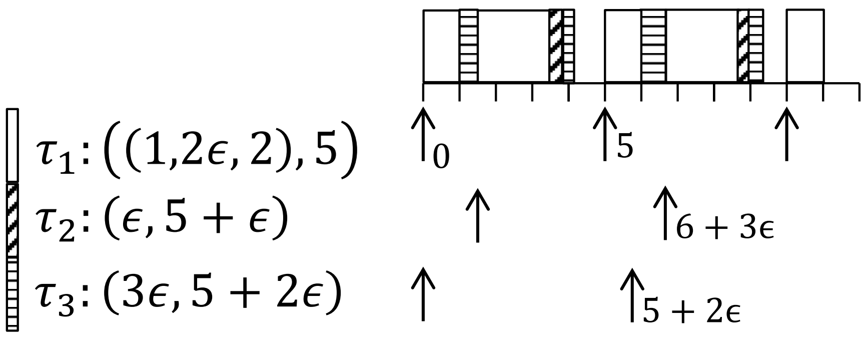


Figure 7.7: An exemplary taskset, where the worst case phasing between τ_2 and τ_1 is different from the one between τ_3 and τ_1.

obtained by solving the following equation that is extended from Equation (7.1).

$$\begin{aligned} R_i(\vec{\phi_i}) = C_i &+ \left\lceil \frac{R_i(\vec{\phi_i}) + \phi_i}{T_1} \right\rceil C_{1,1} - \left\lceil \frac{\phi_i}{T_1} \right\rceil C_{1,1} \\ &+ \sum_{j=2}^{s_1} \left\lceil \frac{R_i(\vec{\phi_i}) + \phi_i - \sum_{k=1}^{j} (C_{1,k} + G_{1,k})}{T_1} \right\rceil C_{1,j} \\ &- \sum_{j=2}^{s_1} \left\lceil \frac{\phi_i - \sum_{k=1}^{j} (C_{1,k} + G_{1,k})}{T_1} \right\rceil C_{1,j} \\ &+ \sum_{j=2}^{i-1} \left\lceil \frac{R_i(\vec{\phi_i}) + \phi_i}{T_j} \right\rceil C_j - \sum_{j=2}^{i-1} \left\lceil \frac{\max(\phi_i, \phi_j)}{T_j} \right\rceil C_j \end{aligned} \tag{7.4}$$

Equation (7.4) is similar to Equation (7.1) except that it considers more non-suspending tasks. The last term of the right-hand side of Equation (7.4) comes from the fact that the tasks that have higher priority than τ_i actually can have different release offsets. The solution of Equation (7.4) can be obtained with Algorithm 13. By going through all possible combinations of $\vec{\phi_i}$, we can find the worst-case response time R_i of τ_i. If $R_i \le D_i$, τ_i is schedulable.

Although we can find the schedulability of Γ_{1s}, the exponential complexity of the given algorithm is not desirable. Lemma 26 gives a useful intuition in this case, where a critical scheduling instant for a taskset can be identified by looking at task parameters. If the critical instant is when all the tasks arrive at the same time, the traditional fixed priority scheduling properties can be applied. In other words, the corollary can help us with easily classifying a taskset with a

Algorithm 13 Response-Time$(\Gamma, i, \vec{\phi_i})$

```
Require: $\Gamma_{1s}$: a taskset including a self-suspending task and $n-1$ non-suspending tasks, $i$: a
    task index, $\vec{\phi_i} = (\phi_2, \phi_3, \ldots, \phi_i)$: an offset vector
Ensure: the response time of $\tau_i$ under $\vec{\phi_i}$
1: ▷ Calculate the initial condition for $\tau_i$.
2: $W_i^0 = \sum_{j=1}^{s_1} C_{1,j} + \sum_{j=2}^{i} C_j$
3: $l = 0$
4: while $W_i^{l+1} \neq W_i^l$ do
5:     ▷ From Equation (7.4).
6:     $W_i^{l+1} = \sum_{j=2}^{s_1} \left\lceil \frac{W_i^l + \phi_i - \sum_{k=1}^{j}(C_{1,k}+G_{1,k})}{T_1} \right\rceil C_{1,j} + \left\lceil \frac{W_i^l+\phi_i}{T_1} \right\rceil C_{1,1} + \sum_{j=2}^{i-1} \left\lceil \frac{W_i^l+\phi_i}{T_j} \right\rceil C_j + C_i -$
       $\left\lceil \frac{\phi_i}{T_1} \right\rceil C_{1,1} - \sum_{j=2}^{s_1} \left\lceil \frac{\phi_i - \sum_{k=1}^{j}(C_{1,k}+G_{1,k})}{T_1} \right\rceil C_{1,j} - \sum_{j=2}^{i-1} \left\lceil \frac{\max(\phi_i,\phi_j)}{T_j} \right\rceil C_j$
7:     $l = l + 1$
8: return $W_i^l$
```

self-suspending task into a category that RMS can be used without any modification.

Lemma 28 *For Γ_{1s1n} that has a self-suspending task with two segments and a non-suspending task, a critical instant occurs when all the tasks are released at the same time when $R_1 = C_{1,1} + G_{1,1} + C_{1,2} < C_2 \leq L_1$.*

Proof From Lemma 26, the critical scheduling instant of τ_{1ss} and τ_2 is same as the conventional critical instant [16] if (1) $[(C_{1,1} \geq C_{1,2}) \wedge (G_{1,1} < L_1)]$ or (2) $[(C_{1,1} < C_{1,2}) \wedge (G_{1,1} < L_1) \wedge (G_{1,1} < C_2 \leq L_1)]$. If $G_{1,1} < C_2 \leq L_1$ is satisfied, therefore, τ_2 experiences the worst-case response time when τ_{1ss} and τ_2 are released at the same time. Since $R_1 = C_{1,1} + G_{1,1} + C_{1,2} > G_{1,1}$, the phasing between τ_{1ss} and τ_2 will still remain the same as before if $R_1 = C_{1,1} + G_{1,1} + C_{1,2} < C_2 \leq L_1$ satisfies. This proves the lemma.

We extend Lemma 28 to be applicable to a taskset including many non-suspending tasks and a self-suspending task with s_1 segments. We let L_1 denote the slack from the completion of the last segment of the first job of τ_{1ss} to the beginning of the next job of τ_{1ss}. In other words,

$T_1 = C_1 + G_1 + L_1$, where $C_1 = \sum_{j=1}^{s_1} C_{1,j}$ and $G_1 = \sum_{j=1}^{s_1-1} G_{1,j}$ as described in Section 7.1. Then, the following property is satisfied.

Theorem 29 *For Γ_{1s} having a self-suspending task and many non-suspending tasks, a critical instant occurs when all the tasks are released at the same time if $R_1 = C_1 + G_1 < C_i \leq L_1$ for $i \in \{i | i \in \mathbb{Z}^+$ and $1 < i \leq n\}$ is satisfied.*

Proof From Equation (7.4), the worst-case response time of each task occurs when all subtracting terms become 0 and all adding terms are positive. At first, all subtracting terms become 0 if all offsets of non-suspending tasks to the self-suspending task are 0. Regarding the adding terms, since we assume the worst-case execution time of each task is greater than the response time of τ_{1ss}, no adding terms become 0. As described in the proof of Theorem 24, rises of the step function (the right-hand side of Equation (7.4)) occur at an integer multiple of T_1 or T_js. Therefore, offsetting the release of non-suspending tasks can make the rises of the step function happen earlier. This can in turn decrease the response-time test. Therefore, the critical instant for tasksets satisfying $R_1 = C_1 + G_1 < C_i \leq L_1$ for $i \in \{i | i \in \mathbb{Z}^+$ and $1 < i \leq n\}$ occurs when all tasks arrive at the same time.

Now, we will provide a least upper bound of utilization for Γ_{1s}^* that satisfies the conditions given in Theorem 29 with rate monotonic policies.

Theorem 30 *For a taskset Γ_{1s}^* with implicit deadlines, Γ_{1s}^* is schedulable if the total utilization of the taskset is less than or equal to*

$$U_{RM-SS}(n,k) = n\left((2+2k)^{\frac{1}{n}} - 1\right) - k \tag{7.5}$$

where n is the number of tasks in Γ_{1s}^*, and k is the ratio of G_1 to T_1 and lies in the range of 0 to $2^{\frac{1}{n-1}} - 1$.

Proof We assume that all non-suspending task periods are less than $2T_1$. We will relax this assumption later. From Theorem 29 and the definition of the conventional critical instant [16], the worst-case response time of a task happens when it is released with its higher priority tasks

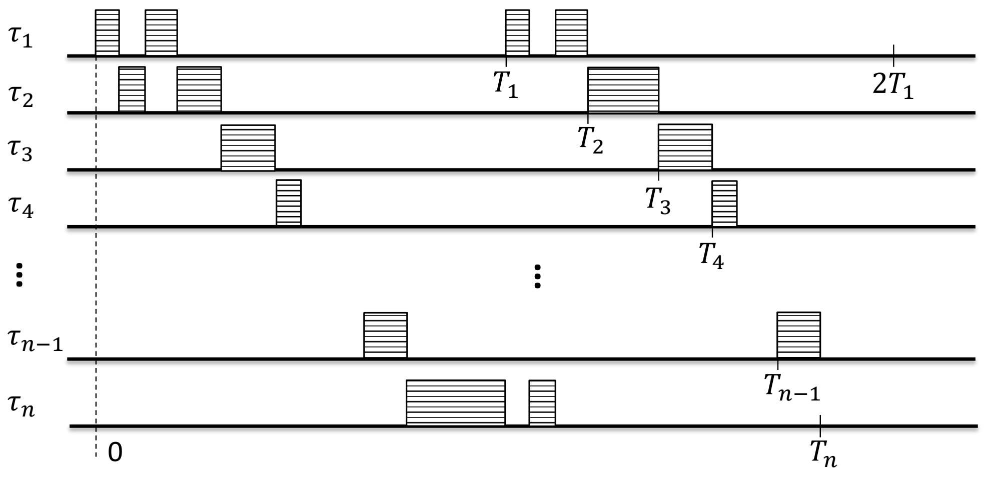


Figure 7.8: The worst-case phasing for a taskset having one self-suspending task and $n-1$ non-suspending tasks.

at the same time. Therefore, we will take a look at a busy interval when all tasks arrive at the same time. Figure 7.8 shows the worst-case taskset[3] for Γ^*_{1s} that satisfies the conditions given in Theorem 29. The worst-case execution times of tasks are given as follows:

$$C_i = \begin{cases} T_2 - T_1 - G_1 & , i = 1 \\ -T_{i+1} - T_i & , 1 < i < n \\ 2T_1 - T_n + 2G_1 & , i = n \end{cases}$$

Then, the total utilization U of Γ^*_{1s} is given like the following:

$$\begin{aligned} U =& \frac{T_2 - T_1 - G_1}{T_1} + \frac{T_3 - T_2}{T_2} + \cdots + \frac{2T_1 - T_n + 2G_1}{T_n} \\ =& \frac{T_2}{T_1} + \frac{T_3}{T_2} + \cdots + \frac{T_n}{T_{n-1}} + \frac{2T_1}{T_n} - n - \frac{G_1}{T_1} + \frac{2G_1}{T_n} \\ =& \frac{T_2}{T_1} + \cdots + \frac{T_n}{T_{n-1}} + \frac{2T_1}{T_n} - n - \frac{G_1(T_n - 2T_1)}{T_1 T_n} \end{aligned}$$

[3]Although we do not go through details here, we can prove that the taskset in Figure 7.8 is actually the worst case. Adding ϵ to C_1 or subtracting ϵ from C_1 increases the total utilization U.

Let k denote $\frac{G_1}{T_1}$, and q_i is the ratio of T_{i+1} to T_i. Then,

$$\begin{aligned} U =& \frac{T_2}{T_1} + \cdots + \frac{T_n}{T_{n-1}} + \frac{2T_1}{T_n} - n - k\left(1 - \frac{2T_1}{T_n}\right) \\ =& \sum_{i=1}^{n-1} q_i + \frac{2(1+k)}{\prod_{i=1}^{n-1} q_i} - n - k \end{aligned} \tag{7.6}$$

From Equation (7.6), we can see that U is a convex function of $n-1$ q_is for $1 \le i \le n-1$. It should have a unique minimum value that is the least upper bound for Γ^*_{1s}. We compute the partial derivative with respect to $n-1$ q_is for $1 \le i \le n-1$. Then, we have the following $n-1$ equations for $1 \le i \le n-1$,

$$\frac{\partial U}{\partial q_i} = 1 - \frac{2(1+k)}{q_i \prod_{j=1}^{n-1} q_j} = 0$$

By solving these equations, we can find the minimum. The solution from the equations above, we get $q_i = (2+2k)^{\frac{1}{n}}$. Substituting this solution to Equation (7.6) gives us the following equation.

$$\begin{aligned} U_{RM-SS}(n,k) =& (n-1)(2+2k)^{\frac{1}{n}} + \frac{2+2k}{(2+2k)^{1-\frac{1}{n}}} - n - k \\ =& n\left((2+2k)^{\frac{1}{n}} - 1\right) - k \end{aligned} \tag{7.7}$$

It should also be noted that k cannot exceed $2^{\frac{1}{n-1}} - 1$ because $k = \frac{G_1}{T_1} < \frac{T_2 - T_1}{T_1} = q_1 - 1$ should be satisfied.

Now we relax the assumption that all non-suspending task periods are less than $2T_1$. Consider a set of tasks that still meet the condition given in Theorem 29, but $T_n \ge lT_1$, where l is an integer greater than 1. If we transform τ_1 so that (1) $T'_1 = lT_1$ and (2) $C'_1 = C_1$ are satisfied, τ'_n can have at most $(l-1)C_n$ more computation time to keep CPU busy. Then, the transformed taskset always has the utilization that is larger than the one before the task transformation[4]. Therefore,

[4]This proof is based on Theorem 5 in [16] and holds good for this case because the taskset still meets the condition given in Theorem 29.

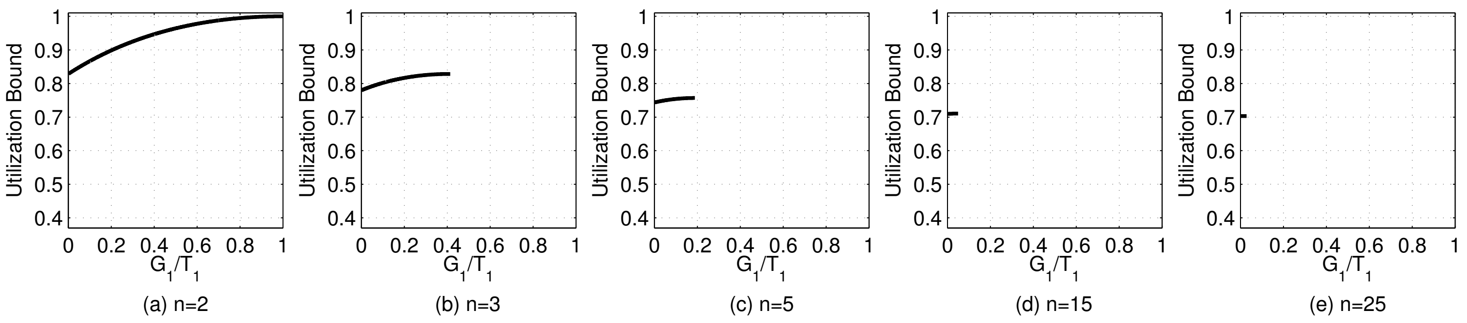


Figure 7.9: Utilization bound for a taskset having one self-suspending task and $n-1$ non-suspending tasks.

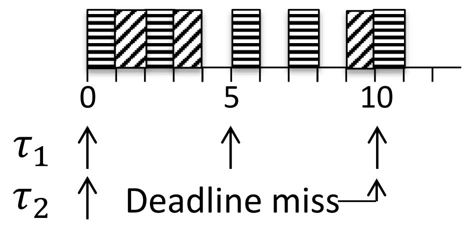
we need to consider the worst case when all non-suspending task periods are less than $2T_1$. This proves the theorem.

Figure 7.9 shows the trend of U_{RM-SS} while n and k vary. One interesting fact is that the utilization bound of Γ^*_{1s} is sometimes larger than the bound for non-suspending tasks. This happens due to the nature of self-suspending behaviors.

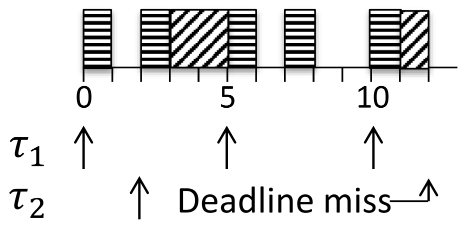
7.2.3 Many Self-Suspending Tasks

We now consider a taskset that has many self-suspending tasks. In the previous section, we have shown that finding the worst-case response times of lower-priority non-suspending tasks is not trivial because all results from all the possible phases need to be compared against each other except for some special cases that we have identified. Therefore, having many self-suspending tasks makes the scheduling problem intractable. In addition, the conventional fixed priority scheduling such as RMS does not account for a different timing requirement per segment. For example, if there is a relatively long suspension time between two segments of a lower priority task and the completion time of the second segment is close enough to its deadline, the task may not easily meet its deadline.

Consider a taskset that is composed of two self-suspending tasks: τ_1: $((1,1,1),5)$ and τ_2:



(a)



(b)

Figure 7.10: Scheduling τ_1: $((1, 1, 1), 5)$ and τ_2: $((2, 5, 2), 10)$ with rate monotonic scheduling.

$((2, 5, 2), 10)$. The executions of τ_1 and τ_2 with RMS are illustrated in Figure 7.10. The boxes filled with horizontal lines represent τ_1, and the boxes filled with diagonal lines represent τ_2. The release of each job is also depicted below the time axis to show the different phasing behaviors. By extending Proposition 27, we can understand that we need to consider four different phases. The case when $\tau_{1,1}$ and $\tau_{2,1}$ arrive at the same time is depicted in Figure 7.10(a). The case when $\tau_{1,2}$ and $\tau_{2,1}$ arrive at the same time is illustrated in Figure 7.10(b), where $\tau_{1,1}$ and $\tau_{2,2}$ are also released at the same time at time 10. The case when $\tau_{1,2}$ and $\tau_{2,2}$ are released together cannot exist for Figure 7.10. Since τ_1 has the shortest period, it has the highest priority. As shown in Figure 7.10, regardless of different phases, τ_2 always misses its deadline. This happens because the conventional fixed priority scheduling does not consider the suspension time between segments. For example, τ_2 has only 5 units of time to execute for 4 units of time due to 5 units of suspension time.

One possible way of resolving this issue is to assign a segment that requires a fast execution a higher priority. Figure 7.11 illustrates the execution behaviors of τ_1 and τ_2 when $\tau_{2,1}$ has the highest priority, $\tau_{1,1}$ and $\tau_{1,2}$ are assigned the priorities in the middle, and $\tau_{2,2}$ is assigned the lowest priority. As shown in Figure 7.11, τ_2 meets its deadline, and the given taskset is schedulable with the proposed scheduling method.

