Title

Safe Adaptive System Reconfiguration in Autonomous Machines

Author

Yi, Saehanseul

Publication Date

2023

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA,
IRVINE

Safe Adaptive System Reconfiguration in Autonomous Machines

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Saehanseul Yi

Dissertation Committee:
Distinguished Professor Nikil Dutt, Chair
Professor Fadi Kurdahi
Professor Tony Givargis

2023

DEDICATION

To my family and friends.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ACKNOWLEDGMENTS

I am very fortunate to have Nik as my advisor for his unrelenting support and encouragement throughout my doctoral journey. His presence was indispensable in helping me navigate through this challenging journey, and I could not have reached the finish line without his support. I extend my gratitude to Prof. Fadi Kurdahi for our several years of discussions that have provided invaluable guidance for my research endeavors. I would like to thank Prof. and Chair Tony Givargis for his feedback during my advancement and the final defense, which has enabled me to move forward with a new perspective. Both as a department chair and manager, he and Holly Byrnes offered invaluable support that I deeply appreciate.

I extend my gratitude to Prof. Anton Burtsev and Prof. Ardalan Amiri Sani for for their continuous assistance over numerous quarters of the course Operating Systems, which has contributed significantly to my growth as an educator. I have gained valuable insights into teaching and learned a great deal.

Thank you Prof. Andreas Herkersdorf from TU Munich for the warm welcome and the opportunity to join as a visiting scientist. The experience proved to be highly rewarding, as it facilitated the expansion of my research and provided the chance to connect with fellow researchers. Among them, thanks to Dr. Anh Vu Doan for guiding me in the realm of optimization and for our countless productive discussions.

I extend my heartfelt gratitude to my family, who always have provided unwavering support from across the Pacific Ocean. Thanks to Jaehun Kim for making Irvine feel like my home country. Thanks to Biswadip Maity, Caio Batista, Sina Labbaf, Kenneth Stewart, Dongjoo Seo and other Dutt Research Group members for many fun PhD nights, as well as our valuable discussions and collaboration efforts. Thanks to all my friends who may not have been mentioned here.

I am grateful for the funding support for my research, which was provided by the Donald Bren School of Information and Computer Sciences, National Science Foundation grant CCF-170485. Lastly, I extend my thanks to IEEE, and ACM for granting permission to incorporate previously published work into my dissertation.

Portions of Chapter 2 and Chapter 3 of this dissertation is a reprint of the material as it appears in *Energy-Efficient Adaptive System Reconfiguration for Dynamic Deadlines in Autonomous Driving*, Saehanseul Yi, Tae-Wook Kim, Jong-Chan Kim, Nikil Dutt, IEEE International Symposium On Real-Time Distributed Computing (ISORC), 2021, used with permission from IEEE.

Portions of Chapter 2 and Chapter 3 of this dissertation is a reprint of the material as it appears in *EASYR: Energy-Efficient Adaptive System Reconfiguration for Dynamic Deadlines in Autonomous Driving on Multicore Processors*, Saehanseul Yi, Tae-Wook Kim, Jong-Chan

Kim, Nikil Dutt, ACM Transactions on Embedded Computing Systems (TECS), 2023, used with permission from ACM.

Portions of Chapter 2 and Chapter 4 of this dissertation is a reprint of the material as it appears in *BoostIID: Fault-agnostic Online Detection of WCET Changes in Autonomous Driving*, Saehanseul Yi, Nikil Dutt, IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC), 2024, used with permission from IEEE/ACM.

VITA

Saehanseul Yi

EDUCATION

Doctor of Philosophy in Computer Science **2023**
University of California, Irvine *Irvine, CA*

Master of Science in Electrical and Computer Engineering **2015**
University of Seoul *Seoul, Korea*

Bachelor of Science in Electrical and Computer Engineering **2013**
University of Seoul *Seoul, Korea*

RESEARCH EXPERIENCE

Graduate Research Assistant **2018–2023**
University of California, Irvine *Irvine, California*

Visiting Scientist **2019–2019**
Technical University of Munich *Munich, Germany*

Graduate Research Assistant **2013–2015**
University of Seoul *Seoul, Korea*

Undergraduate Research Assistant **2012–2013**
University of Seoul *Seoul, Korea*

TEACHING EXPERIENCE

Teaching Assistant **2018–2023**
University of California, Irvine *Irvine, CA*

INDUSTRY EXPERIENCE

Software Engineering Intern **2022–2022**
Intel Corporation *Santa Clara, CA*

Associate Research Engineer **2015–2018**
Dasan Network Solutions. *Seongnam, Korea*

REFEREED JOURNAL PUBLICATIONS

EASYR: Energy-Efficient Adaptive System Reconfiguration for Dynamic Deadlines in Autonomous Driving on Multicore Processors **2023**
ACM Transactions on Embedded Computing Systems

Chauffeur: Benchmark Suite for Design and End-to-End Analysis of Self-Driving Vehicles on Embedded Systems **2021**
ACM Transactions on Embedded Computing Systems

HESSLE-FREE: Heterogeneous Systems Leveraging Fuzzy Control for Runtime Resource Management **2019**
ACM Transactions on Embedded Computing Systems

REFEREED CONFERENCE PUBLICATIONS

BoostIID: Fault-agnostic Online Detection of WCET Changes in Autonomous Driving **2024**
IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)

Information Processing Factory 2.0 - Self-awareness for Autonomous Collaborative Systems **2023**
Design Automation & Test in Europe (DATE)

Demand layering for real-time dnn inference with minimized memory usage **2022**
IEEE Real-Time Systems Symposium (RTSS)

Energy-Efficient Adaptive System Reconfiguration for Dynamic Deadlines in Autonomous Driving **2021**
IEEE International Symposium on Real-Time Distributed Computing (ISORC)

The information processing factory: a paradigm for life cycle management of dependable systems **2019**
CODES+ISSS

BOOK CHAPTERS

Reflecting on Self-aware Systems-on-Chip **2020**
A Journey of Embedded and Cyber-Physical Systems

ABSTRACT OF THE DISSERTATION

Safe Adaptive System Reconfiguration in Autonomous Machines

By

Saehanseul Yi

Doctor of Philosophy in Computer Science

University of California, Irvine, 2023

Distinguished Professor Nikil Dutt, Chair

An autonomous machine operates on its own by continuously sensing the environment, making decisions, and executing actions. Due to this attribute, it is well-suited for managing critical tasks that demand rapid responses without human intervention such as avionics, surgical robots, and autonomous driving. Given that failures in such tasks can have catastrophic consequences, these autonomous machines are meticulously designed with stringent timing safety guarantees, which have been extensively studied and integrated into real-time systems. As hardware and software complexities increase, the fundamental concepts in real-time systems, namely worst-case behaviors and determinism, are becoming extremely challenging to analyze. The recent trend in autonomous driving exacerbates this challenge even further. As tasks previously performed by humans are now delegated to real-time systems, the workloads are increasingly demanding in terms of performance, which in turn necessitates the use of modern complex hardware. The widening gap between worst case scenarios and typical operating conditions results in pessimistic resource planning and high energy consumption. Furthermore, design-time analyses and static design struggle to address the dynamic nature of the physical world, replete with ever-changing situations.

In this dissertation, I delve into the concept of operation modes as an approach to tackle

the challenges in designing real-time systems that interact with the dynamic physical world. First, I introduce the EASYR framework, which leverages adaptive system reconfiguration to enhance energy efficiency. EASYR reacts to velocity changes and associated end-to-end deadline changes, holistically adapting scheduling parameters such as task period and speed factor while satisfying timing constraints. The possible deadline range is divided into several discrete operation modes, each with its maximum end-to-end latency guarantees. EASYR not only optimizes the energy efficiency within each mode but also provides a safe transition between these modes. Consequently, in real-world driving scenario, simulation results show that EASYR achieved 62% reduction in energy consumption compared to classical static deadline optimization.

Next, I introduce the FRTS framework, which exploits adaptive system reconfiguration to improve resilience of timing safety against WCET changes. Carefully designed real-time systems are confronting diverse faults and aging effects resulting from their increased lifespan. Safety critical systems must be safe when designed, and continue to be safe as conditions change. In FRTS, faults are manifested by a statistical distribution changes in execution times. Testing the assumption of independent & identical distribution (i.i.d) is well-suited for proactively sensing these changes prior to actual violations. Once a fault is detected, the system enters the emergency mode, during which system utilization is secured as a buffer against WCET changes. With the buffer, the system is capable of assessing the fault impact and performing reconfiguration to an optimized operating point. As a result, the system exhibits greater resilience compared to a conventional fault-tolerance technique with an improvement of 18% with a small energy overhead in simulation results.

These two frameworks demonstrate a great potential to improve autonomous operations by facilitating adaptation to dynamic situations. This is a stepping stone toward a true autonomous reconfiguration of safety-critical machines.

Chapter 1

Introduction

Autonomous machines are designed to operate on its own with minimal user intervention. These machines are required to operate continuously without arbitrary stops, making it suitable for handling critical jobs: surgical robots, military drones, self-driving cars, and so forth. The recent advancement in computing hardware shows a great potential of these machines. However, safety is a major road block for these machines to spread.

The failure of safety-critical autonomous machines often results in catastrophic accidents. To prevent such failures, it is necessary to remove potentially dangerous operation modes at design time. In that sense, there are mainly two sets of constraints for safe design: (i) timing and (ii) functional correctness. The former guarantees the system responses within a given time. The latter guarantees the correct output of a functionality, which encompasses the fault-tolerance of hardware modules.

As system complexity grows, ensuring timing correctness becomes more challenging. Traditionally, two key concepts are involved in ensuring timing correctness of a system. The first key concept is *worst-case behaviors*. Given that failures generally occur outside regu-

lar operation, it is imperative to identify possible worst-case scenarios and be prepared for them. Particularly, it is crucial to determine the worst-case execution time (WCET) of tasks as a prerequisite for subsequent timing analyses. The second key concept is *determinism*. Once the worst-case behaviors are identified, the subsequent step involves identifying when these behaviors could potentially occur. Consequently, determinism enhances predictability, leading to a reduction in uncertainties in the system. The concept of determinism can be implemented at low levels, such as cache policies, or extended to higher levels, such as task schedulers. It is not uncommon to trade off performance in order to minimize uncertainty, sometimes necessitating specialized hardware to mitigate this trade-off. As a result, safety-critical systems becomes a rigid monolith that has tightly integrated structures constructed upon stringent timing analyses and policies for deterministic behaviors, all aimed at enforcing timing constraints with predictability. In avionics, timing correctness is provided by time and spatial partitioning of resources as suggested in ARNIC 653 software specification. The implementation from VxWorks [78] utilizes several guest operating systems on top of the main real-time operating system. It is worth bearing such a huge overhead to ensure safety as long as workloads are not computationally demanding.

However, the rigid monolith approach faces several challenges in light of the advancements in hardware and software for autonomous systems. Notably, the recent development in computer vision technology has prompted both academia and industry to pursue the realization of autonomous driving (AD). AD serves as an excellent example to bring autonomous systems to the next level, especially considering the novel environment it must navigate through: other vehicles, traffic lights, pedestrians, and more. Indeed, the foremost challenge in AD lies in perception of the surrounding environment, which in itself poses significant hurdles to the existing design methodologies for autonomous systems. The core component of perception pipeline in AD relies on deep neural network (DNN), which are inherently data-intensive and

compute-intensive. These workloads demand cutting-edge, commercial-off-the-shelf (COTS) hardware that previous timing analyses have not yet fully encompassed.

	2008	2010	2014	2018	2020
Autonomous level			2	3	4
# Features	5	8	14	18	21
Key features	Lane Departure Warning	Forward Collision Warning	Animal Detection	Mapping, Localization	Full Image Detection
Perf. (TOPS)[1]	0.0044	0.026	0.256	2.5	24
Power (W)	2.5	2.5	2.5	3	10
Semiconductor	180 nm CMOS	90 nm CMOS	40 nm CMOS	28 nm FD-SOI	7 nm FinFET

[1]TOPS: Trillion operations per second

Table 1.1: The evolution of Mobileye's ADAS

For example, Table 1.1 shows the evolution of Mobileye's Advance Driver Assistance Systems (ADAS) the precursor of AD, over time. In the very beginning, it supported very basic features with a low-spec processor made out of an old semiconductor technology (180 nm) even at the time. The more advanced features are added, the more ADAS processor look like COTS in terms of both power and performance. Still, ADAS hasn't reached the full autonomy (level 5), so this trend is expected to continue. Tesla, another leading AD company, follows the same trend but more aggressively. Their vehicle is known to use vision-based perception, which performs better on GPUs than multicore processors. An NVIDIA GPU was added in their second generation of product (HW2) and they claimed that it is sufficient for full self-driving. However, as more features are added, such as stop sign recognition, the hardware upgrade was unavoidable. In HW2.5, a second GPU was added, consuming 57W of power. Recently, they announced HW3 adding two custom-designed neural processing unit (NPU) to support added cameras and even more features such as seeing traffic cones. With NPU, Tesla claims that the frames-per-second (FPS) performance increased by 21x. Their newest computer system reportedly consumes 72W of power, which is worrisome for battery-operated systems. As can be seen, the computational demand for AD is ever-increasing with new features constantly added. The adoption of the latest COTS hardware and software

makes the gap between typical operation and worst-case scenarios continues to grow. This leads to a severe system underutilization or even impossible to meet performance requirements with target hardware[51]. To minimize the gap, there have been endeavors to enable different operation modes, instead of the rigid monolith system approach.

Vestal et al.[77] proposed a pioneering approach wherein a mixed-criticality system (MCS) was introduced, incorporating varying degrees of assurance. This approach was designed to mitigate the increased development cost associated with extensive timing analyses arising from increasing hardware complexity. In Vestal's model, tasks are classified based on their criticality. Low-criticality tasks require lower level of assurance, which cost less. In a dangerous situation, such systems elevate the criticality and drop low-criticality tasks to ensure the timing correctness of high-criticality tasks. It is an early effort of enabling different *operation modes* in the safety-critical system, i.e. *system reconfiguration*. Inspired by the MCS, this dissertation introduces two system reconfiguration methods with AD as a primary example. One method targets energy efficiency, while the other concentrates on enhancing reliability of timing safety in the face of long lifespan of autonomous machines.

Chapter 2

Background and Related Work

2.1 System and Task Model

2.1.1 Uniprocessor system model

This section describes the uniprocessor system model used in Chapter 3 and 4. Our model assumes a computing system with a single CPU supporting DVFS with a speed factor s in the range of

$$0 < s_{min} \leq s \leq 1, \tag{2.1}$$

where s_{min} denotes the minimum speed factor used for the CPU idle time while 1 (i.e., 100%) indicates the maximum processing speed. The uniprocessor model assumes that s can be any continuous real value in that range such that general optimization techniques can be applied to. The practical consideration with commercial off-the-shelf (COTS) CPUs supporting only

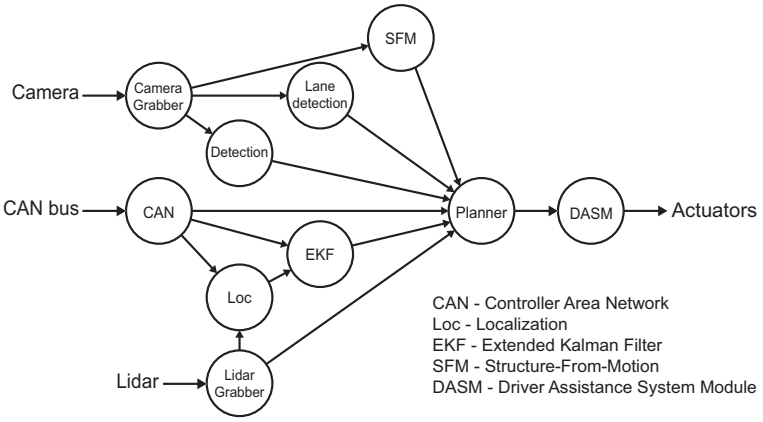


Figure 2.1: A DAG from Bosch as a reference autonomous driving system in the WATERS industrial challenge 2019 [47]

a number of discrete frequency levels will be discussed in multicore model in Section 2.1.2.

The system executes a set of n implicit-deadline periodic tasks

$$V = \{\tau_1, \tau_2, \cdots, \tau_n\}, \tag{2.2}$$

where their read-write dependencies are represented by a DAG

$$G = (V, E \in V \times V) \tag{2.3}$$

with its nodes V and directed edges E representing the task set and dependencies, respectively. Then, each task τ_i is characterized by its period p_i and worst-case execution time (WCET) e_i assuming s=1. Fig. 2.1 shows an example DAG with ten tasks with complex dependencies beginning from sensors to actuators. Tasks communicate with each other with asynchronous message passing. Due to the multi-rate nature, oversampling or undersampling can happen to communication buffers, where newly arrived data always overwrite existing ones. This task model has been commonly used in many studies for automotive systems [40, 35].

When dealing with speed factors, we use the inter-task DVFS method, where s can only be changed at context switching between tasks [11]. Then, each task τ_i in the runtime is characterized by

$$\tau_i = (p_i, e_i, s_i), \tag{2.4}$$

where s_i is the per-task speed factor at the moment. Among them, only e_i is a given value, whereas p_i and s_i are design variables. Thus, a complete *system configuration* π can be described by a vector of tuples

$$\pi = ((p_1, s_1), \cdots, (p_n, s_n)). \tag{2.5}$$

We assume the earliest deadline first (EDF) scheduling algorithm such that the L&L utilization bound [61] can be used to test the system schedulability as

$$U(\pi) = \sum_{i=1}^{n} \frac{e_i}{p_i s_i} \leq 100\%, \tag{2.6}$$

which has s_i in the denominator reflecting the effective WCET that is inverse-linearly proportional to s_i. Based on the above schedulability test, we can guarantee every task's *periodicity*.

2.1.2 Multicore System Model

This section outlines the multicore system used in Section 3, which is an extension of the uniprocessor model incorporating gang scheduling. The multicore system is equipped with m homogeneous cores and Voltage-Frequency Island (VFI) architecture. The CPU clock

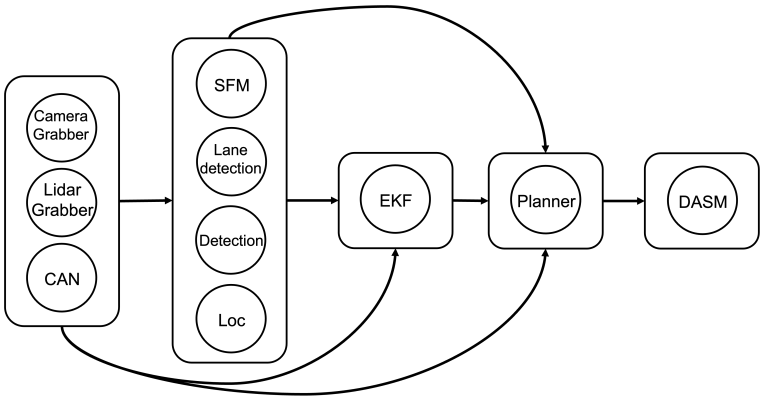


Figure 2.2: Example gangs and their read-write dependencies

frequency is shared across the cores, which is expressed as a speed factor S in the range of

$$0 < S_{min} \leq S \leq 1, \tag{2.7}$$

similar to the uniprocessor model. The system executes a set of n implicit-deadline periodic gangs of tasks

$$V = \{w_1, w_2, \cdots, w_n\}, \tag{2.8}$$

where each gang w_i contains m tasks, where m is at least one. Every gang has one or more dedicated CPU cores to avoid scheduling inside gangs [7]. The read-write dependencies between gangs are inherited from the gang members. For instance, Fig. 2.1 shows a DAG with ten tasks from sensors to actuators, where its nodes and directed edges represent the task set and dependencies, respectively. Then, Fig. 2.2 shows an example of five gangs and their inherited dependencies.

Depending on how tasks are grouped into gangs, so-called a gang formation, the end-to-end latency may vary because the dependency between gangs also changes. Finding a gang formation with the minimum end-to-end latency is known to be NP-hard and will be discussed

in later sections.

Given a gang formation, each gang w_i is characterized by its period P_i, per-gang speed factor S_i, and worst-case execution time (WCET) function E_i:

$$w_i = (P_i, S_i, E_i), \tag{2.9}$$

where a gang of tasks shares the same P_i and maintains S_i for E_i period of time. The speed of processor can only be changed at context switching between gangs to avoid energy and timing overhead of excessive frequency transitions [11]. This inter-task DVFS is typically known to have negligible latency overhead due to the small number of switchings. Therefore, we do not consider DVFS overhead in this work. For E_i, it takes an input S_i and outputs the maximum WCET among gang members at a given speed factor:

$$E_i(S_i) = \max\left(e_x^{mem} + \frac{e_x^{comp}}{S_i}\right), \forall x \in w_i, \tag{2.10}$$

where e_x^{mem} is a speed-independent portion of WCET of gang member x, and e_x^{comp} is the portion that scales inverse-linearly with S_i. Many studies assume the entire WCET to be inverse-linearly proportional to S_i [17] since it provides a safe upper bound. However, it can be largely inaccurate[12] because memory and I/O operations are less affected by S_i. Thus, we use the task model from [9] which considers speed-independent components into account. For notational convenience, we define *speed-independent ratio* r_i as below for discussions in later sections.

$$r_i = e^{mem}/(e^{comp} + e^{mem}) \tag{2.11}$$

Among gang parameters in Eq. (2.9), only P_i and S_i are design variables of our optimization

problem. Thus, we define *multicore system configuration* Π as

$$\Pi = ((P_1, S_1), \cdots, (P_n, S_n)). \tag{2.12}$$

which is a vector of tuples of each gang's period and speed factor. There could exist multiple system configurations under various deadline constraints, which will be further discussed in the next section.

Finally, we use the EDF scheduling with the one-gang-at-a-time policy [7] such that the scheduling unit is a gang instead of a task. That is, a DAG of tasks is transformed into another graph where each node is a gang by using gang formation algorithm. Then each gang is assigned its period and scheduled dynamically based on their implicit deadline with a preemptive EDF scheduler, occupying the entire multicore processor at a time. The L&L utilization bound [61] can be used to test the system schedulability whether all the gangs can be scheduled without violating their respective implicit deadline (i.e., before the next period starts). With the gang WCET model above, our L&L utilization bound is as follows:

$$U(\Pi) = \sum_{i=1}^{n} \frac{E_i(S_i)}{P_i} \leq 100\%, \tag{2.13}$$

where the sum of gang utilizations (the gang WCET over period) should not exceed the upper bound 100%. The upper bound can vary depending on the scheduler used.

2.2 Gang Scheduling

In Chapter 3, we utilize gang scheduling to implement our uniprocessor approach for energy-efficient system reconfiguration. Traditional gang scheduling was proposed to minimize the

makespan of parallel tasks by reducing synchronization overheads through scheduling interacting threads together as a gang [41]. Then, it was picked up by the real-time community to support parallel tasks because of increasing computing demands and the need for parallel processing [4, 76, 28, 38, 43]. These early works on real-time gang scheduling focus on schedulability improvement such that more tasks can be accepted with minimal deadline miss. In practice, however, predictability becomes an issue due to inter-core interference between co-running tasks that is difficult to predict. Recently, to mitigate the interference issue, RT-Gang scheduling was proposed [7] for periodic real-time tasks. RT-Gang is a more restrictive form of gang scheduling because of its one-gang-at-a-time policy restricting the co-scheduling of gangs to prevent interference. Thus, RT-Gang scheduling can make the interference manageable which is suitable for real-time systems where predictability is important. Even though RT-Gang considers task precedence and successfully minimizes the end-to-end deadline, it does not consider energy consumption or system reconfiguration such as DVFS. Our work combines gang scheduling and adaptive system reconfiguration for energy efficiency.

2.3 Power Model

2.3.1 Uniprocessor Power Model

We use the following popular power model from [11, 16]:

$$P(s) = P_s + P_d(s) = \beta + \alpha s^{\gamma}, \tag{2.14}$$

where P_s is the static power and $P_d(s)$ is the dynamic power parameterized by a speed factor s. The static power is expressed as β independent of other parameters, whereas the dynamic power depends on s while α and $\gamma \in [2, 3]$ are CPU-dependent parameters. In this work, we do not use CPU sleep states to reduce the static power, so our focus is to minimize the average dynamic power by minimizing s as long as satisfying the dynamic deadlines, although we also include and compare static power in the evaluation.

