Dependable and energy-efficient cyber-physical systems by graceful degradation

Dissertation zur Erlangung des akademischen Grades Doktor der Ingenieurwissenschaften (Dr.-Ing.) im Fach Elektrotechnik und Informationstechnik

ROBERT SCHMIDT

1. Gutachter	Prof. Dr.-Ing. Alberto García-Ortiz
2. Gutachter	Prof. Dr.-Ing. Görschwin Fey
Eingereicht am:	2022-02-10
Tag des Promotionskolloquiums:	2022-05-18

Universität Bremen

Für Anka, Ida und Mirta

Abstract

Factory automation, civil infrastructure monitoring, medical wearables, process- and environmental control are essential applications of computer systems in a modern society. By integration of computer systems with the environment, society transfers major trust to such cyber-physical systems, which are required to dependable deliver their expected service. Designing dependable computer systems is a challenge since the inception of the computer, but today's applications limit one major resource required for dependability: Energy.

Where early dependable systems could spend large energy budgets, today's cyber-physical systems are often battery powered and required to be energy-efficient. To enable their widespread adoption in society, we developed a design approach which integrates low-power with dependable system design. We leverage the energy efficiency of modern commodity components by safeguarding them with temporal redundancy.

The resulting cyber-physical systems are energy-efficient and yet dependable, and their real-time guarantees empower the integration of mixed-criticality functions with verifiable quality of service guarantees. Our static, design-time and certification-friendly approach is a breeze for designers and certification authorities, as no assumptions about hardware features and error probabilities are necessary, which allows application- and cost optimal component selection.

German version

Fabrikautomatisierung, Infrastrukturüberwachung, tragbare medizinische Computersysteme sowie Prozess- und Umweltsteuerung sind unverzichtbare Anwendungen von Computersystemen in einer modernen Gesellschaft. Durch die Zusammenführung von Computersystemen mit der Umgebung überträgt die Gesellschaft großes Vertrauen auf solche cyber-physischen Systeme, welche im Gegenzug ihre Dienste zuverlässig bereitstellen müssen. Der Entwurf zuverlässiger Computersysteme ist eine Herausforderung seit der Entstehung des Computers, aber die heute geforderten Anwendungen limitieren eine für Zuverlässigkeit wichtige Ressource: Energie.

Wo erste zuverlässige Systeme noch große Energiebudgets zur Verfügung hatten, sind die heutigen cyber-physischen Systeme oft batteriebetrieben und müssen entsprechend energiesparsam sein. Um eine großflächige Einführung dieser Systeme in der Gesellschaft zu ermöglichen haben wir einen Entwurfsansatz entwickelt welcher energiesparsamen und zuverlässigen Entwurf von Computern vereint. Wir nutzen die Energieeffizienz moderner seriengefertigter Hardwarekomponenten, indem wir sie durch zeitliche Redundanz schützen.

Die damit verwirklichten cyber-physischen Systeme sind energiesparsam und trotzdem zuverlässig, und ihre Echtzeitgarantien ermöglichen die Integration

von Funktionen unterschiedlicher Kritikalität mit verifizierbaren Dienstgüten. Unser komplett statischer und zertifikationsfreundlicher Entwurfsansatz macht es Entwickelnden sowie Zertifizierungsstellen einfach, da weder Annahmen über Hardwarefunktionen noch Fehlerwahrscheinlichkeiten notwendig sind, was eine anwendungs- und kostenoptimale Komponentenauswahl ermöglicht.

Contents

Acknowledgments

This thesis is a product of my journey and the many people of which I had the pleasure to meet during my research activities. I was fortunate to have the opportunities I had, and I am aware of the circumstances which allowed me to write this thesis.

First and foremost, I would never had a chance to begin and end my journey without my loving wife *Ann-Katrin Maetze-Schmidt*, who supported me during the darkest and brightest times, working unfathomable hours with endless love and greatest endurance to take care of our beloved childs and myself. *Ida* and *Mirta*, thank you for reminding me every day why I wrote this thesis, and for your patience with me when I had to write instead of spending time with you.

I am grateful to have met Prof. Dr.-Ing. *Alberto García-Ortiz* during my studies at the University of Bremen, as he inspired and supported me early on. Thanks for all our mutual time discussing and advancing ideas together, your patience, and all your work you invested in creating opportunities for me.

After my graduation I had the great chance to work with Prof. Dr.-Ing. *Görschwin Fey* at the German Aerospace Center, who provided me with room to grow, trust, and guidance. Thanks for connecting me to many bright minds, broadening my horizon by bringing me to many new places around the world, and the freedom to pursue my research activities.

During my time at the German Aerospace Center I was fortunate to work together with *Jan-Gerd Meß*, Dr. *Frank Dannemann*, *Falk Nohka*, *Gökçe Aydos*, and *Carl-Johann Treudler*, from whom I learned a lot and enjoyed creating with day to day. I enjoyed the company and inspiration from talking to all my coworkers at the institute, and thank you all for the great time.

My visit at the Tallinn University of Technology was especially memorable and pleasant by the hospitality of Prof. Dr. *Jaan Raik*, who welcomed and introduced me to his centre for dependable computing systems, Tallinn, and Estonia. Thanks for sharing many ideas with me, and helping me develop my own.

After my return to the University of Bremen I was lucky to work together with *Yanqiu Huang*, *Wanli Yu*, *Lennart Bamberg*, *Amir Najafi*, *Ardalan Najafi*, *Yarib Nevarez* and *Kerstin Janssen* — thank you all for the great time and your support! Amir and Ardalan, thanks for your kindness and calming me down with many sweets and emotional support, the nice barbeques, and your thoughts. Yanqiu and Wanli thank you for teaching me how to eat spicy food without crying, cheering me up, and your inspiring continous progress in research. Yarib, thanks for letting me use your stash of electronic components, teaching me how to use software without going insane, sharing your ideas and stories with me, and beating me at GoldenEye 007 on the N64. Lennart, thank you for our mutual time in our office, the conversations, reflection, your perspective on life and research, and your Netflix account for the movie Ida urged to see. Kerstin, thanks for handling all the administrative burdens, your positive attitude, and keeping

tabs on many details I would have missed.