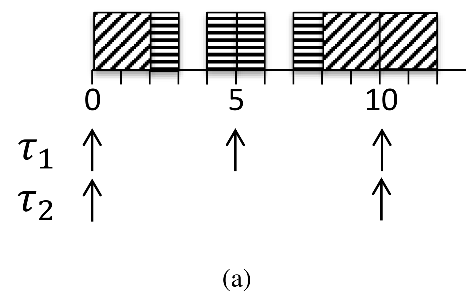


Figure 7.11: Scheduling τ_1: $((1,1,1),5)$ and τ_2: $((2,5,2),10)$ with segment-fixed priority scheduling.

7.3 Segment-Fixed Priority Scheduling

We propose the segment-fixed priority scheduling, where we decompose a self-suspending task into multiple segments and assign them different priorities. In this section, we also relax the assumption of a constant gap that was made in Section 7.2 so that $G_{i,j}^{Min} \leq G_{i,j}^{Max}$ is allowed. In other words, suspension time can vary during run-time, but it is bounded. Although this is a more realistic assumption, variable suspension time easily makes the analysis intractable. We have shown that different phases among tasks need to be considered, so variable suspension time gives myriads of different phase differences. This ends up being hard-to-predict jitters in tasks. This issue can be avoided by leveraging a phase enforcement scheme [13, 14] that guarantees a computing segment of a self-suspending task τ_i arrives at time ϕ_i time units after the arrival of the job of task τ_i and hence a segment does not arrive before its enforced phase time. This reduces jitter. We first provide an optimal method to determine phases and priorities to support segment-fixed priority scheduling.

7.3.1 Fast Deadline and Phase Assignment using Heuristics

Although the optimal priorities and phases can be obtained using the above-mentioned method, the execution time of the algorithm grows exponentially with the number of tasks and segments. To overcome this, we propose four heuristics in this subsection. The high-level ideas are (1)

Algorithm 14 ED(Γ)

```
Require: $\Gamma$: a set of $n$ self-suspending tasks
Ensure: $\Delta$: a set of segment-level relative deadlines
Ensure: $\Phi$: a set of segment-level phase offsets
1: for $i = 1$ to $n$ do
2:     ▷ Calculate the actual amount of processing time for $\tau_i$ with the suspension-time consideration.
3:     $D_i = D_i - G_i$
4:     $\phi_{i,0} = 0$
5:     for $j = 1$ to $s_i - 1$ do
6:         ▷ Assign $\tau_{i,j}$ $D_{i,j}$ so that $\forall j$, $\frac{C_{i,j}}{D_{i,j}}$ is all same.
7:         $D_{i,j} = \frac{C_{i,j} D_i}{C_i}$, $\phi_{i,j} = \phi_{i,j-1} + D_{i,j} + G_{i,j}$
8:         $\Delta \leftarrow D_{i,j}$, $\Phi \leftarrow \phi_{i,j}$
9:     $D_{i,s_i} = D_i + G_i - \phi_{i,s_i-1}$, $\Delta \leftarrow D_{i,s_i}$
10: return $\Delta$ and $\Phi$
```

taking into account only available CPU time for a task after subtracting suspension time from its deadline, (2) distributing its slack to each segment based on computation demands, (3) assigning a segment a deadline with a phase, and (4) scheduling each segment using Deadline-Monotonic Scheduling (DMS). The four heuristics are about how to distribute the slack of a self-suspending task to assign a segment a deadline, hence assigning the segment a priority.

To effectively show how the algorithms work, we introduce few new notations. Since we want to assign intermediate segment-level deadlines to determine the priorities of task segments, we let $D_{i,j}$ denote the segment-level deadline of $\tau_{i,j}$ relative to its release time that is represented as $\phi_{i,j-1}$. Then, we define a segment density as the ratio of the worst-case execution time of the task segment to the task period. We also define U_j^{Tot} as $\sum_{i=1}^{n} \frac{C_{i,j}}{T_i}$, which is the total utilization of the j^{th} segments of all tasks. We use these terms to define the following heuristics.

- **ED** (Equal Density): Assign $\tau_{i,j}$ a segment deadline so that all segment densities for τ_i are same. In other words, there is a certain value $\nu_i = \frac{C_{i,1}}{D_{i,1}} = \frac{C_{i,2}}{D_{i,2}} = \cdots = \frac{C_{i,s_i}}{D_{i,s_i}}$.
- **MTD** (Minimize Total Density): Assign $\tau_{i,j}$ a segment deadline so that the total density for τ_i is minimized. That is to find $D_{i,j}$s that minimizes $\sum_{j=1}^{s_i} \frac{C_{i,j}}{D_{i,j}}$.
- **ES** (Equal Slack): Assign $\tau_{i,j}$ a segment deadline so that $D_{i,1} - C_{i,1} = D_{i,2} - C_{i,2} = \cdots = D_{i,s_i} - C_{i,s_i}$ is satisfied.
- **PS** (Proportional Slack): Assign $\tau_{i,j}$ a segment deadline so that $\forall j \in \{j | 1 \leq j < s_i, j \in \mathbb{Z}^+\}, D_{i,j} - C_{i,j} : D_{i,j+1} - C_{i,j+1} :: U_{i,j} : U_{i,j+1}$ is satisfied.

Outputs of the heuristics are a set of segment deadlines that will determine priorities of task segments under DMS policy. The shorter the relative deadline is, the higher the priority is. The release phases are determined based on the segment deadline. For example, if $\tau_{i,j}$ is assigned a segment deadline $D_{i,j}$, the release phase for $\tau_{i,j+1}$ is $\phi_{i,j-1} + D_{i,j} + G_{i,j}$. One of the heuristic implementations are presented in an algorithmic format in Algorithm 14.

7.4 Evaluation

We have provided the optimal method to determine priorities and phases for computation segments of self-suspending tasks. We have also proposed four heuristics that have lower computation complexity compared to the MILP-based optimal method. In this section, we show the evaluation results of (1) the Rate-Monotonic policy (`RM`), (2) the MILP-based optimal solver (`OPT`), and (3) four heuristics (`ES`, `ED`, `MTD`, and `PS`).

We vary the number of tasks from 2 to 16 while randomly picking a period that is uniformly distributed between 10 and 100. Then, we randomly choose the worst-case execution time of each task segment, where the worst-case execution time is uniformly distributed between 0 and its period. We then scale the tasks so that the total utilization of the taskset does not exceed the maximum value defined in each test scenario. The maximum utilization ranges from 0.1 to

1. In terms of the suspension time, we show two cases: $\frac{G_i}{T_i} = 0.1$ and $\frac{G_i}{T_i} = 0.6$ for $\forall i, 1 \leq i \leq n$. It turns out that the ratio of the suspension time to the task period is important to the schedulability of the given taskset. We fixed the number of task segments to 2, hence, there is one suspension stage between two computing segments. With these configuration parameters, we randomly generate 100 tasksets per test scenario, 8000 tasksets in total. We then apply (1) the Rate-Monotonic policy (task-level fixed priority assignment), (2) the MILP-based optimal method, and (3) four heuristics to see if the randomly generated tasksets are schedulable under those methods.

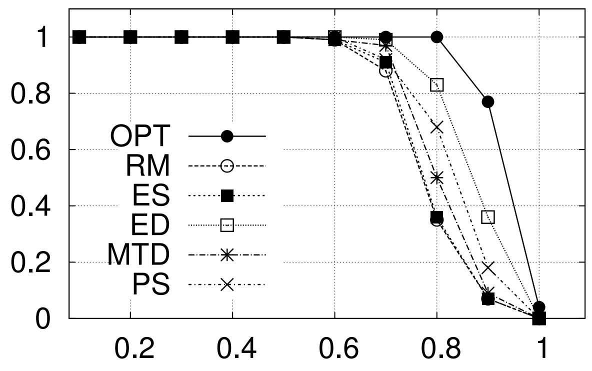
Figure 7.12 shows the schedulability analysis results when the ratio of G_i to T_i for all randomly generated tasks is bounded by 0.1. As shown in the figure, all heuristics perform better than `RM`. `ES` shows almost similar performance to `RM` because `ES` divides the slack by the number of segments, hence not giving more room to execute for a segment that has a longer execution time. The performance of `ED` is the best among heuristics and `RM`. By balancing densities of task segments, `ED` also minimizes the maximum segment density that has large impact on the schedulability. In Figure 7.12(d), `ED` performs about 40 times better than `RM`. `MTD` is designed to reduce the maximum total density of a task. By minimizing the total density, the peak density among segments can be reduced, but it does not perform as well as `ED`. `MTD` would work better if all the segments arrive at the same time. `PS` is intended to deal with assigning more slacks the *competitive* segments by looking at the sum of utilization of the j^{th} segment of all tasks, but it does not perform as well as `ED` or `MTD`. The performance different between `OPT` and `ED` gets larger as the total utilization becomes large. When the total utilization is large, the exact analysis is required to deal with tight deadlines and phases.

Figure 7.13 shows the schedulability analsys results when $\max_{\forall i \in \Gamma} G_i/T_i = 0.6$. The trend shown in this figure is similar to the one shown in Figure 7.12 in terms of the algorithm performance. One interesting aspect is that even `OPT` starts failing to schedule tasksets quite early. This can be interpreted as the impact of the suspension time on the schedulability. As the suspension time becomes larger, it is difficult for tasks to meet their deadlines due to their tightness regardless of the amount of CPU idle time.

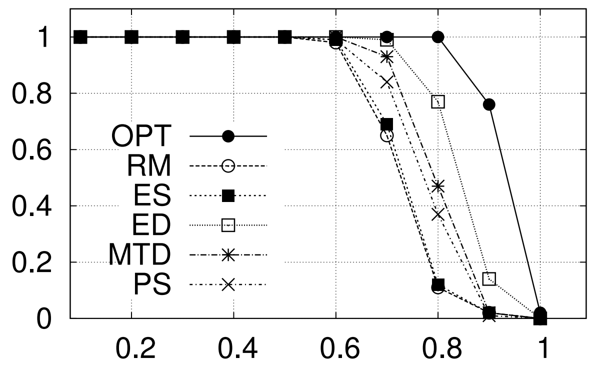
Figure 7.12(d) and 7.13(d) do not have the results of `OPT` because `OPT` did not generate the solution due to its exponential complexity. This aspect motivates the necessity of heuristics, and `ED` performs well as shown in the evaluation results.

7.5 Summary

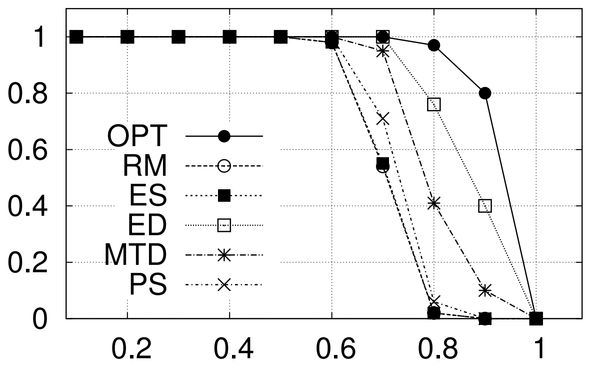
We have provided schedulability analyses and proposed a new method called segment-fixed priority scheduling for self-suspending tasks. We have identified a condition that allows us to leverage the conventional task-fixed priority scheduling such as Rate-Monotonic Scheduling (RMS). However, the condition is narrow, and RMS is shown to not be the optimal scheduler in many cases for self-suspending tasks. This is mainly caused by (1) reduced available CPU time due to self-suspension and (2) unknown suspension time during run-time. These two issues are addressed by utilizing segment-level priority assignment and phase enforcement. To determine the priority and phase per task segment, we have proposed the MILP-based optimal method and four heuristics. The evaluation results show that one of our heuristics performs 40 times better than RMS at best. The heuristics perform well as long as tasksets are not in a very tightly scheduled region that requires the optimal method for the correct parameters. The heuristics could also be complementary to the optimal method because they do not require significant CPU time to get the results. A quick check can be done by using heuristics, and the optimal solver can be used if needed. Future work to be done includes implementing segment-level fixed priority scheduling on a real system [125, 128] so that special-purpose processors can be used in a predictable and analyzable way.



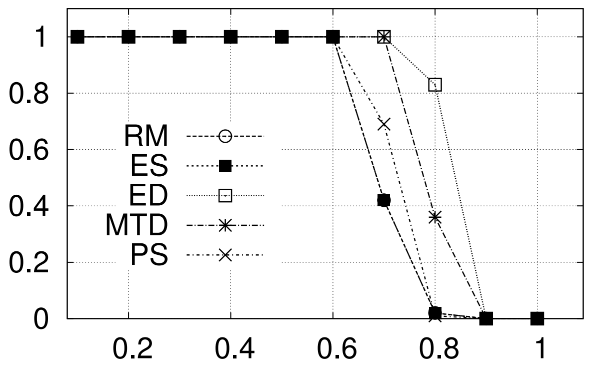
(a) $n = 2$



(b) $n = 4$

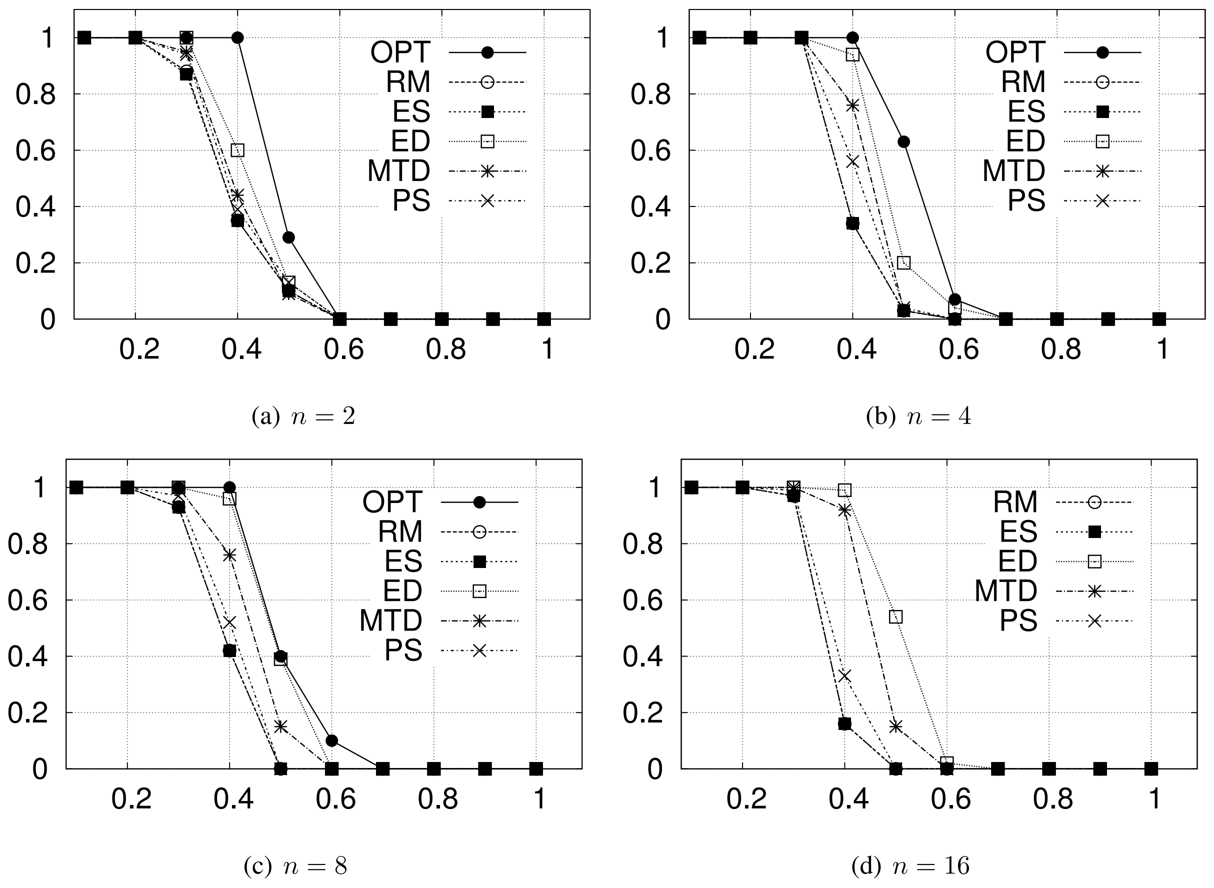


(c) $n = 8$



(d) $n = 16$

Figure 7.12: Schedulability analysis results when $\max_{\forall i \in \Gamma} G_i/T_i = 0.1$. The number of tasks varies from 2 to 16. The x-axis represents the maximum total utilization of the randomly generated tasksets. The y-axis denotes the ratio of the number of schedulable tasksets to the number of generated tasksets. For example, 0.5 means that half of the randomly generated tasksets are schedulable.



(a) $n = 2$

(b) $n = 4$

(c) $n = 8$

(d) $n = 16$

Figure 7.13: Schedulability analysis results when $\max_{\forall i \in \Gamma} G_i/T_i = 0.6$. The same axis definition as Figure 7.12 is used.

Chapter 8

Runtime Support for Fault-tolerance Features

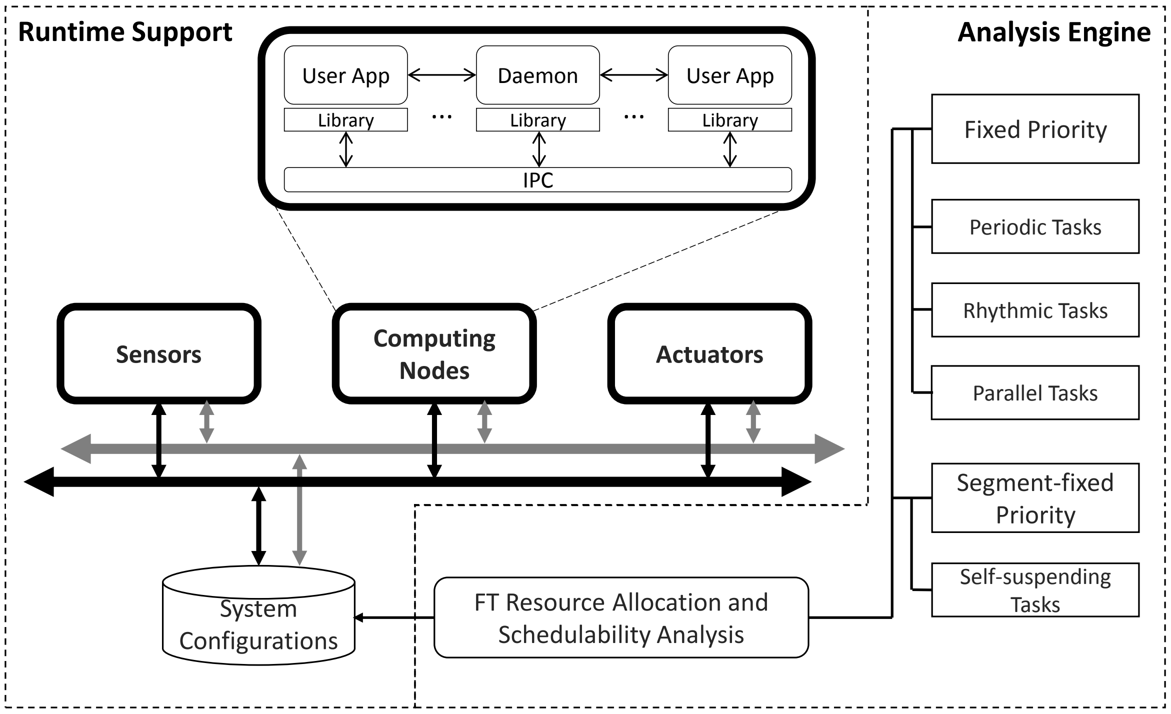


Figure 8.1: Runtime support for fault-tolerance features in the dissertation overview.