For that, the average dynamic power can be calculated as follows: For each task τ_i, its instantaneous dynamic power αs_i^γ is maintained during task's effective execution time e_i/s_i affected by the speed factor. Since the power pattern repeats by its period p_i, the average power consumed in a unit time while executing τ_i can be calculated as

$$P_i(p_i, s_i) = \frac{\alpha s_i^\gamma \times \frac{e_i}{s_i}}{p_i} = \frac{\alpha {s_i}^{\gamma-1} e_i}{p_i}, \tag{2.15}$$

which is a function of p_i and s_i. Then, by summing up n such average powers with the static power and the idle-time CPU power at s_{min}, the total average power for the system is given as a function of a system configuration π as

$$P(\pi) = \beta + \alpha \sum_{i=1}^{n} \frac{s_i^{\gamma-1} e_i}{p_i} + \alpha s_{min}^\gamma \left(1 - \sum_{i=1}^{n} \frac{e_i}{p_i s_i}\right). \tag{2.16}$$

2.3.2 Multicore Power Model

We use the uniprocessor power model Eq. (2.14) as a base for multicore power model. The average dynamic power for multicore system with gang scheduling can be calculated as follows: for each gang w_i, its instantaneous dynamic power αS_i^γ is maintained during gang WCET (Eq. 2.10). Since the power pattern repeats by its period P_i, the average power

consumed in a unit time while executing w_i can be calculated as

$$\mathcal{P}_i(P_i, S_i) = \alpha S_i^{\gamma} \times \frac{E_i(S_i)}{P_i} \tag{2.17}$$

which is a function of P_i and S_i. Then, by summing up n such average powers and the idle-time CPU power at S_{min}, the average dynamic power of core k is given as a function of a system configuration Π as

$$\mathcal{P}^k(\Pi) = \alpha \sum_{i=1}^{n} S_i^{\gamma} \frac{E_i(S_i)}{P_i} + \alpha S_{min}^{\gamma} \left(1 - \sum_{i=1}^{n} \frac{E_i(S_i)}{P_i}\right). \tag{2.18}$$

For the homogeneous multicore processors, the total average power consumption can be described as the sum of individual core's power consumption as in [15]

$$\mathcal{P}(\Pi) = \sum_{k}^{m} \mathcal{P}^k(\Pi), \tag{2.19}$$

where m is the number of cores in the homogeneous multicore processor.

2.4 Dynamic Deadlines

Recent studies [59, 49] presented motivating examples of dynamic deadlines in autonomous driving, where object detection systems are commonly proposed that adapt themselves to varying deadlines, demonstrating the unique potential of autonomous driving systems. More specifically, Lee and Nirjon [59] support dynamic deadlines with selective subgraph executions by considering varying time budgets. Heo et al. [49] support dynamic deadlines by selectively executing multiple forward propagation paths of a neural network with different execution times. Both studies trade dynamic deadlines (or slacks) for improving object de-

tection accuracy. However, little work has been done with dynamic deadlines for the energy optimization of autonomous driving.

In our system model, when referring to deadlines, they always mean the *end-to-end deadlines* from sensors to actuators, not the implicit deadlines that are equal to periods. Thus, even when the system is schedulable, satisfying every gang's P_i, it does not mean deadlines will be guaranteed. To formally define our notion of deadlines, let us assume n_s *sensor tasks* (i.e., source nodes) and n_a *actuator tasks* (i.e., sink nodes) in G. Then we say there are $n_s \times n_a$ unique *flows*, each of which has at least one *path* that is a sequence of adjacent tasks fully connecting a flow. The set of paths in G is denoted by

$$\mathbb{P} = \{\delta_1, \delta_2, \cdots, \delta_{|\mathbb{P}|}\}, \tag{2.20}$$

where each path δ_i denotes an ordered set of task (or gang) *indices* following the path. For example, in Fig. 2.1, the original DAG has three (3×1) flows and eight paths. Similarly, for the corresponding gang formation in Fig. 2.2, there exists a linear ordering of gangs that preserves task dependency for each path δ from the original DAG. Then deadlines are imposed upon the paths such that newly arrived sensor data at time t_1 propagates through the DAG until it first gets out of an actuator task at time t_2 within a given deadline d (i.e., $t_2 - t_1 \leq d$). In the automotive industry, the above notion is commonly referred to as *reaction time constraints* [1].

Fig. 2.3 shows continuous dynamic deadlines as the vehicle velocity changes, where vertical dashed lines depict discrete sensor arrivals. At each k-th sensor data arrival at time $t[k]$, its dynamic deadline $d[k]$ is decided as a red point by the vehicle velocity $v[k]$ with a given mapping function (e.g., $d(v)$ in Eq. (3.1)). Thus, each sensor data arrival can be denoted by $(t[k], d[k])$ for $k \geq 1$. Although many variables representing other physical states can be

considered, this study focuses on the velocity-dependent deadlines as an initial effort toward a more general framework.

To efficiently manage dynamic deadlines, we employ a multi-mode approach, where a feasible deadline range is partitioned into m discrete *modes*, where each mode guarantees the shortest deadline within its deadline range. For notational convenience, the modes are denoted by the per-mode shortest deadlines

$$\{d^1, d^2, \cdots, d^m\}. \tag{2.21}$$

In Fig. 2.3, its deadline domain is partitioned into six equal-length modes, and at each sensor data arrival, the system mode is decided, possibly triggering mode changes. While the system is in a particular mode, the mode's shortest deadline is guaranteed, as depicted by the thick blue line.

The mode's shortest deadline is the minimum *end-to-end latency* from sensors to actuators and is formally defined as follows. We borrow a widely used model from [35], which expresses the worst-case delay of a path δ as an accumulation of periods (P_is) and worst-case response times (WCRTs) of every task in δ, like the rightmost part in

$$D_\delta(\Pi) = \sum_{i \in \delta} 2P_i \approx \sum_{i \in \delta} (P_i + WCRT_i), \tag{2.22}$$

where $D_\delta(\Pi)$ denotes the approximated worst-case delay of a path δ assuming a system configuration Π. We approximate the original delay model to a linear form $\sum_{i \in \delta} 2P_i$ for simplicity. Among the per-gang delay components $2P_i$, one P_i is for the waiting time until the task reads the sensor data (*waiting delay*), and the other is for processing the data (*processing delay).* Then, given tasks and gang formation, the *end-to-end latency* is defined

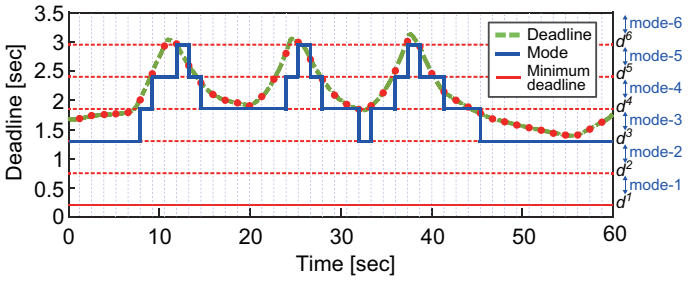


Figure 2.3: Dynamic deadlines with discrete mode changes

as the longest D_δ among paths in $\mathbb{P}$.

2.5 Probabilistic WCET

pWCET is part of measurement-based probabilistic timing analysis (MBPTA)[31], which is gaining widespread adoption due to the limited scalability of static analyses. pWCET is commonly described as a Generalized Extreme Value (GEV) statistical distribution, which is well-studied to capture worst-case behaviors. GEV has three parameters: shape, location (mean), scale. These parameters can be estimated after the block-maxima (BM) method, which filters the maximum value out of every n-sample window. Depending on *shape* value, GEV can exhibit three types of distributions: Gumbel (light tail), Weibull (cut off), or Frechet (heavy tail). Then, pWCET is obtained by extracting a value from the cumulative density function (CDF) of these distributions at a certain probability. In the rest of the dissertation we use the terms WCET and pWCET interchangeably.

2.6 Boosting

Boosting is a machine learning technique that combines the predictions of multiple weak classifiers with weights to create a strong predictive model. Iteratively, the boosting algorithm

adjusts the weights, emphasizing misclassified samples to achieve improved accuracy. Extereme gradient boosting (XGBoost)[26] is a state-of-the-art boosting technique, renowned for its unparalleled speed and performance. In Chapter 4, we aim to obtain a more accurate detector by considering each i.i.d test as a weak classifier and combining them with XGBoost. The classifiers are often evaluated using *F1 score*, a comprehensive metric that involves both false positives (FP) and false negatives (FN):

$$F_1 = \frac{2TP}{2TP + FP + FN} \tag{2.23}$$

where TP represents true positives. The highest possible F1 score is 1.0, which cannot be achieved in the presence of FP and FN.

2.7 Energy Efficient Real-time Systems

There have been many efforts to develop energy-efficient real-time systems, most of which, however, assume only static deadlines. Broadly, there are two frequently used energy saving approaches in hard real-time systems: DVFS and *dynamic power management* (DPM). In DVFS approaches, there is a body of literature to find speed levels through offline optimization [32, 11, 46, 72]. These approaches try to find critical speed factors under a static deadline constraint, which is the lowest frequency satisfying the given static timing constraint. In contrast, our method finds the critical speed for each deadline through multiple modes with different timing constraints and employs a safe mode change protocol to freely go back and forth between them. Another body of real-time DVFS approaches tries to reclaim *dynamic slacks*[56, 13], which is not to be confused with our dynamic deadlines; instead, they define dynamic slacks as the difference between the worst-case and the actual execution times.

Note that dynamic slack reclaiming does not conflict with our approach and could be used together for further energy reductions.

In DPM approaches, cores are switched off during idle periods to reduce energy consumption. It is useful when the system is underutilized. However, each idle period should be large enough to offset the energy overhead of frequent switching on and offs. Many scheduling methods have been proposed to create large idle periods [82, 52, 8, 75]. Lee et al.[82] proposed leakage control EDF scheduling, but it requires additional hardware to calculate the sleep period. This impractical assumption was avoided in [8] by using a simpler sleep period calculation method and the enhanced race-to-halt (ERTH) algorithm. With ERTH, tasks run at full speed to secure longer sleep time. DPM and DVFS approaches are not orthogonal and can be used in conjunction. However, satisfying the timing constraints during mode changes with DPM approaches requires non-trivial considerations. Thus, we leave integrating DPM for our future work.

2.8 Fault-Tolerant Real-time Systems

The realm of fault-tolerant systems encompasses a comprehensive body of literature focused on software-implemented hardware fault tolerance (SIHFT) [2]. Fault tolerance typically includes both fault detection and recovery. There exists an extensive range of literature in the field of fault tolerance, which varies based on fault models and the level in which fault has occurred. Traditionally in real-time systems, fault tolerance has been studied from a functionality perspective. Recent research has attempted to include the effect of timing as well. Especially, a specific type of fault is carefully analyzed and their effect on execution times are assessed during the design phase. Nikiema et al. [67] propose a low-level fault-tolerant hardware technique that recovers from various transient faults with near-zero

overhead. Hardy et al. [48] incorporate fault detection and recovery together in the pWCET estimation, focusing on faulty cache blocks. Chen et al. [25, 24] propose a static probabilistic timing analysis (SPTA) approach to estimate an increased pWCET for random caches in the presence of both transient and permanent faults. The same authors show that different online fault detection technique can greatly affect pWCET estimations, over 200% [23]. Huang et al. [51] emphasize that perfect fault detection with a 100% coverage incurs a significant overhead and may not be a realistic design decision, under some reliability requirements. While these approaches are effective in tolerating transient and permanent faults, they often result in inefficiencies during normal operation due to their inherently pessimistic design.

Another branch of fault tolerance is control flow checking (CFC). Some fault injection studies have shown that control flow errors can take up to 77% of computer system errors [73]. These methods typically utilize a hardware unit to reduce the significant overhead associated with monitoring instructions. Wolf et al. [79] devised a hardware-software hybrid CFC technique with the latency of only 22.1 processor cycles, which can detect over 65% of injected faults uncaught by processor exceptions. Zhang et al. [86] propose a low overhead CFC that begins with the WCET with full CFC and gradually reduce the WCET by carefully selecting code blocks to monitor, while achieving a sufficient fault detection coverage. However, these methods lack the ability to capture the inherent statistical nature of execution times and can only detect timing violations after they have already occurred.

In mixed-criticality systems, a key research focus is on efficiently allocating a budget within the high-criticality (HC) mode, which may involve dropping or missing deadlines for low-criticality (LC) tasks. Burns et al. [21] introduced a quantitative concept of robustness, a job failure (JF) count, which establishes a threshold for transitioning between different fault tolerance modes such as fail operational, fail robust, and so on. The JF count corresponds to the number of HC jobs that overrun. In their study, they analyze the system schedulability

under a fixed-priority scheduler for each fault-tolerance mode, while tasks have varying levels of degradation. Baruah et al. [14] proposed an algorithm called EDF-Virtual Deadline (EDF-VD), which is the state-of-the-art scheduler for mixed-criticality uniprocessor system. EDF-VD employs the strategy of assigning shorter deadlines to HC tasks. This ensures that these tasks have sufficient time to complete their execution before their original deadline in the event of a change in operation mode. Gu and Easwaran [45] enhanced EDF-VD with EDF-UVD (Universal VD), which extends the concept of assigning virtual deadlines even to LC tasks. With EDF-UVD, the dynamic budget generated by LC tasks is collectively managed for HC tasks. This approach minimizes the allocation of unnecessary budgets as opposed to EDF-VD. As a result, EDF-UVD showcases increased fault tolerance capability and prolonged postponement of mode-switches compared to fixed budget allocation methods. Huang et al. [53] integrate EDF-VD with DVFS technique to achieve energy efficiency in mixed-criticality systems. During the design phase, they determine the optimal combination of the normal CPU frequency and a deadline shortening factor x. In situations of task overruns, the system operates at the maximum CPU frequency. These approaches primarily address fault tolerance related to implicit deadline violations rather than considering the comprehensive end-to-end deadline of the entire system, which involves the sensor-decision-actuation loop.

Chapter 3

Safe Adaptive System Reconfiguration for Energy Efficiency

3.1 Motivation and Challenge

Due to the enormous amount of computing required for autonomous vehicles, their inherent high energy consumption has become one of the major hurdles when designing such computing systems. For example, Gawron et al. [42] transformed a commercial electric vehicle (EV) into a connected and automated vehicle (CAV) by installing sensors such as camera, radar, and LiDAR in addition to computers and a short-range communication device to evaluate the energy consumption and greenhouse gas emission of the future CAVs. According to their report, the computer system is taking 80% of the added power consumption. The computer system of CAV composed of multiple central processing units (CPUs) and graphics processing units (GPUs) reportedly consumes more than 2 kW of energy, reducing an EV's driving range up to 12% [60]. The current energy consumption of EVs is around 200 Wh/km on

average of total 245 cars [34]. That is 10kW energy consumption at the speed of 50 km/hour (about 30 mph). Therefore, even though the current CAV has not reached full automation, its computer system can consume up to 20 % of the vehicle's power consumption. As autonomous driving technologies evolve, more sensors will be added, requiring more processing and computing power, so the portion of computing systems in energy consumption is expected to increase. In addition to that, according to a recent article regarding vehicle computing [63], the United States national energy consumption of EVs is approximately equal to the total energy consumption of 15 representative technology companies' data centers, each of which can consume 12 terawatt-hours, taking the significant portion of the overall energy consumption. Since the volume of EV energy consumption is huge, even a small improvement would be meaningful in terms of global carbon emission reduction. With this challenge, some energy optimization methods have been proposed that simultaneously try to satisfy the stringent real-time requirements of automotive systems [57, 81]. However, since they commonly assume just (fixed) *static deadlines*, they do not reflect recent autonomous driving applications with time-varying *dynamic deadlines*. Such applications include the localization system with its dynamic latency constraint as a function of velocity [71] and the truck platooning system where its control response times are adjustable to varying driving conditions [37].

With the above motivation, we aim to develop an energy-efficient software optimization method and a runtime framework that can specifically exploit the dynamic nature of deadlines found in many autonomous driving applications. This study focuses on minimizing CPU energy consumption using dynamic voltage and frequency scaling (DVFS). Although conventional automotive microcontrollers often lack such features, recent application processors for autonomous driving mostly support DVFS.

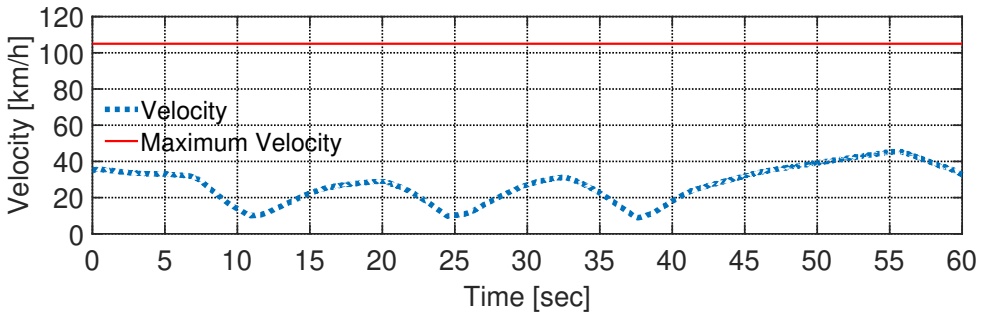
To demonstrate our basic idea, Fig. 3.1 shows an example time history of a vehicle's velocity

and its corresponding dynamic deadline, assuming a velocity-deadline mapping function

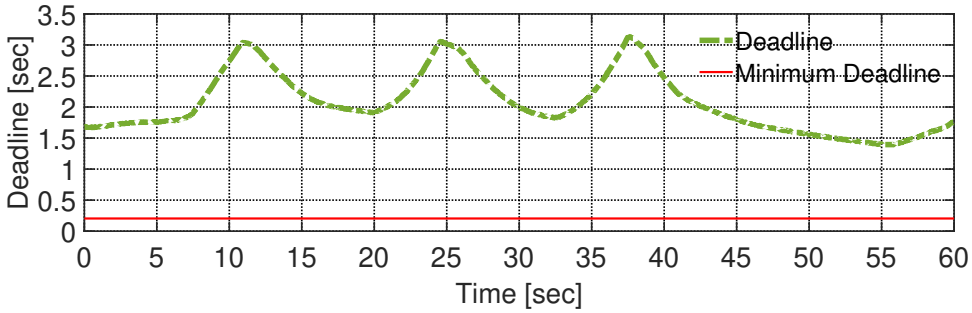
$$d(v) = \frac{-v + \sqrt{v^2 + 2\lambda a^{max}}}{a^{max}}, \tag{3.1}$$

which represents the minimum time for a vehicle at its initial velocity v to advance a fixed distance λ assuming its maximum acceleration a^{max}. It is especially useful in truck platooning where trucks maintain a fixed longitudinal gap between them across various driving velocities [36] such that safe control decisions can be made more efficiently in terms of a maximum travel distance (i.e., λ) between sensing and actuation rather than by a rigid timing constraint. Hence, in the figure, the faster the vehicle runs, the shorter the deadline gets. The minimum deadline depicted by a red line is determined by the maximum velocity enforced by traffic regulations. Here, our basic idea is to trade the area between the time-varying dynamic deadline and the minimum deadline to reduce energy consumption by adaptively slowing down the CPU to the extent that guarantees the dynamic deadline, rather than adhering to the static deadline [57, 81] as in the traditional energy optimization methods.

In the automotive industry, complex control applications composed of multiple independent real-time tasks are commonly modeled with directed acyclic graphs (DAGs) whose nodes are periodic tasks, and edges are read-write dependencies between tasks. Fig. 2.1 shows an example DAG from Bosch [47] for their reference autonomous driving system from sensors to actuators. Upon such a DAG, its dynamic deadlines are imposed by its worst-case end-to-end latency from sources to sinks, which is collectively decided by the periods of individual tasks. Then, our objective is to minimize the average power consumption while guaranteeing such dynamic deadlines. When doing that, we must satisfy (i) the system schedulability constraint (i.e., task periods) as well as (ii) the end-to-end deadline constraint. At first, we solve the problem by assuming a static deadline constraint. For that, we formulate it as a geometric



(a) Example time history of vehicle velocity [29]



(b) Corresponding dynamic deadlines (λ=20 m and a^{max}=2.5 ms^{-2})

Figure 3.1: Example dynamic deadlines

programming (GP) problem [19], which is a special form of non-convex optimization that can be efficiently solved by a transformation into a convex problem.

To extend the above optimization method to time-varying dynamic deadlines, we partition the feasible deadline range into a number of discrete *modes*, where the system is separately optimized for each mode, assuming each mode's *shortest* deadline, respectively. Then, our runtime framework provides a safe mode change protocol that changes each task's period when the vehicle slows down or speeds up crossing across different modes. Our mode change protocol is designed not to miss any deadline if the mode change is from a shorter deadline to a longer one. However, we found that extra delays are unavoidable in the opposite direction (i.e., longer to shorter deadlines). Even in that case, however, we provide a mode change delay analysis method from which we can reserve enough safety margins to hide away the extra delays while guaranteeing safety.

The basic formulation for uniprocessors was done in our previous work [83] and will be introduced in Section 3.3.1. In Section 3.3.2, we achieve the same goal in multicore processors with VFI architecture, where all the CPU cores in the same island share the same voltage-frequency level. VFI is most common in commercial processors because the complex design for supporting per-core DVFS cannot justify the potential energy gain [50].

Transitioning from uniprocessor to multicore introduces several challenges: inter-core interference and multicore real-time schedulability. In addition to the NP-hard task-to-core assignment, the inter-core inferences and complex architecture of multicore processors make timing analysis notoriously difficult. Because of that, it led to strict timing constraints based on a highly pessimistic estimation of worst-case execution time (WCET) resulting in less number of schedulable tasks and low system utilization. Recently, Real-time Gang (RT-Gang) scheduling was proposed to address the inter-core interference and schedulability issues. In RT-Gang, the tasks are grouped into a gang, enforcing a one-gang-at-a-time policy, which makes the interference smaller and manageable since only a predefined set (i.e. a gang) of tasks can occupy the processor at a time.

Moreover, it enables the use of well-studied unicore timing analysis and real-time schedulers to increase system schedulability. Even though RT-Gang's strict policies hinder exploiting the full capacity of hardware, we choose RT-Gang as it improves predictability and schedulability which are crucial in a real-time systems context.

However, it entails another problem: gang formation. It can be described as partitioning a DAG of tasks into multiple gangs, which decides the minimum end-to-end delay of a given DAG. Minimizing end-to-end delay is important because the shorter the delay, the more room we have for slowing down the processor until the end-to-end delay meets the deadline. Gang formation is NP-hard combinatorial problem, and thus we developed a greedy heuristic

to handle it.

3.2 EASYR Framework Overview

Given a DAG G of n periodic tasks and m discrete deadline constraints, our problem is decomposed into three parts: (i) gang formation, (ii) period and speed factor optimization, and (iii) safe mode change protocol. First, we obtain a gang formation $W = \{w_1, w_2, ..., w_k\}$ that can minimize the end-to-end latency of the given DAG considering task WCET and dependency. Second, given a gang formation, we find the gang periods and speed factors that minimize average power consumption while satisfying the end-to-end deadline constraint for each mode. The solution can be described by following two matrices:

$$\begin{pmatrix} P_1^1 & P_2^1 & \cdots & P_k^1 \\ \vdots & \vdots & \ddots & \vdots \\ P_1^m & P_2^m & \cdots & P_k^m \end{pmatrix} \text{ and } \begin{pmatrix} S_1^1 & S_2^1 & \cdots & S_k^1 \\ \vdots & \vdots & \ddots & \vdots \\ S_1^m & S_2^m & \cdots & S_k^m \end{pmatrix}, \tag{3.2}$$

where P_i^j and S_i^j represent gang i's optimal period and speed factor at the j-th mode, respectively. In other words, each row represents mode j with system configuration Π_j that guarantees the end-to-end latency shorter than d_j in Eq. (2.21). For uniprocessor solution, it is as straightforward as replacing P_i^j and S_i^j with p_i^j and s_i^j from the above two matrices. Finally, the solution should satisfy safe mode changes such that the system can freely go back and forth between modes without violating the dynamic deadline requirements. For that, a safe runtime mode change protocol is proposed in Section 3.5.