I am thankful to call *Daniel Gregorek* and *Sebastian Schmale* my friends, and appreciate their continous support, open ears, and many shared moments with me. *Maze*, *Onno*, and *Helge*, without your faithful and loving friendship I would not exist as the person that I am today. Thank you for 23 years of reflection, growth, and joy!

1 The problems of modern fault-tolerant system design

Computer systems integrated with the environment, or cyber-physical systems (CPSs), are increasingly relied upon in monitoring civil infrastructure, electronic health care, aerospace, factory automation, or process- and environmental control [1–6]. Due to their inherent integration with the physical world, dependability is paramount for CPSs [7, 8].

In the taxonomy of dependable and secure computing, dependability describes a general desirable property that includes the attributes of reliability, safety, availability, integrity, maintainability, and security [9]. Dependable computer systems, and especially CPSs, are required to deliver their expected service despite the occurrence of faults [10].

Moreover, CPSs in battery powered internet-of-things (IoT) applications like wireless sensor networks, wearables, and electronic health care devices are required to be energy-efficient, and therefore resort to low-power design techniques [11, 12].

For example, a state of the art low-power design technique emerging in production tapeouts of system-on-chip (SOC) designs for CPSs is near-threshold voltage scaling to safe energy during operation, at the expense of increased performance- and delay variance [13]. Yet delay variance increases the risk for timing violations, which can compromise the circuit's dependability [14].

Although traditional dependable system design weights penalties in power, performance, and system complexity against the consequences of system failure [15], the dependability-to-energy trade-off is not researched to the extend required by CPSs [16].

While traditional dependable systems are developed for applications with higher dependability requirements, CPSs target emerging critical applications. These applications, exemplarily listed in Figure 1.1, differ in their lower budget for dependability improvements, mission durations, and failure costs.

For example, reliability requirements for a rocket are high, due to the high costs of mission failure, which is reflected in the observed reliability of roughly $R = 0.95$ [17]. In contrast, commercial mobile phones fail every 11 days on average [18]. Both rocket and mobile phone represent totally different applications and corresponding systems: Rockets are non-repairable after launch, have short mission durations, high costs of mission failure, and large budgets for reliability improvements. Mobile phones are mostly repairable, are used for longer periods, have negligible failure costs, and smaller budgets for reliability improvements. In between, emerging critical applications like medical in-body CPSs are required to be reliable for longer periods, but do not have as large budgets for reliability improvements as systems in traditional critical applications.

With limited budgets for dependability improvements and severe energy

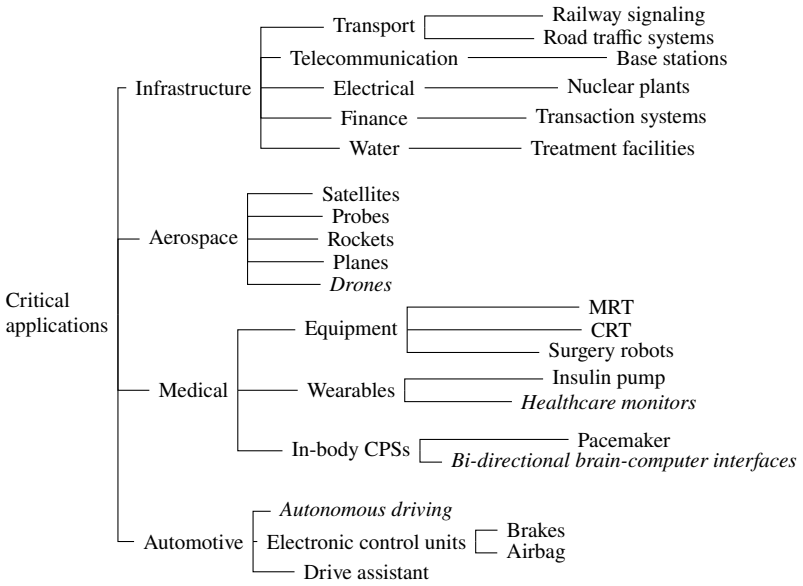


Figure 1.1: Examples for traditional and emerging critical applications. Emerging applications are highlighted with *italic text*.

constraints, CPSs rule out traditional modular redundancy and majority voting approaches, despite their need to detect and correct errors during system operation. Moreover, their energy constraints in combination with limited budgets require that CPSs are developed with commercial off-the-shelf (COTS) components. As CPSs are used for process- and environmental control, they require real-time guarantees, and they have to integrate different functionalities due to economic, size, and weight reasons.

Accordingly, holistic development of CPSs is a major challenge, which requires to integrate real-time and dependable system design under severe constraints, as shown in Figure 1.2. Therefore, instead of spatial redundancy, we propose to resort to temporal redundancy for error detection and correction, in conjunction with a real-time scheduling approach suitable for COTS components, which provides real-time guarantees and supports the integration of different functionalities in our COTS-based CPSs.

The following section provides a precise description of the problem statement, which is picked up in the thesis statement, followed by a description of our approach and methods used during the research activities that led to this thesis.

1.1 Problem definition and thesis statement

For a precise description of the problem statement it is necessary to highlight the relationships between faults, errors, and failures. A fault is an anomalous physical condition which might manifest as an error, in which the logical state of an element differs from its intended value. Failure denotes the inability of an element to perform its designed function because of errors in the element [15].