This chapter describes our distributed layer called SAFER (System-level Architecture for Failure Evasion in Real-time applications) used to incorporate the proposed algorithms in Chapters 3 through 6 as depicted in Figure 8.1. To realize dependable CPS, SAFER is designed to achieve multiple goals. Most importantly, *no single point of failure* is permitted. In other words, a task/processor failure should not lead to system failure. Secondly, *failure recovery within a guaranteed duration* should be achieved. Since cyber-physical systems are usually tightly connected to the physical world, failure recovery without predictable timing behavior could return unpredictable results in the physical world. Apart from these two goals, *predictive fault discovery and notification, resource isolation, ease of use of abstraction, ease of application development, and sensor/actuator control* are other factors considered.

SAFER incorporates configurable task-level fault-tolerance features to tolerate fail-stop *processor* failures and *task* failures for distributed embedded real-time systems in a *timely* manner. To detect failures, SAFER monitors the health status and state information of each task and broadcasts the information. When a failure is detected using either *time-based failure detection* or *event-based failure detection*, SAFER reconfigures the system to retain the functionality of the whole system using task-level fault-tolerance techniques. More specifically, SAFER provides the following features: (a) Each task can have zero, one or more backup(s), (b) Each backup can be either a hot standby or a cold standby, (c) Failure detection and recovery latencies can be guaranteed, (d) A primary and each of its backup(s) are always allocated to run on independent processor boards to avoid common failure modes, (e) State transfer is managed for seamless recovery from failures.

To integrate SAFER with our fault-tolerant task allocation schemes [100, 129] and verify that the system works well with SAFER, we have implemented new features in a model-based development tool, SysWeaver developed at CMU [15]. Our analysis engine based on the formal timing analysis given in Section 8.1.5 has been added to SysWeaver, and we have added a simulation capability of SAFER features under the presence of failures. Specifically, by injecting failures, we can simulate the timing behavior of the system and verify its operation with different models and system parameters.

SAFER has been implemented as a proof of concept on Ubuntu 10.04 LTS and deployed on Boss, an award-winning self-driving car developed at CMU [8]. We provide a case study showing the quality of mobile robotics algorithms running on the vehicle. The results are measured using the autonomous driving simulation scenarios used during the 2007 DARPA Urban Challenge. We also provide extensive measurement results in TTR and the overhead of SAFER.

The major contributions of this chapter are as follows:

1. The design, implementation and evaluation of a real-time fault-tolerant distributed architecture, SAFER, to provide task-level fault-tolerance techniques.
2. The analyses of the worst-case timing behaviors of SAFER features.
3. Modeling of a system equipped with SAFER to analyze timing characteristics through a model-based design tool, SysWeaver.
4. A case study showing the quality of mobile robotics algorithms of Boss in the presence of failures after the SAFER layer is integrated.

The rest of this chapter is organized as follows. Section 8.1 describes the architecture of SAFER and its implementation, and it also provides detailed analysis of the SAFER layer on failure detection and recovery time. Section 8.2 describes a modeling technique on SysWeaver to find proper system parameters for SAFER. The evaluation results and the case study on Boss will follow in Section 8.3 and 8.4, respectively. We conclude our chapter in Section 8.7.

8.1 The Architecture of SAFER

The overall architecture of the SAFER layer is illustrated in Figure 8.2. The SAFER layer is composed of SAFER daemons, one running on each processor, and a library supporting a task execution environment. The library enables any task launched on the SAFER layer to be periodically executed, with configurable parameters. The daemons have a master-slave architecture, and the master SAFER daemon controls the slave SAFER daemons responsible for managing tasks on each node and monitoring its health status. The configurable parameters for each task are given to the library when the task is launched by a SAFER daemon. For the underlying communication layer, an inter-process communication primitive can be used.

SAFER uses multiple techniques for recovering from processor or task failures. For a task failure without a processor failure, SAFER tries to re-spawn the failed task several times. If the task fails to run, the re-spawning process can be restarted with a different software state or with new inputs because the failure might have been caused by the current software state or/and the current inputs. This re-spawning process can be configured for different applications. The SAFER layer also supports task-level replication techniques, where *selective* tasks on failed processors are recovered on other live processors. Replicas must therefore be placed on independent nodes, a constraint that is referred to as a *placement constraint* [100]. The major benefit of using selective task-level recovery is its flexibility. Since we can selectively recover tasks, we can increase the reliability of highly critical tasks by adding more hot/cold standbys for those tasks. We can also efficiently manage the available computing resources by not replicating less-critical tasks, thus enabling an affordable solution.

SAFER provides process group management which is applicable to the SAFER daemons and application tasks. SAFER has two different types of groups: one that is formed by the SAFER daemons and the other formed by each application. The SAFER daemons have a master-slave relationship and manage tasks on each machine. Hence, when the master SAFER daemon fails, one of the slave SAFER daemons will be promoted to become the master SAFER daemon. Also, any application task that has at least one hot standby forms a group that includes the primary and its hot standbys. When the primary fails, one of the hot standbys will become the primary. Therefore, one generic group management method can be used for the SAFER daemons and application tasks in the system.

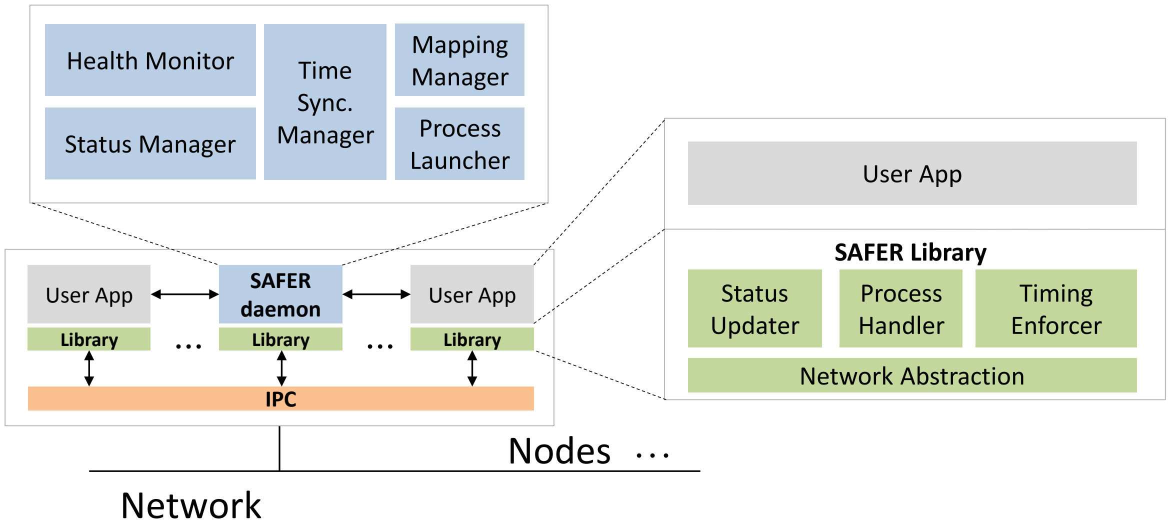


Figure 8.2: The overall architecture of SAFER.

8.1.1 Group Membership Protocol

The process group management is done using a group membership protocol [130, 131]. Our group membership protocol is implemented as a separate thread in the SAFER library used by both the SAFER daemons and application tasks. The membership protocol of SAFER is specifically designed to provide predictable timing behavior and deterministic recovery times. In this membership protocol, all members send heartbeat messages to the master, where the master of a group can be either the master SAFER daemon or the primary of an application group. The master broadcasts its own health status including the list of group members to the group members. Based on this information, only the master will decide who is in the group. The master can detect the failure of a group member if no heartbeat messages are received from that member. The failure of the master can be detected by the group members due to the absence of status messages, and one of the group members will be promoted to become the master by following a pre-determined sequence of group members. If the failed master rejoins the group, it will broadcast a message to obtain group information and send a message to the current primary. After successful state transfer, the current primary will be demoted, and the rejoined primary takes the role of the primary. If a failed group member rejoins, it will be detected through its heartbeat messages by the current primary, which in turn will add it to the group. These steps are important to provide consistent group management when simultaneous failures occur. Suppose an application group is composed of a primary and two hot standbys. If the primary fails and the heartbeat messages from the first hot standby to the second hot standby are lost, each hot standby may try to become the primary. When they see each other, the pre-determined sequence decides who has precedence. Therefore, we can guarantee that we have only one primary in the group, hence maintaining consistency.

8.1.2 The SAFER Library

As illustrated in Figure 8.2, the SAFER library is a task execution environment composed of a status updater, process handler, timing enforcer and network abstraction. User threads and the SAFER daemons run on top of the library.

Status Updater

The SAFER group membership protocol is implemented as a separate thread in the status updater. The status updater supports task-level replication techniques by managing state information between the primary (the master SAFER daemon) and its backups (the slave SAFER daemons). The role of the status updater changes based on whether a task it monitors is a primary (e.g. the master SAFER daemon) or a backup (e.g. the slave SAFER daemon). The status updater of the primary task periodically sends a heartbeat message, its internal state information and the list of group members to its hot/cold standbys, where the update period is configurable. The status updater at the backup node sends out heartbeat signals to the primary such that the primary can decide who is in the group. The rejoining process is also dealt with by the status updater. Therefore, the status updater enables the members in the group to agree upon the availability of each node.

Process Handler

The process handler of the SAFER library promotes a backup to be the primary when it receives the corresponding request from the master SAFER daemon or the status updater. When a backup is promoted, the new primary starts generating outputs for use and confirms its promotion to the requester. It must be noted that a hot standby is always running and its outputs are filtered by the network abstraction of the library in Figure 8.2 under the control of the process handler.

Timing Enforcer

The timing enforcer enables tasks to have guaranteed and protected access to required processing resources in a timely manner based on Linux/RK [121]. In Linux/RK, a shared resource is reserved and enforced by the following parameters: computation time C every T time-units within a deadline D. We refer to these parameters $\{C, T, D\}$ as explicit parameters of our reservation model. These C units of usage time will be guaranteed to be available for consumption before D units of time after the beginning of every periodic interval.

8.1.3 The SAFER Daemon

As illustrated in Figure 8.2, a SAFER daemon is composed of a health monitor, a status manager, a time synchronization manager, a mapping manager and a process launcher.

Health Monitor

The health monitor of the master SAFER daemon monitors the health status of the other daemons and their processors. Once the daemon detects and notifies the failure of a processor, other SAFER daemons can trigger the recovery procedures using the process launcher and process handler unless it is already recovered by the hot standbys of the tasks running on the failed processor.

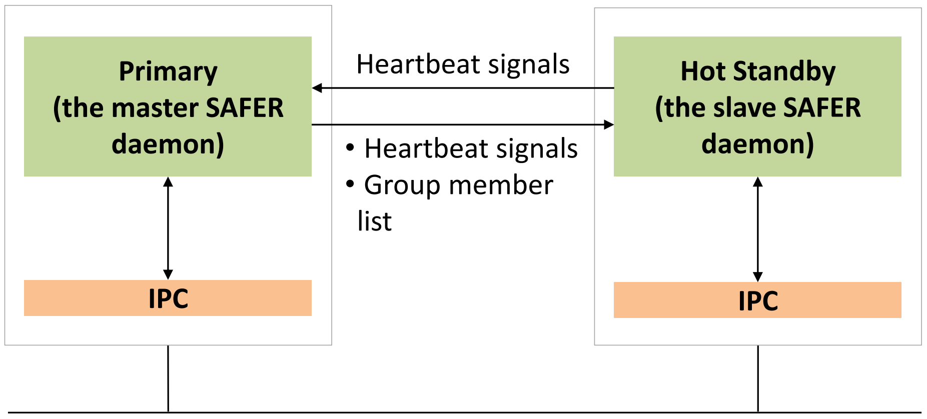


Figure 8.3: The primary-backup architecture.

Status Manager

The status manager tracks the current status of tasks running on its own node and issues failure notification if there is a task failure (say due to a segmentation fault) by capturing the OS signal, which may trigger the failed task re-spawning process on the local node. If there is any cold standby on the processor where the daemon runs, the status manager stores the state information of that cold standby from its primary as depicted in Figure 8.4. It should be noted that the daemons themselves cannot have cold standbys because the daemon should operate as long as the processor is alive.

Time Synchronization Manager

The SAFER layer offers a global time service using a service similar to NTP [132] used for time synchronization over the Internet. The master SAFER daemon behaves as a time server, and each slave becomes a client for this service and listens to messages from the time server. This service is essential to synchronize all the daemons so that failure recovery occurs within the given timing requirement[1] between the primary task and its hot/cold Standbys. This also enables the timing enforcer of the SAFER library to have a smaller penalty in resource scheduling.

[1]Please see Section 8.1.5.

Process Mapping Manager and Launcher

The process mapping manager and launcher are responsible for automatically deploying tasks on the nodes of the SAFER layer based on system configuration parameters. The system configuration includes information about where tasks are allocated and the resource requirements of each task. It also contains the location of the primary and hot/cold standbys if the tasks are selected to have backups. The process mapping manager maintains and updates the system configuration information. Changes to this information can occur due to processor failures, resource demand changes, task completions, and so forth. Based on the up-to-date information from the process mapping manager, the process launcher loads tasks on different processors. The process mapping manager and launcher can be connected to a user-interface application that provides a global view of the system with the current health status of each task on each node. As an example, the information from the process mapping manager and launcher are visualized on TROCS [124], the operator interface of Boss [8].

8.1.4 Failure Detection and Recovery

Heartbeat signals from the status updater of each task will be used for detecting task/processor failures. The status updater of the SAFER library at a backup node will decide the failure of the primary if heartbeat messages of its primary are missed a specified number of times. We call this failure detection scheme as *time-based failure detection*. A task failure may be directly detected by the status manager of the SAFER daemon by catching a signal generated by the OS when a task has unexpectedly failed. Then, an appropriate recovery will be initiated. We name this failure detection scheme as *event-driven failure detection*. It should be noted that event-driven detection cannot be used for processor failure detection.

The recovery from a failure is done by using either the task re-spawning process or the task-level replication techniques of hot and cold standbys. The re-spawning process relaunches the failed task with a different software state and/or a random back-off time to apply new inputs,

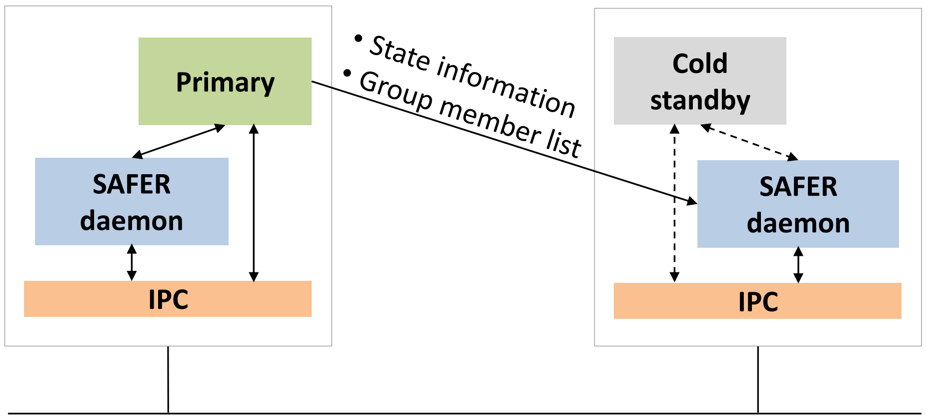


Figure 8.4: The cold standby operation.

which could be different from the inputs causing the failure. This process can be suspended by either the limited number of retries or the task recovery, where the task recovery can be achieved by either the re-spawning process or one of the backups on different processors.

A hot standby receives the same inputs as the primary with no failure, and the user threads of the hot standby run normal operations except that the output from them is filtered by the network abstraction[2]. In the presence of any task failure detected by a hot standby, one of the live backups will promote itself to become the primary based on the predetermined precedence information. Then, it will send a notification to the SAFER daemons.

In the case of a cold standby, without a failure, a cold standby node daemon periodically receives and stores the state information of the primary coming from the status updater of the primary. The disadvantage of using a cold standby is that the recovery latency could be long when there is a failure detected by the master SAFER daemon. Conversely, since it runs only on demand, it saves computing resources in the absence of failures.

[2]We do not generate outputs from hot standbys because we assume the fail-stop failure model. To relax the failure assumption model so that we can check if the outputs from the primary are valid, the network abstraction can be modified to compare the results of the primary with the results of its hot standbys.

8.1.5 Worst-Case Analysis and Admission Control

Assigning a proper standby to a task is of vital importance to meet the fail-over requirements of the given set of tasks. Regardless of the detection methods we provide, we have to consider the worst-case behavior to meet the fail-over requirements. We explicitly consider the time-based failure detection method. Event-based failure detection takes no longer than time-based failure detection and will not be analyzed directly.

For this analysis, we assume a synchronous network [130], and we define a time delay d to represent the maximum network delay of the heartbeat signal packets. We let $T_{heartbeat}$ denote the interval between two consecutive heartbeat signals. Then, the status updater of the SAFER library at a backup node decides the death of the primary unless it hears a heartbeat signal from a processor within $d + kT_{heartbeat}$, where k is an adjustable positive integer based on the underlying communication medium and protocol.

We consider a set of tasks, Γ composed of n tasks, τ_1, τ_2, ..., τ_n. Each task is in one of three sets: *Hard Recovery Task* set, Γ_H, *Soft Recovery Task* set, Γ_S, and *Best-effort Recovery Task* set, Γ_B. A task τ_i is represented by (C_i, T_i, D_i, μ_i) [100]. τ_i will compute for a maximum of C_i time-units every T_i time-units within a relative deadline D_i, and μ_i denotes the ratio of *recovery instance* to its deadline. The recovery instance is defined as the time instance when the failed job is completed by the new primary, which used to be the backup. For example, if $\tau_i \in \Gamma_H$ or $\mu_i = 1$, the failed job should be recovered within D_i. R_i denotes the worst-case response time of τ_i in the absence of a failure, and we assume an implicit deadline of $D_i = T_i$ in this chapter. s_i denotes the worst-case slack-to-recovery time of τ_i, i.e., the time duration between the failure and the recovery instance. If τ_i runs on processor, P_l, the response time of τ_i on P_l is represented as $R_i^{P_l}$. τ_i can have $n^H(\tau_i)$ hot standbys and $n^C(\tau_i)$ cold standbys. Then, $\tau_{i,j}^H$ denotes the j^{th} hot standby of τ_i, and $\tau_{i,k}^C$ is the k^{th} cold standby of τ_i. Either $\tau_{i,0}^H$ or $\tau_{i,0}^C$ can represent τ_i. We also have a set of processors, P, composed of p processors, P_1, P_2, ..., P_p, and $\tau_i \in P_l$ means that τ_i is running on P_l. Let Π represent the allocation information. We use Π_{ij}^H to represent the processor allocated to τ_{ij}^H. Π_{ik}^C is the processor that contains τ_{ik}^C.