3.3 Period and Speed Factor Optimization

This section formulates and solves the offline multi-mode system optimization problem for both uniprocessors and multicore processors (assuming gang formation is given and fixed). We begin by finding the optimal configuration for a single mode for a uniprocessor without considering safety constraints in the transient phase between the modes. Then, we extend the optimization method such that it can go back and forth between the modes without violating safety constraints (Section 3.3.1). Next, multicore formulation is introduced in a similar way, followed by the optimization solver we used (Section 3.3.3). Finally, a realistic condition of discrete CPU frequencies is discussed (Section 3.3.4).

3.3.1 Uniprocessor Formulation

This section explains how we can formulate a single-mode optimization for a uniprocessor as a baseline for the multi-mode system optimization. As the objective function, the average power in Eq. (2.16) is used, without the constants α and β, where π denotes two sets of decision variables p_is and s_is with constrained domains as $p_i > 0$ and $s_{min} \leq s_i \leq 1$, respectively.

Then, we have two explicit constraints: (i) schedulability constraint as already discussed in Eq. (2.6) and (ii) deadline constraint, for which we need to devise an end-to-end delay analysis model. We borrow a widely used model from [35], which expresses the worst-case delay of a path δ as an accumulation of periods (p_is) and worst-case response times (WCRTs) (r_is) of every task in δ, like the rightmost part in

$$D_\delta(\pi) = \sum_{i \in \delta} 2p_i \approx \sum_{i \in \delta} (p_i + r_i), \tag{3.3}$$

where $D_\delta(\pi)$ denotes the approximated worst-case delay of a path δ assuming a system configuration π. We approximate the original delay model to a linear form $\sum_{i\in\delta} 2p_i$ as calculating the exact r_i involves complex non-convex operations. Then, the deadline constraint d is considered for every path δ in $\mathbb{P}$ (i.e., the set of paths in G). Among the per-task delay components $2p_i$, one p_i is for the waiting time until the task reads the sensor data (*waiting delay*), and the other is for processing the data (*processing delay).*

Then our single-mode formulation is given as follows:

$$\underset{\pi}{\text{minimize}} \quad \mathcal{P}(\pi) = \sum_{i=1}^{n} \frac{s_i^{\gamma-1} e_i}{p_i} + s_{min}^{\gamma}(1 - \sum_{i=1}^{n} \frac{e_i}{p_i s_i}) \tag{3.4}$$

$$\text{subject to} \quad U(\pi) = \sum_{i=1}^{n} \frac{e_i}{p_i s_i} \leq 1 \tag{3.5}$$

$$D_\delta(\pi) = \sum_{i\in\delta} 2p_i \leq d \quad (\forall \delta \in \mathbb{P}). \tag{3.6}$$

For a multi-mode formulation, naively, the single-mode one can be repeatedly used to find every row of the multi-mode solution matrices in Eq. (3.2). However, we cannot directly use this approach for a multi-mode system since it does not guarantee a safe transition between modes. Specifically, when old- and new-mode tasks coexist during a mode change, schedulability violations can occur even if each mode is schedulable in isolation [27]. Thus, we add a new constraint called *per-task utilization invariability*, meaning every task's utilization $u_i = e_i/(p_i s_i)$ is invariant across modes as

$$\frac{e_i}{p_i^1 s_i^1} = \frac{e_i}{p_i^2 s_i^2} = \cdots = \frac{e_i}{p_i^m s_i^m} = u_i^* \ (\forall i = 1, 2, ..., n), \tag{3.7}$$

where u_i^* denotes identical utilization for τ_i across modes. In that manner, even in the tran-

sient interval, the system's instantaneous utilization is maintained unchanged, which in turn guarantees the system schedulability [3]. Now, we use u_i^*s as our decision variables, replacing s_i^js, which can be decided later by $s_i^j = e_i/(p_i^j u_i^*)$. Then our multi-mode optimization can be formulated as follows:

$$\begin{aligned}
\underset{\hat{\pi}}{\text{minimize}} \quad & \mathcal{P}(\hat{\pi}) = \sum_{j=1}^{m}\sum_{i=1}^{n} \frac{e_i^\gamma}{(p_i^j)^\gamma (u_i^*)^{\gamma-1}} + s_{min}^\gamma (1 - \sum_{i=1}^{n} u_i^*) && (3.8)\\
\text{subject to} \quad & U(\hat{\pi}) = \sum_{i=1}^{n} u_i^* \leq 1, && (3.9)\\
& D_\delta^j(\hat{\pi}) = \sum_{i\in\delta} 2p_i^j \leq d^j \quad (\forall j \in [1,m], \forall \delta \in \mathbb{P}), && (3.10)
\end{aligned}$$

where $\hat{\pi}$ denotes the newly defined multi-mode system configuration with p_i^js and u_i^*s. The objective function is the sum of average power in each mode, after eliminating the α and β from Eq. (2.16) for the notational simplicity. The first constraint is the system schedulability now expressed by u_i^*s. The second constraint is for the dynamic deadlines across m modes.

3.3.2 Multicore Formulation

Thanks to gang scheduling, transitioning from uniprocessor formulation to multicore one is as straightforward as substituting variables e_i, p_i, and s_i with $E_i(S_i)$, P_i, and S_i, respectively. Our single-mode formulation for multicore processors is given as follows:

$$\begin{aligned}
\underset{\Pi}{\text{minimize}} \quad & \mathcal{P}(\Pi) = \sum_{i=1}^{n} \frac{S_i^\gamma E_i(S_i)}{P_i} + S_{min}^\gamma (1 - \sum_{i=1}^{n} \frac{E_i(S_i)}{P_i})\\
\text{subject to} \quad & U(\Pi) = \sum_{i=1}^{n} \frac{E_i(S_i)}{P_i} \leq 1 && (3.11)\\
& D_\delta(\Pi) = \sum_{i\in\delta} 2P_i \leq d \quad (\forall \delta \in \mathbb{P}).
\end{aligned}$$

Then, the *utilization invariability*, meaning every gang's utilization (i.e., $u_i^j = E_i(S_i^j)/p_i^j$) is invariant across modes as

$$\frac{E_i(S_i^1)}{P_i^1} = \frac{E_i(S_i^2)}{P_i^2} = \cdots = \frac{E_i(S_i^m)}{P_i^m} = u_i^* \ (\forall i = 1, 2, ..., k), \tag{3.12}$$

where u_i^* denotes identical utilization for each gang w_i across modes. In that manner, even in the transient interval, the system's instantaneous utilization is maintained unchanged, which in turn guarantees the system schedulability [3]. Now, we use u_i^*s as our decision variable replacing P_i^js with $E_i(S_i^j)/u_i^*$. Then our multi-mode optimization can be formulated as follows:

$$\underset{\hat{\Pi}}{\text{minimize}} \quad \mathcal{P}(\hat{\Pi}) = \sum_{j=1}^{m}\sum_{i=1}^{n}(S_i^j)^\gamma u_i^* + S_{min}^\gamma(m - \sum_{j=1}^{m}\sum_{i=1}^{n} u_i^*) \tag{3.13}$$

$$\text{subject to} \quad U(\hat{\Pi}) = \sum_{i=1}^{n} u_i^* \leq 1, \tag{3.14}$$

$$D_\delta^j(\hat{\Pi}) = \sum_{i\in\delta'} \frac{2E_i(S_i^j)}{u_i^*} \leq d^j \quad (\forall j \in [1, m], \forall \delta' \in \mathbb{P}'), \tag{3.15}$$

where $\hat{\Pi}$ denotes the newly defined multi-mode system configuration with P_i^js and u_i^*s. The objective function is the sum of average power in each mode, after eliminating the α and β from Eq. (2.18) for the notational simplicity. The first constraint is the system schedulability now expressed by u_i^*s. The second constraint is for the dynamic deadlines across m modes.

Note that the average power of a mode from multi-mode solution could be worse than the one from single-mode solution. In order to achieve a mode's optimal average power, each gang should freely choose its period and speed factor. However, if we fix the utilization value across modes, the period and speed factor are not independent anymore but constrained by the relationship $u_i^* = E_i(S_i^j)/p_i^j$. This adverse effect of utilization invariability is further

discussed in the experiment section.

3.3.3 Geometric Programming (GP)

Our multi-mode optimization problem can be efficiently solved by GP, which is a mathematical optimization method for solving specially formed optimization problems through logarithmic transformations into convex ones. As a result, GP always finds the (true, globally) optimal solution when the problem is feasible [19]. To use GP, the objective function and inequality constraints should be constructed by the special form *posynomial*, as in $f(x) = \sum_{k=1}^{K} c_k x_1^{a_{1k}} x_2^{a_{2k}} \cdots x_n^{a_{nk}}$, with decision variables x_is, non-negative coefficients c_ks, and real-valued exponents $\{a_{11}, \cdots, a_{nK}\}$. Our objective functions and constraints are in posynomial forms except for the idle-time CPU power terms in the rightmost part of $\mathcal{P}(\Pi)$ in Eq. (3.11) and $\mathcal{P}(\hat{\Pi})$ in Eq. (3.15) (for uniprocessors, $\mathcal{P}(\pi)$ in Eq. (3.3.1) and $\mathcal{P}(\hat{\pi})$ in (3.8)).

However, the idle power terms can be removed without affecting optimality as long as $\exists S_i \neq S_{min}$. The optimal solutions in such cases always have 100% system utilization without any idle time. To prove it intuitively, assume the system utilization $U < 100\%$, if we pick a certain gang w_i and decrease S_i, thus increasing E_i, until U reaches 100%, the average power of w_i will decrease and the idle power term will disappear, eventually saving more energy than the original configuration. However, when we cannot reduce processor speeds ($\forall i : S_i = S_{min}$), the optimal case would have $U < 100\%$ as P_i increases. Such cases only occur when the deadline is extremely long after all S_is are bounded by S_{min}. We are not considering such extreme cases in this work.

3.3.4 Extension to Discrete CPU Frequency Levels

Our optimization method yields speed factors in the continuous range between S_{min} and 1. However, because most CPUs in practice support only a predefined set of discrete frequency levels, we need to adapt the resulting speed factors to the discrete domain. One possible approach is to emulate the exact speed factors by modulating between two neighboring discrete frequency levels [55, 18] to obtain the near-optimal energy reduction similar to the continuous frequency solution. However, using such intra-task DVFS entangles other practical considerations such as extra time and energy overhead associated with excessive frequency transitions [80] and possible transient faults [85, 33].

In light of this, we propose a more conservative but safer method that uses the closest frequency level that is higher than the corresponding optimal speed factor. Then every task's actual utilization is less than or equal to the ideal utilization in accordance with the utilization invariability constraint. Thus, even though the approximated system will consume more energy than the continuous one, it safely ensures schedulability and end-to-end latency constraints during mode changes.

3.4 Gang Formation Heuristic for Multicore Processors

Gang scheduling in real-time systems brings many benefits such as mitigating inter-core interference, tighter WCET estimation, and increased system utilization so that the system can accept more tasks given end-to-end deadlines. However, it entails another NP-hard integer programming problem: the gang formation. It can be described as grouping tasks

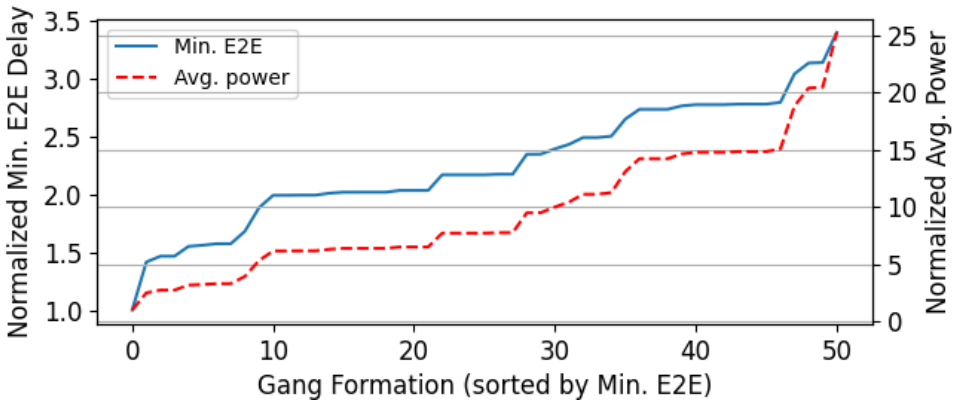


Figure 3.2: Gang formation problem space. A shorter end-to-end latency results in a lower power consumption.

into multiple gangs while minimizing the end-to-end latency in Eq. (2.22). We employ the state of the art gang scheduling [7] that enforces one-gang-at-a-time policy and restricts co-scheduling inside a gang (i.e. the number of tasks in a gang does not exceed the number of cores) for minimizing inter-core interference. Depending on how gangs are formed, they may not fully utilize the multicore processor (e.g. the shorter task creates a core idle period until the longer task finishes), which leads to an increased delay and failure to meet end-to-end deadline. More importantly, in our dynamic deadline context, the minimum end-to-end latency decided by a gang formation is closely related to power saving as the gap between the delay and the deadline provides opportunities to slow down the CPU. Therefore, it is important to create a gang formation that minimizes the makespan and a couple of heuristics have been proposed [5, 6].

Given a DAG of n tasks, their WCETs, and homogeneous m cores, we want to group tasks into an arbitrary number of gangs while minimizing the end-to-end latency defined in Eq. (2.22). The reason for minimizing end-to-end latency here is that it is closely related to minimizing power. In Fig. 3.2, we exhaustively searched gang formation space for an example 5-task DAG. All possible gang formations are sorted by their minimum end-to-end

latency and numbered sequentially over the x-axis. The average power consumption on the right y-axis is calculated using the period optimization in the next section under the same end-to-end deadline for all the gang formations. We discovered that the power consumption increases monotonically as the minimum end-to-end latency increases. This is because the bigger the gap between the minimum end-to-end latency and the deadline, the more room for CPU to slow down, resulting in smaller power consumption.

Intuitively, to get better gang formation, it is desirable to pack gangs as dense as possible to fully utilize m cores, while trying to avoid grouping dependent tasks to the same gang. If tasks in dependency constraints are in the same gang, the gang has to be repeated to propagate data to the consumer task which was not possible in the previous execution because it is synchronized to start with the provider task by gang scheduling. This adds an additional delay to the overall end-to-end latency.

The gang formation proves NP-hard integer programming problem[5] like bin packing. Therefore, the authors from [6] proposed the *Virtual Gang Heuristic (VGH)*. In VGH, a DAG consists of tasks on m homogeneous cores and is partitioned into multiple gangs minimizing the completion time of all the gangs. It introduces the notion of family to avoid grouping dependent tasks. A family of tasks τ_i consists of all tasks that are connected with τ_i's ancestor or descendant. VGH strongly restricts grouping tasks in a family relationship to avoid the repetition of gang execution.

However, VGH does not fit well for our problem because it has a different objective. It minimizes the total completion time of gangs which is equal to the sum of gang WCETs along the paths. On the other hand, our objective is to minimize end-to-end latency considering the worst data propagation pattern which is analyzed in Eq. (2.22). A simple 2-task example is depicted in Fig. 3.3. In the example, tasks A and B are in a dependency constraint. For

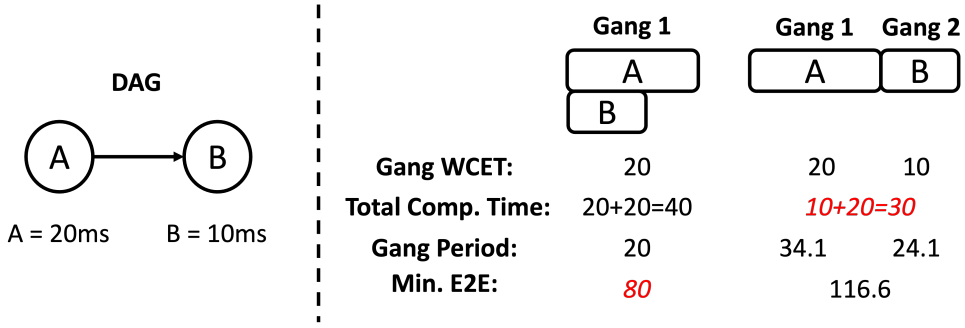


Figure 3.3: Gang formation example with different objectives (total completion time vs. minimum end-to-end latency)

VGH, it makes sense to avoid packing A and B together in the same gang because it would increase the total completion time. On the other hand, the gang periods tend to increase significantly as we add more gangs because of the utilization bound in Eq. (2.13). Basically, the utilization bound implies that the period should be large enough to have time for the execution of other gangs. In this case, A's period (34.1 ms) is significantly greater than its WCET (20 ms) to give task B (10 ms) a chance to execute. Therefore, for the minimum end-to-end latency which is based on gang periods, it is beneficial to pack A and B together despite the dependency relationship.

Moreover, VGH does not consider WCET changes over processor speed, especially with the speed-independent ratio r_i defined in Eq. (2.11). Specifically, the gang WCET is defined as the longest task's WCET in the gang and if its r_i is high the dominant task can be changed to another one as depicted in Fig. 3.4. As the speed factor increases, task 1's WCET shrinks slower than task 2's and task 1 becomes the dominant task in the gang. In worse cases, the gap between the grouped tasks' WCETs becomes big enough to significantly decrease the multicore utilization, which can be deemed as the quality degradation of gang formation over S_i changes. This would not happen without r_i as the tasks will scale at the same ratio. At certain S_i, the gang WCET might change significantly and the gang formation obtained at $S_i = 1$ may not perform well. To address this issue, we try to find *the base speed*

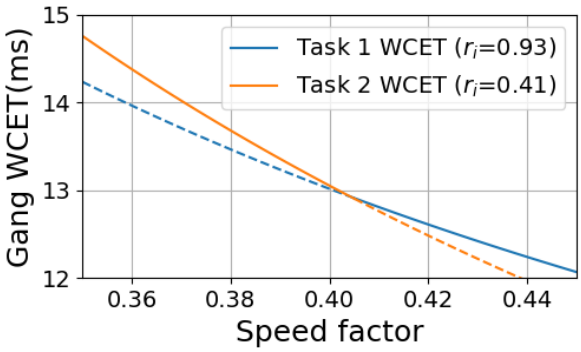


Figure 3.4: An example gang WCET curve with $r_i = e_i^{mem}/e_i$. Solid lines represent the dominant task at the time.

factor (S_{base}) and obtain a gang formation from the scaled task WCETs that minimizes the degradation of the gang formation quality over S_i changes. For example, the gang formation with $S_{base} = 0.5$ might degrade less than the one with $S_{base} = 1.0$ as the possible S_i change is smaller (0.5-s_{min} vs. 1.0-s_{min}).

As a result, VGH can be improved in three ways for our problem. (i) The strict family concept should be alleviated as there are advantageous cases when tasks in dependency constraints are in the same gang as we have seen in the previous example. (ii) An estimation of gang periods must be taken into account as our objective is calculated based on periods, not WCETs. (iii) The gang WCET changes over S_i should be considered as the heuristic solution may perform significantly worse at different S_i.

We developed a greedy heuristic, *Dynamic Deadline Gang Heuristic (DDGH)*, as shown in Algorithm 1. **Line 1** scales the task WCETs with the base speed factor (S_{base}). **Line 2** sorts tasks by its WCET in descending order. Sorting tasks helps to group tasks with similar WCETs, trying to fully utilize all the cores during gang execution. **Line 3** initializes an empty gang formation. As a greedy approach, a single task is picked (**Line 6**) and then assigned to a gang (**Line 12**) at a time. **Lines 7-10** create a temporary gang formation (*new_gangs*) to add the current task to an existing gang for evaluation. The iteration with

Algorithm 1 Dynamic Deadline Gang Heuristic (DDGH)

Require: Task WCETs(e), speed-independent ratio(r), base speed factor(S_{base}), paths in the DAG($\mathbb{P}$)
Ensure: Gang formation ($gangs$)

```
1: new_wcet = scale_wcet_at_sf(e, r, S_base)
2: q = sort_task_by_wcet(new_wcet)
3: gangs = ()
4: while not q.empty() do
5:     candidates = ()
6:     task = q.pop()
7:     for g ∈ {gangs + ∅} do
8:         new_gangs = add_gang_member(gangs, g, task)
9:         delay = calc_delay_proxy(new_gang, e, ℙ)
10:        candidates.push(delay)
11:    end for
12:    best_g = min(candidates)
13:    gangs = add_gang_member(gangs, best_g, task)
14: end while
15: return gangs
```

$g = \emptyset$ means creating a new gang with the current task. Finally, **Line 13** decides the best gang ($best\_g$) for the current task and updates the gang formation ($gangs$). The process is repeated until the queue (q) is empty.

For evaluating temporary gang formations, *calc_delay_proxy* function in Algorithm 2 is used to provide a proxy of end-to-end latency. It is a proxy because it is difficult to estimate the exact end-to-end latency which is the optimization result in the next section. Instead, we exploit the knowledge of optimization solution form to just estimate the relative end-to-end latencies between different gang formations. **Lines 2-3** calculate gang WCETs (E) for a given gang formation. **Lines 5-9** iterate over each path in the DAG and sum up our metric in **Line 8**. The metric is the core part of the heuristic and consists of gang WCET and the sum of gang WCETs. With both terms multiplied, it acts as a proxy for gang period which is a basic component in end-to-end latency. The first term, gang WCET, contributes to minimizing the individual gang's WCET. The period is typically proportional to WCET,

Algorithm 2 Evaluate temporary gang formations

Require: Gang formation($gangs$), task WCETs(e), paths in the DAG($\mathbb{P}$)
Ensure: Proxy of end-to-end latency

```
1: function CALC_DELAY_PROXY(gangs, e, ℙ)
2:     for gang ∈ gangs do
3:         E(gang) = get_gang_wcet(gangs)
4:     end for
5:     max_delay = 0
6:     for δ ∈ ℙ do
7:         max_delay = 0
8:         for task ∈ δ do
9:             delay+ = E(task_to_gang(task)) * sum(E)
10:        end for
11:        if max_delay < delay then
12:            max_delay = delay
13:        end if
14:    end for
15:    return max_delay
16: end function
```

so the smaller the gang WCET, the smaller the period. The second term, the sum of gang WCETs, is another critical part to estimate the period. When a new gang is added, one of the gang periods must be greater than the sum of gang WCETs as there should exist enough time for other gangs to execute. This is implied in the utilization bound in Eq. (2.13) ($U = \sum E_i/P_i < 1$). It also increases other gang's period as we have seen with the example in Fig. 3.3, so the heuristic is discouraged to create a new gang by this term. Therefore, DDGH does not strictly prohibit the grouping of tasks in precedence relationships. Instead, it uses a proxy delay to assign tasks flexibly and discourages creating a new gang, so gangs are packed as densely as possible.

3.5 Safe Mode-change Protocol

For safe system reconfigurations with dynamic deadlines, the followings should be respected even during mode changes:

- **Periodicities.** Every gang period before and after the mode change should be guaranteed, which can be satisfied by the utilization invariability constraint introduced in Section 3.3.
- **Deadlines.** Unfortunately, however, the above periodicities do not guarantee dynamic deadlines, which span across multiple gangs possibly with different modes during a mode change.

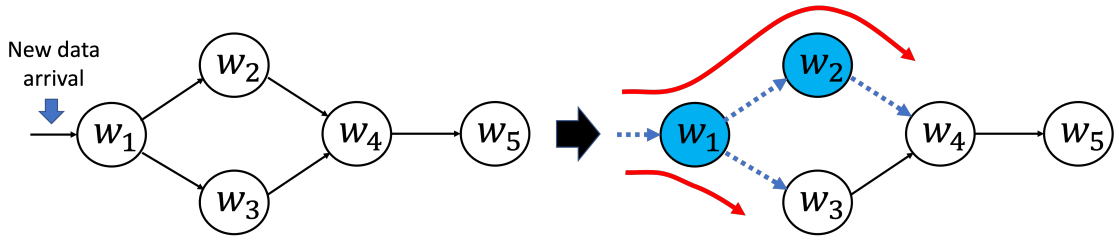
Thus, this section focuses on developing a safe mode change protocol in terms of end-to-end dynamic deadlines based on already guaranteed per-gang periodicity. Even though variables from the multicore formulation are used in this section, the same protocol can be readily applied to uniprocessors by replacing gang variables with task variables.