For CPS, fault tolerance is of utmost importance to avoid failures, as the consequences of failure, or lack of service, in critical applications are dire. But size, weight, and power (SWaP) constraints prohibit traditional spatial fault

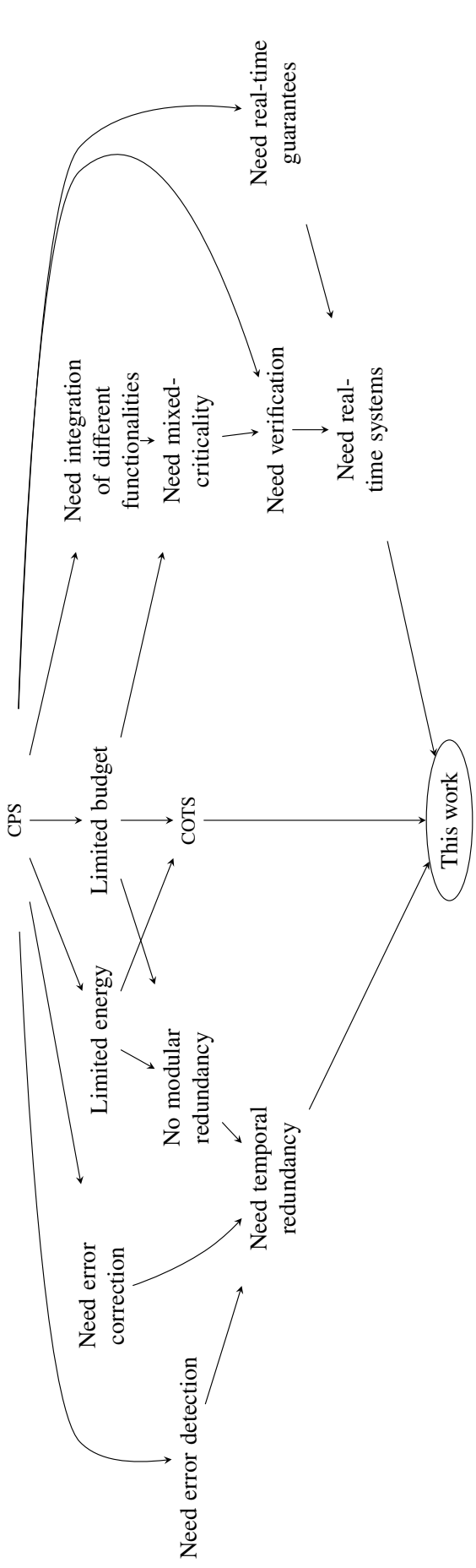


Figure 1.2: Problems in holistic CPSs design. This work picks up all leaf problems stemming from fault-tolerant and low-power real-time system design.

tolerance approaches, while energy- and economic constraints enforce the use of COTS components and integration of different functionalities. This raises the following question:

Problem. *How can we combine dependable, low-power, and real-time system design for COTS-based systems to develop the CPSs required for emerging critical applications*?

We propose that this problem is solvable by design for graceful degradation, which accepts reduced service for an extended operational lifetime:

Thesis statement. *Design for graceful degradation improves the dependability-to-energy ratio of cyber-physical systems, within controlled performance levels, by replacing traditional overdesign for dependability with trade-off capabilities leveraging application margins, to enable the widespread adoption of cyber-physical systems for the benefit of society.*

Our approach to design for graceful degradation, further introduced in Section 1.2, integrates low-power and dependable design, and enables COTS-based fault-tolerant real-time systems which exploit temporal redundancy to provide error detection and correction.

1.2 Approach and methods

Design for dependability and design for low-power are well known oppositional approaches. With the rising demand for energy-efficient and dependable CPSs, traditional approaches in both fields need to be evaluated for their potential to solve the aforementioned challenges.

1.2.1 Integration of dependable and low-power design

Integration of dependable and low-power design by graceful degradation requires error detection. Without error detection, systems lack vital information to evaluate if further operation is safe and reliable, which is required for graceful degradation within application constraints. Furthermore, error detection provides essential information for possible higher-level error correction and containment techniques, fault diagnosis, or failure prediction, to minimize system state corruption and to return to correct state. Because minimization of further state corruption needs to be initiated as soon as possible to prevent system failure, error detection is required to be online, during system operation, and with minimal latency.

Ideally, error detection would be able to detect all relevant types of errors, have perfect coverage, and no design degradation in terms of complexity, performance, or energy. But real-world error detection techniques are likely to degrade performance or energy consumption, and increase the design complexity. Moreover, they might not scale with design size, increased process variations, or the number of detectable errors.

It is not sufficient to consider error detection alone, instead error detection needs to be evaluated in the context of an architecture. Computer architecture research today is driven by the switch to disruptive, emerging technologies like

fin field-effect transistors (FinFETs) and 3D integration, and strong emphasis of machine learning workloads [19, 20]. General purpose computing architectures are required to deliver increasing performance by leveraging available transistors under tight power budgets and thermal constraints. Therefore current architectures strive to exploit massive parallelism, resulting in many-core architectures [21, 22]. While parallelization is essential, it is not a silver bullet because workloads with large sequential calculations do not profit from the potential speedup. Meanwhile, CPSs and the internet of things spur the shift from general purpose to more specialized architectures, resulting in connected, heterogeneous computer networks with high safety and security requirements [7].