On SAFER, the following theorems are satisfied.

Theorem 31 *Given τ_i and its $n^H(\tau_i)$ hot standbys, at least one of the hot standbys will detect and recover*[3] *the failure (only) of the primary if $T_{heartbeat} \leq \frac{(s_i - d)}{k}$, where d is a network delay. The primary is marked as failed if k consecutive heartbeat signals are missing.*

Proof The worst case of recovering a failure happens when a task fails just before the released job completes. Under this circumstance, the slack-to-recovery time is minimized as $s_i = \mu_i T_i - R_i^{\Pi_i}$. Then, the worst-case slack-to-recovery time s_i should be greater than the failure detection time, $d + kT_{heartbeat}$ so that one of the hot standbys becomes the primary and sends out the computed outputs from the failed job. Hence, $T_{heartbeat} \leq \frac{(s_i - d)}{k}$ is satisfied.

Theorem 32 *Given τ_i and its $n^C(\tau_i)$ cold standbys, at least one cold standby will detect and recover the failure of the primary if $T_{daemon} \leq \frac{(s_i - R_i^{\Pi_{i,j}^C} - d - d_S)}{k}$, where T_{daemon} is the period of the SAFER daemon, d_S is the time required for copying and processing the state information from the daemon and j is determined based on the precedence sequence.*

Proof A similar proof to that of Theorem 1 can be applied. First, we have to consider the response time of the cold standby on $\Pi_{i,j}^C$ because the cold standby will start running after the node receives the fault notification. This will further decrease s_i. Also, the period of the daemon should be applied instead of $T_{heartbeat}$ from the SAFER library, since the cold standby is managed by the SAFER daemons. Then, $T_{daemon} \leq \frac{(s_i - R_i^{\Pi_{i,j}^C} - d - d_S)}{k}$ is satisfied.

These two theorems provide significant information to task partitioning algorithms because these properties should be applied to the admission test of hot standby and cold standby along with the relevant schedulability-based admission tests. Based on these two theorems, a schedulability test is given in Algorithm 15.

[3]In order to fully recover within the given value in this theorem, the hot standby should always keep the outputs until the next period starts. Otherwise, the recovery can be off by the period of the hot standby.

Algorithm 15 System-Schedulability-Test-(Γ, P, Π)

```
Require: $\Gamma$: a taskset, $P$: a set of processors and $\Pi$: allocation information between $\Gamma$ and $P$
Ensure: Schedulability of $\Gamma$ on $P$ with $\Pi$
 1: for $i = 1$ to $n$ do
 2:     ▷ Do the response time test for $\tau_i$ and its standbys
 3:     $R_i \leftarrow$ the response time of $\tau_i$ on $\Pi_i$.
 4:     if $R_i \leq D_i$ then
 5:         ▷ The primary is schedulable, so check its standbys
 6:         for $j = 1$ to $n^H(\tau_i)$ do
 7:             $R_j \leftarrow$ the response time of $\tau^H_{i,j}$ on $\Pi^H_{i,j}$.
 8:             if $R_j \leq D_i$ then
 9:                 ▷ This hot standby is schedulable.
10:         ▷ Check its slack-to-recovery time
11:         if $R_i \leq \mu_i D_i - kT_{heartbeat} - d$ then
12:             ▷ The primary and hot standbys are recoverable
13:         for $j = 1$ to $n^C(\tau_i)$ do
14:             $R_j \leftarrow$ the response time of $\tau^C_{i,j}$ on $\Pi^C_{i,j}$.
15:             if $R_j \leq D_i$ then
16:                 ▷ This cold standby is schedulable.
17:                 ▷ Check its slack-to-recovery time
18:                 if $R_i \leq \mu_i D_i - R_j - kT_{heartbeat} - d - d_S$ then
19:                     ▷ Mark this cold standby recoverable
20: if all tasks schedulable and recoverable then
21:     return TRUE
22: else
23:     return FALSE
```

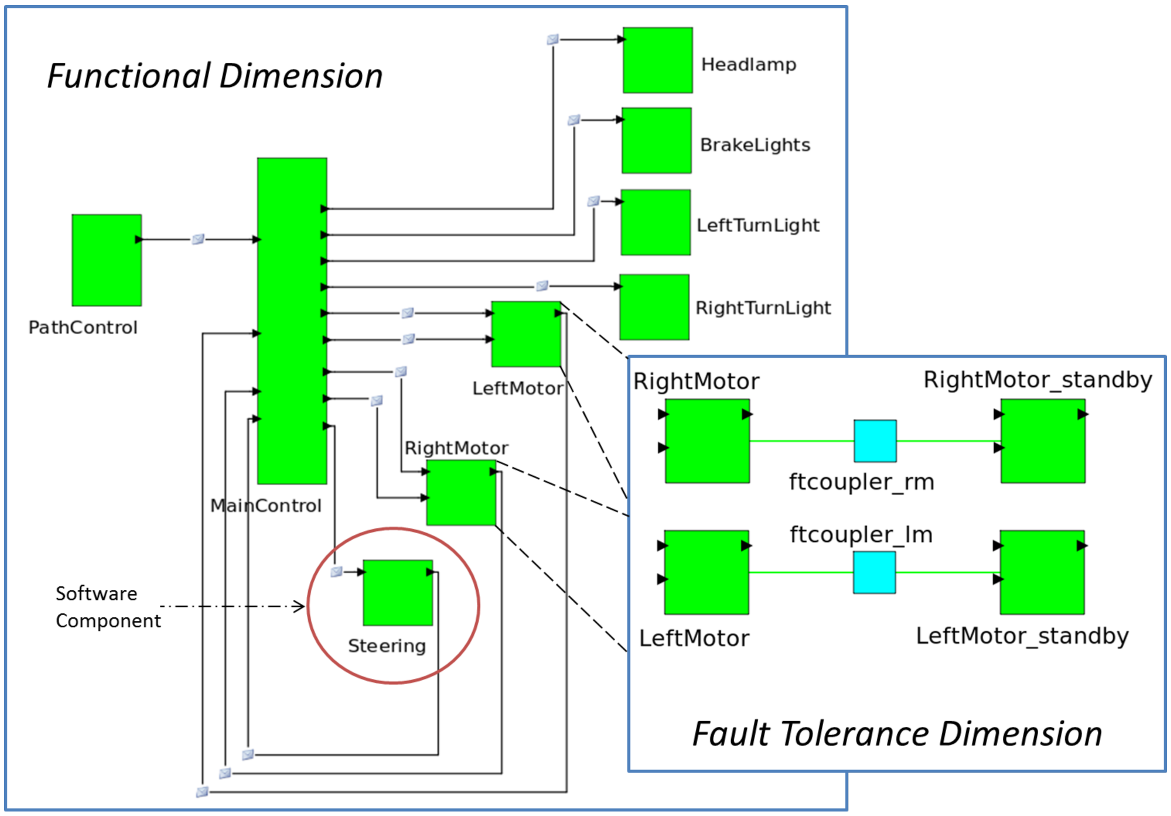


Figure 8.5: Fault-tolerance dimension in SysWeaver.

8.2 SysWeaver Integration

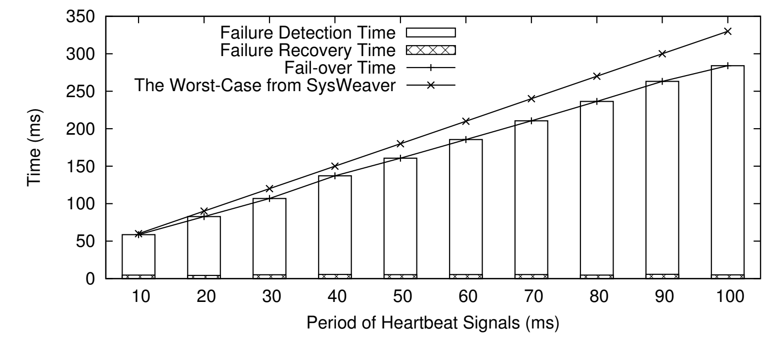
SysWeaver is a model-based design, integration, and analysis framework introduced by de Niz et. al [15] for embedded real-time systems. It explicitly captures the para-functional behaviors such as timeliness and dependability, and their impact on the functional aspects of a system. It uses a view-based representation of various system dimensions such as Functional, Timing, Fault-Tolerance and Deployment. Different domain experts can work on each of these different dimensions, while SysWeaver resolves dependencies among the different views automatically. The framework enables the use of analysis plugins and task-level system simulation capabilities to evaluate and verify system properties along with automatic system code generation. This provides the ability to automatically generate distributed system-oriented glue code which ties together the distributed functional code.

We use SysWeaver to model and capture the SAFER fault-tolerance properties of the system in the Fault-Tolerance Dimension. This view enables us to model the fault-tolerance strategy as well as give us a component-level mapping of the backups running in the system. Figure 8.5 shows an example of the Fault-Tolerance Dimension in SysWeaver. A system model is created within SysWeaver and we then use fault-tolerance task allocation algorithms described in our previous work [100, 129] to provide a fully deployed system.

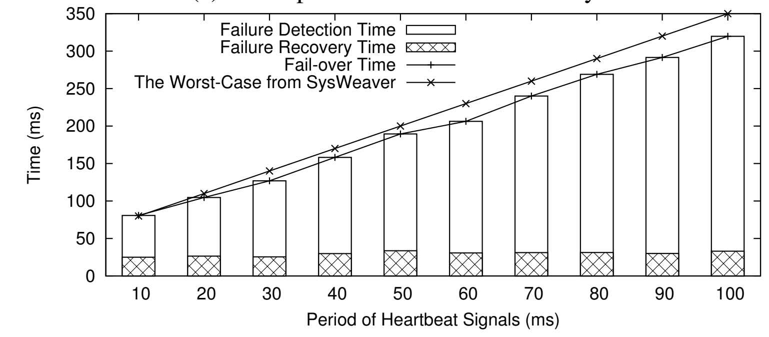
As part of the verification of the SAFER layer, we have implemented the theoretical analysis as an analysis plugin in SysWeaver. Using the system properties represented in the different dimensions, the analysis engine provides an evaluation of the fault-tolerance techniques and analytic results that are then fed back into the system model. To evaluate these analytical results, we then use the distributed system simulation engine within SysWeaver. The simulation engine uses task level properties such as execution time and task priorities along with communication delays to simulate the system response. A fault injection framework has been integrated into the simulation engine that enables us to inject faults and evaluate fault-based mode changes for tasks and captures system response times and deadlines of primary tasks in the presence of faults. The analytical results from SysWeaver for SAFER will be shown in the next section.

8.3 Evaluation

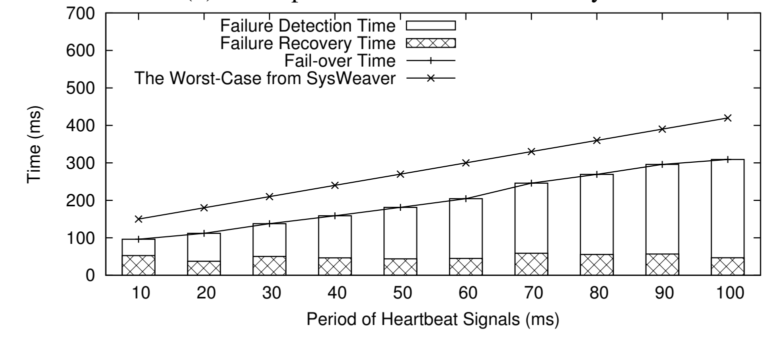
A proof-of-concept implementation of SAFER runs on Linux and x86 hardware. SAFER is deployed and has been running on Boss [8]. To measure the performance of the SAFER layer with the presence of a failure, we have built a cluster composed of three Intel Quad-Core machines. We ran a scenario used to test Boss during the competition in 2007 without the perception system. The artificial intelligence algorithms for behavior and planning along with vehicle control were run on the cluster. By injecting processor failures through a script, we measured the fault detection and fault recovery times for different tasks with different periods. The fail-over time is the summation of failure detection time and recovery time. For a hot standby, the fault detection



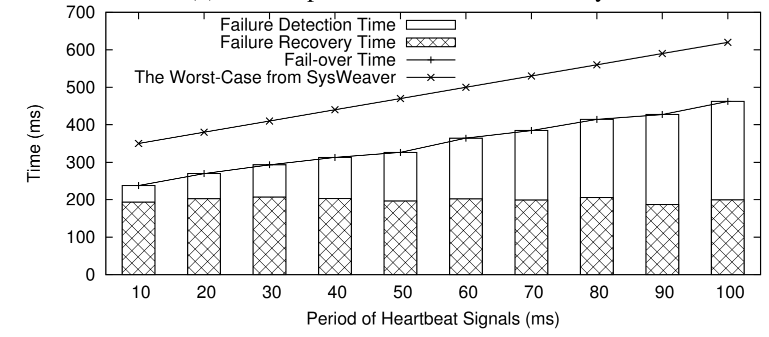
(a) 10ms-period task with hot standby



(b) 10ms-period task with cold standby



(c) 100ms-period task with hot standby



(d) 100ms-period task with cold standby

Figure 8.6: Fail-over time measurements when time-based detection is used.

time is the time duration between when a failure happens and when the backup detects the failure. Its fault recovery time is the time duration between when the backup detects the failure and when the failed task is completely recovered. For a cold standby, the fault detection time is the time duration between when a failure happens and when the master SAFER daemon detects the failure; its fault recovery time is the time duration between when the master SAFER daemon detects the failure and when the failed task is completely recovered, where the recovery procedure includes state information copy, initialization and one re-execution.

For the measurements, we chose two tasks, `BehaviorTask` and `LocalPlannerTask`, which have $10ms$ and $100ms$ as their respective periods. Since Boss has a set of harmonic tasks composed of 10ms-period tasks, 50ms-period tasks and 100ms-period tasks, those two tasks are appropriate to understand the behavior of Boss with SAFER. We measured the fail-over time with a hot standby and a cold standby[4] with two different detection methods, time-based detection and event-based detection. Each point corresponds to the average of 50 iterations.

8.3.1 Time-based Failure Detection

Figure 8.6 shows the fail-over time measurements when the time-based detection is used for detecting a failure, while the period of heartbeat signals from the primary varies from $10ms$ to $100ms$. The hot standby (the master SAFER daemon for cold standby) declares a failure if it misses three continuous heartbeat signals from the primary with a delay d of $20ms$ that includes network and node-side delay. From the data in the figures, it is seen that the failure detection time highly depends on the period of the heartbeat signals of the primary. The failure detection time linearly increases as the period of heartbeat signals increases because the waiting time from the backup linearly increases. The worst-case of the failure detection time calculated in SysWeaver

[4]The fail-over time measurements with the re-spawning process are not described in this dissertation because the fail-over time with cold standby is longer than the time with the re-spawning process due to the network delay. If the process fails to recover the failure after several retries, it could take longer; however, it is beyond the scope of this dissertation.

is also depicted. As shown in the figures, our measurements in average case are under the worst-case.

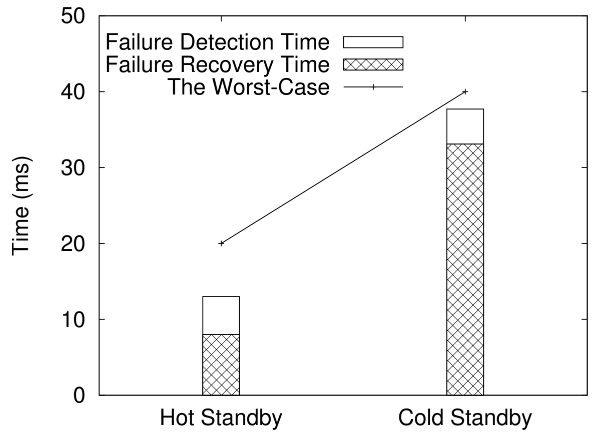
The recovery time of a task is related to its task period because the process handler of its hot (cold) standby should be able to receive the command from the status updater thread (the master SAFER daemon). For the cold standby measurements, the recovery time is much longer because of state recovery and re-execution to recover the failure. The state recovery could depend on each application. For example, the state recovery of the 100ms-period task (`LocalPlannerTask`) in Figure 8.6(d) takes a long time because the task has a large amount of state information and it has a long initialization sequence.

8.3.2 Event-based Failure Detection

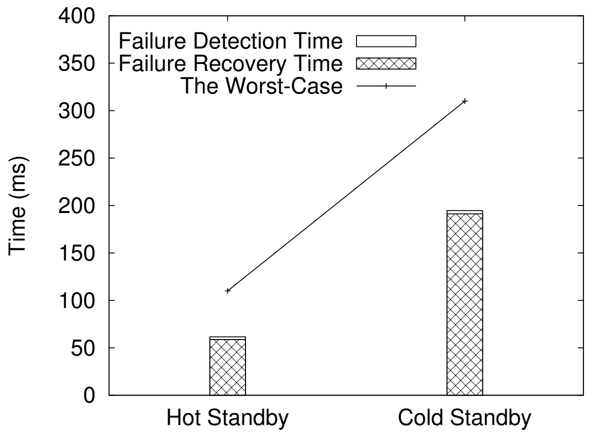
Figure 8.7 shows the measurements when event-driven detection is used when the period of SAFER daemon is $10ms$. Since the local SAFER daemon detects local task failure, the failure detection time is hugely reduced. The local SAFER daemon captures the signal from the task with failure and reports it to the master SAFER daemon. Therefore, the failure detection time is measured as $5ms$ in average. If the period of SAFER daemon increases, the failure detection time will also be increased. The recovery time of a task is related to its period as shown previously. The worst-case from SysWeaver is also depicted as solid lines in Figure 8.7(a) and 8.7(b).

8.3.3 Overheads of SAFER

We have analyzed the worst-case timing behavior using SysWeaver and measured the average-case timing behavior on our simulation cluster. We must also consider the overheads such as additional CPU utilization, network bandwidth and memory consumption that are imposed by the SAFER services. When hot standbys are used, the CPU utilization increases linearly as the number of hot standbys increases. Hot standbys also add to the network load by sending regular heartbeat signals. We limited the size of a heartbeat message to 132 $bytes$ in order to



(a) 10ms-period task



(b) 100ms-period task

Figure 8.7: Fail-over time measurements when event-based detection is used.

minimize the impact of heartbeat messages. Since the failure detection time heavily depends on the period of heartbeat messages, the trade-off between the network load and the failure detection time should be considered. In the case study described in Section 8.4, for example, 25.78 $Kbps$ and 2.578 $Kbps$ of additional network load are added for `BehaviorTask` and `LocalPlannerTask`, respectively.