Assume the system is switching from an *old* mode to a *new* mode, represented by each mode's shortest deadlines, respectively, by

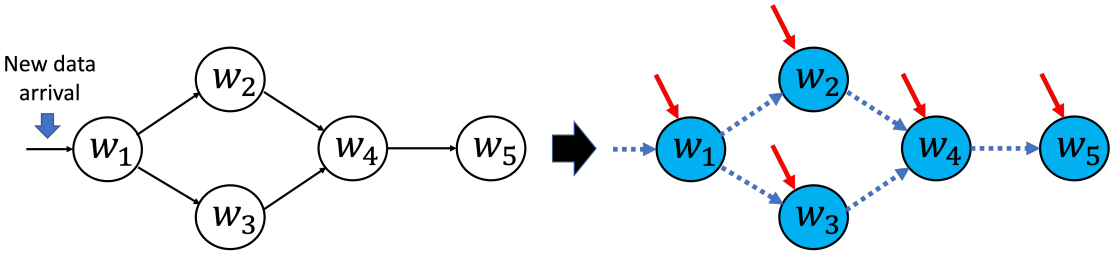
$$d^{old} \rightarrow d^{new}. \tag{3.16}$$

When $d^{old} < d^{new}$, it is termed as *relaxing deadlines* and in the opposite as *shrinking deadlines*. System configurations for each mode are denoted by $\Pi^{old} \rightarrow \Pi^{new}$, that is,

$$((P_1{}^{old}, S_1{}^{old}), \cdots, (P_n{}^{old}, S_n{}^{old})) \rightarrow ((P_1{}^{new}, S_1{}^{new}), \cdots, (P_n{}^{new}, S_n{}^{new})) \tag{3.17}$$



(a) As Late As Possible (ALAP)



(b) As Early As Possible (AEAP)

Figure 3.5: ALAP: tasks gradually change modes (white → blue) as the new data progress along paths (curved red arrows) possibly at different speeds. AEAP: the new sensor data arrival immediately triggers (straight red arrows) every task's mode change.

in its expansion form. The system mode change is triggered by a sensor data arrival at time t_0 with its dynamic deadline falling above or below the old range. In terms of individual gangs, their mode changes only happen at period boundaries. That is, when a gang period has ended and the new one is about to start, the scheduler adjusts its internal structure to the new mode's period and speed factor for that gang. This is because, if we change the mode for a gang when its period has not finished yet, the instantaneous utilization $(E_i(S_i)/P_i)$ for the gang disturbs the overall schedulabilty (Eq. (2.13)), which defeats the purpose of utilization invariability in our optimization constraints and may lead to a deadline violation. Then, we consider the following mode change methods, as depicted in Fig. 3.5:

- **ALAP (As Late As Possible)** individually triggers per-gang mode changes after the new data make progress to every incoming edge of it. The actual mode changes will happen at the nearest period boundary after the trigger.
- **AEAP (As Early As Possible)** immediately triggers every task regardless of the

new data's progress. The mode change completes by $t_0 + \max_i({P_i}^{old})$ in the worst case when the longest period of Π^{old} began right before t_0.

We deal with the following two cases: (i) relaxing and (ii) shrinking deadlines with the above methods, respectively.

(i) Relaxing deadline. In this case, we use **ALAP** such that the mode changes do not adversely affect the already ongoing progress of old sensor data. Besides, we need to ensure that the new sensor data do not violate d^{new}. Note that the new sensor data may progress through tasks possibly with different modes, which happens due to different speeds of different paths. For example, in Fig. 3.5(a), τ_4 has two incoming edges, where the upper path requests the mode change while the slower lower path still retains the old mode. Then the upper path $w_1 \rightarrow w_2 \rightarrow w_4 \rightarrow w_5$ can have a mixture of both modes while handling the new sensor data. Thus, the worst-case delay for the new sensor data during **ALAP** mode changes can be calculated as in the following:

$$D^{new}(\Pi^{old} \rightarrow \Pi^{new}) = \max_{\forall \delta \in \mathbb{P}'} \left(\sum_{i \in \delta} 2 \cdot \max({P_i}^{old}, {P_i}^{new}) \right) \leq \max_{\forall \delta \in \mathbb{P}'} \left(\sum_{i \in \delta} 2{P_i}^{new} \right) = d^{new}, \tag{3.18}$$

which is less than d^{new} since $\forall i : {P_i}^{old} \leq {P_i}^{new}$ that is true when relaxing deadlines. This is because, when the gang utilization(u_i^*) is fixed across modes, if S_i decreases in Eq. (2.13), P_i should increase to offset the change. Therefore, P_i increases monotonically to decrease S_i^{γ} in Eq. (3.8), reducing the average power in the next longer deadline mode.

(ii) Shrinking deadlines. In this case, which basically makes the situation more challenging, we use **AEAP** to quickly finish mode changes, minimizing possible extra delays. Delays for the old sensor data are naturally kept less than d^{old} by the same rationale in Eq. (3.5)

since $\forall i : {P_i}^{old} \geq {P_i}^{new}$ when shrinking deadlines. However, regarding the new sensor data, it can suffer extra delays if any gang's old period instance that began before the new sensor data arrival persists long enough such that the new data's progress is unexpectedly delayed by that persisting old gang instance. Algorithm 3 calculates the worst-case delay considering such negative effects for each path δ, which is an ordered set of gang indices in each path. Among the calculated delays, we can find the longest. The algorithm gradually accumulates delays by gangs in δ. **Line 1** indicates that it is unavoidable for the new sensor data to be *waited* by the old period at the first gang. Then we have two cases for the remaining gangs: (i) its mode is already changed before the data progress arrives (**Line 4**) and (ii) an old instance persists (**Line 6**). In the former, we simply accumulate the new delay component $2{P_i}^{new}$. In the latter, the persisting old (long) period ${P_i}^{old}$ hides away the accumulated delay up to then, resetting it to ${P_i}^{old} + {P_i}^{new}$.

Algorithm 3 Finding the worst-case delay for AEAP

Require: $\{(p_1^{old}, \cdots, {P_n}^{old}), ({P_1}^{new}, \cdots, {P_n}^{new}), \delta\}$
Ensure: The worst-case delay of new sensor data for δ
1: $D \leftarrow P_{\delta[1]}^{old} + P_{\delta[1]}^{new}$ ▷ $\delta[1]$ denotes its first element
2:
3: **for** $i \in \delta \setminus \{\delta[1]\}$ **do**
4: **if** $D + 2{P_i}^{new} > {P_i}^{old} + {P_i}^{new}$ **then**
5: $D \leftarrow D + 2{P_i}^{new}$
6: **else**
7: $D \leftarrow {P_i}^{old} + {P_i}^{new}$
8: **end if**
9: **end for**
10: **return** D

By the above analyses, we claim that when relaxing deadlines with **ALAP**, there is no end-to-end deadline miss for both the already ongoing progress and new ones. When shrinking deadlines with **AEAP**, the already ongoing progress rather benefits from it, whereas new sensor data can suffer extra delays. However, we can analyze the worst-case extra delays, which can be used when planning appropriate safety margins in design time. For example,

instead of using direct mapping from velocities to modes, the system can change its mode a little earlier at a lower velocity to offset the extra delays.

3.6 Evaluation

3.6.1 Experimental Setup

Table 3.1: Workload information at maximum speed ($s_i = 1$)

Task	Camera Grabber	Lidar Grabber	CAN	SFM*	Lane Detection*	Object Detection*	Loc.*	EKF*	Planner	DASM
WCET(ms)	0.25	25.7	0.6	30.6	27.1	294.8	175.5	1.6	21.0	1.9
r_i	0.5	0.5	0.5	0.15	0.45	0.29	0.20	4e-9	1e-15	1e-15

* from Chauffeur benchmark suite [64]. WCETs of unmarked applications are derived from [58]

Table 3.2: Gang formation of WATERS workloads on quad-core CPU using the heuristic from [6]

Gang	1	2	3	4	5
Tasks	SFM,Lane Det., Obj Det., Loc.	Cam. Grabber, Lidar Grabber, CAN	Planner	DASM	EKF
WCET at $\forall S_i = 1$	294.8	25.7	21.0	1.9	1.6
r_i **at** $\forall S_i = 1$	0.29	0.5	1e-15	1e-15	4e-9

Workload. When we need an exemplar DAG, we use the one in Fig. 2.1 with ten tasks, where their WCETs and the speed-independent ratio r_is are listed in Table. 3.1 [58]. Unfortunately, WATERS industrial challenge applications are IP protected, so we were not able to run them and obtain r_is. We borrowed available applications from Chauffeur autonomous driving benchmark [64] and measured their WCETs and $r_i s$ on Nvidia Jetson TX2 platform (A57 cores). We did a frequency sweep to get r_i and applied curve fitting. All the r_is have R-squared greater than 0.99 in their curve fitting. The WCETs of unmarked applications in Tab. 3.1 are derived from actual measurements on the same Jetson platform from [58] and

we made an educated guess about their r_is.

DAG generation. We use GGen random graph generator [30] to generate multi-source and -sink graphs in layer-by-layer method: 500 DAGs for each input size n=5, 10, 20 and edge probability p=0.5, 0.25, 0.125. All the generated DAGs are non-isomorphic to each other (i.e., structurally different). WCETs are generated with a uniform distribution and, for each WCET, three sets of r_is are generated from different ranges: low [0.0, 0.5], high [0.5, 1.0], and mixed [0.0, 1.0].

Power Model. We empirically found the power parameters for Eq. (2.14), as α=842.04, β=232.81, and γ=2.64, on the same hardware platform in [58].

Discrete frequency levels. We use 12 evenly spaced frequencies between 345 MHz and 2 GHz from the same hardware platform [58] for real-world driving data simulation.

Driving Scenarios. We use real-world driving scenarios from the comma.ai driving dataset [29], where we picked ten 60-second driving logs with their velocity from 0 km/h to 114 km/h as depicted in Fig. 3.15.

Dynamic Deadlines. Each DAG has its shortest and longest end-to-end latency on the target platform when $\forall S_i = 1$ and $\forall S_i = s_{min}$, respectively. For the fair comparison between DAGs, we converted velocities into deadlines for each DAG using Eq. (3.1) by picking values for λ when a^{max}=2.5 ms^{-2} such that the shortest latency of the DAG becomes the shortest deadline at the highest velocity (114 km/h) in our scenarios, whereas the longest latency of the DAG which is bounded by $s_{min} = 0.17$ becomes the longest deadline. Then, the number of modes is chosen arbitrarily to 10, partitioning the range with equal length.

Optimization. For the GP solver, we use MATLAB CVX [44] convex programming package. On a laptop with a quad-core Intel Core i7@2.6 GHz CPU, it took 3.07, 7.97, and 23.11 seconds for {n=5,p=0.5}, {n=10,p=0.25}, and {n=20,p=0.125} taskset, respectively.

Simulation. We implemented a simulator supporting the EDF scheduling and our mode change protocol, by which we can precisely estimate the exact task schedules, end-to-end

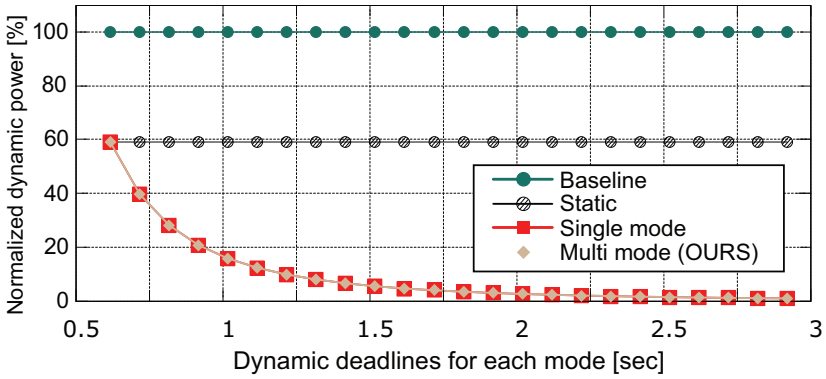


Figure 3.6: Dynamic power optimization results

latencies, and energy consumption. Our simulator takes 0.1ms time step in each iteration and in each step the EDF scheduler routine is invoked. Gangs are submitted to the scheduler queue at the start of the simulation and they resubmit themselves at the end of their period. The EDF scheduler (1) decreases the current gang's remaining WCET, (2) check if the current gang's WCET is equal to or below 0 and schedule a new task from the queue if needed, (3) set the speed factor and period for the new task, and finally (4) maintain a queue in the earliest first deadline order.

3.6.2 Uniprocessor Results

The goal of our uniprocessor evaluation is twofold: (i) illustrate the benefit of using multiple frequency levels in dynamic deadline situation over conventional methods that utilize only one or two frequencies (ii) evaluate the effectiveness of our mode change protocol. It is of no use if it compromises safety. The following three methods are compared in the evaluation:

- **Baseline**: $\forall i : s_i = 1$;
- **Static**: optimized for the minimum deadline (617 ms);

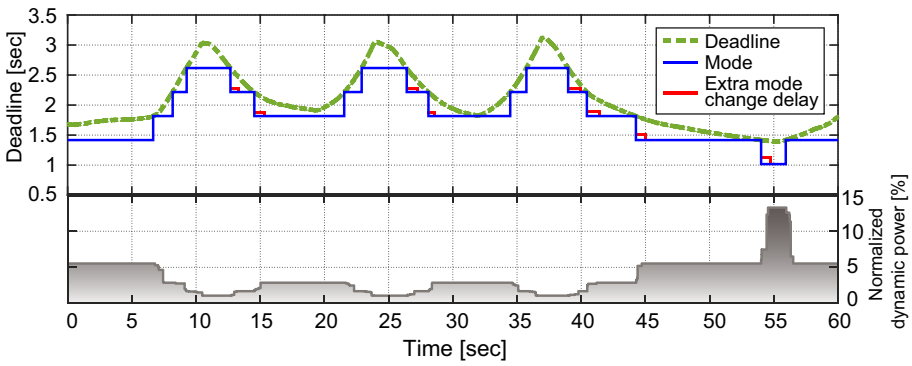


Figure 3.7: Simulation results with a real-world driving scenario

- **Multi mode (OURS):** method in Section 3.3.1.

Fig. 3.6 shows the dynamic power optimization results of each method. For *Single mode*, we solved the problem in Section 3.3.1 repeatedly for each deadline. In the leftmost mode with the shortest deadline, the three methods except the baseline method show an equal result (58.9%). However, as the deadline increases, the dynamic power plunges to 1.0% in the rightmost mode with the longest deadline. Another interesting observation is that our multi-mode results do not reveal visible performance degradation compared with the single-mode results, meaning that the per-task utilization invariability constraint does not critically affect the optimization results. On top of that, there is a 75% reduction in overall optimization time when using multi-mode formulation thanks to fewer decision variables.

Fig. 3.7 shows the simulation results with one of our driving scenarios. Only in this experiment, we partition the deadline range into six equal length modes for better visualization. As the deadline changes, as depicted by the green dashed line, mode changes are sporadically triggered, depicted by the rising and falling edges of the blue line that roughly follows the dynamic deadline. Note the small red staircase shapes at each falling edge, which depicts the extra delays when shrinking deadlines analyzed by Algorithm 3. More specifically, their height represents the extra delay, while their width represents the transient interval for each

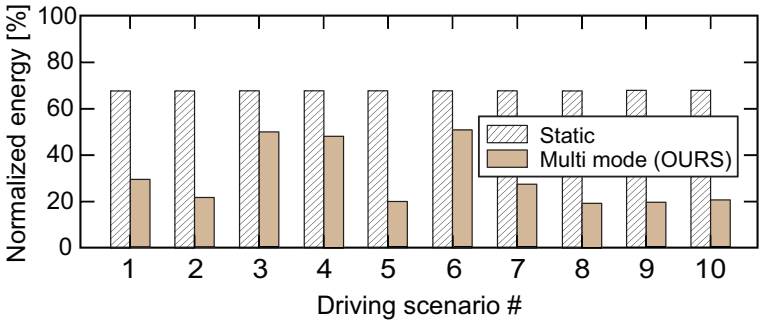


Figure 3.8: Energy consumption with varying driving scenarios (continuous frequency)

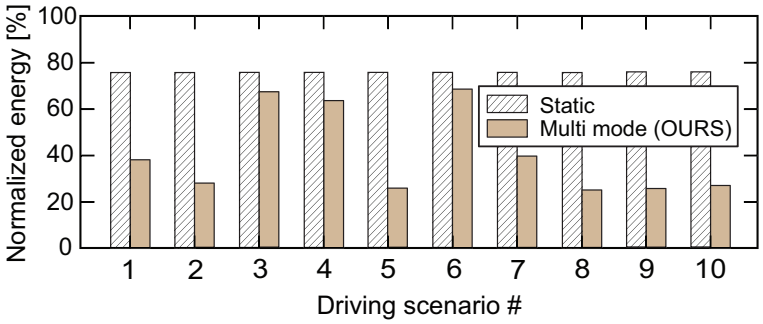


Figure 3.9: Energy consumption with varying driving scenarios (discrete frequencies)

mode change. As shown in the figure, we do not have any deadline miss in this scenario even after considering the extra mode change delays. Additionally, the bottom plot shows how the normalized dynamic power varies (between 1.0% and 13.3%) depending on the mode changes.

Fig. 3.8 shows the energy consumption, including static and dynamic power, in driving scenarios using continuous frequency. Though *Static* achieves a significant energy reduction of 32% from baseline, *Multi mode* reduces further by timely utilizing lower frequencies. Scenarios #3, #4, and #6 show relatively low energy reductions since they maintain the vehicle within a high-velocity range most of the time, making the system remain in the short deadline modes. In the others, more than half of the total energy was saved. On average, our method achieved 69.3% and 54.8% energy reductions compared with the baseline and the static methods, respectively.

Fig. 3.9 shows the energy consumption using discrete frequencies as discussed in Section 3.3.4. A discrete frequency that is greater than or equal to the optimal one is assigned to each task. Therefore, we expect the energy consumption will increase in both methods. *Static* achieves 24% of energy reduction, which is smaller than the previous case. *Multi mode* still shows a significant reduction from *Static*, however, the energy gap between them becomes smaller. In scenarios #3, #4, and #6, the reduced gap is very notable. This is caused by the nonlinearity in the power model in terms of speed factor. The difference between the optimal and mapped discrete frequency yields bigger difference in power consumption in the higher frequency range. On average, there was a 31.8% increase in energy consumption when mapped to discrete frequencies. Our method achieved 59.3% and 46.4% energy reductions on average compared with the baseline and the static methods, respectively.

Throughout the experiments with the industry-level applications and real-life driving scenarios, we could not find any deadline violation with our mode change protocols. However, as we discussed in Section 3.5, the extra delays when shrinking deadlines are unavoidable and they could be an issue depending on vehicle's maximum acceleration and mode lengths. In such cases, the violations can be avoided by selecting a shorter deadline mode considering possible mode change delays analyzed in Section 3.5 (i.e. a safety margin) in design time analysis. We leave the choice of optimal margin for our future work since it requires a thorough analysis.

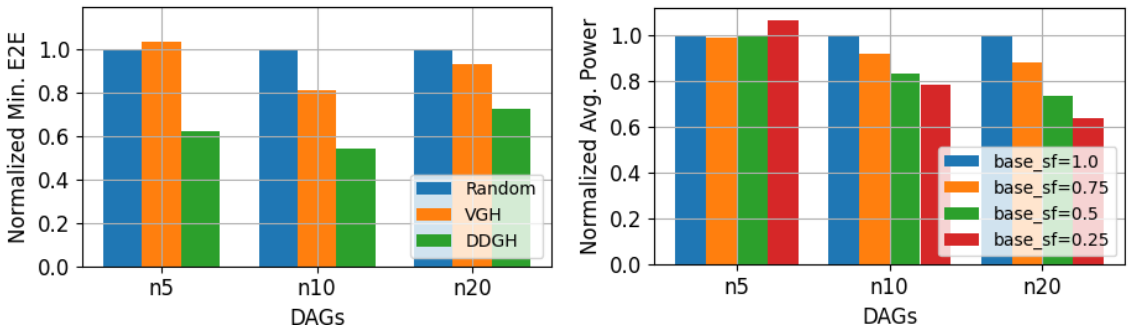
3.6.3 Multicore Results

We first present gang formation results, by which we decide gang formations for each DAG for the rest of the evaluation. For the period and speed factor optimization, we compare power savings in individual modes with our multi-mode optimization method. Then, we

vary the parameters, such as the number of tasks, the number of edges, and the speed-independent ratio r_i, to evaluate their effect on our power optimization. Our multi-mode solution is then evaluated with real-world driving scenarios in a simulator with our safe mode change protocol. Finally, the efficacy of our safe mode change protocol is evaluated with an end-to-end deadline violation example. The following three methods are compared in the evaluation:

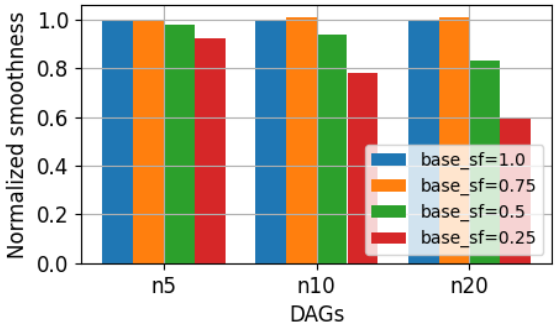
- **Baseline**: $\forall i : S_i = 1$;
- **DPM (conventional)**: $\forall i : S_i = 1$ while executing a gang, otherwise CPU is in sleep state;
- **Multi mode (OURS):** The method in Subsection 3.3.2.

One of the traditional methods of DVFS-based power optimization in real-time systems is to set a static deadline, assumed to be the shortest deadline that could possibly occur, and find the critical speed to save dynamic power consumption. The power saving of such a method largely depends on how far away the static deadline is from the minimum deadline that the given system is able to handle and thus the result could be subjective. Therefore, we set the minimum deadline as the static deadline, labeled as *Baseline*, representing the maximum possible power saving from the traditional static deadline optimization methods. *DPM* [74] is another conventional method that we are comparing to, which could reduce both dynamic and static power consumption. In general, it is believed to have a better power efficiency than DVFS in periodic real-time systems[10]. Although there are many power-efficient scheduling algorithms for DPM, they do not consider safe mode change and thus cannot be used directly in our setting. Therefore, we use our safety-guaranteed solution and mode change for DPM experiments. Specifically, they have the same gang periods as our multi-mode solution such that end-to-end deadlines are guaranteed. In each mode, a



(a) Minimum end-to-end latency

(b) Average mode dynamic power



(c) Gang WCET curve smoothness

Figure 3.10: Gang formation heuristic comparison and base speed factor variation experiments

gang will be given the time for execution based on Eq. (2.10). However, the gang will be executed at the full speed and the remaining slack is used for sleeping to cut off the static power consumption. For simplicity, we do not consider the overhead of changing CPU sleep states.

3.6.4 Gang Formation Heuristics Evaluation

In Fig. 3.10a, we evaluate three gang formation heuristics: random, VGH, and our DDGH with 500 DAGs for each taskset (n=5, 10, 20) and mixed values of r_is. The results are normalized to random formation results for each DAG and then averaged. The random method randomly picks a task and assigns it to a gang, including both existing and new

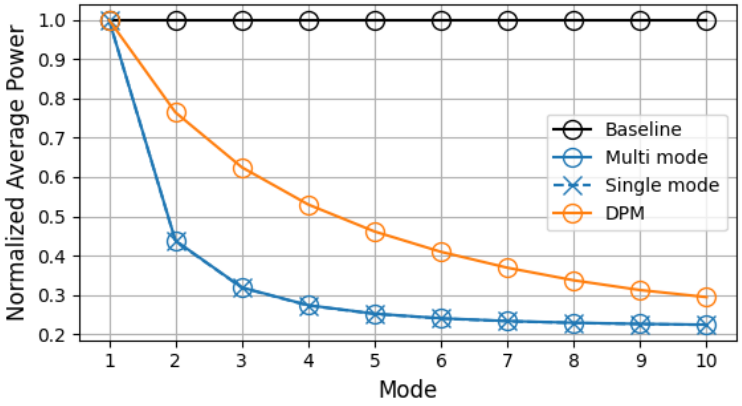
gangs, with an equal probability. VGH is the current state-of-the-art method but it aims to minimize the total completion time, exhibiting slight improvement for end-to-end latency compared to the random formation. It is worse than random when a DAG is small and simple as in n=5 taskset. When the length of paths in a DAG is not too long, it is advantageous to pack dependent tasks together as we have seen in the example (Fig. 3.3), which is not possible with VGH because of the family policy. However, as the paths get longer as the number of tasks increases in n=10 taskset, VGH's family policy takes a positive effect and performs better than the random method. With n=20 taskset, even though the gang formation problem gets significantly difficult due to combinatorial explosion, our DDGH still performs better than VGH because of its flexibility in grouping dependent tasks and better end-to-end latency estimation. Overall, DDGH performs 32% better on average compared to VGH in all tasksets.