Computer architecture affects dependability and power consumption, necessitating thorough consideration of the dependability-to-energy ratio of CPSs. Current evaluations are isolated from architecture and application considerations, and focus on single aspects like communication [23], error detection [24], or noise [25]. Integrating approaches which consider error detection susceptibility to process variations and near-threshold voltage scaling are a step in the right direction [26], and proposed metrics like fault coverage to power are currently investigated to rate different error detection techniques [27]. Nevertheless both studies only focus on self-voting modular redundancy error detection techniques, and verification is limited to fault simulation only. The cross-layer resilience evaluation framework (CLEAR) further advances the systematic evaluation of the dependability versus energy trade-off [28], but targets only general purpose processor cores, rising the need for further investigations in face of specialized architectures common in emerging critical cyber-physical systems.

Considering the aforementioned problems, we propose to integrate dependable and low-power design, systematically evaluate the dependability-to-energy ratio of cyber-physical systems, and test the hypothesis that graceful degradation enables better dependability-to-energy ratios.

1.2.2 Required resources and materials

This thesis requires access to software and hardware for the design, implementation, verification, and evaluation of digital circuits. Software as provided by Europractice and admission to the department's hardware laboratory should satisfy nearly every need. Moreover, a workstation and access to the department's high-performance server architecture are required for simulation experiments.

1.2.3 Data collection and analysis

Data collection is part of all experiments and is considered already during experiment design. Experiments are designed to be reproducible, and are carried out following scientific and laboratory best practices [29]. To aid further research, data is published where feasible, and archived in open formats [30].

Data analysis is also considered during the design of reproducible experiments and is automated with the help of custom-written programs to avoid error-prone manual efforts.

1.2.4 Assumptions and range of validity

This thesis limits assumptions to reduce the risk of erratic research activities. Thorough literature research safeguards the validity of the problem description and hypothesis. Wherever possible dependencies to resources are minimized to limit assumptions about soft- and hardware tool availability. Experimental design, data collection, and evaluation are part of a reproducible workflow to preserve the validity of all obtained results.

1.2.5 Ethics

Serving society, attending to the welfare and progress of the majority, and holding paramount the safety, health, and welfare of the public are recognized values of all professional engineering societies. Therefore engineers are expected to commit themselves „to the highest ethical and professional conduct“ [31]. In research, one particular ethical dilemma is the dual-use problem. Dual-use arises if artifacts of scientific research have „the potential to be used for bad as well as good purposes“ [32]. In electrical engineering, the potential reuse or transfer of knowledge from it's civil domain of origin to military or similar terrorist domains poses such a threat.

While the research activities outlined in this proposal explicitly exclude military applications, the generated knowledge still needs to be examined for its dual-use potential. Therefore ethic self-assessments are suggested to handle dual-use issues if they should arise during the proposed thesis.

1.3 Structure of this thesis

The remainder of this work starts by summarizing the current state of the art approaches and their theoretical background in Chapter 2. Section 2.1 introduces the reader to errors in fault-tolerant systems, including sources of errors, their modeling, and metrics. Next, power dissipation in CMOS and fundamental low-power design approaches are presented in Section 2.3, followed by a detailed introduction and classification of fault tolerance techniques in Section 2.2. Furthermore, graceful degradation is motivated in Section 2.4, and the fundamentals of real-time scheduling are summarized in Section 2.5.

Chapter 3 introduces our solution by a system architecture overview, and Chapter 4 presents our developed tools which we used to design our envisioned CPSs. Chapters 5 and 6 apply these tools and show experimental results, which we discuss and conclude with in Chapter 7.

1.4 Conclusion

The design of CPSs is challenging, as it has to combine oppositional design approaches with conflicting goals. We propose to integrate dependable- and low-power system design into design for graceful degradation, by use of temporal redundancy for error detection and correction. Our graceful design approach works with COTS components, to leverage their benefits in energy consumption

and performance, and allows to integrate functions of different criticality with verifiable quality of service (QoS) and real-time guarantees.

2 Preliminaries

This thesis proposes to integrate fault-tolerant and low-power design for real-time systems to develop graceful degrading CPS. Therefore this chapter provides the necessary theoretical foundation and related approaches which are essential to our work.

The first part of this chapter deals with errors in fault-tolerant systems, explains their origins, and their modeling. After defining the required terminology in Section 2.1, we present the interaction of radiation and semiconductor devices in Section 2.1.1, and how we can model radiation environments, radiation-induced faults, and resulting errors. Next, we introduce fault tolerance techniques, followed by low-power design fundamentals, and motivate graceful degradation, to finally present the state of the art in dynamic real-time scheduling.

2.1 Errors in fault-tolerant systems

To discuss errors and fault-tolerance, we present a set of basic terms first, which are listed in our glossary in Chapter 8 as well. A system interacts with other entities, where these entities could be humans or other systems [9]. The external entities interacting with a system are its environment, and the common frontier between the system and its environment is the system boundary [9].

The intended function of a system, or system function, is described in a functional specification. We call what the system does to implement its system function the system behavior. The system structure enables the system to display its behavior. The system behavior perceived by another system, or user, is the delivered service. The event if delivered service deviates from correct service is a service failure [9].

To understand the relationship between failures, errors, and faults, it is necessary to introduce the concept of state. For CPSs, or more general computer systems, the total state contains the state of computation, communication, stored information, interconnection, and physical condition [9]. The perceivable part of total state at a service interface is the system's external state, and the remaining part is its internal state. We can refine the definition of delivered service with the definition of external state, as delivered service is a sequence of external states [9]. This allows us to refine service failure as well because we can express service failure as a transition from correct- to incorrect service due to a state deviation of one or more external states. The external state deviations are errors, and their causes are faults [9].

Such faults are either active or dormant: An active fault causes an error, and a dormant fault not. Moreover, errors are not limited to external state, as internal state deviations are possible as well. Nevertheless, errors can propagate within a system by computation processes, and once they reach the service interface, they

result in service failures [9], as shown in Figure 2.1 on 11.