The network load is also increased by state transfer requirements for cold standbys. Minimizing the amount of data to transfer reduces network congestion and is aided by the support provided by SAFER to easily configure the specific state information to transfer. The period of data transfer is another dimension to consider. If cold standbys are used in the case study, 112.89 $Kbps$ and 41.29 $Kbps$ of the network load are added when 1024 $bytes$ for `BehaviorTask` and 4096 $bytes$ for `LocalPlannerTask` are required for state transfer. Furthermore, the SAFER daemons storing state information from primaries also consume more CPU resources. The computation time of daemons increases by 2 to 5 percent[5] depending on the number of cold standbys that each SAFER daemon manages. Cold standbys also consume memory resources even if they are dormant in system memory.

[5]This value could be different on different configurations.

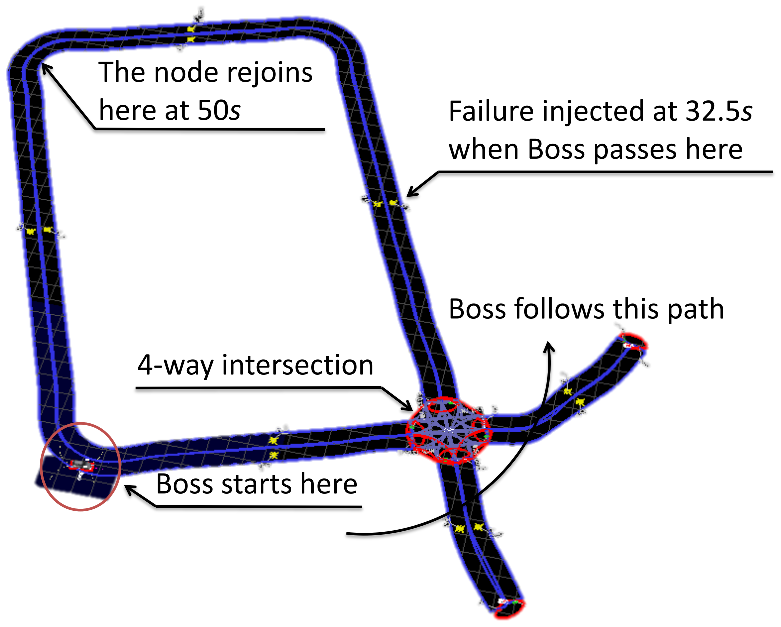

The node rejoins here at 50*s*
Failure injected at 32.5*s* when Boss passes here
Boss follows this path
4-way intersection
Boss starts here

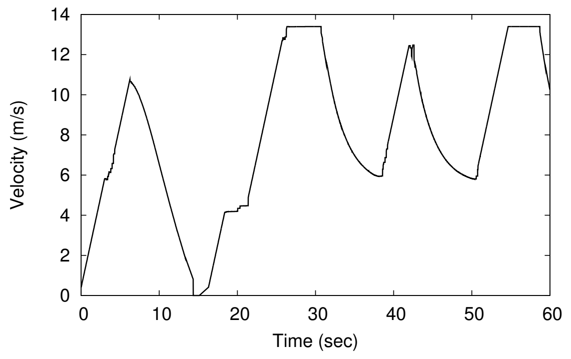
Figure 8.8: The map Boss follows during the simulation.

8.4 Case study on Boss

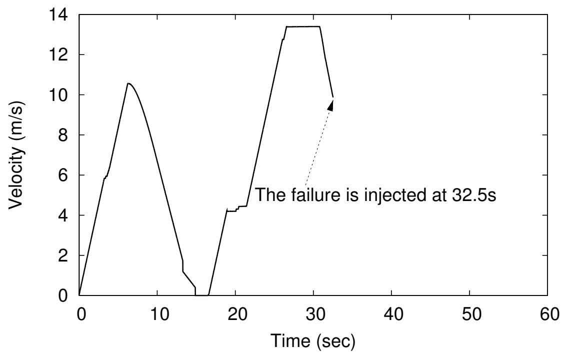
The SAFER layer has been heavily tested on Boss. As a case study, we show results from a scenario used during the 2007 DARPA Urban Challenge. The scenario uses the layout of our test track located at Robot City in Hazelwood, Pittsburgh, PA, where we test our self-driving car at intersections, stop signs, U-turns, two-lane roads, curvy roads and parking lots. In the scenario illustrated in Figure 8.8, Boss will start from the point depicted at the bottom left of the figure. Boss will follow the road, cross an intersection governed by stop signs, increase the velocity while following the straight road and make turns on the curves. The same scenario file is also used on the real vehicle, but only the tasks that interact with the sensors are replaced with the simulated tasks.

The scenario is composed of nine tasks, excluding the SAFER daemons: `BehaviorTask`, `ControllerTask`, `LocalPlannerTask`, `MissionPlannerTask`, `Planner3DTask_1`, `Planner3DTask_2`, `RoadBlockageDetector`, `RobotClient` and `ServerTask`. Then, `MissionPlannerTask` decides where to go. `BehaviorTask` decides the behavior such as turning, lane changing and intersection traversal, and `LocalPlannerTask` sends commands to the vehicle actuators such as accelerator, brake and steering wheel. `ControllerTask` receives those actuator commands and directly interfaces with the vehicle hardware. On the simulation cluster, this task runs in simulation mode. `Planner3DTask_1` and `Planner3DTask_2` are mainly used in unstructured driving conditions such as parking and are therefore not heavily used in this scenario. To test SAFER with this scenario, we replicated `BehaviorTask` and `LocalPlannerTask` using hot standbys. The period of heartbeat signals is fixed at $10ms$, and the period of the SAFER daemon is also $10ms$. We run the scenario while Boss navigates the map. In reference to Figure 8.9, we inject a failure by disconnecting one of the three cluster machines at $t = 32.5s$ after Boss turns left at the intersection. Then, the disconnected machine rejoins the cluster at $t = 50s$ when Boss is at the top left corner.

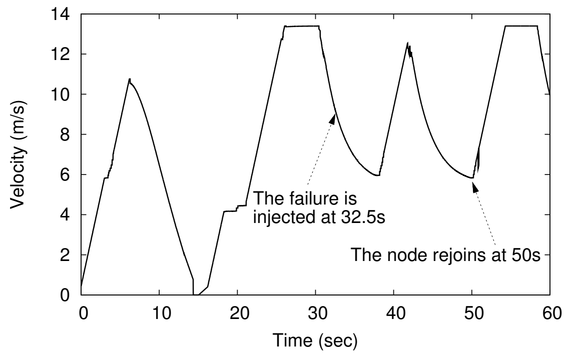
Figure 8.9 shows the velocity profile of Boss measured from `ControllerTask`. The velocity is an important variable to visualize the behavior of a self-driving car. During normal operations, as depicted in Figure 8.9(a), Boss stops at the intersection controlled by the stop sign at $t = 15s$, hence the velocity becomes zero. The velocity graph shows valleys when Boss decreases its speed before and along curves at $t = 39s$ and $t = 50s$. Boss then increases its speed along the straight road. The velocity cannot be sent to `ControllerTask` without `BehaviorTask` and `LocalPlannerTask` running. With SAFER disabled, therefore, Boss completely stops when a node failure is injected as depicted in Figure 8.9(b). Figure 8.9(c) shows the case with SAFER enabled. The node failure is injected at $t = 32.5s$, and the node rejoins at $t = 50s$. As can be seen, there is very little if any behavioral difference at the level of driving. At a microscopic level, the graphs zoomed into the time interval $[32, 52]$ are shown in Figure 8.10. It can be seen that the differences between Figure 8.10(a) and 8.10(b) are subtle as SAFER detects and recovers from the failure within the original deadline. However, the velocity profile shows a small amount of jitter during the process rejoining step as the new primary becomes hot standby again when the old primary rejoins.



(a) Normal velocity trace without a failure injection



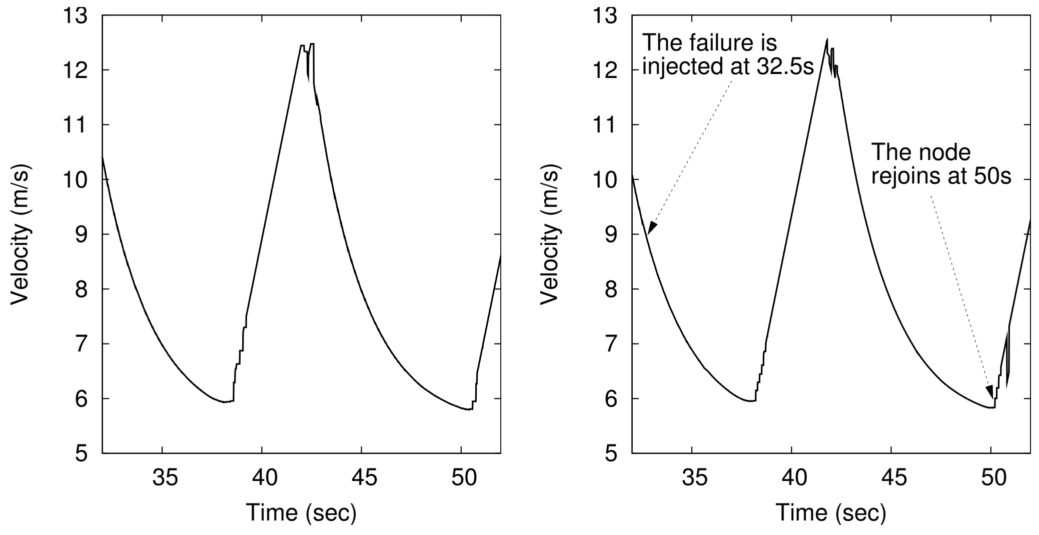
(b) Failure injected at $32.5s$ with SAFER disabled



(c) Failure injected at $32.5s$ with SAFER enabled

Figure 8.9: The velocity trace of Boss measured from `ControllerTask`.

We have leveraged the benefits of SAFER on the vehicle. Boss is equipped with various sensors, but the Velodyne HDL-64E is the most critical sensor on the vehicle due to its wide field of view (360° horizontal by 26.8° vertical). This provides enough data to reconstruct a good three-dimensional view of the world around the vehicle. After the competition, we saw that the processing board running the task for Velodyne frequently crashed. With the SAFER layer in place, this undesirable situation can be avoided. We also sometimes suffered from over-heated processing nodes in the harsh environment. When the densely deployed processing nodes are over-heated, cooling time is required not to damage the hardware. SAFER is also able to handle this unwanted event. Although SAFER is a generic framework, it may be limited by physical constraints. For example, a machine that is used exclusively to interface with a sensor may not be recovered even with SAFER when the machine fails.



(a) No failure injection (b) With SAFER enabled

Figure 8.10: The scaled version of Figure 8.9(a) and 8.9(c).

8.5 Actuator Failure Recovery

SAFER provides an adaptive and affordable way of tolerating processor and/or task failures on distributed real-time embedded systems. However, it has a limitation in tolerating a failure of processor that has a dedicated connection to an actuator,[6] which plays an essential role in CPS applications. SAFER uses the publish-subscribe model for data transfer among all system nodes. Any node connected to a network including an actuator is recoverable using SAFER. Even though distributed control on a publish-subscribe network is getting popular [89, 133], most actuators require separate I/O connections for enabling, controlling and disabling themselves. For instance, many electric motors receive direct pulse-code modulation (PCM) signals from analog input ports to control dynamics. Hence, it is common to have a separate embedded controller for such a motor. Then, the failure of this controller itself cannot be tolerated by SAFER because those signals are not connected to the network. In other words, the controller becomes a single point of failure.

[6]A failure of processor having a direct connection to a sensor cannot be tolerated, neither. Although this section focuses more on an actuator side, a similar approach can be used for a processor dedicatedly connected to a sensor.

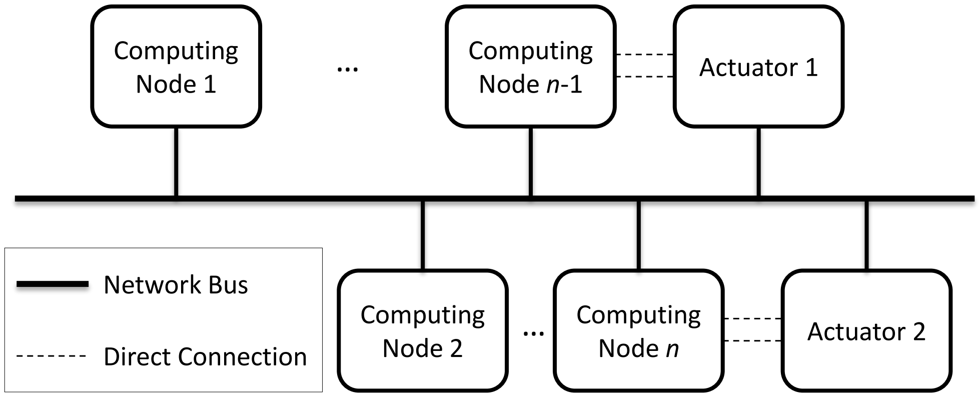


Figure 8.11: An example of actuator connections on SAFER.

This section provides a method that relaxes this limitation by (1) deploying a simple piece of hardware to avoid a dedicated connection between a processor and an actuator, (2) adding a software module that monitors and controls the hardware, and (3) enhancing the failure detection and recovery mechanisms of SAFER to support these changes.

8.5.1 Actuator Connections on SAFER

In this subsection, we will describe how actuators are used with SAFER taking the Boss implementation as an example. SAFER assumes the publish-subscribe model for data delivery among all computing units. Since SAFER is capable of tolerating a task-level failure, the failure of any running task with standby(s) connected to a network can be recovered from. On SAFER, an actuator can also subscribe to its publishers that control the actuator. This architecture is becoming common for distributed control [89], and it can be a desirable way to move forward [133]. For example, all motors that control the steering, acceleration, brake and gear shift of Boss are connected to a CAN bus, and all control messages are transferred via the bus.

A problem, however, arises when an actuator requires dedicated I/O connections. For instance, all electric motor units used on Boss receive three individual direct inputs and one output in addition to the control messages from CAN. The inputs are used for enabling/disabling the motor and controlling its rotation. The output represents the current status of the motor. A

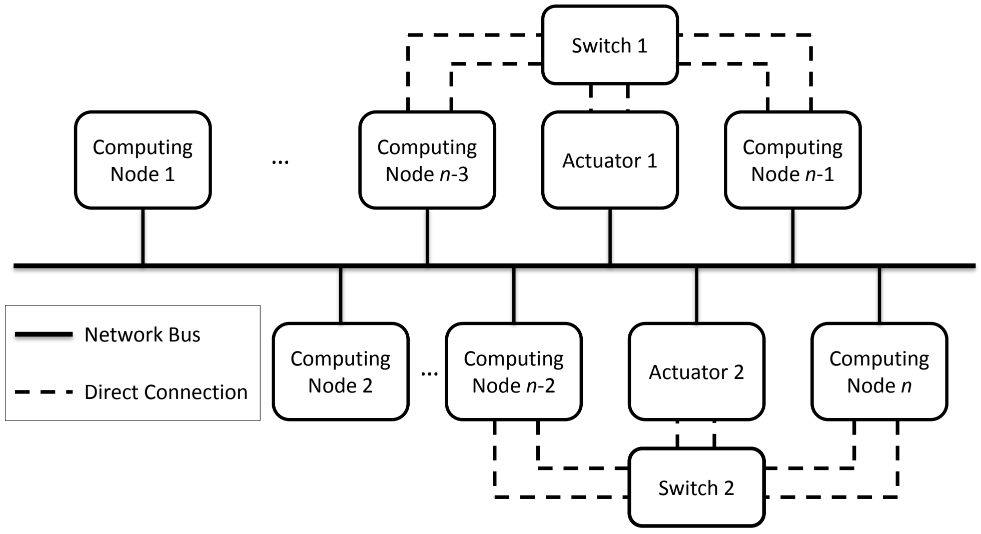


Figure 8.12: An example of actuator connections with switches on SAFER.

high-level connection diagram of Boss is depicted in Figure 8.11. From the figure, Actuator 1 has direct connections to Computing Node $n - 1$, and Actuator 2 also has direct connections to Computing Node n. Then, (say) the failure of Computing Node n may cause a single point of failure because Actuator 2 cannot be used anymore[7]. This cannot be resolved even though SAFER is enabled and Actuator 2 does not have its own fault.

8.5.2 The Proposed Method

We address the problem stated in the previous section by adding a simple switch, designing a new software module for controlling the switch and enhancing the SAFER layer to support these changes.

Switch Design and its Control

The main idea of the proposed solution is that we do not allow any direct connection between an actuator and its controlling unit. In other words, we enable more than one computing unit to be able to control any actuator. An actuator unit usually has several input and output connections.

[7]Boss is designed to be *fail-safe* when this happens.

Any number of computing units can be connected to the input/output ports from the actuator unit[8]. This resembles a publish-subscribe model, and several computing units can monitor the output of the actuator unit using a relatively simple circuit. The challenge arises from the input side of the actuator unit because only one unit should control the actuator at any given time. To satisfy this requirement, we need to have an additional switch which multiplexes one and many possible inputs (outputs from controlling CPUs going to the actuator) and enables only one set of inputs to the actuator. This architecture is illustrated in Figure 8.12 when a computing unit for an actuator has one standby[9].

The reader may note that this switch itself is a single point of failure, but in practice a very simple unit can be built to have less probability of failure than a complex unit. Let p denote the probability of failure for a switch. Let q denote the probability of failure for a computing unit. We assume that (1) actuators do not fail and (2) all tasks have standbys so that they can tolerate two processor failures. We expect that in practice $p << q << 1$. This is a reasonable assumption because a computing unit on a CPS usually has a probability of failure that is much less than 1. Since the switch is supposed to be a very simple circuit, p is even less than q. Then, from the architecture in Figure 8.11, the probability of system failure is $2q$. For the architecture using the switch from Figure 8.12, the probability is $2(q^2 + p)$. Based on the assumption $p << q << 1$, $2(q^2 + p) << 2q$ satisfies, and the switch architecture from Figure 8.12 is significantly better in terms of reliability.

Figure 8.13 shows a more detailed diagram focusing on a primary computing unit (Computing Node 1), its standby (Computing Node 2) and an actuator unit (Motor Controller and Motor). For the switch design, we recommend using a relay due to its simplicity and high reliability reliability. As shown in the figure, the connection is controlled by the primary. When the standby fails, the primary is still in place. When the primary fails, its standby will take over. The switch can be designed to automatically make connections between the actuator and the standby when

[8]It is common that an actuator has an output port for its current status.

[9]The number of standbys affects the switch design.

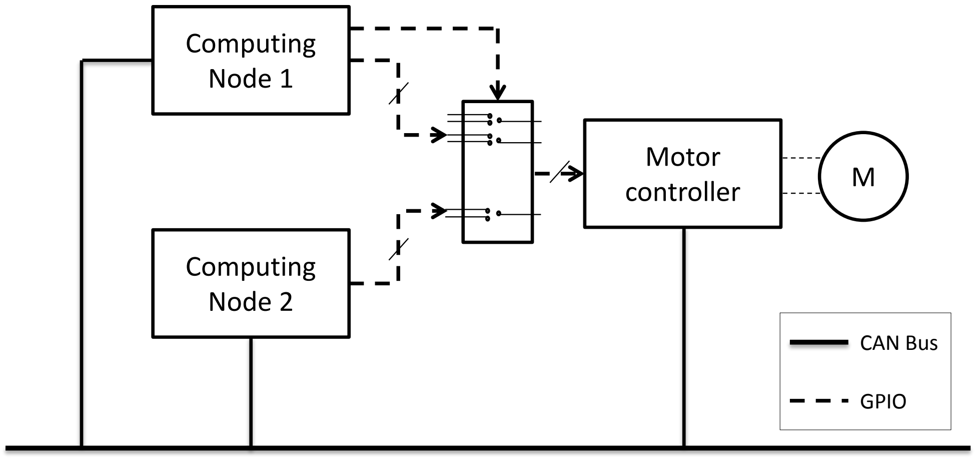


Figure 8.13: A detailed switch diagram with the primary and standby nodes.

the control signal from the primary is absent. A daisy chain configuration can be used to support multiple standbys. Thus, we can tolerate a failure of a node that controls an actuator that requires direct input and output connections.