Next, we evaluated the effect of the base speed factor (S_{base}) in Algorithm 2. In Fig. 3.10b, we sampled four values of S_{base} from 0.25 to 1.0 to see the trend. For each S_{base}, we apply multi-mode optimization to the gang formation to obtain power consumption in each mode. The number of modes was set to 10 each of which has a corresponding end-to-end deadline for which all the S_{base} gang formations were optimized equally. Since it requires a holistic evaluation across modes, the average mode power consumption is used for evaluation. Our initial hypothesis was that the sweet spot would exist around $S_{base} = 0.5$ so that the gang WCET curve change is minimized towards both ends of the mode range: the shortest and the longest modes where S_i changes are maximum. However, we discovered that the lower S_{base} is, the less degradation it gets by S_i changes, resulting in a smaller average mode power. The reason is that, when task WCETs are scaled to a lower S_{base} value, DDGH identifies the tasks whose WCETs are significantly scaled up and tends to group them together. In this way, even if there is a change in S_i, the overtaking of a dominant task happens less

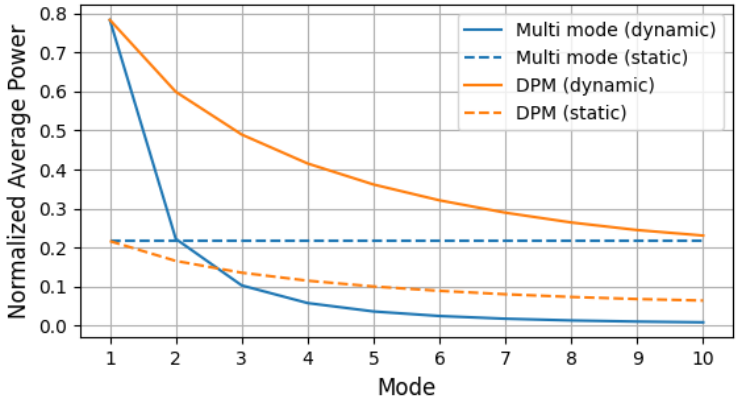
frequently and the quality of gang formation is maintained stable over S_i changes. We measured the smoothness of gang WCET curve in S_{base} solutions for each taskset. The smoothness is defined as the e_{comp} differences between the two tasks when dominant task switching happens as e_{comp} is the only component affected by S_i. The larger the difference is, the greater the impact on gang formation quality and the resulting end-to-end latency, so a smaller value is desired for smoothness. For n=5 taskset, because the DAGs are simple, gang WCET changes rarely occur so the smoothness did not change much and our S_{base} method did not have a major impact. However, for larger tasksets, smoothness declines steeply, exhibiting a similar trend with the average mode power consumption. As a result, for n=10 and n=20 with $S_{base} = 0.25$, the power consumption is reduced by 21.8% and 36.1%, respectively, compared to $S_{base} = 1.0$ gang formation. Following this trend, S_{base} is set to S_{min} for the rest of the experiments that involve DDGH.

3.6.5 Multi-mode Average Power Optimization

Fig. 3.11a shows normalized average power in each mode, averaged across 500 randomly generated 10-task DAGs with mixed values of r_is. The modes are numbered in order from the shortest to the longest deadline, equally partitioning the range from minimum to maximum end-to-end latency of each DAG. For the comparison between single-mode and multi-mode optimization, the only difference is the utilization invariability constraint. However, as it can be seen in the figure that the two curves are almost perfectly overlapped, showing its effect on power consumption is negligible. Even if the freedom of gang utilization variable is limited in multi-mode optimization, the other decision variable P_i is flexibly adjusted to lower S_i as much as possible, still reaching the near-optimal power consumption. In the first mode with the shortest deadline, both DPM and our multi-mode approach show the maximum power consumption as there is no slack for slowing down nor shutting off the CPU



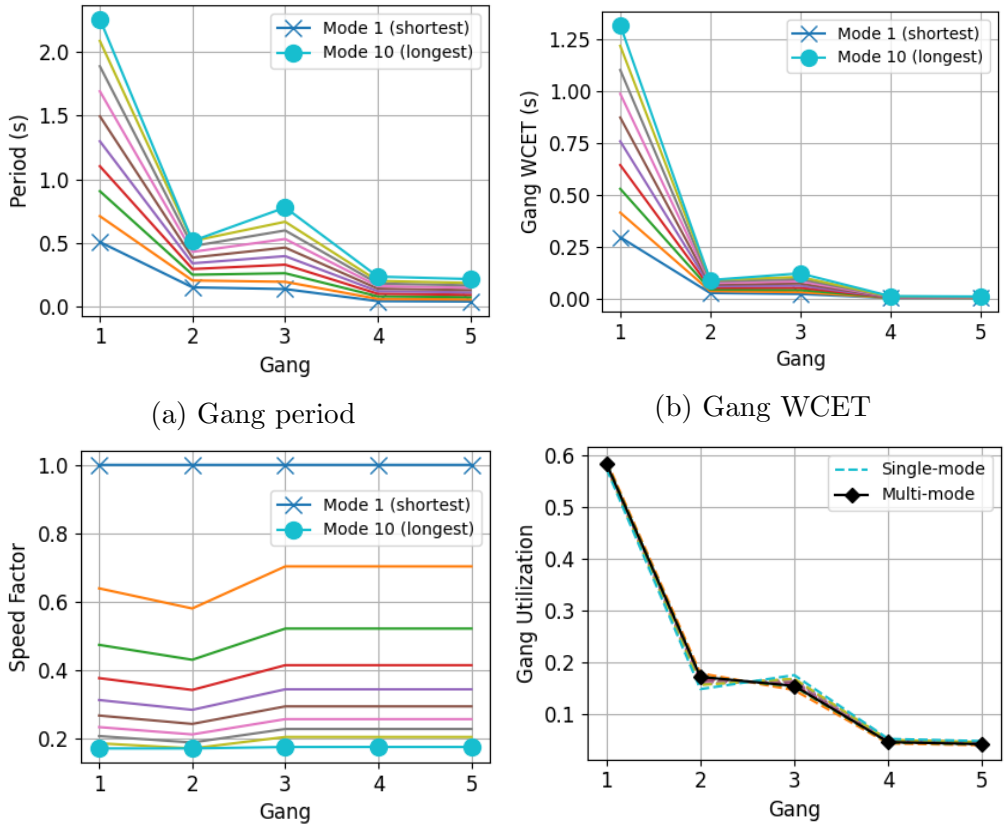
(a) Total power consumption



(b) Dynamic and static power consumption

Figure 3.11: Power optimization results including both dynamic and static power. Mode 1 has the shortest deadline

cores. As the deadline gets longer, the slacks also become longer. To exploit them, DPM inactivates the CPU cores to save static power as can be seen in Fig. 3.11b. Since the ratio of execution burst at full speed to the size of slack gets smaller, the average dynamic power of DPM also decreases but not as fast as our multi-mode method. Even though the CPU cores are always on consuming static power, the dynamic power reduction is much greater than DPM resulting in overall smaller average power of 22.4% against the maximum power consumption compared to 30.8% of DPM in the longest deadline mode.



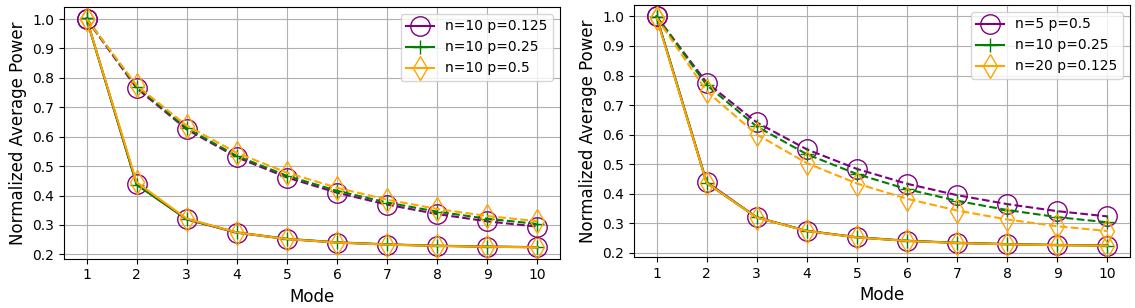
(a) Gang period

(b) Gang WCET

(c) Gang speed factor

(d) Gang utilization

Figure 3.12: Gang parameters in each mode: gang period, WCET, speed factor, and utilization



(a) Varying number of edges

(b) Varying number of tasks

Figure 3.13: Power optimization results with varying number of tasks and edges. Mixed values of r_is are used. The dashed lines are DPM counterparts

3.6.6 Gang Parameter Details in Each Mode

Fig. 3.12 depicts the exemplar DAG's gang parameters in each mode after applying our multi-mode optimization. Only the first and the last mode are marked for readability. Basically, each of these parameters tends to scale linearly from the shortest mode. The gang period (Fig. 3.12a) and WCET (Fig. 3.12b) are positively correlated as it can be seen starting from the bottom of the figures that they increase similarly as the mode goes higher. Note that for gangs 2 and 3, their WCETs are similar but gang 3 is more sensitive to speed factor due to its low speed-independent ratio r_i. Eventually, in the longest deadline mode, gang 3 has a longer WCET and period than gang 2. For speed factors (Fig. 3.12c), all the gangs try to maintain a similar value. This is because the exponent(γ) of the speed factor in power Equation (2.18) is typically greater than 2, so the power is more sensitive to an increase in speed factor than a decrease. Therefore, if all the WCETs scale at the same rate with S_i, i.e., all r_is are the same across tasks, the solver cannot favor a specific S_i which may lead to an increase in another gang's S_i and worse overall power saving. In result, it tries to find a uniform value for all S_is. In Fig. 3.12c, however, the speed factors of gangs 1 and 2 are lower than others. Given utilization and period, these gangs with higher r_i can afford lower S_i due to the low sensitivity of WCET to S_i. Finally, the gang utilization (Fig. 3.12d) is maintained the same across modes in the multi-mode solution. Note that even though the optimal gang utilization from single-mode solutions depicted in dashed lines does not deviate too far from the multi-mode solution, they could possibly violate the utilization bound during mode change. However, the effect of the deviation on power saving is negligible as can be seen in Fig. 3.11.

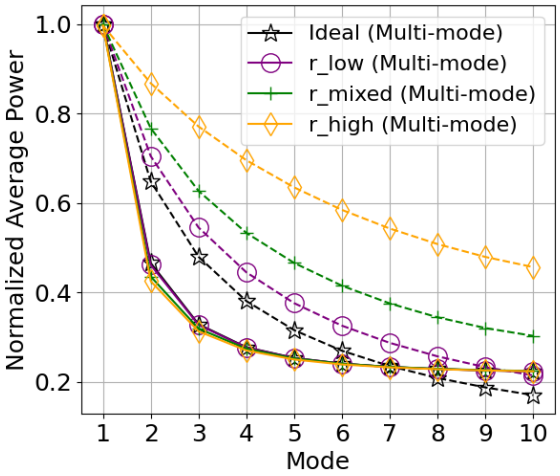


Figure 3.14: Power optimization results with varying speed-independent ratio r_is. The dashed lines are DPM counterparts

3.6.7 Effect of the number of Edges and Tasks

When generating random DAGs, we tried to have a similar shape to the exemplar DAG meaning that the ratio of the number of edges to the number of nodes should be similar. This can be controlled by the edge probability parameter provided by GGen which resulted in different values for each taskset (n=5, 10, 20) as specified in Section 3.6.1. In Fig. 3.13a, we first fix the number of tasks and only vary the edge parameter to see the effect on our multi-mode optimization compared to the DPM method in dashed lines. Each taskset's result is averaged across 100 randomly generated DAGs with mixed values of r_is. As can be seen in the figure, the number of edges does not affect the amount of power saving in our multi-mode approach meaning that it is not an important parameter of our optimization. Likewise, in Fig. 3.13b, we vary the number of tasks with selected edge probabilities and they result in the same power saving as in Fig. 3.13a. In both experiments above, our multi-mode optimization achieves strictly better power saving than the DPM method.

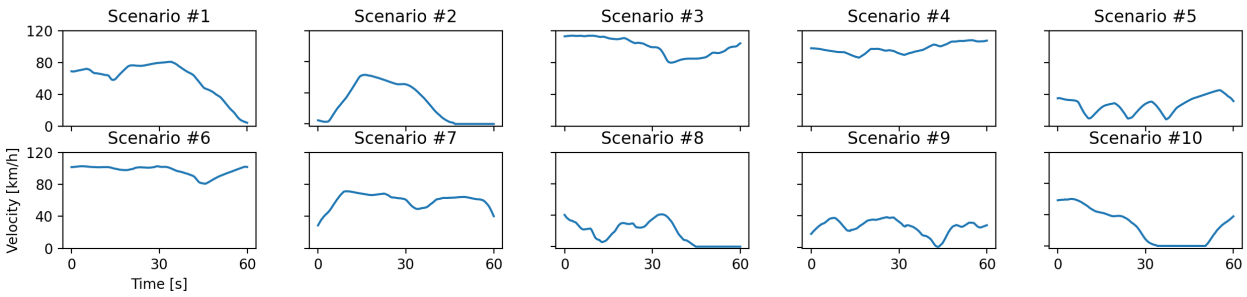


Figure 3.15: Real-world driving scenarios from comma.ai dataset

3.6.8 Effect of Speed-independent Ratio r_i

So far, we have been using mixed values of speed-independent ratio r_is in our experiments. This experiment investigates the effect of r_is on our multi-mode optimization by comparing four different sets of r_is. In addition to low, high, and mixed set specified in Section 3.6.1, we added the ideal set where all r_is are set to zero. In Fig. 3.14, the DPM counterparts in dashed lines vary significantly with r_is while our multi-mode method shows stable power saving result. The power consumption of DPM increases as r_i gets higher because when r_i is high the WCET does not scale well, resulting in longer execution time hence more static power consumption. On the other hand, our multi-mode method can flexibly adjust the speed of the processor to save dynamic power regardless of the r_is. It is interesting to note that DPM outperforms our multi-mode optimization in long deadline modes with the ideal r_i set. In reality, however, it is unrealistic to have the ideal set of r_is. The exemplar DAG's r_is are close to the low r_i set in the figure which could perform equal to or better than DPM in every mode.

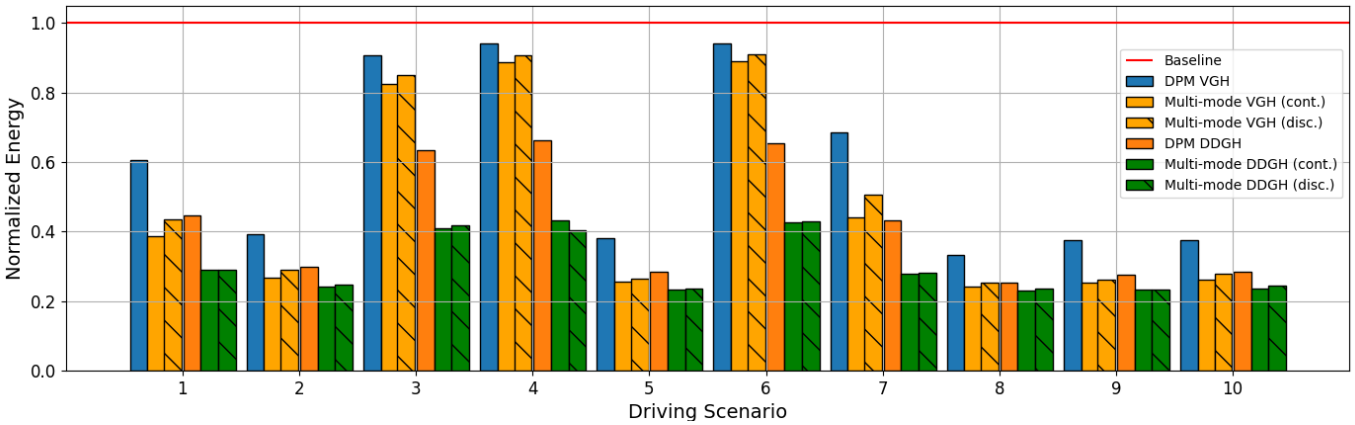


Figure 3.16: Energy consumption with various driving scenarios

3.6.9 Energy Optimization with Real-world Driving Scenarios

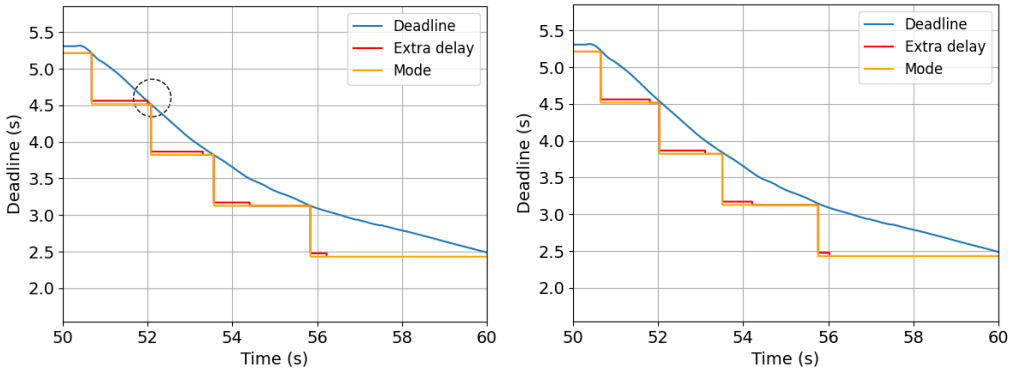
Fig. 3.16 shows the energy consumption, including static and dynamic power, in driving scenarios using continuous and discrete frequencies. The exemplar DAG with its r_is and our multi-mode optimization were used in this experiment. For each driving scenario, the first three bars are the energy consumption of DPM and multi-mode methods using VGH while the latter three bars are using our DDGH. We start our analysis with VGH solutions first and then discuss DDGH ones later. With continuous frequency, though DPM VGH achieves a significant energy reduction of 40.6% on average from baseline, the multi-mode VGH reduces further by timely utilizing lower frequencies. For discrete frequencies, we use the frequency that is greater than or equal to the optimal one assigned to each gang as discussed in Section 3.3.4. Therefore, we expect the energy consumption will increase slightly in our multi-mode method while the DPM method is not affected because it uses only the maximum frequency. As a result, the energy consumption is increased by 6.2% on average for the multi-mode solution with discrete frequencies but it still shows a significant reduction of 20.4% from DPM VGH. Scenarios #3, #4, and #6 show relatively low energy reductions since they maintain the vehicle within a high-velocity range most of the time, making the system remain in the short deadline modes. On average, our multi-mode VGH

method achieved 52.9% and 30.7% energy reductions compared with the baseline and the DPM VGH method, respectively. In the others, more than half of the total energy was saved.

On top of that, our DDGH can minimize energy consumption further by providing shorter end-to-end latency gang formation for both DPM and multi-mode optimization. The shorter end-to-end latency means there is more slack from the end-to-end deadline, and thus a lower CPU frequency can be used in case of multi-mode optimization, or for the longer period the core is turned off for the DPM method. As a result, DDGH enables 27.8% and 25.5% improvement from VGH solutions for the DPM and the multi-mode method, respectively. For high-velocity scenarios #3, #4, and #6, the improvement by DDGH is significant because it enables the lower CPU frequencies for the same deadline. For other scenarios, the improvement is relatively small because they are already exploiting low CPU frequencies and the power consumption difference between these modes is small as depicted in Fig. 3.11a. For discrete frequencies, the energy consumption is increased by 0.9% on average. However, we noticed that for scenario #4, the energy consumption is actually decreased from its continuous frequency counterpart. The high-velocity scenario #4 has small changes in velocity and the continuous solution utilizes total of two modes for the entire 60 seconds. With discrete frequencies, slightly higher frequencies are mapped which in result adds one more available mode whose deadline falls into scenario #4's deadline range. This is an exceptional case related to the mode granularity and if we increase the number of modes the discrete frequency solution will consume more energy as expected. On average, our method achieves 54.9% and 30.3% total energy reductions compared with the baseline and the traditional optimization with the current state-of-the-art gang formation heuristic (DPM VGH), respectively.

3.6.10 Safe Mode Change Evaluation

In Section 3.5, we analyzed that the end-to-end deadline violation could only occur when the deadline gets shorter in shrinking deadline situations. The old instances in longer periods can cause extra delays in response time and we provided an algorithm to predict the worst-case delay. To evaluate our algorithm and mode change safety, because we couldn't find any violation throughout the driving scenario experiments, we generated an artificial deadline graph with a sharp slope as in Fig. 3.17 in addition to worst-case EDF scheduling. It is possible to happen in the real driving scenario but it is not included in the dataset we used. The artificial driving scenario is basically a strong acceleration for continuous 10 seconds with the worst data propagation patterns. The same exemplar DAG and its multi-mode solution were used for this experiment. As the deadline changes, as depicted by the blue line, mode changes are sporadically triggered, depicted by the rising and falling edges of the orange line that roughly follows the dynamic deadline. Note the small red staircase shapes at each falling edge, which depicts the extra delays when shrinking deadlines analyzed by Algorithm 3. More specifically, their height represents the extra delay, while their width represents the transient interval for each mode change. Not only the mode deadline curve (orange) should not go above the dynamic deadline (blue), but also the extra delay curve should not (red staircase). In Fig. 3.17a, an end-to-end violation occurred at the end of the staircase at $t = 52.1$ marked with a circle. In Fig. 3.17b, we shifted the mode deadline curve to the left by applying a small safety margin which triggers mode change a little earlier such that the red staircase shape is not overlapped with the deadline curve. As a result, the end-to-end deadline violation was avoided.



(a) (b)

Figure 3.17: An example of end-to-end deadline violation with AEAP method

3.7 Summary

We presented EASYR: Energy-Efficient Adaptive System Reconfiguration for Dynamic Deadlines in Autonomous Driving on Multicore Processors. Our work is motivated by emerging autonomous driving applications with time-varying dynamic deadlines, where the computing system's excessive energy consumption is a major concern. Unlike traditional energy optimization methods assuming rigid static deadlines, our EASYR approach utilizes the dynamic slack obtained by adaptively relaxing deadlines considering the vehicle's physical state. Towards that end, EASYR's GP-based optimization method proactively exploits the dynamic deadlines to find energy-efficient multi-mode system configurations. To enable our framework on multicore processors, we use RT-Gang scheduling to eliminate the challenges introduced by multicore processors: predictability and schedulability which are important in the context of real-time systems. Our experimental results show that our gang formation heuristic performs 32% better on average than the state-of-the-art heuristic. Our multi-mode optimization reduces the average total energy consumption by up to 54.9% in various real-world driving scenarios compared with the maximum energy consumption configuration and 30.3% compared with the conventional method using dynamic power management (DPM). Moreover, our extensive simulation experienced no deadline miss due to our safe

mode change protocol and delay analysis method. To the best of our knowledge, our work is one of the first attempts to optimize the energy consumption in multicore systems for autonomous driving, explicitly focusing on dynamic deadlines.

This study's contributions can be summarized as follows:

- We developed a greedy heuristic to obtain gang formations which could lead to better energy savings.
- We formulate an optimization problem for energy-efficient multicore computing systems for autonomous driving with time-varying dynamic deadlines and provide a GP-based optimal solution.
- We provide a safe mode change protocol that guarantees analyzable (if any) overheads, which can be safely manipulated in the design time.
- Our experiments use a realistic autonomous driving taskset from industry and actual measurements from the target computing system in addition to randomly generated tasksets for extensive analysis for comparison with conventional real-time energy optimization methods.

Based on the theoretical foundation presented with EASYR, in the future, we plan to extend EASYR's system model considering heterogeneous CPU clusters and other accelerators such as GPUs. Also, instead of just velocity, we plan to consider other factors that can dynamically affect end-to-end deadlines such as adaptive redundant execution for fault tolerance. Other future extensions would be to jointly support both real-time and non-real-time tasks, in addition to supporting other popular schedulers such as the rate-monotonic fixed-priority scheduler with a different schedulability test and a utilization bound.