If a system, which provides multiple services, experience partial failure, the remaining service leaves the system in a degraded mode [9]. This can be accidentally, or deliberately planned for in the system's specification. We pick up this idea in detail by graceful degradation in Section 2.4.

Faults, errors, and failures are threats to dependability. Dependability is either „the ability to deliver service that can justifiably be trusted“ [9], or „the ability to avoid service failures that are more frequent and more severe than is acceptable“ [9]. As an integrating concept dependability includes the attributes of availability, reliability, safety, integrity, and maintainability [9], as shown in Figure 2.2.

Availability describes the readiness of a system for correct service, and reliability the continuity of correct service [9]. Safety and integrity are the absence of harmful consequences and improper system alterations, respectively. Maintainability is the system's ability to receive modifications and repairs [9]. For CPSs, the absence of unauthorized disclosure of information, or confidentiality, is an additional attribute to consider.

The terminology defined so far allows us to classify faults according to eight viewpoints, as shown in Figure 2.3. Awareness of fault classes enables their inclusion in a dependability specification, and guides the selection of proper fault tolerance techniques [9]. For example, we can differentiate faults by the system's boundary in internal- and external faults: Internal faults originate inside the system, and external faults originate outside the system's boundary and propagate errors into the system [9]. Another classification example is intent: Deliberate faults are the result of harmful decisions, and non-deliberate faults are introduced without awareness [9]. Human-made faults are especially interesting, as they are the result of human actions and absence of required actions. Depending on the objective of the human's interaction with the system, we differentiate non-malicious and malicious faults, where the latter is introduced with the objective to cause harm [9].

To attain the attributes of dependability, we distinguish four major categories of approaches: Fault prevention, fault tolerance, fault removal, and fault forecasting [9]. Fault forecasting estimates the present and future number of faults in a system, and their likely consequences. Approaches which reduce the number and severity of faults in a system are called fault removal approaches. Fault prevention is concerned with preventing the occurrence or introduction of faults, while fault tolerance strives to avoid service failures in the presence of faults [9]. The latter is fundamental to our design of fault-tolerant CPSs, and includes error detection and error handling, as shown in Figure 2.4.

Error detection identifies the presence of an error, which can happen concurrently during operation or preemptive during system suspension [9]. Recovery by error- or fault handling transforms a system state with faults and errors into an usable state which allows restoring delivered service. Fault handling keeps faults from being activated again by diagnosis, isolation, and reconfiguration, where reconfiguration either switches in spare components or distributes work to non-failed components [9]. Reinitialization, as part of fault handling, is concerned with updating the system's record of faults and configuration.

Error handling removes errors from the system state by any combination

Figure 2.1: Error propagation [9]

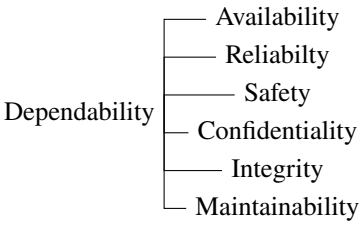


Figure 2.2: The attributes of dependability [9]

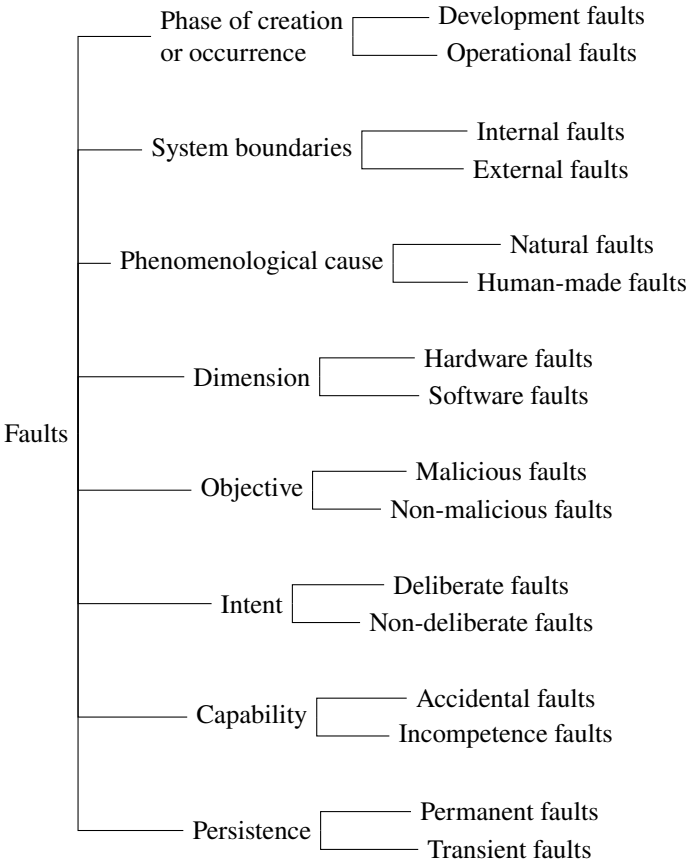


Figure 2.3: Fault classification [9]