SAFER Modifications

SAFER needs to be modified to support the above-mentioned switch. Two main modifications are required: (1) the switch added in Figure 8.13 should be supported by the SAFER layer; and (2) the failure detection and recovery scheme should be extended.

SAFER must control connections between an actuator unit and its computing units based on which node is the primary. In other words, when the primary takes control, the relay should maintain the connection between an actuator and the primary. If the standby is promoted to the primary due to the failure of its primary, the relay should alternate the connection. This operation can be done on SAFER by modifying appropriate software modules. For hot standby as the type of standby, the status updater of the SAFER library needs be modified to be able to set a GPIO pin[10]. It is important to promptly control the switch when the primary failure is detected. For

[10]This is for the current implementation. Depending on how SAFER is implemented, the modification may be different.

cold standby as the standby type, this should be managed by the SAFER daemon. Therefore, the status manager of the SAFER daemon should manage the switch status, and the process launcher of the daemon should handle the switch when it activates the cold standby.

The SAFER library was originally implemented to subscribe to other tasks via an inter-process communication primitive using Ethernet. Currently, it can send heartbeat signals using only one communication interface, and this is also a limitation. For example, Boss has a few controllers only on CAN. To make them work together with computing units on an Ethernet, the network abstraction should be modified to support two or more different network types at the same time. Then, a standby on the Ethernet can subscribe to the heartbeat signals of the primary on CAN. In this case, additional care must be given to time-based failure detection because the network characteristics are different between Ethernet and CAN. The failure recovery scheme also requires more design-time analysis. The original SAFER assumes all binaries of the various standbys are the same as the primary's. However, when a different network is used, the binary itself and configuration parameters can be different, hence making design-time analysis more time-consuming.

8.6 Sensor Failure Recovery

We have shown how SAFER can tolerate processor, task, or actuator failures so far. In this section, we extend the SAFER layer to tolerate sensor failures. Unlike processors, it is not trivial to duplicate sensors on CPS due to the lack of enough physical room in many cases. Sensors such as LIDARs, radars, and cameras can be rather expensive to be duplicated. One way of tackling this challenge is to leverage different sensor modalities that are already deployed on CPS. Recent consumer vehicles, for example, are already equipped with forward-looking radars and cameras for various safety features. The front-facing radars are usually used for ACC (Adaptive Cruise Control), and the cameras are often leveraged for lane departure warning, pedestrian detection, sign detection, and so on. As the sensors are already looking at the same direction in the example, they can be interchangeably used for different functionalities in the presence of a radar/camera failure as long as appropriate algorithms are provided. When the ACC radar failed, for instance, a vision algorithm detecting cars can be used for ACC. The quality of driving might be worse as vision outputs are usually noisier; however, having such a capability is essential to ensure safety. To this end, SAFER should be able to (1) monitor sensor health status, (2) detect sensor failures, (3) notify algorithms of any failure, and (4) reconfigure the system correspondingly. We also have a case study using an algorithm picking up a target for ACC running on our automated SRX. The major modifications made to SAFER in this section is quite important as the SAFER layer now can support *application-aware* fault-tolerance features in addition to tolerating sensor failures.

8.6.1 Sensor Failure Detection

In Sections 1.2 and 2.3.1, we assumed a hard sensor failure model. In this section, we further classify the model into two different categories: *fail-stop* and *stuck-at* failures. For the sensors that fail in a fail-stop manner, they just crash and do not output wrong values. For the sensors that fail in a stuck-at manner, their output values are stuck at a certain value that is potentially wrong. This classification is important because the fail-stop sensor failures can be directly dealt with by the SAFER layer. For the stuck-at sensor failures, however, a coordination between SAFER and the perception subsystem might be necessary because SAFER may not be capable of directly reading the sensor output values.

We use both periodic heartbeat signals and proactive pings to detect fail-stop sensor failures. Most sensors such as LIDARs, radars, and cameras that are connected to a network can be configured to send out periodic messages that can be leveraged as heartbeat signals. For example, the radar sensors deployed on the autonomous car at CMU emit periodic track messages over a CAN network regardless of any track presence. SAFER reads such messages from the CAN network and monitors the health status of those radar sensors. SAFER also does a series of

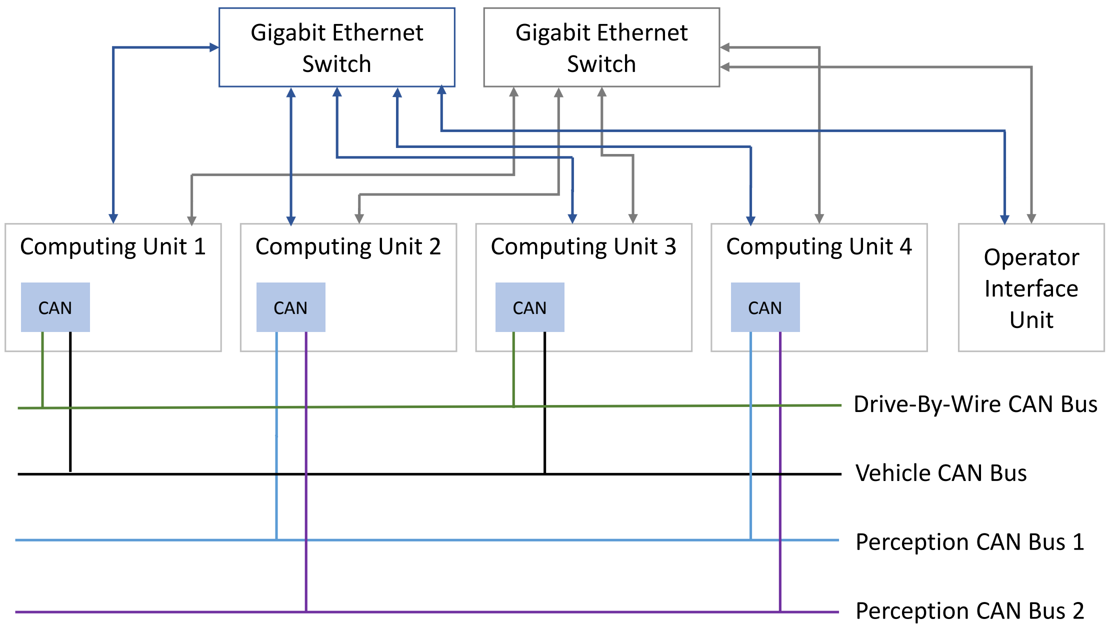


Figure 8.14: An abstracted computing hardware architecture of the automated SRX.

similar actions over the in-vehicle Ethernet for the LIDAR sensors. Figure 8.14 depicts the abstracted computing hardware architecture of the autonomous vehicle and explains why SAFER can read the heartbeat signals from the LIDAR and radar sensors. The LIDAR sensors send their outputs over the Ethernet switches, and the radar sensors emit their outputs over the perception CAN network. Some cameras do not have a capability of sending out periodic messages, so SAFER is also able to periodically ping such cameras. If SAFER does not receive any response within a predefined time window even with multiple retries, a failure is declared. Then, the following measured data become invalid.

To detect stuck-at sensor failures, we exploit a data grabber for each sensor type that converts raw sensor data to an internal format for the perception subsystem of our autonomous vehicle. While the grabbers read the sensory data, they detect stuck-at failures by using the methods proposed in [75, 79][11]. The grabbers then notify the SAFER layer about the detected failure(s). To avoid a single point of failure, each grabber can also have a hot standby supported by SAFER.

[11]The algorithms detecting stuck-at failures are the beyond the scope of this dissertation, so we do not provide the details of the algorithms used in the grabbers.

The SAFER layer sends out this information to the corresponding modules that use the failed sensor(s). Another benefit of using the data grabbers is to reduce the amount of data going through the networks on SAFER. Raw sensory data usually contain large amounts of information, and the grabbers extract the required data only and hence help to reduce the network traffic caused by the sensor data. The standbys of algorithms using the sensor data also benefit from having sensor data grabbers because we assume a publish-subscribe model and the standbys can be easily configured to run.

8.6.2 SAFER Extensions for Sensor Failure Recovery

SAFER leverages different sensor modalities to recover from sensor failures and hence improve the system dependability. We assume that all the sensors are connected to at least one network on the SAFER layer. After a sensor failure is detected, the SAFER layer lets the corresponding algorithms know that they cannot rely on the data from the failed sensor anymore. This is very useful in different aspects. We can avoid the waste of computing resources. When a camera fails in an autonomous vehicle, for example, we do not need to run vision algorithms that are usually compute-intensive. Also, we can control the quality of the perception outputs. Consider a sensor fusion algorithm for autonomous driving that merges radar data with LIDAR point clouds. When the radar sensor fails, we have to rely only on the LIDAR sensors. Without an appropriate adjustment, the perception outputs from the fusion algorithm will be extremely degraded, hence making autonomous driving potentially dangerous. If the perception subsystem cannot provide good enough data for autonomous driving, a human operator of the system should be alerted to the failure to ensure safety.

SAFER needs modifications to the SAFER daemon to support sensor failure recovery. The health monitor and status manager should have entries for the sensors used by the user applications. Compared to a processor/task entry, each sensor entry has extra information: sensor type, coverage, list of applications that use the sensor, and criticality. The sensor type is used for dis-

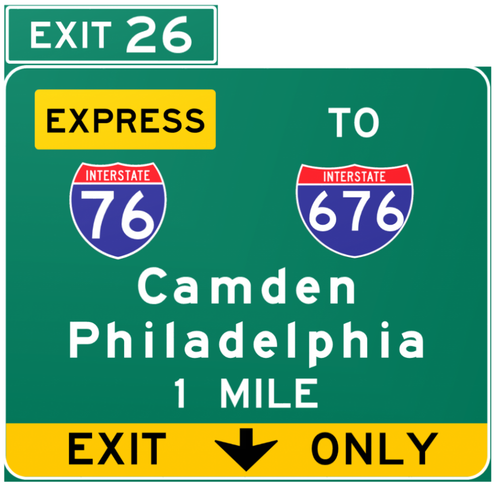


Figure 8.15: An example of interchange advance guide signs.

tinguishing different sensor modalities and includes LIDARs, radars, cameras, and thermostats. The coverage contains information about where the sensor is placed. The list of applications that use the sensor is necessary for SAFER to notify them about the sensor failure or another status change. The criticality is used to represent the importance of the sensor. For example, if there is a sensor that may cause a single point of failure, that should be the most critical sensor and the SAFER layer may even change the mode of the behavior/planning algorithms to react to the failure. If the system with the failed sensor cannot operate reliably, the human operator should take over the system so that the system does not become unsafe. The above-mentioned information can be added during the design time.

When current sensor status changes due to a sensor failure, the mapping manager of the SAFER daemon must reconfigure the system accordingly. If there are user applications that solely rely on the failed sensor, these applications should be turned off not to waste processing/networking resources. Other applications that partially sue the failure sensor should be reconfigured not to use the failed sensor. If there is a mode of operation that can make up the lack of the failed sensor, this mode should be activated even though the mode requires a higher portion of system resources. When this mode is activated, SAFER considers the resources retained from the halted applications that were using the failed sensor. Consider a map-matching algorithm localizing an autonomous vehicle using map data and GPS coordinates. If the GPS receiver fails, SAFER can activate an algorithm that uses camera images to roughly localize the vehicle location using interchange advance guide signs depicted in Figure 8.15. The process launcher actually sends out control signals to each application so that the SAFER library reconfigures individual applications. A case study using this framework will be discussed in the next subsection.

8.6.3 Case Study: Selecting a Target for Adaptive Cruise Control

We apply the framework proposed in the previous section to an algorithm responsible for picking up a target used for ACC. The algorithm is called `ACCTargetSelector` and has already been running on our automated SRX. The algorithm takes LIDAR point clouds, radar point targets, vision objects, the current/intended lane for autonomous driving, and map information to determine a target that is used for ACC. An ACC target is important to autonomous driving because the target provides important context-aware information. For example, our automated vehicle can just follow the ACC target, or we can avoid its misbehavior by moving to another lane.

The overview of `ACCTargetSelector` is depicted in Figure 8.16. As mentioned above, raw data from sensors are converted to an appropriate internal format and transferred to the algorithm. Using the map information, `ACCTargetSelector` identifies a list of candidate targets in the current/intended lane detected by each type. Therefore, three sets of targets are identified. Then, the closest target to the vehicle is selected among the validated targets using domain knowledge such as road rules. The KF (Kalman Filter)-based sensor fusion algorithm takes the chosen target as a measurement. `ACCTargetSelector` periodically selects a target, and the estimated target will be the output of the algorithm.

Since `ACCTargetSelector` equally relies on three different types of sensors when picking up a candidate from three sets of candidate targets, the presence of a sensor failure can

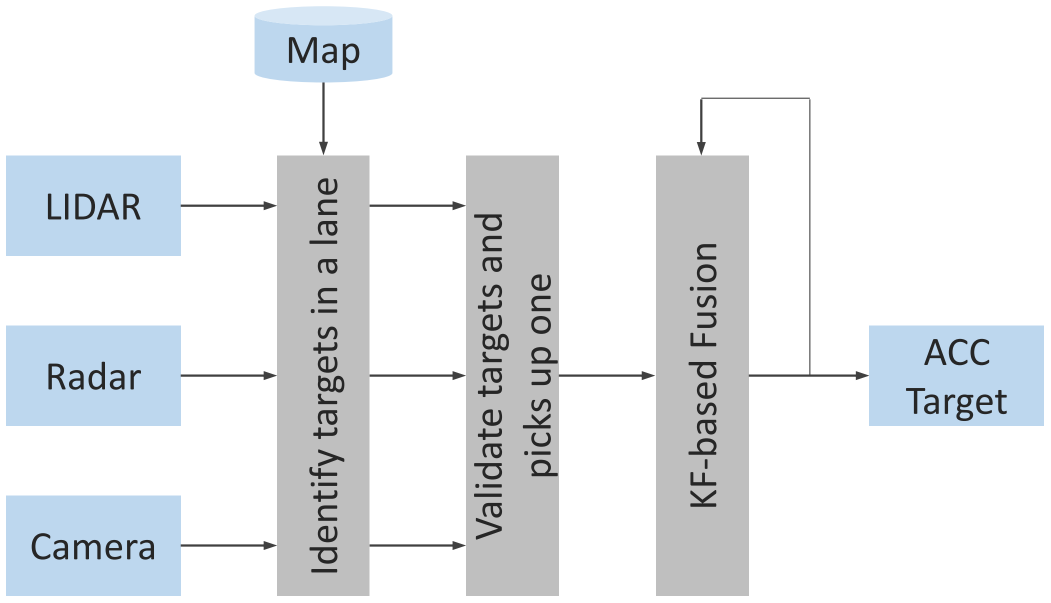


Figure 8.16: The procedure block diagram of `ACCTargetSelector`.

make the algorithm fail to output an ACC target. To resolve this issue, we have modified `ACCTargetSelector` to leverage the SAFER layer. When a sensor failure is detected, SAFER reports the failure to `ACCTargetSelector`. Then, when the closest target is chosen for the KF-based sensor fusion, the validation algorithm runs in a different mode to incorporate the sensor failure.

8.7 Summary

We have proposed a layer called SAFER (System-level Architecture for Failure Evasion in Real-time applications) to incorporate configurable task-level fault-tolerance features using hot standbys and cold standbys in order to tolerate fail-stop processor and task failures for distributed embedded real-time systems. SAFER is implemented on Ubuntu 10.04 LTS and integrated into a self-driving car developed at Carnegie Mellon. The formal analyses of the worst-case timing behavior are also provided, and our analysis engine is integrated with a model-based design tool, SysWeaver. We have presented measurements along with analytical results from the driving simulation scenarios used during the 2007 DARPA Urban Challenge. A case study on Boss has also shown that there is no noticeable behavioral difference even when node failure is injected and the failed node later rejoins. Future work to be done includes supporting graceful degradation based on load and resource changes and providing a generic infrastructure to recover from different types of sensor and actuator failures.

We have also proposed a method to realize a fault-tolerant embedded controller on distributed real-time systems. We have avoided a direct connection between an actuator and a computing unit so that the actuator can be controlled by other computing units in the distributed real-time system. To apply this idea, we have proposed a way to modify SAFER (System-level Architecture for Failure Evasion in Real-time applications). We achieve this goal by (1) adding simple relay circuits controlling the connections between actuators and computing units, (2) adding a software module for maintaining the relay circuits, and (3) enhancing the failure detection and recovery schemes of SAFER. The hardware design has been completed, but the implementation of the SAFER extension is on-going.

Chapter 9

Conclusions and Future Work

In this dissertation, we have studied the problem of realizing dependable CPS (Cyber-Physical Systems). We have designed and implemented a system-level software framework that enables essential system components including processors, networks, and sensors the ability to meet their timeliness and reliability requirements. New computational models proposed in this dissertation catalyze the *analyzability* in different types of CPS tasks. The task models capture periodic, dynamic, parallel, and self-suspending CPS attributes, and the corresponding schedulability analyses provide deterministic timing properties. Even in the presence of resource failures, the framework reliably operates systems using software redundancy. Coupled with a model-based design tool, a distributed layer called SAFER (System-level Architecture for Failure Evasion in Real-time applications) substantiates the framework in practice and enables the *predictability* in system behaviors with and without the resource failures. The framework is also deployed on an autonomous vehicle developed at Carnegie Mellon to show how the proposed techniques make the real-world vehicle dependable. This dissertation contributes to advances in response-time analyses, resource augmentation bounds, and utilization bounds for CPS.

9.1 Contributions

The major contributions of this dissertation fall into one of the following categories:

- Resource allocation for fault-tolerant computing
- Schedulability analyses for cyber-physical systems
- Runtime support for fault-tolerance features

The details of the contributions are presented below.

9.1.1 Resource Allocation for Fault-Tolerant Computing

A task-partitioning strategy for allocating software replications to processors has been developed to improve system reliability. R-BFD (Reliable Best Fit Decreasing) is proposed to allocate hot standbys. R-BATCH (Reliable Bin-packing Algorithm for Tasks with Cold standby and Hot standby) is developed to further reduce the resource over-provisioning required for task reliability using the notion of virtual tasks by consolidating standbys. We have categorized real-time CPS tasks based on their recovery time requirement into hard recovery, soft recovery, and best-effort recovery tasks. To tolerate fail-stop processor or task failures, the task categorization is used to determine the appropriate recovery type: hot standby, cold standby, or re-execution. Evaluation results show that R-BFD can save up to 37% on computing resources, in comparison to prior approaches. When the virtual tasks can meet the recovery time requirements of their primary tasks, R-BATCH can save even 45% more on the computing resources, while maintaining the same level of reliability as prior techniques.