Chapter 4

Safe Adaptive System Reconfiguration for Resilient Timing Safety

4.1 Motivation and Challenges

Autonomous driving is anticipated to deliver a promising green future characterized by low fatalities and enhanced traffic flow. Electric vehicles (EVs) have emerged as a fundamental platform for autonomous driving, and are expected to have longer lifespans due to the absence of failure-prone mechanical combustion engines and transmissions. Indeed, a US Bureau of Transportation Statistics report [20] cites an increase in the average lifespan of US vehicles from 8.4 years in 1995 to 12.2 years in 2022, with this vehicle longevity trend expected to grow, given the rapid growth in EV sales. As EV lifespans increase, and with scaling of circuit technology, the electronic components in these vehicles become more susceptible to various permanent faults and aging effects, posing safety risks regardless of the quality of initial design. For example, P_{fail} that affects SRAM cells is increased to 10^{-4} at 12nm from

10^{-13} at 45nm [66]. Aged NVMe SSDs in Alibaba datacenter are experiencing fail-stop and fail-slow failures [62].

Accordingly, the Worst-case Execution Time (WCET) of safety-critical tasks may be affected during vehicle operation, which in turn may compromise overall vehicle safety. Classical approaches estimate WCET at design time, and build stringent timing guarantees via design-time schedulability analysis, which are difficult to modify after the product is released, particularly in the face of run-time faults that affect WCET. Once the system utilization exceeds the upper bound by faults, i.e., overloaded, tasks begin to miss their deadlines, and this situation tends to only deteriorate over time due to the domino effect. To avoid such scheduling failures, traditional recovery methods often involve the use of safety margins or the deprioritization of less critical tasks.

Even if an appropriate recovery measure was triggered, there is no assurance that the WCET will remain unchanged, especially in cases of permanent faults. A highly optimized real-time system design implies a tight margin in system utilization, rendering the system vulnerable to such faults. Indeed, permanent faults have the potential to invalidate initial rigorous design effort of these long-life EVs. To address this issue, a system reconfiguration approach similar to graceful degradation could be a viable solution.

A run-time system reconfiguration entails the transition of operation modes at run-time. For example, in mixed-criticality system [77], LO- and HI-criticality modes are predefined during design phase, switching back and forth as needed. In the work of Yi et al. [84], the system can adapt to dynamically changing deadlines by also making use of predefined modes. While these predefined modes can be as fine-grained as designers intend to, they may not encompass all possible scenarios as the design space scales.

We present **FRTS**, a fault-tolerant Framework for Resilient Timing Safety that enables a

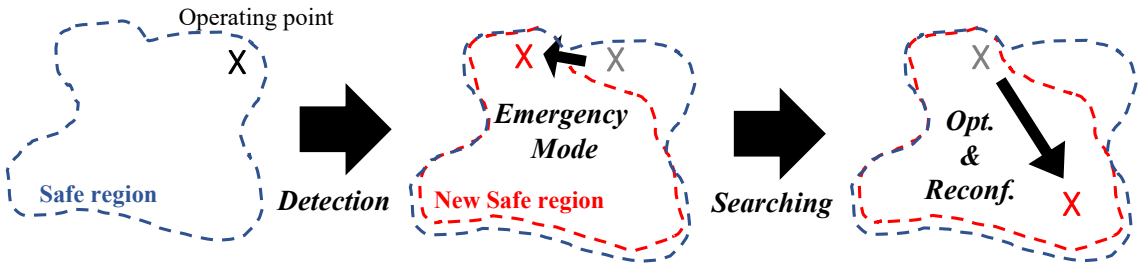


Figure 4.1: Overview of FRTS framework

proactive fault detection and a transition back to a safe, optimized system configuration. The objective of FRTS is to increase the survivability of the system, recognizing that no system is immune to faults, and there is no perfect fault-tolerance technique with zero latency and complete coverage. FRTS itself does not incorporate a fault recovery technique, but it can be employed in conjunction with other fault recovery techniques. Rather than focusing on specific type of faults, however, FRTS focuses on finding new safe yet optimized scheduling parameters following the persisting or recovered faults.

The rest of this chapter is organized as follows. The next section presents a brief overview of the framework, and Section 4.3 describes our proactive BoostIID approach for fault-agnostic detection of WCET changes. Section 4.4 describes identifying new safe, energy efficient configuration. The overhead of framework is analyzed in Section 4.5 followed by experimental results in Section 4.6

4.2 FRTS Overview

FRTS consists of four states: (i) detection of change in WCET, (ii) transitioning to emergency mode, (iii) seeking a new safe, optimized state, and (iv) system reconfiguration. Fig. 4.1 depicts the process. Initially, a system operates within a designated safe region, which is predetermined during the design phase. When a permanent fault occurs, the safe

region shrinks, rendering the current operating point unsafe. In response, the system enters an emergency mode, transitioning to a new operating point that potentially falls within a new safe region since the impact of fault is unknown. While in emergency mode, the system identifies the impact of fault in terms of pWCET (probabilistic WCET) by collecting execution time samples, and subsequently determines an optimized operating point. Finally, the system triggers a system reconfiguration to the new operating point.

Along this process, FRTS tasks must be running in addition to, and concurrently with the original workloads. The detection employs multiple i.i.d tests, which also serve as a prerequisite for finding a new operating point. Then, when a sufficient number of samples are amassed, parameter estimation of the statistical distribution is conducted for the tasks affected by fault. Once the parameters are verified through a Goodness-of-Fit (GoF) test, system scheduling parameters are optimized using the new WCETs obtained. These tasks operate in the background with lower frequency, yet they remain an integral part of the design process to ensure timely reconfiguration.

4.3 BoostIID: Proactive Fault-agnostic Detection of Change in WCET

In the detection phase of FRTS, we view faults in the system as statistical distribution changes of execution times in a time series. Fig. 4.2 illustrates a motivational example that lists execution time measurements (samples) over time. A fault occurred at 500th sample, resulting in a slight change in both the distribution and pWCET estimation. However, changes in distribution often result in subtle variations in individual data points, making it challenging to distinguish. With conventional monitoring approaches, the detection would

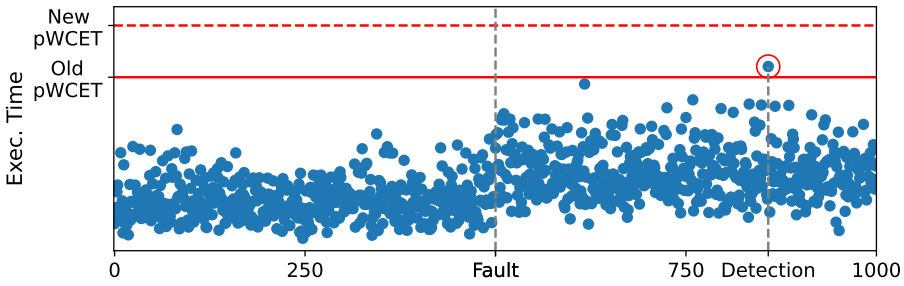


Figure 4.2: Example fault detection scenario

occur at the 858th sample after the original WCET has been violated. *Our objective is to proactively detect potential violations by monitoring execution times and sensing statistical distribution change.* This allows time for triggering recovery procedures. Furthermore, using less pessimistic WCETs without incorporating fault impacts would lead to increased system utilization and energy efficiency.

We discovered that i.i.d tests perform reasonably well for this problem. Our WCET detector periodically performs i.i.d tests on the dataset of execution times, and promptly detects changes as soon as a sufficient number of affected samples due to faults are added to the dataset.

4.3.1 Independent and identically distributed (i.i.d) tests

The i.i.d tests are commonly used to verify whether the samples originate from the same distribution or not. They serve as a prerequisite for MBPTA (Measurement-based Probabilistic Timing Analysis) , ensuring the dataset's integrity. Additionally, most i.i.d tests run very efficiently, making them suitable for online monitoring. In this dissertation, i.i.d tests will be applied on a dataset with more than one distribution. Depending on the ratio of samples from one distribution to another, different characteristics may emerge in the dataset, leading to the utilization of various i.i.d tests. In this study, we emlpoyed three exemplar i.i.d tests:

- *Kwiatkowski–Phillips–Schmidt–Shin (KPSS)* tests if a time series is stationary around a unit root
- *Rescaled Range (RS)* tests for the presence of long-term correlations or trends in a time series
- *Ljung-box* tests for autocorrelation at various lags

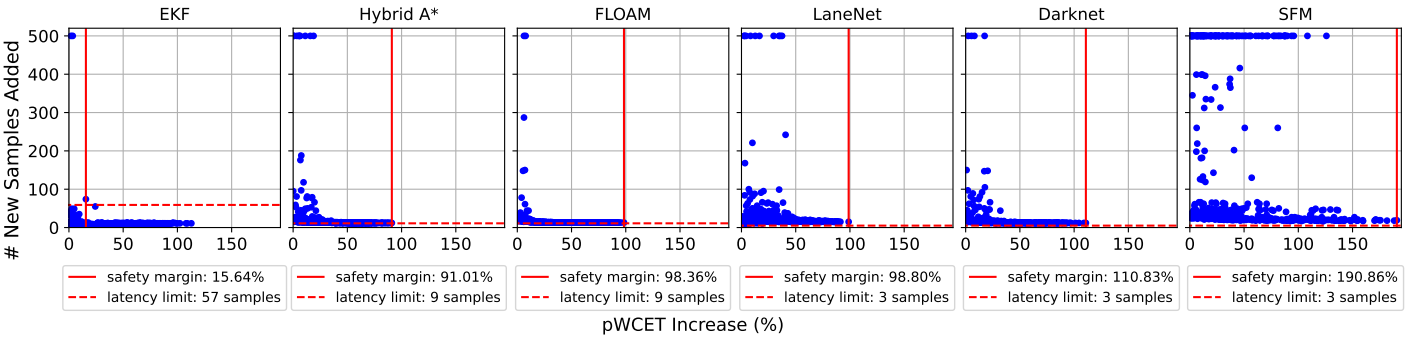


Figure 4.3: Example detection accuracy and latency using KPSS test

4.3.2 Detection latency & Safety margin

The proposed fault-agnostic WCET detector considers a scenario where permanent faults occur, leading to an impact on the GEV distribution of tasks. Prior to the fault occurrence, the detector maintains a history of execution time samples. Then, samples from the modified GEV distribution will be added to the dataset. Fig. 4.3 displays the example detection result using KPSS i.i.d test, where each data point is associated with a single GEV distribution, randomly modified from the original one shown in Table 4.1. For each data point, 500 samples are consistently retained in a first-in-first-out manner to serve as input for the KPSS test. The x-axis displays the increase in pWCET due to the faults, while the y-axis represents the number of new samples from the modified distribution added to the dataset until the KPSS test fails. Cases where the pWCET becomes shorter are considered harmless

and excluded from the analysis. Notably, when pWCET change is small, detecting the distribution change becomes challenging, necessitating a larger number of samples. Then, the *detection latency* can be defined as the number of new samples required to detect the statistical distribution change. The *latency limit* for each task is a design parameter; a detection is deemed unsuccessful if the detection latency exceeds the limit. In this example, the latency limit is set equal to the end-to-end deadline of the system. Due to the varying task periods, the latency limit for each task can be expressed as the number of samples during the limit, as depicted in Fig. 4.3 (dashed horizontal line). Then, the *safety margin* point is sought, beyond which the number of required samples remains below the latency limit as depicted in Fig. 4.3 (solid vertical line). The safety margin is added to the fault-free pWCET, effectively masking the small changes in pWCET which are difficult to detect.

Table 4.1: GEV parameter and pWCET estimation based on measurements

Task	**EKF**	**Planner**	**Loc.**	**Lane D.**	**Obj. D.**	**SFM**
Impl.	EKF	H. A*	FLOAM	LaneNet	Darknet	SFM
dist.[1]	W	W	F	F	W	W
shape	-0.14	-0.02	0.03	0.03	-0.07	-0.13
scale	0.07	0.72	0.82	1.11	1.06	6.01
loc.	2.36	22.83	86.40	18.64	27.88	6.58
WCET[2]	2.74	28.91	94.91	27.03	34.94	39.41

[1]GEV distributions. W: Werber, F: Frechet [2]unit: msec

Tasks exhibit varying sensitivity to KPSS test depending on their GEV parameters. Among the tasks, EKF demonstrates a strong fit with the KPSS test, showcasing the safety margin of 15.64%. Other tasks shows poor detection accuracy, with the safety margin hovering around 100%. Particularly, the KPSS test struggles a lot with SFM with the safety margin of 190.86%. Among GEV parameters in Table 4.1, SFM has the largest *scale* parameter. Notably, other two poor performing tasks, LaneNet and Darknet, also have relatively high *scale* values.

4.3.3 Boosting multiple i.i.d tests

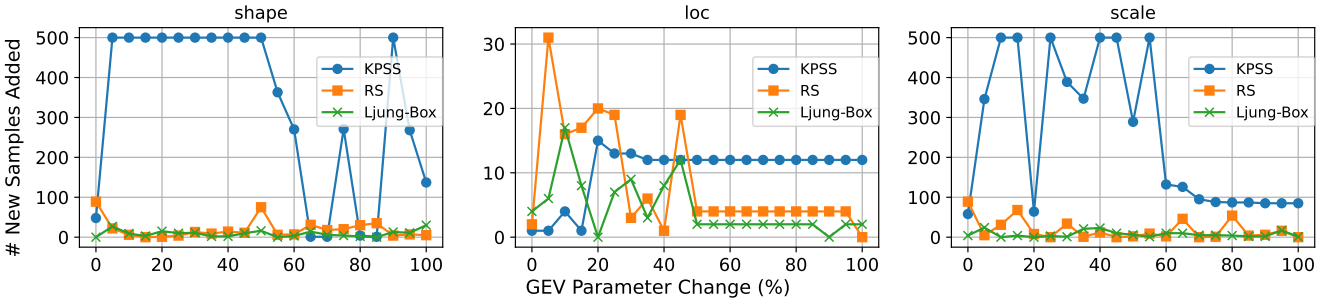


Figure 4.4: Characterization of i.i.d tests with GEV parameter sensitivity using Darknet

As can be seen, KPSS performs well for certain tasks and poorly for others. We also explored two other i.i.d tests: *RS*, and *Ljung-Box*. Fig. 4.4 characterizes each i.i.d test by systematically altering GEV parameters. The results also display varying sensitivity for different tasks, however, we only show the Darknet result for brevity. KPSS shows a low senesitivity to small changes in the *shape* and *scale* parameters. On the contrary, it detects small changes in location (mean) as it is known for verifying the stationarity of a time series. While both RS and Ljung-Box perform relatively well on *shape* and *scale*, their curves continuously fluctuate, competing for the best method. In addition, their curves often reach zero, which indicates false negative cases. Although it is uncommon for only one GEV parameter to change, this still raises concerns.

As a result, we combine i.i.d tests with a boosting algorithm [26], expecting a synergy among varying GEV parameter sensitivity. For training, similar to the experiment in Fig. 4.3, GEV parameters are randomly modified and the results of all the i.i.d tests for the same dataset are fed into the boosting algorithm as input. To minimize the training bias, half of the input consists of datasets with samples from a single distribution. The resulting classifier from the boosting is expected to lower the detection latency and safety margin. Furthermore, to reduce false negative and false positive cases as in Fig. 4.4, we included the previous n i.i.d test results as part of the boosting input.

4.4 A New Safe Energy-Efficient Configuration

Upon identifying a potential timing violation, the system enters the emergency mode, aiming to maximize system utilization to serve as a buffer against the uncertain effects of the fault. In DVFS enabled systems, the emergency mode could be simply maximizing the CPU frequency that was previously lowered for energy optimization. During emergency mode, if a timing violation occurs, FRTS cannot proceed further and a fallback maneuver must be initiated. Otherwise, if the effects of faults can be mitigated by the maximum CPU frequency, FRTS seeks a new set of scheduling parameters that is both safe and energy efficient, ultimately triggering a system reconfiguration.

4.4.1 Identifying new pWCETs affected by faults

A pWCET for a task is derived by extracting a point with a specific probability from a CDF of a GEV distribution. In that way, the probability of the task execution time exceeding the pWCET is regulated. The typical pWCET estimation process involves (i) collecting execution time samples, (ii) performing i.i.d tests to assess the integrity of samples, (iii) estimating GEV parameters, and (iv) executing GoF test to verify the GEV estimation. If the i.i.d tests fail, the subsequent steps become invalid. As a part of BoostIID, the i.i.d tests are invoked periodically. When triggered by BoostIID's detection, the sample collection process is initiated and continues until a certain number of samples are accumulated. The necessary number of samples is determined by either the minimum sample requirement for the i.i.d test or that for the GoF test. In this study, we will use the minimum sample size of 2,500 for the MAD GoF test. The GEV paramter estimation and the GoF test are executed during the emergency mode; however, ensuring schedulability at design time is crucial to identify the maximum fault tolerance, which will be discussed in Section 4.5.

4.4.2 Redundant execution for acceleration of sampling

If the impact of faults can be mitigated by the maximum CPU frequency during the emergency mode, then there will be idle periods in the system that can be utilized. However, this slack has not been identified yet so it cannot be allocated to safety-critical tasks. Instead, performing redundant task execution during the slack time can effectively reduce the duration of emergency mode, leading to quicker sampling and preventing excessive energy consumption due to maximum CPU frequency. To prioritize the safety-critical tasks within an EDF scheduler, the redundant tasks could be assigned very large deadlines, making them susceptible to preemption by tasks with imminent deadline. Then, a challenge arises in determining which task should be selected for redundant execution during the slack period. Even without the redundant execution policy, as the system operates, sampling will naturally occur, but at different rates for different tasks due to variations in task periods and WCETs. An ineffective task selection policy would imbalance the sampling rate and may incur an oversampling of certain tasks. An optimal redundant task execution policy would invoke the precise number of instances for each task, thereby preventing oversampling.

To tackle this challenge, we first formulate an optimization problem as follows:

$$\underset{x_i}{\text{minimize}} \quad t = \sum_i x_i e_i + \frac{t}{p_i} e_i \tag{4.1}$$

$$\text{subject to} \quad y_i(t) = k - \frac{t}{p_i} - x_i \leq 0, \quad y_i(0) = k, \quad \forall i, \tag{4.2}$$

where t represents the time to acquire at least k samples for each task, x_i is the decision variable that represents the number of redundant instances for task i, and $y_i(t_0)$ is the number of remaining samples at time t_0. e_i and p_i are the WCET and period of task i, respectively. Given time t, y_i calculates the number of remaining samples by subtracting (i) the number

of instances by the natural execution rate (t/p_i), and (ii) x_i from the total minimum required samples (k). In summary, the optimization problem finds the values of x_is that minimize t such that $y(t) \leq 0$.

Nonetheless, the formulation encounters several challenges. The optimization solvers are often not computationally scalable, which can become problematic when dealing with a growing number of variables. More seriously, the assumption of WCET e_i tends to be overly pessimistic with the actual execution time distribution, resulting in a conservative value of y_i and an increased value for t. Therefore, inspired by the formulation, we devise a heuristic to mimic the optimal behavior. Algorithm 4 provides a pseudo code for an EDF scheduling

Algorithm 4 Pseudo code - EDF scheduling with FRTS redundant execution

Require: EDF scheduler (*edf*), current time (t), safety-critical tasks (*sched_tasks*), redundant task set (*red_task*), max. deadline (MAX_DL), min. required samples (k)
Ensure: Next task to run (*next_task*)

```
1: for task ∈ sched_tasks do
2:     if task.deadline < t then
3:         task.start_new_job()
4:         task.set_deadline(t + task.p)
5:         edf.submit(task)
6:     end if
7: end for
8: for task ∈ red_tasks do
9:     if task.finished() == True and task.y < k then
10:        task.start_new_job()
11:        task.set_deadline(MAX_DL − task.p ∗ task.y)
12:        edf.submit(task)
13:    end if
14: end for
15: next_task = edf.schedule()
16: return next_task
```

approach that incorporates a redundant execution policy. For **lines 1-7**, safety-critical tasks are submitted to the EDF scheduler if their period has ended. In **line 4**, a deadline is set with respect to the task period and the current time t. Redundant tasks (*red_tasks*) are

managed separately in **lines 8-14**. The difference from the case with safety-critical tasks is *if* condition in **line 9**. For *red_tasks*, the algorithm checks if the previous job has finished instead of monitoring the period end. This approach ensures seamlessly task submission and maximizes the utilization of available slack time. Additionally, the algorithm also checks whether the required number of sample has been collected, thus preventing the submission of unnecessary redundant tasks. The central concept of the heuristic is encapsulated in **line 11**. By setting the task deadline near MAX_DL, the safety-critical tasks are given higher priority, while enabling scheduling between redundant tasks. Therefore, redundant tasks can only be executed when there are no safety-critical tasks in the scheduling queue. When the deadlines are the same for more than one task, the EDF scheduler will pick one randomly to break a tie. The heuristic favors the redundant task with a larger task period and a greater number of remaining samples to collect. Finally, in **line 15**, the EDF scheduler selects the task with the earliest deadline to be executed next. In the evaluation section, various heuristic policies will be compared with the optimization problem approach.

4.4.3 Scheduling parameter optimization & Safe reconfiguration

Once a sufficient number of samples collected for each task, the GEV parameters will be estimated and verified by the GoF test. Then, the system acquires the final pWCETs that account for the impact of faults. With the new WCETs, the P&S optimization described in Section 3.3 employs geometric programming techniques to identify the best period and speed factor for each task, aiming to reduce energy usage. The optimization process relies an iterative interior-point method to incrementally approach to the optimal solution. This can be time-consuming if the initial estimates for the variables are significantly off from their optimal values. Under the assumption that a fault would not dramatically shift the optimization parameters, we can use the previously identified optimal point as an initial

guess for the optimization solver (i.e., *warm start*). Then, the optimization solving time will be significantly shorter compared to the design-time optimization if the solver starts searching for the solution from the previous operating point.

For timing safety during transitioning to a new operating point, both the system utilization bound and end-to-end deadline are considered. In emergency mode, the end-to-end deadline is satisfied as long as each task is finished before their implicit deadlines. In that case, the total system utilization is lower than the bound as there is slack time. However, the reconfiguration occurs task-wise and the instant utilization can cause the system utilization to exceed the upper bound. Therefore, the P&S optimization should add the constraint of *utilization invariability* that limits the task utilization to be unchanged. This constraint might affect the solution's energy efficiency. However, its impact has been shown to be negligible, as evidenced in Fig. 3.12d from 3.6.6.

4.4.4 Evaluation Metric: Survivability

The primary objective of the FRTS framework is to improve the survivability in presence of faults by timely reconfiguration of the system with new optimized scheduling parameters that better utilize the system. The survivability in the literature is typically measured by *fail operational (FO)* and *fail robust (FR)* counts [22]. Both metrics assess the system's ability to tolerate a certain number of job overruns without leading to dropped jobs or missed deadlines. The two metrics are aggregated into a *job failed (JF)* count, which then guides the system in determining the severity of faults and subsequently initiating appropriate fault-tolerant techniques at different levels. However, FRTS is designed to adapt to the faults by quantifying the severity of faults in terms of changes in WCET, necessitating a more coarse-grained measurement of survivability. In the evaluation section, alongside JF

counts, the analysis also aims to identify the maximum allowable WCET change that can be accommodated by a fault-tolerant technique, thereby simulating different levels of fault severity.

4.5 FRTS Overhead Analysis: System Utilization

Real-time systems adhere to rigorous schedules determined by timing analyses. Therefore, periodically running multiple i.i.d tests should be considered in the design as well. We evaluate the overhead of our proposed detector with a real-time system described in Section 2.1.1.

From the two constraints of P&S optimization, the end-to-end deadline and the system utilization, the FRTS tasks do not contribute to the end-to-end deadline (Eq. 3.6). Therefore, only the system utilization should be re-distributed over all tasks, including FRTS ones. The revised constraint is

$$\sum_{i=1}^{n} \frac{e_i}{p_i s_i} + \sum_{i=1}^{m} \frac{e_i^d}{f_i^d p_i^d} \leq U_{upper} = 100\%, \tag{4.3}$$

where m represents the number of FRTS tasks, which include i.i.d tests, estimation, GoF, and optimization for each original task, while e^d represents the WCET of those FRTS tasks. $f^d p^d$ represents the WCET and the effective period of FRTS tasks, where it implies that for every f^d executions of the corresponding original task, the FRTS task is executed once. To avoid oversampling, p^d is set equal to the period of the corresponding task. We assume FRTS tasks always run at maximum frequency for simplicity. As a result, the utilization of the original tasks will decrease compared to when FRTS was not considered. Then, each task will have either (i) an increased period, (ii) an increased speed factor (CPU frequency), or (iii)

a combination of both. However, the period is bounded by the end-to-end deadline, making it more likely the speed factor to increase. According to the average power consumption (Eq. (3.4)), the system now consumes more energy at an increased CPU frequency to meet the same end-to-end deadline. Therefore, the overhead of our framework also can be translated into an increased energy consumption. However, as we will demonstrate in the following experiments, this overhead is small and amortized by the larger system energy savings achieved by BoostIID.