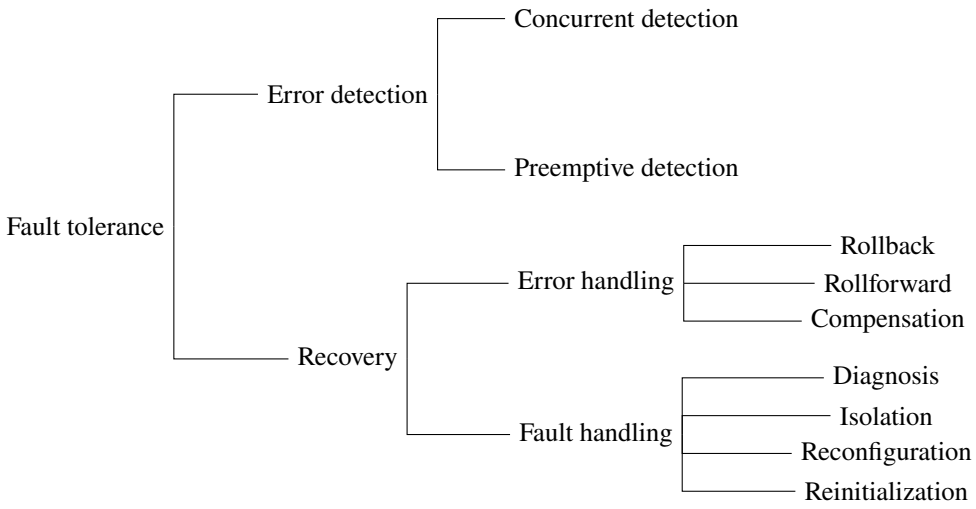


Figure 2.4: Fault tolerance techniques [9]

of rollback, rollforward, and compensation [9]. Reverting to a known good state, or checkpoint, is the concern of rollback, triggered after error detection. Rollforward advances the system to an error-free state, possibly skipping partial delivered service. Compensation leverages redundancy to mask the errors in the system state [9].

Our CPS design approach is based on concurrent error detection, compensation, and reinitialization. As it implements error handling followed by fault handling, our approach is a system recovery fault tolerance strategy.

2.1.1 Radiation and semiconductors

Radiation-induced faults are natural, external hardware faults, and a source of errors even for terrestrial CPS. They emanate from ionizing radiation, which is the product of cascading interactions of galactic cosmic ray (GCR) and solar particles with atomic nuclei located in the top layers of earth's atmosphere [33]. These GCR originate somewhere in our galaxy and are particles with energies up to several thousand GeV [33]. But the off-chip environment is not the only contributor to emissions, as alpha particle emitting impurities in integrated circuit (IC) packaging and fabrication materials with emissivity lower than $5 \times 10^{-4}\,1/\mathrm{cm}^2\mathrm{h}$ can be found [34]. This section summarizes the interaction of ionizing radiation with semiconductor material as a foundation for the error models in Section 2.1.2.

If high-energy neutrons are reacting with silicon by inelastic scattering, they transfer energy to the silicon nucleus, which puts the nucleus into an exited energy state [35]. This exited state is unstable, and the nucleus commonly emits magnesium, lithium, and helium ions (alpha particles) to enter a stable energy state [36].

All ions transfer energy into the silicon due to loss while traveling through the material. The magnitude of this effect is described by the mass stopping power, which is the mean energy lost $\mathrm{d}E$ over distance $\mathrm{d}l$ and the material's density ρ [37, p. 19]:

$$\frac{S}{\rho} = \frac{1}{\rho}\frac{\mathrm{d}E}{\mathrm{d}l} \tag{2.1}$$

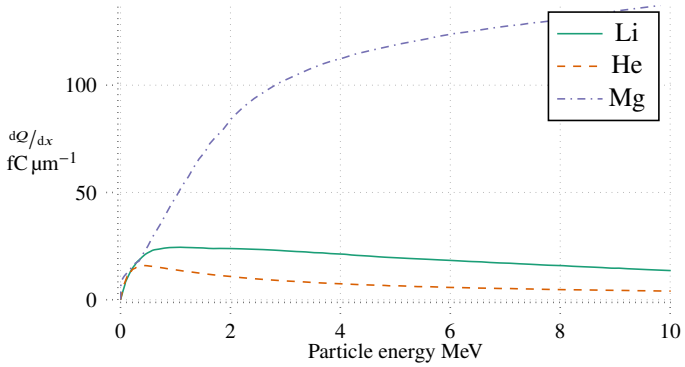


Figure 2.5: Charge generation per linear distance for different ions in silicon [36].

The disturbance caused by an ion, or transferred energy into silicon, depends on the ion's linear energy transfer (LET). LET, or restricted linear electronic stopping power $L_\Delta = {}^{\mathrm{d}E_\Delta}/_{\mathrm{d}l}$, can be thought of as locally transferred energy because energy carried away by energetic secondary electrons with an initial kinetic energy greater Δ is excluded [37].

For every energy of 3.6(3) eV transferred into the silicon an electron hole pair is produced [38, p. 1082], forming a cylindrical track of electron hole pairs along the ion's path [36]. This links energy to charge, allowing to express restricted linear electronic stopping power in unit of energy over distance ($\mathrm{J\,m^{-1}}$) as charge over distance ($\mathrm{C\,m^{-1}}$). For practical use, one elementary positive charge e is $1.602\,176\,634 \times 10^{-19}$ C [39, p. 128], and one electronvolt is $1.602\,176\,634 \times 10^{-19}$ J [39, p. 145], resulting in the following expression for silicon:

$$\frac{\mathrm{d}Q}{\mathrm{d}x} = L_\Delta \cdot \frac{1\,\mathrm{eV}}{1.602\,176\,634 \times 10^{-19}\,\mathrm{J}} \cdot \frac{1.602\,176\,634 \times 10^{-19}\,\mathrm{C}}{3.6(3)\,\mathrm{eV}} \tag{2.2}$$

Figure 2.5 presents the charge deposition in silicon per distance traveled, linking the ion energy to circuit quantities.

Within a few microns of a p-n junction the electric field collects charges between 1 fC and several 100 fC [36]. First, carriers are rapidly collected, leading to large current and voltage transients at the junction's circuit node. After the rapid collection diffusion prevails, as shown in Figure 2.6. Charge sharing among nodes is common because in actual circuits nodes are in close proximity to one another [36]. Reverse-biased junctions are especially sensitive, particularly if the junction is floating or weakly driven.