To incorporate CPS task dependencies, we have defined an abstraction called an *application flow*, which enables timing analysis. A processor assignment methodology called R-FLOW (Reliable application-FLOW-aware SW-C partitioning algorithm) is proposed to consider task dependencies and network bandwidth. A key observation is that consolidating tasks that communicate each other can save computing/networking resources as the consolidation reduces the processing and networking overhead. Our evaluation shows that R-FLOW can save up to 60% of computing resources as compared to prior work that does not incorporate the task dependencies. R-FLOW is also adapted to an AUTOSAR-compliant platform to see how the proposed algorithms can be used in the automotive context.

9.1.2 Schedulability Analyses for Cyber-Physical Systems

Conventional real-time theories are in general applicable to CPS; however, they do not incorporate highly dynamic physical attributes and hence do not provide tight schedulability analyses. We have identified three dominant factors affecting the CPS timeliness: dynamically varying periods, parallelism, and self-suspensions.

Tasks with Continually Varying Periods

A periodic real-time task model widely used in the literature is often too conservative for handling a CPS task that depends on physical attributes that dramatically change the task period/workload. We have defined a new task model called *Rhythmic Tasks* for modeling such tasks having continually varying periods/workloads depending on external physical events. We provide response-time analysis techniques for rhythmic tasks under constant speed, accelerating speed, and decelerating speed. We have also derived utilization bounds for some simple cases. We offer comprehensive guidelines to find schedulable utilization levels for the rhythmic task model. We have applied our schedulability analyses and guidelines to a case study in engine control. The case study shows how the proposed techniques can be used effectively.

Tasks with Parallel Threads

Parallel real-time tasks can significantly help emerging CPS meet demanding computational requirements while guaranteeing their timeliness constraints. Since there has not been much research on parallel real-time scheduling, we have formally defined a fork-join parallel real-time task model that has a varying number of parallel threads depending on physical attributes. A task transformation algorithm is proposed to improve the schedulability when parallel real-time tasks are co-located with periodic real-time tasks. When global deadline-monotonic scheduling is used, we obtain a resource augmentation bound of 3.73, which means that any task set that is feasible on m unit-speed processors can be scheduled by the proposed algorithm on m processors that are 3.73 times faster. The proposed scheme is implemented on Linux/RK as a proof of concept and applied to a case study in autonomous driving. The case study shows that our proposed scheme improves the quality of autonomous driving from the speed and curvature perspectives.

Tasks with Self-Suspensions

Segment-fixed priority scheduling is proposed to deal with self-suspending real-time tasks. A task with self-suspensions alternates between a computing segment and a suspending stage. The segment-fixed priority scheduling decomposes self-suspending tasks into multiple segments and assigns them different priorities and phases if needed. This mitigates the negative effects caused by (1) reduced available CPU time due to self-suspensions and (2) unknown suspension time during runtime. We have provided four heuristics and a MILP-based optimal algorithm that decides the priority and phase per task segment. Evaluation results show that one of our heuristics performs up to 40 times better than RMS, in comparison to task-fixed priority scheduling such as rate-monotonic scheduling. We have analytically identified the reason why the proposed scheduling algorithm is better in general by providing schedulability analyses and utilization bounds.

9.1.3 Runtime Support for Fault-Tolerance Features

We have proposed a layer called SAFER (System-level Architecture for Failure Evasion in Real-time applications) to incorporate configurable task-level fault-tolerance features using hot standbys and cold standbys in order to tolerate fail-stop processor and task failures for CPS. SAFER is implemented on the Linux operating system and integrated into an autonomous vehicle developed at Carnegie Mellon. The formal analyses of the worst-case timing behavior with and without failures are also provided, and our analysis engine is integrated with a model-based design tool, SysWeaver. We have presented measurements along with analytical results from the driving simulation scenarios used during the 2007 DARPA Urban Challenge. A case study in autonomous driving has also shown that there is no noticeable behavioral difference even when node failure is injected and the failed node later rejoins.

9.2 Future work

As CPS represent a relatively new area, there are still many possible research directions for future work relevant to this dissertation. We discuss below some of research topics that need to be studied in depth.

9.2.1 Adaptive Graceful Degradation

Graceful degradation is a well-established approach to maintain limited functionality in a system with a component failure. The basic idea behind this is to avoid potential undesirable effects by providing downgraded functionality to accommodate the reduction in available resources due to a failure. When it comes to autonomous vehicles, for example, graceful degradation should be appropriately adjusted depending on different situations. Suppose a failure takes down a processing board running vision algorithms to detect pedestrians. If a vehicle with the failure is driving on a highway, the vehicle may notify its driver of the failure and keep driving. If the vehicle is in an urban area, pedestrians are highly likely to be present. Hence, the vehicle may run the pedestrian detection algorithms in a degraded mode (possibly with a lower frame rate) on another live processing board and also slow down the vehicle. It is important to apply graceful degradation in an *adaptive* manner so that a system can recover a failure even with fewer resources.

In CPS such as autonomous vehicles, most algorithms (tasks) deal with a periodic sequence of perception, computation and control. The periods of such tasks play an important role in determining how much resources are required in the system. By dynamically adjusting task periods, we can regulate overall resource *utilization*. Lowering the utilization of a task creates more room for other tasks to use. In other words, a framework for *adaptive graceful degradation* is indispensable to run critical tasks with limited resources caused by a failure. For example, the vision algorithms mentioned above can be run on another live processing board along with tasks that are adjusted to have lower utilization.

9.2.2 Smart Sensor Control

Dealing with sensor and actuator failures is vital for obtaining dependable CPS. Since many accidents in avionics and automotive industries have been traced to unexpected sensor failures, how CPS can be tolerant to them should be investigated. CPS with cost and space constraints may not always be able to have redundant sensors. Thus, different sensor modalities can be leveraged when sensors, such as cameras and radars, have overlapping fields of view. Failure conditions for each sensor or actuator type must be identified, and methods must be developed to detect failures. Any detected failure will be notified to higher layers so that algorithms can be reconfigured to use fewer sensors or actuators while generating less accurate but useful outputs.

Many analog sensors are also prone to intermittent faults, so using different sensor modalities is better than duplicating the same type of sensors because different types of sensors typically react to the same environmental condition in diverse ways. Suppose a vehicle is equipped with radars for blind spot detection. If a backward-looking radar does not work properly, a vision algorithm detecting obstacles from images obtained through a rear-facing camera needs to be used. An autonomous vehicle may use a low-grade sensor with complex data-processing algorithms after a high-grade sensor with simple algorithms fails, until the vehicle can safely stop.

9.2.3 Runtime Support

Realizing adaptive graceful degradation and smart use of sensor/actuator modalities requires a runtime framework with *flexible* configuration options. SAFER must be extended to support adaptive graceful degradation. When a processing board failure is detected, the standbys of the primary tasks on the failed board can be activated to run elsewhere. If resources are limited, SAFER will make sure that all required tasks are executed in a degraded manner. The schedulability of the adjusted tasks can be guaranteed by using admission control algorithms or response-time tests that can handle varying periods using the rhythmic task model. Depending on a given condition, the best configuration parameters can be adaptively set by SAFER.

SAFER must also be extended to effectively use sensor/actuator modalities. The SAFER layer should have the capability to detect sensor anomalies that are different for each type of sensors, which can be a plug-in module for the SAFER layer. When a sensor failure happens, SAFER can trigger a different configuration using different types of sensors to recover from the failure. Since different data-processing algorithms are mandatory for different types of sensors, the logical combination among algorithms are given *a priori* as configuration parameters. Then, SAFER can assign the suitable amount of resources to tasks.

9.2.4 Occlusion-free Perception using Communications as a Sensor

Large-scale CPS should operate reliably . One of the possible approaches is to accomplish this by addressing the lack of global perception. As CPS stand now, each node constituting large-scale CPS can only access local sensors. Communications can be leveraged as a sensor to make local sensory information globally available. In intelligent transportation systems, for example, it will generate an occlusion-free perception system for safety (avoiding accidents), lower delays (avoiding congested routes), and fuel efficiency (avoiding sudden acceleration/deceleration). In virtual hospitals, a remote surgeon will be able to perform a surgery. Cooperative citywide surveillance systems will be able to find missing kids, prevent theft and robbery, and rescue people in danger. To this end, timely and reliable interactions among the CPS nodes will play a major role. The effects of any resource failures on the entire system must be minimized by isolating such failures. It is also important to understand how CPS can work appropriately with people, where uncertainties arise not just from the physical environment but also from the humans operating the system.

9.2.5 Workload Estimations for Cyber-Physical Systems

To verify CPS timeliness, a tool for predicting CPU and network loads is essential. A tight coupling between CPS algorithms and the physical inputs can be used to infer the loads. For example, a motion-planning algorithm for mobile robots requires more CPU utilization on curvy roads. Using this idea, methods can be developed to predict resource usage. By properly identifying dominant physical inputs (features), a machine learning based approach can be potentially leveraged.

9.2.6 Real-Time Scheduling for Heterogeneous Computing Architectures

Running complex computations on many-core processors, such as general-purpose graphics processing units (GP-GPU), will become common in CPS. However, this trend poses a challenge in guaranteeing timeliness. Using such hardware may lead to unpredictable suspension delays, hence reducing the benefits of using parallel computations. A runtime framework must be developed for a real-time task that can use both CPU and GPGPU. Depending on workload needs and system status, the task can run either CPU- or GPGPU-based implementations. This research will lead to new directions for efficient resource usage.

Bibliography

[1] K. Lakshmanan, S. Kato, and R. Rajkumar, "Scheduling Parallel Real-Time Tasks on Multi-core Processors," in *IEEE Real-Time Systems Symposium*, 2010, pp. 259–268.

[2] J. Wei, J. Snider, J. Kim, J. Dolan, R. Rajkumar, and B. Litkouhi, "Towards a Viable Autonomous Driving Research Platform," in *IEEE Intelligent Vehicles Symposium*, 2013.

[3] N. Budhiraja, K. Marzullo, F. Schneider, and S. Toueg, "The primary-backup approach," *Distributed systems*, vol. 2, pp. 199–216, 1993.

[4] R. Schlichting and F. Schneider, "Fail-stop processors: an approach to designing fault-tolerant computing systems," *ACM Transactions on Computer Systems*, vol. 1, no. 3, pp. 222–238, 1983. [Online]. Available: http://portal.acm.org/citation.cfm?id=357371

[5] T. X. Mei, M. Shafik, R. Lewis, H. Walilay, M. Whitley, and D. Baker, "Fault tolerant actuation for steer-by-wire applications," in *Institution of Engineering and Technology Conference on Automotive Electronics*, 2007, pp. 1–8.

[6] Z. Gu, S. Wang, J. Kim, and K. Shin, "Integrated Modeling and Analysis of Automotive Embedded Control Systems with Real-Time Scheduling," in *SAE 2004 World Congress*, 2004. [Online]. Available: http://papers.sae.org/2004-01-0279

[7] S. Oikawa and R. Rajkumar, "Portable RK: a portable resource kernel for guaranteed and enforced timing behavior," in *IEEE Real-Time and Embedded Technology and Applications Symposium*, 1999, pp. 111–120.

[8] C. Urmson, J. Anhalt, H. Bae, D. Bagnell, C. Baker, R. Bittner, T. Brown, M. Clark, M. Darms, D. Demitrish, J. Dolan, D. Duggins, D. Ferguson, T. Galatali, C. Geyer, M. Gittleman, S. Harbaugh, M. Hebert, T. Howard, S. Kolski, M. Likhachev, B. Litkouhi, A. Kelly, M. McNaughton, N. Miller, J. Nickolaou, K. Peterson, B. Pilnick, R. Rajkumar, P. Rybski, V. Sadekar, B. Salesky, Y.-W. Seo, S. Singh, J. Snider, J. Struble, A. Stentz, M. Taylor, W. Whittaker, Z. Wolkowicki, W. Zhang, and J. Ziglar, "Autonomous driving in urban environments: Boss and the Urban Challenge," *Journal of Field Robotics Special Issue on the 2007 DARPA Urban Challenge, Part I*, vol. 25, no. 1, pp. 425–466, June 2008.

[9] T. Gu and J. M. Dolan, "On-Road Motion Planning for Autonomous Vehicles," in *Intelligent Robotics and Applications*, 2012, vol. 7508, pp. 588–597. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-33503-7_57

[10] J. W. Liu, *Real-time systems.* Prentice Hall PTR, 2000.

[11] H. Cho, P. Rybski, and W. Zhang, "Vision-based 3D bicycle tracking using deformable part model and Interacting Multiple Model filter," in *IEEE International Conference on Robotics and Automation*, 2011, pp. 4391–4398.

[12] M. McNaughton, C. Urmson, J. M. Dolan, and J.-W. Lee, "Motion planning for autonomous driving with a conformal spatiotemporal lattice," in *IEEE International Conference on Robotics and Automation*, 2011, pp. 4889–4895.

[13] R. Rajkumar, *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, 1991.

[14] K. Lakshmanan and R. Rajkumar, "Scheduling Self-Suspending Real-Time Tasks with Rate-Monotonic Priorities," in *IEEE Real-Time and Embedded Technology and Applications Symposium*, 2010, pp. 3–12.

[15] D. de Niz, G. Bhatia, and R. Rajkumar, "Model-Based Development of Embedded Systems: The SysWeaver Approach," in *IEEE Real-Time and Embedded Technology and*

Applications Symposium, 2006, pp. 231–242.

[16] C. L. Liu and J. W. Layland, “Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment,” *Journal of The ACM*, vol. 20, pp. 46–61, 1973.

[17] S. Dhall and C. Liu, “On a real-time scheduling problem,” *Operations Research*, vol. 26, no. 1, pp. 127–140, 1978.

[18] M. Dertouzos and A. Mok, “Multiprocessor Online Scheduling of Hard-Real-Time Tasks,” *IEEE Transactions on Software Engineering*, vol. 15, pp. 1497–1506, 1989.

[19] K. Ramamritham, J. A. Stankovic, and P. Shiah, “Efficient Scheduling Algorithms for Real-Time Multiprocessor Systems,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, pp. 184–194, 1990.

[20] B. Andersson, S. Baruah, and J. Jonsson, “Static-Priority Scheduling on Multiprocessors,” in *IEEE Real-Time Systems Symposium*, 2001, pp. 193–202.

[21] S. Baruah and T. Baker, “Schedulability analysis of global EDF,” *Real-Time Systems*, vol. 38, pp. 223–235, 2008.

[22] N. Fisher, S. K. Baruah, and T. P. Baker, “The Partitioned Scheduling of Sporadic Tasks According to Static-Priorities,” in *Euromicro Conference on Real-Time Systems*, 2006, pp. 118–127.

[23] K. Lakshmanan, R. Rajkumar, and J. P. Lehoczky, “Partitioned Fixed-Priority Preemptive Scheduling for Multi-core Processors,” in *Euromicro Conference on Real-Time Systems*, 2009, pp. 239–248.

[24] S. Kato and N. Yamasaki, “Real-Time Scheduling with Task Splitting on Multiprocessors,” in *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2007, pp. 441–450.

[25] B. Randell, “System Structure for Software Fault Tolerance,” *IEEE Transactions on Software Engineering*, vol. 1, pp. 221–232, 1975.

[26] J. Laprie, J. Arlat, C. Bounes, and K. Kanoun, “Definition and Analysis of Hardware and Software-Fault-Tolerant Architectures,” *IEEE Computer*, vol. 23, pp. 39–51, 1990.

[27] B. Natarajan, A. S. Gokhale, S. Yajnik, and D. C. Schmidt, “DOORS: Towards High-Performance Fault Tolerant CORBA,” in *International Symposium on Distributed Objects and Applications*, 2000, pp. 39–48.

[28] G. Manimaran and C. S. R. Murthy, “An Efficient Dynamic Scheduling Algorithm For Multiprocessor Real-Time Systems,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, pp. 312–319, 1998.

[29] S. Punnekkat, A. Burns, and R. I. Davis, “Analysis of Checkpointing for Real-Time Systems,” *Real-time Systems*, vol. 20, pp. 83–102, 2001.

[30] S. Ghosh, R. Melhem, D. Moss, and J. S. Sarma, “Fault-Tolerant Rate-Monotonic scheduling,” *Real-Time Systems*, vol. 15, no. 2, pp. 149–181, 1998. [Online]. Available: http://dx.doi.org/10.1023/A:1008046012844

[31] C. M. Krishna and K. G. Shin, “On Scheduling Tasks with a Quick Recovery from Failure,” *IEEE Transactions on Computers*, vol. 35, no. 5, pp. 448–455, 1986.

[32] A. A. Bertossi, L. V. Mancini, and F. Rossini, “Fault-Tolerant Rate-Monotonic First-Fit Scheduling in Hard-Real-Time Systems,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, pp. 934–945, 1999.

[33] J. Chen, C. Yang, T. Kuo, and S. Tseng, “Real-Time Task Replication for Fault Tolerance in Identical Multiprocessor Systems,” in *IEEE Real-Time and Embedded Technology and Applications Symposium*, 2007, pp. 249–258.

[34] A. Mok, “Fundamental design problems of distributed systems for the real-time environment,” *Ph.D. thesis, Dept. of Electrical Engineering and Computer Science, MIT, Cambridge, Mass.*, 1983.

[35] J. Lehoczky, “Fixed priority scheduling of periodic task sets with arbitrary deadlines,” in

IEEE Real-Time Systems Symposium, 1990, pp. 201–209.

[36] T. Gillespie, *Fundamentals of Vehicle Dynamics*. Society of Automotive Engineers, 1992.

[37] C. Li and Z. Jianwu, "WCET analysis for gasoline engine control," in *IEEE International Conference on Mechatronics and Automation*, 2005, pp. 2090–2095.

[38] G. C. Buttazzo, G. Lipari, M. Caccamo, and L. Abeni, "Elastic Scheduling for Flexible Workload Management," *IEEE Transactions on Computers*, vol. 51, pp. 289–302, 2002.

[39] R. Guerra and G. Fohler, "A gravitational task model for target sensitive real-time applications," in *Euromicro Conference on Real-Time Systems*, 2008, pp. 309–317.

[40] H. Wei, K. Lin, W. Lu, and W. Shih, "Generalized rate monotonic schedulability bounds using relative period ratios," *Information Processing Letters*, vol. 107, no. 5, pp. 142 – 148, 2008. [Online]. Available: http://www.sciencedirect.com/science/article/B6V0F-4RY6WVS-4/2/71afe80fbb74c1d78c351bb2de655001

[41] L. George and J. Hermant, "A Norm Approach for the Partitioned EDF Scheduling of Sporadic Task Systems," in *Euromicro Conference on Real-Time Systems*, 2009, pp. 161–169.

[42] T. Abdelzaher, V. Sharma, and C. Lu, "A utilization bound for aperiodic tasks and priority driven scheduling," *IEEE Transactions on Computers*, vol. 53, no. 3, pp. 334–350, 2004.