4.6 Evaluation

4.6.1 Experimental Setup

We measured the execution time of tasks in Table 4.1 and the i.i.d tests on a NVIDIA Jetson TX2 platform: the pWCET for KPSS, RS, and Ljung-Box are 1.16 ms, 0.63 ms, and 1.03 ms, respectively. We used the Chauffeur benchmark suite[65] for self-driving vehicles, which implements the representative automotive tasks from the Bosch WATERS industrial challenge[47]. The i.i.d tests are from Chronovise 1.0, a C++ framework for MBPT [70]. The estimation of GEV parameters was done by Chronovise, using Maximum Likelihood Estimation (MLE) method on 500 measurements with a BM block size of 5. When randomly modifying GEV parameters, a uniform distribution is used in a range from 0 to 200% of its original value. MATLAB 2022b was used to generate random samples from various GEV distributions. The pWCET estimation was performed using Chronovise with a probability of 10^{-4} and cross-checked with MATLAB's gevcdf. Finally, the boosting was performed on 100,000 datasets using XGBoost 1.7.6 python package. For hyperparameter tuning for each task, we used GridSearchCV from scikit-learn 1.2.1 to find best values for

max_depth, min_child_weight, gamma, subsample, colsample_bytree, and *reg_alpha*. The rest of the training parameters were set to default.

For evaluation of FRTS framework, we have implemented a simulator with EDF scheduler equipped with redundant execution policies. The simulator executes 10 tasks from WATERS DAG with task periods and speed factors for a designated duration of time, and can be configured to simulate using only WCETs, or random execution times drawn from GEV distributions. The FRTS tasks – i.i.d tests, GEV estimation, P&S optimization, and GoF test – utilizes the ones from Chronovise framework. We measure the system's fault resilience, in addition to average energy consumption in three phases – pre-fault, reconfiguration, and post-fault – and compared them to a EDF-Virtual Deadline (EDF-VD) method with DVFS optimization from [54]. In EDF-VD approach, a virtual deadline is set for each task, which triggers a utilization of maximum CPU frequency to mitigate the fault impact. The deadline shortening parameter x is multiplied to the original (implicit) deadline. In this study, we used two configurations for the value of x, 0.5 and 0.75.

4.6.2 BoostIID Evaluation

Fig. 4.5 presents the detection evaluation of BoostIID on six representative automotive tasks (EKF, HybridA*, FLOAM, LaneNet, Darknet, SFM) shown earlier in Table 4.1. The x-axis represents a pWCET increase of a randomly modified GEV distribution from the original one, while the y-axis represents the count of additional samples for BoostIID to detect statistical distribution changes. The latency limits are set equal to the ones in Fig. 4.3, the number of periods within the end-to-end deadline. Throughout the x-axis, the data points are generally moved downwards, indicating a decrease in latency overall. As a result, the safety margin (solid vertical line) beyond which latencies are maintained below the limit

(horizontal dashed line) has decreased.

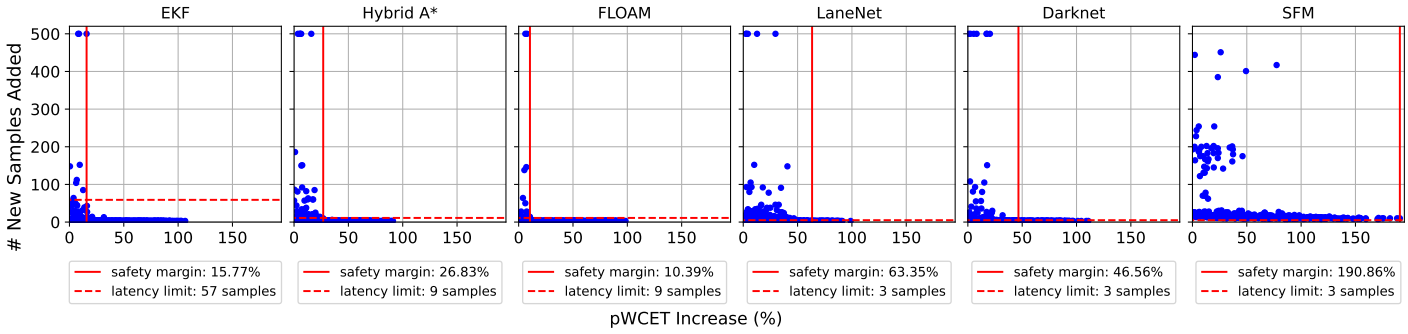


Figure 4.5: Detection accuracy and latency of BoostIID

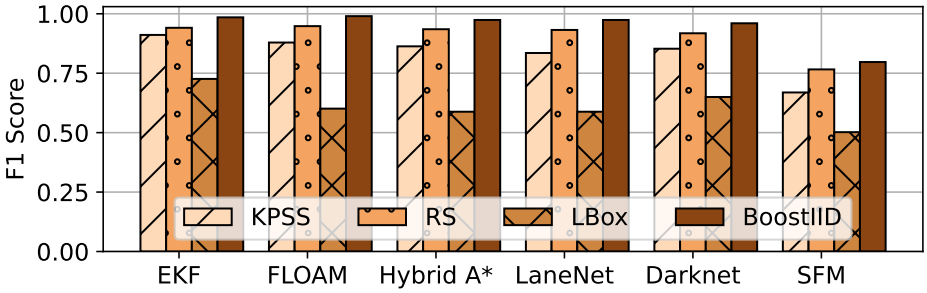


Figure 4.6: F1 scores for various detector configurations

The safety margin will be added to the fault-free WCET when performing a single-mode P&S optimization in Section 3.3.1, as it is the blind spot for our detector. While SFM displays a notable visual improvement, most data points still exceed the latency limit. This is because SFM execution times exhibit a high standard deviation, requiring a relatively large number of samples with the current set of i.i.d tests to effectively distinguish the change in statistical distribution. Fig. 4.6 shows the classification accuracy of BoostIID and the individual i.i.d tests for the same automotive tasks. The y-axis shows the F1 score described in Section 4.3.3, derived using randomly selected 20% of the 100,000 input datasets. KPSS and RS demonstrates strong F1 scores ranging from 0.85 to 0.95 except for SFM, with some false positive cases which are mostly masked with the safety margin. On the other hand, Ljung-box (LBox) exhibits the lowest F1 scores around 0.6, generating numerous false negative cases due to its excessive sensitivity. BoostIID outperformed the other tests with F1 scores

ranging from the peak of 0.99 for FLOAM to a minimum of 0.96 for Darknet, excluding SFM at 0.80. In the case of SFM, the accuracy is poor even with the boosting, allowing us to revert to the fault-aware method.

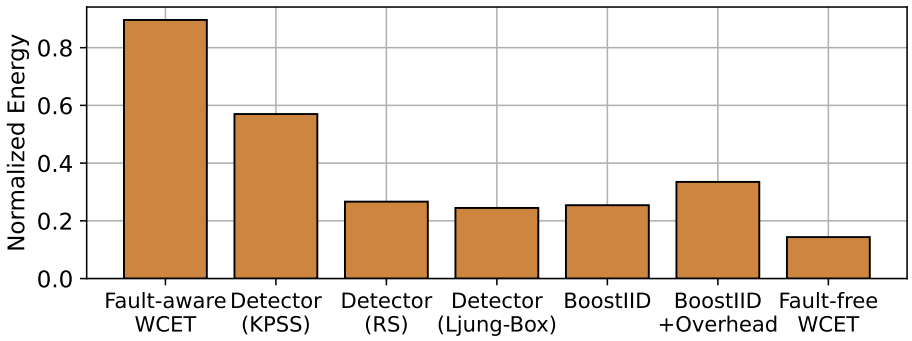


Figure 4.7: Comparison of average dynamic power consumption on a real-time system with various WCET estimations

We measured the average dynamic power consumption on the real-time system described in Section 3.3.1, using Eq. (3.4) after performing P&S optimization with the safety margins obtained from our analysis. In Fig. 4.7, the average power consumption is normalized with respect to the system's maximum power consumption, representing normalized energy consumption in a unit time. The first bar, fault-aware WCET, assumes that the WCET has doubled for all the tasks, equivalent to the safety margin of 100%. The fault-aware WCET method operates at near maximum CPU frequency to meet the same deadline, drawing 89.6% of the maximum energy consumption. The last bar represents the system with fault-free WCET which consumes only 14.4% of the system's maximum energy consumption, contrasting the energy efficiency of the fault-aware WCET method. The following three detectors that configured with a single i.i.d test show a significant reduction in energy consumption by lowering the safety margins. Among i.i.d tests, KPSS exhibits the worst performance at 57.0%, followed by RS and Ljung-box at 26.65% and 24.47%, respectively. BoostIID achieved 25.42% of the maximum energy consumption, showing 71.2% reduction from the fault-aware one. While Ljung-Box demonstrates the lowest energy consumption,

this is attributed to its high sensitivity as discussed before, rendering it undesirable for standalone use. In sum, boosting led to higher accuracy in consistent with the highest F1 score, while also achieving a slight improvement in energy efficiency. When the detector overhead is considered, other tasks run at a slightly higher CPU frequency, resulting in an increased energy consumption at 33.49% of the maximum. Nevertheless, BoostIID shows energy reduction of 62.6% over the classical fault-aware approach, demonstrating amortization of the detector's energy overhead.

4.6.3 FRTS Redundant Execution Evaluation

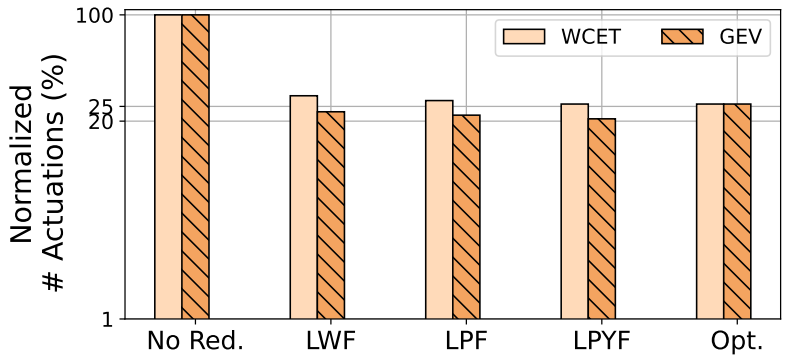


Figure 4.8: Comarison of various redundant execution policies

Utilizing redundant execution is a method to harness any unutilized slack within the system, aiming to expedite the sampling of execution times during the emergency mode. Consequently, a reduction in the duration of the emergency mode leads to decreased energy consumption. Given that the maximum CPU frequency is being employed, the energy consumption is directly linked to the length of the emergency mode. In Fig. 4.8, different redundant execution policies are compared based on the number of actuations, a platform-independent metric that quantifies time. That is, the count of actuation decisions from the DASM task in Fig. 2.1 is represented on the y-axis when all of the desired number of samples for each task have been collected. Two sets of simulations were conducted. The

first set examined cases where tasks consistently displayed their WCET during execution (namely, *WCET configuration*), which is useful to check the optimality of the formulation from Section 4.4.2. In the second set of simulations, task execution times were randomly drawn from their GEV distribution (namely, *GEV configuration*) to simulate the normal operational conditions, which is useful to compare average energy consumptions. The policies are implemented by substituting Line 11 from Algorithm 4 with the following strategies: (i) Longest-WCET-First (LWF), (ii) Longest-Period-First (LPF), (iii) Largest-Period*Y-First (LPYF), and (iv) optimization formulation. When redundant execution is absent, there is no variation in the collection time between the two sets of simulations, as the sample acquisition rate for each task remains constant due to their fixed task periods. In the WCET configuration, *Opt.* demonstrates the optimal performance, closely followed by the *LPYF* policy, which achieved near-optimal results at 99.9% of the *Opt.* solution. However, the *Opt.* policy exhibits the poorest performance in the GEV configuration. This is attributed to the fact that, in the *Opt.* case, the number of redundant executions for each task is predetermined during the design phase and is calculated based on their WCETs. In contrast, the other policies are dynamic in nature such that a redundant task can cease execution prematurely once the desired number of samples has been collected. This adaptability effectively counteracts the randomness of execution times. In result, the *LPYF* policy achieved the highest 79.3% reduction in the duration of emergency mode. It is worth noting that the disparity between the WCET and GEV configurations might have been more pronounced if a lower value for the pWCET probability had been used, thus further increasing the gap between the randomly sampled execution time and the WCET.

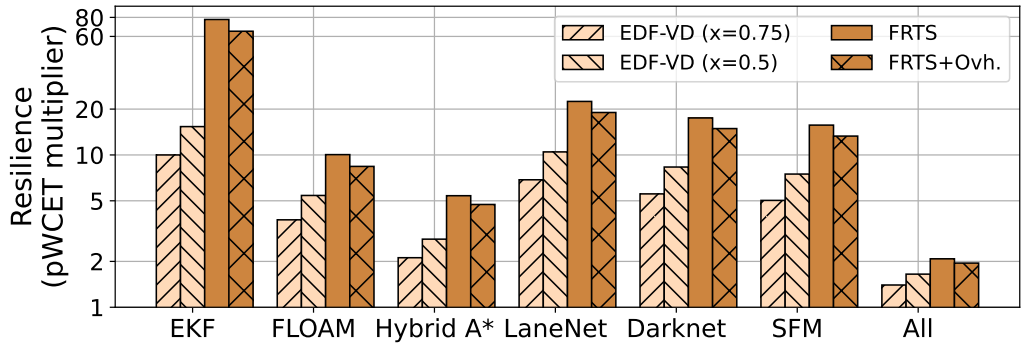


Figure 4.9: Resilience comparison between EDF-VD and FRTS using representative automotive tasks from WATERS DAG

4.6.4 FRTS Resilience Evaluation & Utilization Overhead

We use a soft real-time system, where implicit deadline violations are tolerable as long as the end-to-end deadline is kept, which is vital for ensuring the timing from sensors to actuators. Fig. 4.9 depicts the resilience for such a system with comparison between the EDF-VD approach and the FRTS framework. Two configurations are used for EDF-VD: $x = 0.75$ configuration triggers a switch to the maximum CPU frequency when the task execution time exceeds 75% of its WCET, while the $x = 0.5$ configuration does so at 50%. Therefore, $x = 0.5$ consumes more energy but is expected to exhibit a higher resilience against more severe degree of task overrun. The resilience was measured by gradually increasing the WCETs for affected tasks until the system starts to exceed an end-to-end delay threshold, which is set equally for all the cases in Fig. 4.9. The simulator was configured for this experiment such that a job always takes its WCET amount of time, simulating the worst-case behaviors. The y-axis is represented on a logarithmic scale, indicating the normalized maximum increase in WCET, relative to the fault-free value of each task. The x-axis displays the scenario where JF= 1, indicating a single job failure (overrun), with six representative automotive tasks, followed by JF= 10, where it assumes all tasks were affected by the fault. Across all configurations, the FRTS approach exhibited the highest resilience, even when considering

the overhead. This was followed by EDF-VD approaches, the $x = 0.5$ configuration and then the $x = 0.75$ one, as expected. This divergence in resilience can be attributed to the FRTS's capability for timely reconfiguration, whereas EDF-VD is constrained by fixed task periods, which in turn sets a lower limit on the possible end-to-end latency. In JF= 1 configuration, the FRTS approach demonstrated a resilience improvement of x2.14 and x3.18 compared to the $x = 0.5$ and $x = 0.75$ configurations, respectively. In JF= 10 configuration, the FRTS approach achieved 18.2% and 39.3% resilience improvement against the $x = 0.5$ and $x = 0.75$ configurations, respectively. The disparity in resilience improvement between the two JF configurations can be attributed to the availability of slack in the JF= 1 case, which the FRTS can leverage – unlike in the JF= 10. We designed an experimental setup where the target system is powerful enough to afford doubled WCETs of all tasks while maintaining the same end-to-end deadline. The FRTS approach can efficiently reconfigure the system close to this limit, which is 1.95 when accounting for overhead as can be seen in Fig. 4.9. The limited improvement observed in the JF=10 configuration is attributable to this constraint. Finally, the resilience of the system with FRTS can be quantified as JF=1x20.9 and JF=10x1.95, while JF=1x8.3 and JF=10x1.65 for the system with EDF-VD x=0.5.

4.6.5 FRTS Warm-start Scheduling Parameter Optimization

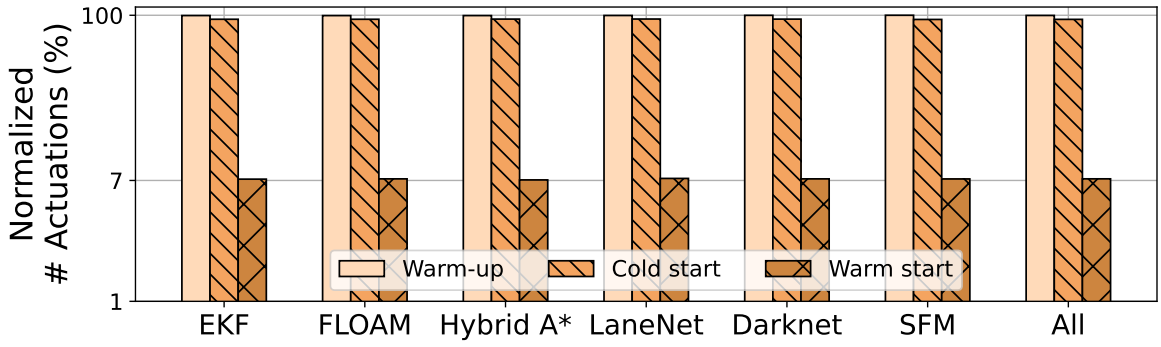


Figure 4.10: Evaluation of warm start in P&S optimization

Under the assumption that a fault would not dramatically shift the optimization parameters, we can use the previously identified optimal point as an initial guess for the optimization solver, which is commonly referred to as a *warm start*. Utilizing a warm start can substantially decrease the number of iterations needed within the solver, thereby reducing the overall time required for solving. In Fig. 4.10, we evaluated the efficacy of the warm start approach by measuring the average optimization runtime over 1,000 runs. This was followed by an evaluation under the JF=10 condition. The measurements were obtained on the NVIDIA TX2 target platform, where we used the Python CVXPY package for solving, as MATLAB was not available on this platform. The experimental results starts with JF=1 scenario, in which a fault was injected into a single task. *Warm-up* refers to the initial run measurement with no fault injection for a fair comparison by eliminating the cold cache effect when solver is first initialized. *Cold start*, in contrast to *warm start*, refers to the runtime measurement of the optimization with no fault injection when the initial guess is made by the solver, which may be far from optimal solution. In the *Cold Start* scenarios, we observed an average 5.86% reduction in the number of actuations compared to the *Warm-up* cases. In the *Warm start* cases, we intentionally introduced the most severe fault for each task, based on our resilience experiment findings, with the goal of distancing the initial guess from the optimal solution. Regardless of the task or fault severity, *Warm start* approach achieved an average reduction of 92.51% in optimization time.

4.6.6 FRTS Energy Overhead Evaluation

In Fig. 4.11, the average energy consumptions during three phases are displayed: *pre-fault*, *system reconfiguration*, and *post-fault* phase. The average energy consumption pertains to normal operations, hence the tasks exhibit random execution times following a GEV distribution throughout the simulation. Two levels of fault severity, x1.25 and x1.5, are applied in

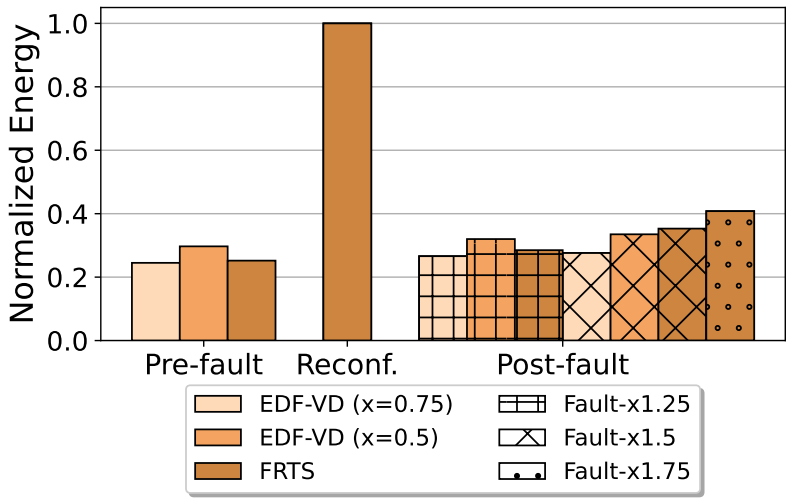


Figure 4.11: Average energy consumption comparison between EDF-VD and FRTS in pre-fault, reconfiguration, and post-fault phases

conjunction with JF=10, affecting all tasks uniformly. For *pre-fault* and *after-fault* phases, the energy consumption was measured for 1,000 seconds in simulation time, equivalent to 33,000 actuation decisions, while *reconfiguration* phase took 4,820 and 7,400 actuation decisions for fault severities of x1.25 and x1.5, respectively. During the *pre-fault* phase, the energy consumption of EDF-VD decreases with increasing values of the deadline shortening parameter x increases. This occurs because, statistically, fewer jobs exhibit execution times that exceed WCETx0.75 compared to those exceeding WCETx0.5, leading to less frequent utilization of the maximum CPU frequency. The system with FRTS operates normally with BoostIID, consuming a similar amount of energy to EDF-VD with $x = 0.75$, yet with much higher resilience as in Fig. 4.9. In the *system reconfiguration* phase, the system enters the emergency mode, during which it collects execution time samples, performs GEV parameter estimation, conducts the GoF test, and optimizes the scheduling parameters (P&S optimization). This is followed by a safe mode change to a new operating point. All of these reconfiguration processes run at the maximum CPU frequency, resulting in the highest average energy consumption, but they occur over a short burst of time. During the post-fault phase, EDF-VD frequently switches to a maximum CPU frequency than before, as the fault

severity becomes more pronounced. The FRTS approach consumes less energy at a fault severity of x1.25 compared to x=0.5, yet it consumes more energy at a fault severity of x1.75. This can be attributed to the fact that the execution time of BoostIID is also impacted by the fault. Despite the brevity of individual execution times, the three i.i.d tests in BoostIID are executed after every job, resulting in significant overhead under conditions of high fault severity. Thus, minimizing the utilization overhead of FRTS could enhance energy efficiency in the post-fault phase. Strategies might include reducing the detection frequency of BoostIID or isolating its execution from faults by employing separate hardware. Finally, only the FRTS approach could withstand a fault severity level of x1.75, albeit at the cost of increased energy consumption.

4.7 Summary

We introduced FRTS: Framework for Resilient Timing Safety against WCET Changes in autonomous driving. Our research is driven by the emerging trend of extended lifespans for autonomous vehicles and the accompanying increase in permanent fault hazards in computer components. The stringent timing safety guarantees, predicated on design-time WCET analysis, are jeopardized by permanent system faults. Traditional detection methods, which target specific faults, evaluate potential fault impacts through targeted analyses and incorporate these assessments into the WCET as a safety margin. These approaches thereby result in limited coverage and energy inefficiency after tolerating faults. Classical fault-tolerant techniques employ this safety margin in high-criticality modes to maintain end-to-end delays below deadlines. However, after tolerating faults, the fixed scheduling parameters determined during the design phase impose a lower limit on end-to-end delays, thereby preventing the hardware from being utilized to its full potential. To address these issues, FRTS leverages

the safety margin (or slack time) to reevaluate the WCET and subsequently reconfigure the system, thereby achieving resilient timing safety. The detection module in FRTS, BoostIID, proactively monitors statistical changes in execution times aiming to detect safety hazards before actual violation happens. By concentrating on the end-effects of faults, namely changes in execution time, BoostIID exhibits a fault-agnostic feature that mitigates the cost and effort required for fault-specific analyses. The limitations of BoostIID in fault tolerance are complemented by the reconfiguration capabilities of the FRTS enabled system. During emergency mode, when the maximum available slack time is generated, FRTS samples execution times to determine new, safe scheduling parameters that fully exploit the system's capabilities. Thanks to its timely reconfiguration, the FRTS system demonstrates an improved resilience against WCET changes, showing a 39.3% compared to the state-of-the-art fault-tolerance technique, EDF-VD with x=0.75. To the best of our knowledge, FRTS is the first attempt to adjust scheduling parameters in real-time for resilient timing safety.