A change in data state is triggered if collected charge Q_{coll} exceeds the node-specific critical charge Q_{crit}. The interaction of Q_{coll} and Q_{crit} needs to be taken into account to model the occurrence of single-event effects (SEEs). Both quantities are dependent on a plethora of factors: The magnitude of Q_{coll} is influenced by *1)* biasing of nodes, *2)* substrate structure, *3)* device size and doping, *4)* type and energy of ion, *5)* initial position of the event and trajectory of the ion; and *6)* state of the device. Node capacitance, operating voltage and feedback transistor strength define the sensitivity of a node expressed in Q_{crit}.

Now with a link from ionizing radiation to unwanted changes in data state, or radiation-induced faults, it is desirable to separately model the device-specific

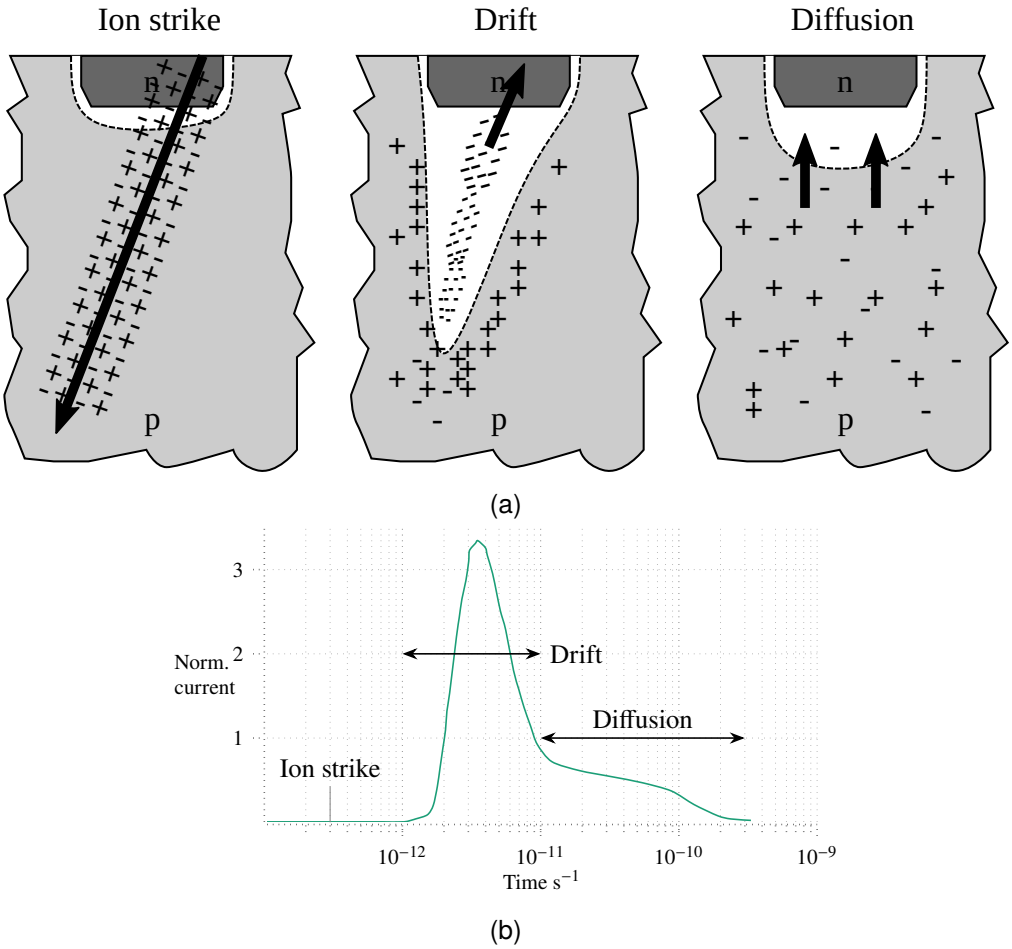


Figure 2.6: Ionizing radiation traversing reverse-biased p-n junction [36] in Figure 2.6a. The charge is collected by drift and diffusion, as shown in Figure 2.6b.

Figure 2.7: Proton- and neutron irradiation facilities at TRIUMF, Canada's national particle accelerator center [40]

susceptibility to radiation-induced faults from the radiation environment. This separation, and the terminology of different models is presented in the next section.

2.1.2 Modeling radiation environments and radiation-induced faults

In this section we separately model the radiation environment and the device-specific susceptibility to radiation induced faults. Moreover, we specify several terms used to differentiate the diverse effects of radiation on circuits, ranging from transient influences to permanent damage.

The simplest radiation environment consists of a monoenergetic radiation source for a specific particle type, such as the monoenergetic particle beams used in testing laboratories, shown exemplarily in Figure 2.7. We describe such a radiation source by the flux $\dot{N}$, or the increment of the emitted particle number $\mathrm{d}N$ in the time interval $\mathrm{d}t$ [37]. If we define the fluence Φ as the the quotient of the increment of number of particles $\mathrm{d}N$ incident on a sphere of cross-sectional area $\mathrm{d}A$ [37], we can describe the radiation environment generated by the beam by a fluence rate $\dot{\Phi}$, or fluence increment $\mathrm{d}\Phi$ during interval $\mathrm{d}t$ [37].