[43] L. Sha, R. Rajkumar, J. Lehoczky, and K. Ramamritham, "Mode change protocols for priority-driven preemptive scheduling," *Real-Time Systems*, vol. 1, no. 3, pp. 243–264, 1989. [Online]. Available: http://www.springerlink.com/content/kv0833w5glr16301/

[44] A. Rowe, M. Berges, and R. Rajkumar, "Contactless sensing of appliance state transitions through variations in electromagnetic fields," in *ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Building*, 2010, pp. 19–24. [Online]. Available: http://doi.acm.org/10.1145/1878431.1878437

[45] S. Funk, J. Goossens, and S. Baruah, “On-line scheduling on uniform multiprocessors,” in *IEEE Real-Time Systems Symposium*, dec. 2001, pp. 183 – 192.

[46] T. Baker, “Multiprocessor EDF and deadline monotonic schedulability analysis,” in *IEEE Real-Time Systems Symposium*, 2003, pp. 120 – 129.

[47] ——, “An analysis of deadline-monotonic schedulability on a multiprocessor,” TR-030201, Department of Computer Science, Florida State University, Tech. Rep., 2003.

[48] ——, “An analysis of EDF schedulability on a multiprocessor,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 16, no. 8, pp. 760–768, 2005.

[49] M. Bertogna, M. Cirinei, and G. Lipari, “New Schedulability Tests for Real-Time Task Sets Scheduled by Deadline Monotonic on Multiprocessors,” in *Principles of Distributed Systems*, 2006, vol. 3974, pp. 306–321. [Online]. Available: http://dx.doi.org/10.1007/11795490_24

[50] ——, “Schedulability Analysis of Global Scheduling Algorithms on Multiprocessor Platforms,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, pp. 553–566, 2009.

[51] R. I. Davis and A. Burns, “A survey of hard real-time scheduling for multiprocessor systems,” *ACM Computing Surveys*, vol. 43, no. 4, pp. 35:1–35:44, 2011. [Online]. Available: http://doi.acm.org/10.1145/1978802.1978814

[52] S. Kato and Y. Ishikawa, “Gang EDF Scheduling of Parallel Task Systems,” in *IEEE Real-Time Systems Symposium*, 2009, pp. 459–468.

[53] A. Saifullah, K. Agrawal, C. Lu, and C. Gill, “Multi-core Real-time Scheduling for Generalized Parallel Task Models,” in *IEEE Real-Time Systems Symposium*, 2011, pp. 217–226.

[54] G. Nelissen, V. Berten, J. Goossens, and D. Milojevic, “Techniques Optimizing the Number of Processors to Schedule Multi-threaded Tasks,” in *Euromicro Conference on Real-Time Systems*, 2012, pp. 321–330.

[55] P. Gai, M. D. Natale, G. Lipari, A. Ferrari, C. Gabellini, and P. Marceca, "A comparison of MPCP and MSRP when sharing resources in the Janus multiple-processor on a chip platform," in *IEEE Real Time Technology and Applications Symposium*, 2003, pp. 189–198.

[56] F. Ridouard, P. Richard, and F. Cottet, "Negative Results for Scheduling Independent Hard Real-Time Tasks with Self-Suspensions," in *IEEE Real-Time Systems Symposium*, 2004, pp. 47–56.

[57] K. Bletsas and N. C. Audsley, "Extended Analysis with Reduced Pessimism for Systems with Limited Parallelism," in *Real-Time Computing Systems and Applications*, 2005, pp. 525–531.

[58] F. Ridouard, P. Richard, F. Cottet, and K. Traore, "Some results on scheduling tasks with self-suspensions," *Journal of Embedded Computing*, vol. 2, pp. 301–312, 2006.

[59] C. Liu and J. Anderson, "An O(m) Analysis Technique for Supporting Real-Time Self-Suspending Task Systems," in *IEEE Real-Time Systems Symposium*, 2012, pp. 373–382.

[60] G. Elliott and J. Anderson, "Globally scheduled real-time multiprocessor systems with GPUs," *Real-Time Systems*, vol. 48, no. 1, pp. 34–74, 2012.

[61] K. Tindell, *Adding time-offsets to schedulability analysis*. University of York, Department of Computer Science, 1994.

[62] J. C. Palencia and M. Gonzalez Harbour, "Schedulability analysis for tasks with static and dynamic offsets," in *IEEE Real-Time Systems Symposium*, 1998, pp. 26–37.

[63] J. Maki-Turja and M. Nolin, "Efficient response-time analysis for tasks with offsets," in *IEEE Real-Time and Embedded Technology and Applications Symposium*, 2004, pp. 462–471.

[64] R. Pellizzoni and G. Lipari, "Feasibility Analysis of Real-Time Periodic Tasks with Offsets," *Real-Time Systems*, vol. 30, no. 1-2, pp. 105–128, 2005.

[65] K. P. Birman, “Replication and fault-tolerance in the ISIS system,” in *ACM Symposium on Operating Systems Principles*, 1985, pp. 79–86. [Online]. Available: http://doi.acm.org/10.1145/323647.323636

[66] Object Management Group, “Fault-Tolerant CORBA,” *OMG Technical Committee Document formal/2001-09-29*, September 2001.

[67] S. Maffeis, “Adding group communication and fault-tolerance to CORBA,” in *USENIX Conference on Object-Oriented Technologies*, 1995, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1268098.1268110

[68] P. Felber and P. Narasimhan, “Experiences, strategies, and challenges in building fault-tolerant CORBA systems,” *IEEE Transactions on Computers*, pp. 467–511, 2004.

[69] S. Sadjadi and P. McKinley, “A survey of adaptive middleware,” *Michigan State University Report MSU-CSE-03-35*, 2003.

[70] G. D. Parrington, S. K. Shrivastava, S. M. Wheater, and M. C. Little, “The Design and Implementation of Arjuna,” *Computing Systems*, vol. 8, pp. 255–308, 1995.

[71] T. Friese, J. P. Mller, and B. Freisleben, “Self-healing Execution of Business Processes Based on a Peer-to-Peer Service Architecture,” in *Systems Aspects in Organic and Pervasive Computing - ARCS 2005*, ser. Lecture Notes in Computer Science, 2005, vol. 3432, pp. 108–123. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-31967-2_8

[72] Z. Cai, V. Kumar, B. F. Cooper, G. Eisenhauer, K. Schwan, and R. E. Strom, “Utility-Driven Proactive Management of Availability in Enterprise-Scale Information Flows,” in *International Conference on Distributed Systems Platforms and Open Distributed Processing/Open Distributed Processing*, 2006, pp. 382–403.

[73] P. Narasimhan, T. Dumitras, A. M. Paulos, S. M. Pertet, C. F. Reverte, J. G. Slember, and D. Srivastava, “MEAD: support for Real-Time Fault-Tolerant CORBA,” *Concurrency and Computation: Practice and Experience*, vol. 17, pp. 1527–1545, 2005.

[74] J. Balasubramanian, S. Tambe, C. Lu, A. S. Gokhale, C. D. Gill, and D. C. Schmidt, “Adaptive Failover for Real-Time Middleware with Passive Replication,” in *IEEE Real-Time and Embedded Technology and Applications Symposium*, 2009, pp. 118–127.

[75] S. Roumeliotis, G. Sukhatme, and G. A. Bekey, “Sensor Fault Detection and Identification in a Mobile Robot,” in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, vol. 3, 1998, pp. 1383–1388.

[76] F. Koushanfar, M. Potkonjak, and A. Sangiovanni-Vincentelli, “On-line fault detection of sensor measurements,” in *IEEE Sensors*, vol. 2, 2003, pp. 974–979.

[77] L. Balzano, “Addressing Fault and Calibration in Wireless Sensor Networks,” Master’s thesis, University of California, Los Angeles, 2007.

[78] A. Fekr, M. Janidarmian, O. Sarbishei, B. Nahill, K. Radecka, and Z. Zilic, “MSE Minimization and Fault-Tolerant Data Fusion for Multi-Sensor Systems,” in *IEEE International Conference on Computer Design*, 2012, pp. 445–452.

[79] P. Maybeck and P. Hanlon, “Performance enhancement of a multiple model adaptive estimator,” *IEEE Transactions on Aerospace and Electronic Systems*, vol. 31, no. 4, pp. 1240–1254, 1995.

[80] N. Mehranbod, M. Soroush, M. Piovoso, and B. A. Ogunnaike, “Probabilistic model for sensor fault detection and identification,” *AIChE Journal*, vol. 49, no. 7, pp. 1787–1802, 2003.

[81] P. Ashok, G. Krishnamoorthy, and D. Tesar, “Guidelines for Managing Sensors in Cyber Physical Systems with Multiple Sensors,” *Journal of Sensors*, vol. vol. 2011, 2012.

[82] J. Frolik, M. Abdelrahman, and P. Kandasamy, “A confidence-based approach to the self-validation, fusion and reconstruction of quasi-redundant sensor data,” *IEEE Transactions on Instrumentation and Measurement*, vol. 50, no. 6, pp. 1761–1769, 2001.

[83] S. J. Wellington, J. Atkinson, and R. P. Sion, “Sensor validation and fusion using the

Nadaraya-Watson statistical estimator," in *IEEE International Conference on Information Fusion*, vol. 1, 2002, pp. 321–326.

[84] L. Jiang, D. Djurdjanovic, and J. Ni, "A new method for sensor degradation detection, isolation and compensation in linear systems," in *ASME International Mechanical Engineering Congress and Exposition*, 2007, pp. 1089–1101.

[85] K. Marzullo, "Tolerating failures of continuous-valued sensors," *ACM Transactions on Computer Systems*, vol. 8, no. 4, pp. 284–304, 1990. [Online]. Available: http://doi.acm.org/10.1145/128733.128735

[86] F. Koushanfar, S. Slijepcevic, M. Potkonjak, and A. Sangiovanni-Vincentelli, "Error-Tolerant Multi-Modal Sensor Fusion," in *IEEE CAS Workshop on Wireless Communication and Networking*, 2002.

[87] D. Hall and J. Llinas, "An introduction to multisensor data fusion," *Proceedings of the IEEE*, vol. 85, no. 1, pp. 6–23, 1997.

[88] R. Isermann, R. Schwarz, and S. Stolzl, "Fault-tolerant drive-by-wire systems," *IEEE Control Systems*, vol. 22, no. 5, pp. 64–81, 2002.

[89] W. Xiang, P. Richardson, C. Zhao, and S. Mohammad, "Automobile brake-by-wire control system design and analysis," *IEEE Transactions on Vehicular Technology*, vol. 57, no. 1, pp. 138–145, 2008.

[90] P. Sinha, "Architectural design and reliability analysis of a fail-operational brake-by-wire system from iso 26262 perspectives," *Reliability Engineering and System Safety*, vol. 96, no. 10, pp. 1349–1359, 2011.

[91] T. Skotnicki, J. A. Hutchby, T. J. King, H. S. Wong, F. Boeuf, and S. T. Microelectronics, "The end of CMOS scaling: toward the introduction of new materials and structural changes to improve MOSFET performance," *IEEE Circuits and Devices Magazine*, vol. 21, no. 1, p. 1626, 2005.

[92] J. Lehoczky, L. Sha, and Y. Ding, “The rate monotonic scheduling algorithm: Exact characterization and average case behavior,” *10th IEEE Real-Time Systems Symposium*, pp. 166–171, 1989.

[93] D. Oh and T. Baker, “Utilization bounds for n-processor rate monotonic scheduling with static processor assignment,” *Real-Time Systems*, vol. 15, pp. 183–192, 1998.

[94] D. S. Johnson, “Near optimal allocation algorithms,” *Ph.D. Dissertation, MIT, Cambridge, MA*, 1973.

[95] D. Johnson, A. Demers, J. Ullman, M. Garey, and R. Graham, “Worst-case performance bounds for simple one-dimensional packing algorithms,” *SIAM Journal on Computing*, vol. 3, p. 299, 1974.

[96] *IEC 61508 - Functional safety of electrical/electronic/programmable electronic safety-related systems*. International Electrotechnical Commission (IEC), Dec. 1998.

[97] R. K. Jurgen, “X-by-wire automotive systems,” *Training*, vol. 2013, pp. 10–14, 2009.

[98] B. d. et dAnalyses, “Final report on the accident on 1st june 2009 to the airbus a330-203 registered f-gzcp operated by air france flight af 447 rio de janeiro–paris,” 2012.

[99] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, “Basic concepts and taxonomy of dependable and secure computing,” *Dependable and Secure Computing, IEEE Transactions on*, vol. 1, no. 1, pp. 11–33, 2004.

[100] J. Kim, K. Lakshmanan, and R. Rajkumar, “R-BATCH: Task partitioning for fault-tolerant multiprocessor real-time systems,” in *IEEE International Conference on Computer and Information Technology*. IEEE, 2010, pp. 1872–1879.

[101] AUTOSAR Administration, “Specification of Operating System V5.0.0 R4.0 Rev 3,” 2011.

[102] D. K. Pradhan, *Fault-tolerant computer system design*. Prentice-Hall, Inc., 1996.

[103] R. Belschner, J. Berwanger, C. Ebner, H. Eisele, S. Fluhrer, T. Forest, T. Führer,

F. Hartwich, B. Hedenetz, R. Hugel *et al.*, "Flexray requirements specification," *FlexRay Consortium, Internet: http://www. flexray. com, Version*, vol. 2, no. 2, 2002.

[104] K. Lakshmanan, G. Bhatia, and R. Rajkumar, "Integrated end-to-end timing analysis of networked autosar-compliant systems," in *Proceedings of the Conference on Design, Automation and Test in Europe*. European Design and Automation Association, 2010, pp. 331–334.

[105] P.-E. Hladik, A. Deplanche, S. Faucou, and Y. Trinquet, "Adequacy between autosar os specification and real-time scheduling theory," in *Industrial Embedded Systems, 2007. SIES'07. International Symposium on*. IEEE, 2007, pp. 225–233.

[106] Freescale Semiconductor, Inc., "MPC8640: MPC8640D Integrated Dual-Core Processor," http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=MPC8640 as of Jan 31, 2013.

[107] OSEK, "OSEK/VDX Operating System Specification 2.2.3," 2005.

[108] *ISO/DIS 26262 - Road vehicles - Functional safety*. International Organization for Standardization (ISO), 2009.

[109] R. Rajkumar, I. Lee, L. Sha, and J. Stankovic, "Cyber-physical systems: the next computing revolution," in *Proceedings of the 47th Design Automation Conference*, ser. DAC '10. New York, NY, USA: ACM, 2010, pp. 731–736. [Online]. Available: http://doi.acm.org/10.1145/1837274.1837461

[110] "Four stroke engine," http://www.ustudy.in/node/3268 as of Feb 2012.

[111] A. Burns and S. Baruah, "Sustainability in real-time scheduling," *Journal of Computing Science and Engineering*, vol. 2, no. 1, pp. 74–97, 2008.

[112] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings, "Applying new scheduling theory to static priority pre-emptive scheduling," *Software Engineering Journal*, vol. 8, no. 5, pp. 284–292, 1993.

[113] M. Joseph and P. Pandya, "Finding response times in a real-time system," *The Computer Journal*, vol. 29, no. 5, p. 390, 1986.

[114] X. Hu, J. D'Ambrosio, B. Murray, and D. Tang, "Codesign of architectures for automotive powertrain modules," *Micro, IEEE*, vol. 14, no. 4, pp. 17–25, 1994.

[115] BMW ConnectedDrive, "Take Over Please!" http://www.bmw.com/com/en/insights/technology/connecteddrive/2010/future_lab/index.html#/1/4 as of Jul 31, 2013, 2010.

[116] S. Pinkow, "Continental Tests Highly-Automated Driving," http://www.conti-online.com/generator/www/com/en/continental/pressportal/themes/press_releases/3_automotive_group/chassis_safety/press_releases/pr_2012_03_23_automated_driving_en.html as of Jul 31, 2013, 2012.

[117] GM Press, "Self-Driving Car in Cadillac's Future," http://media.gm.com/media/us/en/cadillac/news.detail.html/content/Pages/news/us/en/2012/Apr/0420_cadillac.html as of Jul 31, 2013, 2012.

[118] S. Thrun, "What we're driving at," http://googleblog.blogspot.com/2010/10/what-were-driving-at.html as of Jul 31, 2013, 2010.

[119] L. de Ambroggi, "Multicore Trends in Automotive Offer Cost Savings, Higher Performance," iSuppli, 2011.

[120] "OpenMP Application Program Interface Version 3.1," http://openmp.org, 2011.

[121] S. Oikawa and R. Rajkumar, "Linux/RK: A portable resource kernel in Linux," in *IEEE Real-Time Systems Sumposium (RTSS) Work-In-Progress*, 1998.

[122] G. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference*. ACM, 1967, pp. 483–485.

[123] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa, "Resource kernels: A resource-centric approach to real-time and multimedia systems," in *SPIE/ACM Conference on Multimedia*

Computing and Networking, 1998.

[124] M. McNaughton, C. R. Baker, T. Galatali, B. Salesky, C. Urmson, and J. Ziglar, "Software infrastructure for an autonomous ground vehicle," *Journal of Aerospace Computing, Information, and Communication*, vol. 5, no. 1, pp. 491 – 505, 2008.

[125] J. Kim, G. Bhatia, R. Rajkumar, and M. Jochim, "SAFER: System-level Architecture for Failure Evasion in Real-time Applications," in *IEEE Real-Time Systems Symposium (RTSS)*, 2012.

[126] J. Kim, K. Lakshmanan, and R. Rajkumar, "Rhythmic Tasks: A New Task Model with Continually Varying Periods for Cyber-Physical Systems," in *IEEE/ACM Third International Conference on Cyber-Physical Systems*, 2012, pp. 55–64.

[127] Y.-K. Chen and S. Kung, "Trend and Challenge on System-on-a-Chip Designs," *Journal of Signal Processing Systems*, vol. 53, 2008.

[128] J. Kim, H. Kim, K. Lakshmanan, and R. R. Rajkumar, "Parallel Scheduling for Cyber-Physical Systems: Analysis and Case Study on a Self-Driving Car," in *ACM/IEEE 4th International Conference on Cyber-Physical Systems (ICCPS)*, 2013.

[129] J. Kim, G. Bhatia, R. Rajkumar, and M. Jochim, "An AUTOSAR-Compliant Automotive Platform for Meeting Reliability and Timing Constraints," *SAE 2011 World Congress*, 2011.

[130] F. Cristian, "Reaching agreement on processor-group membership in synchronous distributed systems," *Distributed Computing*, 1991. [Online]. Available: http://dx.doi.org/10.1007/BF01784719

[131] R. Rajkumar and M. Gagliardi, "High availability in the real-time publisher/subscriber inter-process communication model," in *Proc. of the 17th IEEE Real-Time Systems Symposium (RTSS)*, 1996.

[132] D. L. Mills, "Internet time synchronization: the network time protocol," *IEEE Transac-*

tions on Communications, vol. 39, no. 10, pp. 1482–1493, 1991.

[133] B. Heck, L. Wills, and G. Vachtsevanos, "Software technology for implementing reusable, distributed control systems," *Applications of Intelligent Control to Engineering Systems*, pp. 267–293, 2009.