The contributions of FRTS can be summarized as follows:

- BoostIID proactively detects the end-results of permanent fail-slow faults, reducing safety margin to improve energy efficiency during normal operations
- BoostIID combines multiple i.i.d tests with a boosting technique to improve detection accuracy and latencies.
- FRTS introduces a method for assessing fault impact in real-time and dynamically reconfiguring the system to achieve resilient timing safety.
- The duration of emergency mode is minimized with our near-optimal sample acceleration heuristic.
- The overhead of FRTS in terms of system utilization and energy consumption are

evaluated.

Expanding upon the core functionalities of FRTS, our future plans include enhancing the system to support a broader range of fault types beyond just fail-slow, as well as integrating recovery techniques such as checkpointing and reexecution. The overhead incurred during the reconfiguration phase can be mitigated by incrementally reconfiguring the system on a task-by-task basis. The overhead incurred during the post-fault phase raises the question of *"Who guard the guards?"*[69] as the fault-tolerant workloads themselves affected may be compromised by the fault. This concern extends beyond energy efficiency, as the correctness of the fault-tolerant workloads may also be compromised. A viable solution is to insulate the FRTS workloads from faults by utilizing dedicated hardware.

Chapter 5

Conclusions

Autonomous machines are increasingly important as they evolve to manage tasks offloaded from humans, performing them with capabilities that surpass human abilities. Particularly when a failure could lead to catastrophic outcomes, the safety guarantees in these machines become increasingly critical. Numerous studies have been conducted in the field of real-time systems to address the safety issues of these machines. Real-time systems have been successfully utilized in a variety of applications to perform safety-critical jobs, including but not limited to avionics, surgical robots, and spaceflights. However, evolving workloads and operating conditions introduce additional challenges. Autonomous driving epitomizes the complexity of modern real-time systems, characterized by tasks that are increasingly demanding in both performance and volume. These vehicles are equipped with an extensive array of sensors to navigate through an unprecedented number of hazards, including moving objects and obstacles, all while operating on battery power. These new environmental dynamics introduce intriguing and novel challenges, such as dynamic deadlines. These evolving dynamic operating conditions amplify the gap between worst-case scenarios and normal operations, thereby escalating design difficulty, costs, and energy inefficiency. Furthermore, the anticipated proliferation of autonomous vehicles stands to intensify existing

energy challenges, given the current energy consumption patterns of electric vehicles. Lastly, the extended lifespan of vehicles also increases the system's exposure to a variety of permanent faults, such as the fail-slow condition of storage devices.

Inspired by mixed-criticality systems[77], the implementation of distinct operational modes and safe transitions between them could serve to mitigate these challenges. This dissertation presented two frameworks that employ adaptive system reconfiguration for energy efficiency and for resilient timing safety. EASYR focuses on dynamic deadlines imposed by the velocity of the ego vehicle and the distance from the front vehicle. By timely utilization of predefined discretized modes that are individually optimized for energy and come with guaranteed latency, EASYR enables the system to adaptively consumes only the amount of energy necessary for the situation. FRTS concentrates on the extended lifespan of vehicles and empowers the system to counteract unpredictable execution time changes at run-time that could compromise safety guarantees established at the design phase. A novel fault-agnostic detector for WCET changes, known as BoostIID, has been introduced as a submodule within FRTS. By proactively identifying potential timing hazard, FRTS seeks to transition the system into emergency mode before any actual violation occurs, consequently facilitating the assessment and counteraction of fault impact. As a result, FRTS enhances the system resilience against WCET changes compared to classical fault-tolerance techniques.

As such, enabling adaptive system reconfiguration offers many advantages for safety-critical real-time systems. Moving forward, EASYR and FRTS open up a variety of intriguing avenues for exploration. Rather than relying solely on a simple dynamic deadline based on the vehicle's velocity and proximity to objects, there could be a multitude of different dynamic deadlines imposed on the vehicle. For instance, the vehicle could be equipped with multiple cameras in various locations. The end-to-end deadline for the front camera may vary from that of the rear camera. In fact, this is contingent upon the specific driving

context, such as whether the vehicle is on a highway, in a downtown area, or in a parking lot. In such contexts, factors like the number of objects, their directions of movement, and their acceleration and velocities become significant considerations. These factors are elegantly formalized in NVIDIA's Safety Force Field [68] for collision avoidance, which calculates the available set of actions for the vehicle based on the surrounding environment. It may be possible to learn dynamic deadlines for each driving context to optimize for energy efficiency using a photorealistic simulator such as CARLA [39]. For BoostIID, the level of proactiveness could be enhanced by delving deeper into its architecture and applying advanced anomaly detection techniques. Even before a fault occurs, an anomaly technique using periodic state machines and trace analyzer could be employed to monitor hardware properties. This would allow BoostIID to adaptively select the appropriate set of i.i.d tests or modify the safety margin, therefore reducing the overhead in system utilization. FRTS could be adapted to manage multiple ongoing faults that maintain the system in emergency mode for an extended duration. To achieve this, task-level optimization is essential rather than system-wide adjustments. The challenge of task-level optimization lies in managing the uncertainties associated with the impact of a fault on other tasks that have not yet been identified. A thorough analysis is essential to determine an energy-efficient safety margin under those circumstances. Additionally, the current FRTS system is resilient up to the level of faults that the system can feasibly manage given tasks, scheduler, and hardware. Should a more severe fault arise, the FRTS system could increase its resilience by opting for alternative tasks. For instance, certain DNN frameworks or models may be less impacted by specific types of faults compared to others, while still maintaining a comparable level of accuracy. If a vehicle is permanently relocated to an area with more severe weather conditions, the system can reduce its dependence on LiDAR sensors, which are vulnerable to rain and snow. Instead, it can prioritize vision-related tasks and reconfigure the system accordingly. Such adaptations could be possible by the run-time reconfiguration capabilities provided by FRTS.

Bibliography

[1] Specification of timing extensions. AUTOSAR Classic Platform 4.3.1.

[2] *Software-Implemented Hardware Fault Tolerance*. Springer US, 2006.

[3] T. F. Abdelzaher, V. Sharma, and C. Lu. A utilization bound for aperiodic tasks and priority driven scheduling. *IEEE Transactions on Computers*, 53(3):334–350, 2004.

[4] A. ahmed Bhuiyan, K. Yang, S. Arefin, A. Saifullah, N. Guan, and Z. Guo. Mixed-Criticality Multicore Scheduling of Real-Time Gang Task Systems. In *2019 IEEE Real-Time Systems Symposium (RTSS)*, pages 469–480, Dec. 2019.

[5] W. Ali, R. Pellizzoni, and H. Yun. Virtual Gang based Scheduling of Real-Time Tasks on Multicore Platforms. *arXiv:1912.10959 [cs]*, Feb. 2020.

[6] W. Ali, R. Pellizzoni, and H. Yun. Virtual Gang Scheduling of Parallel Real-Time Tasks. In *2021 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 270–275, Feb. 2021.

[7] W. Ali and H. Yun. RT-Gang: Real-Time Gang Scheduling Framework for Safety-Critical Systems. In *Proc. IEEE RTAS*, pages 143–155, 2019.

[8] M. A. Awan and S. M. Petters. Enhanced Race-To-Halt: A Leakage-Aware Energy Management Approach for Dynamic Priority Systems. In *23rd Euromicro Conference on Real-Time Systems (ECRTS)*, pages 92–101, July 2011.

[9] H. Aydin, V. Devadas, and D. Zhu. System-level energy management for periodic real-time tasks. In *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*, pages 313–322, 2006.

[10] M. Bambagini, M. Bertogna, and G. Buttazzo. On the effectiveness of energy-aware real-time scheduling algorithms on single-core platforms. In *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*, pages 1–8, Sept. 2014.

[11] M. Bambagini, M. Marinoni, H. Aydin, and G. Buttazzo. Energy-aware scheduling for real-time systems: A survey. *ACM Transactions on Embedded Computing Systems (TECS)*, 15(1):1–34, 2016.

[12] M. Bambagini, M. Marinoni, H. Aydin, and G. Buttazzo. Energy-aware scheduling for real-time systems: A survey. *ACM Trans. Embed. Comput. Syst.*, 15(1), Jan. 2016.

[13] M. Bambagini, F. Prosperi, M. Marinoni, and G. Buttazzo. Energy management for tiny real-time kernels. In *International Conference on Energy Aware Computing*, pages 1–6, 2011.

[14] S. Baruah, V. Bonifaci, G. D'angelo, H. Li, A. Marchetti-Spaccamela, S. Van Der Ster, and L. Stougie. Preemptive uniprocessor scheduling of mixed-criticality sporadic task systems. *J. ACM*, 62(2), may 2015.

[15] R. Basmadjian and H. de Meer. Evaluating and modeling power consumption of multi-core processors. In *2012 Third International Conference on Future Systems: Where Energy, Computing and Communication Meet (e-Energy)*, pages 1–10, 2012.

[16] A. Bhuiyan, D. Liu, A. Khan, A. Saifullah, N. Guan, and Z. Guo. Energy-efficient parallel real-time scheduling on clustered multi-core. *IEEE Transactions on Parallel and Distributed Systems*, 31(9):2097–2111, 2020.

[17] A. Bhuiyan, S. Sruti, Z. Guo, and K. Yang. Precise scheduling of mixed-criticality tasks by varying processor speed. In *Proceedings of the 27th International Conference on Real-Time Networks and Systems*, RTNS '19, page 123–132, New York, NY, USA, 2019. Association for Computing Machinery.

[18] E. Bini, G. Buttazzo, and G. Lipari. Speed modulation in energy-aware real-time systems. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS)*, page 3–10, 2005.

[19] S. Boyd, S.-J. Kim, L. Vandenberghe, and A. Hassibi. A tutorial on geometric programming. *Optimization and engineering*, 8(1):67, 2007.

[20] U. S. bureau of transportation statistics. Average age of automobiles and trucks in operation in the u.s. `https://www.bts.gov`, 2022.

[21] A. Burns, R. I. Davis, S. Baruah, and I. Bate. Robust Mixed-Criticality Systems. *IEEE Transactions on Computers*, 67(10):1478–1491, Oct. 2018.

[22] A. Burns, R. I. Davis, S. Baruah, and I. Bate. Robust mixed-criticality systems. *IEEE Transactions on Computers*, 67(10):1478–1491, 2018.

[23] C. Chen, J. Panerati, and G. Beltrame. Effects of online fault detection mechanisms on probabilistic timing analysis. In *2016 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pages 41–46, 2016.

[24] C. Chen, J. Panerati, I. Hafnaoui, and G. Beltrame. Static probabilistic timing analysis with a permanent fault detection mechanism. In *2017 12th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 1–10, 2017.

[25] C. Chen, L. Santinelli, J. Hugues, and G. Beltrame. Static probabilistic timing analysis in presence of faults. In *2016 11th IEEE Symposium on Industrial Embedded Systems (SIES)*, pages 1–10, 2016.

[26] T. Chen and C. Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 785–794, New York, NY, USA, 2016. Association for Computing Machinery.

[27] T. Chen and L. T. X. Phan. Safemc: A system for the design and evaluation of mode-change protocols. In *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 105–116. IEEE, 2018.

[28] H. S. Chwa, J. Lee, K.-M. Phan, A. Easwaran, and I. Shin. Global EDF Schedulability Analysis for Synchronous Parallel Tasks on Multicore Platforms. In *2013 25th Euromicro Conference on Real-Time Systems*, pages 25–34, July 2013.

[29] comma.ai. driving dataset. `"https://archive.org/details/comma-dataset"`, 2016.

[30] D. Cordeiro, G. Mounié, S. Perarnau, D. Trystram, J.-M. Vincent, and F. Wagner. Random graph generation for scheduling simulations. In *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques*, SIMUTools '10, Brussels, BEL, 2010. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[31] L. Cucu-Grosjean, L. Santinelli, M. Houston, C. Lo, T. Vardanega, L. Kosmidis, J. Abella, E. Mezzetti, E. Quiñones, and F. J. Cazorla. Measurement-based probabilistic timing analysis for multi-path programs. In *2012 24th Euromicro Conference on Real-Time Systems*, pages 91–101, 2012.

[32] C. Da-Ren, C. Young-Long, and C. You-Shyang. Time and Energy Efficient DVS Scheduling for Real-Time Pinwheel Tasks. *Journal of Applied Research and Technology*, 12(6):1025–1039, Dec. 2014.

[33] Dakai Zhu, R. Melhem, and D. Mosse. The effects of energy management on reliability in real-time embedded systems. In *IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 35–40, 2004.

[34] E. V. Database. Energy consumption of full electric vehicles.

[35] A. Davare, Q. Zhu, M. Di Natale, C. Pinello, S. Kanajan, and A. Sangiovanni-Vincentelli. Period optimization for hard real-time distributed automotive systems. In *Proceedings of the 44th annual Design Automation Conference*, pages 278–283, 2007.

[36] A. Dávila and M. Nombela. Sartre-safe road trains for the environment reducing fuel consumption through lower aerodynamic drag coefficient. Technical report, SAE Technical Paper, 2011.

[37] L. Davis. Stability of adaptive cruise control systems taking account of vehicle response time and delay. *Physics Letters A*, 376(40-41):2658–2662, 2012.

[38] Z. Dong and C. Liu. Analysis techniques for supporting hard real-time sporadic gang task systems. *Real-Time Systems*, 55(3):641–666, July 2019.

[39] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun. CARLA: An open urban driving simulator. In S. Levine, V. Vanhoucke, and K. Goldberg, editors, *Proceedings of the 1st Annual Conference on Robot Learning*, volume 78 of *Proceedings of Machine Learning Research*, pages 1–16. PMLR, 13–15 Nov 2017.

[40] N. Feiertag, K. Richter, J. Nordlander, and J. Jonsson. A compositional framework for end-to-end path delay calculation of automotive systems under different path semantics. In *IEEE Real-Time Systems Symposium*. IEEE Communications Society, 2009.

[41] D. G. Feitelson and L. Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and Distributed Computing*, 16:306–318, 1992.

[42] J. H. Gawron, G. A. Keoleian, R. D. De Kleine, T. J. Wallington, and H. C. Kim. Life Cycle Assessment of Connected and Automated Vehicles: Sensing and Computing Subsystem and Vehicle Level Effects. *Environmental Science & Technology*, 52(5):3249–3256, Mar. 2018.

[43] J. Goossens and P. Richard. Optimal Scheduling of Periodic Gang Tasks. *Leibniz Transactions on Embedded Systems*, 3(1):04:1–04:18, June 2016.

[44] M. Grant and S. Boyd. CVX: Matlab software for disciplined convex programming, version 2.1, Mar. 2014.

[45] X. Gu and A. Easwaran. Dynamic Budget Management with Service Guarantees for Mixed-Criticality Systems. In *2016 IEEE Real-Time Systems Symposium (RTSS)*, pages 47–56, Nov. 2016.

[46] Z. Guo, A. Bhuiyan, D. Liu, A. Khan, A. Saifullah, and N. Guan. Energy-Efficient Real-Time Scheduling of DAGs on Clustered Multi-Core Platforms. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 156–168. IEEE, Apr. 2019. 00009.

[47] A. Hamann, D. Dasari, and F. Wurst. Waters industrial challenge. 2019.

[48] D. Hardy, I. Puaut, and Y. Sazeides. Probabilistic wcet estimation in presence of hardware for mitigating the impact of permanent faults. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 91–96, 2016.

[49] S. Heo, S. Cho, Y. Kim, and H. Kim. Real-time object detection system with multi-path neural networks. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 174–187. IEEE, 2020.

[50] S. Herbert and D. Marculescu. Analysis of dynamic voltage/frequency scaling in chip-multiprocessors. In *Proceedings of the 2007 international symposium on Low power electronics and design (ISLPED '07)*, pages 38–43, 2007.

[51] J. Huang, K. Huang, A. Raabe, C. Buckl, and A. Knoll. Towards fault-tolerant embedded systems with imperfect fault detection. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, page 188–196, New York, NY, USA, 2012. Association for Computing Machinery.

[52] K. Huang, L. Santinelli, J. Chen, L. Thiele, and G. C. Buttazzo. Periodic power management schemes for real-time event streams. In *Proceedings of the 48h IEEE Conference on Decision and Control (CDC) held jointly with 2009 28th Chinese Control Conference*, pages 6224–6231, Dec. 2009.

[53] P. Huang, P. Kumar, G. Giannopoulou, and L. Thiele. Energy efficient DVFS scheduling for mixed-criticality systems. In *Proceedings of the 14th International Conference on Embedded Software*, EMSOFT '14, pages 1–10, New York, NY, USA, Oct. 2014. Association for Computing Machinery.

[54] P. Huang, P. Kumar, G. Giannopoulou, and L. Thiele. Energy efficient dvfs scheduling for mixed-criticality systems. In *Proceedings of the 14th International Conference on Embedded Software*, EMSOFT '14, New York, NY, USA, 2014. Association for Computing Machinery.

[55] T. Ishihara and H. Yasuura. Voltage scheduling problem for dynamically variable voltage processors. In *Proceedings. 1998 International Symposium on Low Power Electronics and Design (IEEE Cat. No.98TH8379)*, pages 197–202, 1998.

[56] R. Jejurikar and R. Gupta. Dynamic slack reclamation with procrastination scheduling in real-time embedded systems. In *Proceedings. 42nd Design Automation Conference*, pages 111–116, 2005.

[57] S. Kehr, E. Quiñones, D. Langen, B. Böddeker, and G. Schäfer. Parcus: energy-aware and robust parallelization of autosar legacy applications. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 343–352. IEEE, 2017.

[58] L. Krawczyk, M. Bazzal, R. P. Govindarajan, and C. Wolff. An analytical approach for calculating end-to-end response times in autonomous driving applications. In *10th International Workshop on Analysis Tools and Methodologies for Embedded and Realtime Systems (WATERS 2019)*, June 2019.

[59] S. Lee and S. Nirjon. Subflow: A dynamic induced-subgraph strategy toward real-time dnn inference and training. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 15–29. IEEE, 2020.

[60] S.-C. Lin, Y. Zhang, C.-H. Hsu, M. Skach, M. E. Haque, L. Tang, and J. Mars. The architectural implications of autonomous driving: Constraints and acceleration. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 751–766, 2018.

[61] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.

[62] R. Lu, E. Xu, Y. Zhang, Z. Zhu, M. Wang, Z. Zhu, G. Xue, M. Li, and J. Wu. NVMe SSD failures in the field: the Fail-Stop and the Fail-Slow. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 1005–1020, Carlsbad, CA, July 2022. USENIX Association.

[63] S. Lu and W. Shi. The Emergence of Vehicle Computing. *IEEE Internet Computing*, 25(3):18–22, May 2021.

[64] B. Maity, S. Yi, D. Seo, L. Cheng, S.-S. Lim, J.-C. Kim, B. Donyanavard, and N. Dutt. Chauffeur: Benchmark suite for design and end-to-end analysis of self-driving vehicles on embedded systems. *ACM Trans. Embed. Comput. Syst.*, 20(5s), sep 2021.

[65] B. Maity, S. Yi, D. Seo, L. Cheng, S.-S. Lim, J.-C. Kim, B. Donyanavard, and N. Dutt. Chauffeur: Benchmark suite for design and end-to-end analysis of self-driving vehicles on embedded systems. *ACM Trans. Embed. Comput. Syst.*, 20(5s), sep 2021.

[66] S. R. Nassif, N. Mehta, and Y. Cao. A resilience roadmap. In *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*, pages 1011–1016, 2010.

[67] P. R. Nikiema, A. Kritikakou, M. Traiola, and O. Sentieys. Impact of Transient Faults on Timing Behavior and Mitigation with Near-Zero WCET Overhead. In A. V. Papadopoulos, editor, *35th Euromicro Conference on Real-Time Systems (ECRTS 2023)*, volume 262 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 15:1–15:22, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[68] D. Nistér, H.-L. Lee, J. Ng, and Y. Wang. Nvidia safety force field. 2019.

[69] F. Reghenzani, Z. Guo, and W. Fornaciari. Software Fault Tolerance in Real-Time Systems: Identifying the Future Research Questions. *ACM Computing Surveys*, 55(14s):306:1–306:30, July 2023.

[70] F. Reghenzani, G. Massari, and W. Fornaciari. chronovise: Measurement-based probabilistic timing analysis framework. *Journal of Open Source Software*, 3:711, 08 2018.

[71] T. G. Reid, S. E. Houts, R. Cammarata, G. Mills, S. Agarwal, A. Vora, and G. Pandey. Localization requirements for autonomous vehicles. *SAE International Journal of Connected and Automated Vehicles*, 2(3), Sep 2019.

[72] A. Saifullah, S. Fahmida, V. P. Modekurthy, N. Fisher, and Z. Guo. CPU Energy-Aware Parallel Real-Time Scheduling. In *32nd Euromicro Conference on Real-Time Systems (ECRTS)*, 2020.

[73] Schuette and Shen. Processor control flow monitoring using signatured instruction streams. *IEEE Transactions on Computers*, C-36(3):264–276, 1987.

[74] S. Z. Sheikh and M. A. Pasha. Energy-efficient multicore scheduling for hard real-time systems: A survey. *ACM Trans. Embed. Comput. Syst.*, 17(6), Dec. 2018.

[75] J. Sun, H. Cho, A. Easwaran, J.-D. Park, and B.-C. Choi. Flow Network-Based Real-Time Scheduling for Reducing Static Energy Consumption on Multiprocessors. *IEEE Access*, 7:1330–1344, 2019.

[76] N. Ueter, M. Günzel, G. von der Brüggen, and J.-J. Chen. Hard Real-Time Stationary GANG-Scheduling. In B. B. Brandenburg, editor, *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021)*, volume 196 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 10:1–10:19, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[77] S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*, pages 239–243, 2007.

[78] I. Wind River Systems. Vxworks 653 multi-core edition, Nov. 2022.

[79] J. Wolf, B. Fechner, and T. Ungerer. Fault detection capabilities of an enhanced timing and control flow checker for hard real-time systems. *VALID 2012 - 4th International Conference on Advances in System Testing and Validation Lifecycle*, pages 57–62, 01 2012.

[80] Woonseok Kim, Jihong Kim, and Sang Lyul Min. Preemption-aware dynamic voltage scaling in hard real-time systems. In *Proceedings of the 2004 International Symposium on Low Power Electronics and Design (IEEE Cat. No.04TH8758)*, pages 393–398, 2004.

[81] G. Xie, H. Peng, Z. Li, J. Song, Y. Xie, R. Li, and K. Li. Reliability enhancement toward functional safety goal assurance in energy-aware automotive cyber-physical systems. *IEEE Transactions on Industrial Informatics*, 14(12):5447–5462, 2018.

[82] Yann-Hang Lee, K. P. Reddy, and C. M. Krishna. Scheduling techniques for reducing leakage power in hard real-time systems. In *15th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 105–112, July 2003.

[83] S. Yi, T.-W. Kim, J.-C. Kim, and N. Dutt. Energy-efficient adaptive system reconfiguration for dynamic deadlines in autonomous driving. In *2021 IEEE 24th International Symposium on Real-Time Distributed Computing (ISORC)*, volume ., pages 96–104, 2021.

[84] S. Yi, T.-W. Kim, J.-C. Kim, and N. Dutt. Easyr: Energy-efficient adaptive system reconfiguration for dynamic deadlines in autonomous driving on multicore processors. *ACM Trans. Embed. Comput. Syst.*, 22(3), apr 2023.

[85] Ying Zhang and K. Chakrabarty. Energy-aware adaptive checkpointing in embedded real-time systems. In *Design, Automation and Test in Europe Conference and Exhibition*, pages 918–923, 2003.

[86] M. Zhang, Z. Gu, H. Li, and N. Zheng. Wcet-aware control flow checking with supernodes for resource-constrained embedded systems. *IEEE Access*, 6:42394–42406, 2018.