Sometimes the terms „flux“ [41], „flux“ [42] [1], or „particle flux“ [43] are used to describe what is termed fluence rate according to the international commission on radiation units and measurements (ICRU).[2] Fluence and fluence

[1] Flux f describes the number of particles p that cross the area a in time t: $f = {}^{p}/_{a \cdot t}$ [42]

[2] „[. . .] [t]he term flux has been employed in other texts for the quantity termed fluence rate in the present Report [. . .].“ [37] To avoid confusions the ICRU definitions of flux, fluence and fluence rate are used instead of the former definitions from the semiconductor and space

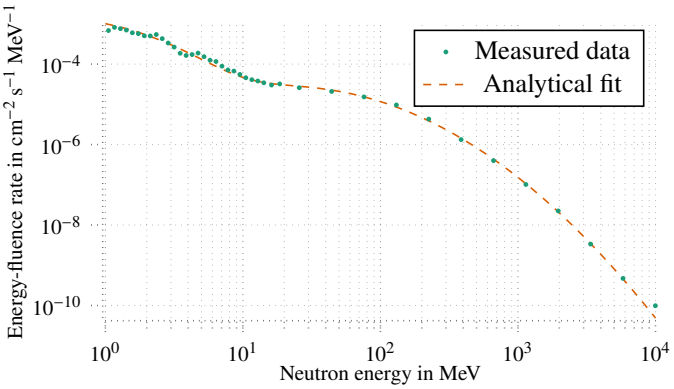


Figure 2.8: Energy-fluence rate in $\mathrm{cm}^{-2}\,\mathrm{s}^{-1}\,\mathrm{MeV}^{-1}$ over neutron energy outdoors at sea level in New York City with mid-level solar activity. This particular energy-fluence rate is the reference spectrum used for testing according to standard JESD89A [41].

rate are therefore related by

$$\Phi = \int \dot{\Phi}\,\mathrm{d}t = \frac{\mathrm{d}N}{\mathrm{d}A} \tag{2.3}$$

with the area A „[. . .] perpendicular to the direction of each particle.“ [37]

These definitions allow us to describe the radiation at the source and at the circuit: Flux, referring to a limited spatial region, lends itself for describing the source, and fluence rate is convenient to describe the radiation environment at the circuit.

If we want to describe a radiation environment of particles with multiple different energy levels, such as the GCR-induced neutrons at sea level in Figure 2.8, the energy-fluence rate $\dot{\Psi}$ is appropriate.[3] Given a radiant energy R, the increment of radiant energy $\mathrm{d}R$ incident on a sphere of cross-sectional area $\mathrm{d}A$ is the energy fluence Ψ [37]. This allows to define the energy-fluence rate $\dot{\Psi}$ as the quotient of the increment in energy fluence $\mathrm{d}\Psi$ by the time interval $\mathrm{d}t$ [37]:

$$\Psi = \int \dot{\Psi}\,\mathrm{d}t = \frac{\mathrm{d}R}{\mathrm{d}A} \tag{2.4}$$

Now with the radiation environment defined, we still need to model the intrinsic quantities of a device which link the radiation environment to the rate of radiation-induced faults. It is common to model these device intrinsic quantities by cross sections. Measured in cm^2/device, they can be interpreted as a surface that is sensitive to a specific particle species. Other common units for cross sections are $\mathrm{cm}^2\,\mathrm{bit}^{-1}$ and $\mathrm{cm}^2\,\mathrm{Mbit}^{-1}$ [41]. As cross sections describe the response to specific particles, it is sensible to introduce the terminology for different responses first.

We introduce the terminology top-down, starting at the root category SEE in the accompanying Figure 2.9. A SEE is „[. . .] [a]ny measurable or observable change in state or performance of a microelectronic device, component, subsystem, or

communities.

[3] In JDEC Standard No. 89A the used term is „differential flux“ [41].

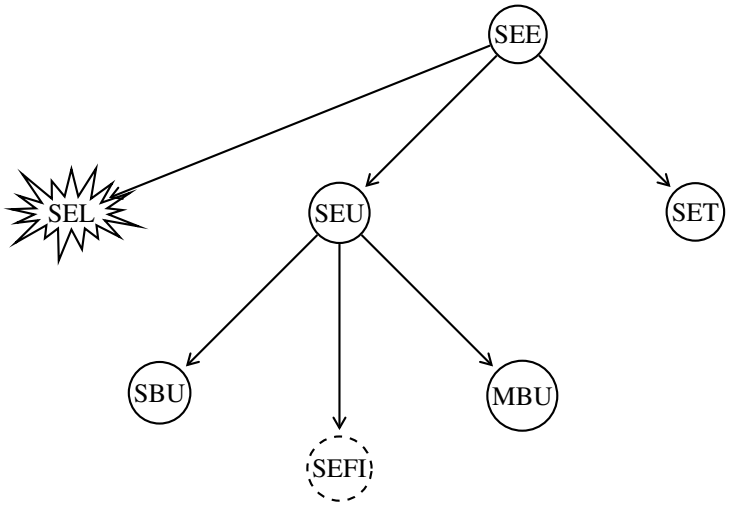


Figure 2.9: SEE is a category of effects that includes specialized terms like SET and SEU. The dashed circle denotes that SEFI specifies a certain system response to a SEU. SEL are highlighted as destructive because of their fatality.

system (digital or analog) resulting from a single energetic particle strike.“ [41] SEEs are categorized by their permanence and destructiveness: Hard errors like single-event latch-ups (SELs) are „frequently destructive“ [33] bringing along *permanent* damage to the circuit, while soft errors like single-event transients (SETs) and single-event upsets (SEUs) can be corrected and are *not permanent*.

SET describes „[. . .] [a] momentary voltage excursion (voltage spike) at a node in an integrated circuit caused by a single energetic particle strike“ [41], and SEU „[. . .] [a] soft error caused by the transient signal induced by a single energetic particle strike“ [41]. We can further differentiate SEUs by the number of affected bits in single-bit upset (SBU) and multiple-bit upset (MBU). Another related term is single-event functional interrupt (SEFI) „which denotes single event upsets that cause the component to reset, lock-up, or otherwise malfunction in a detectable way. SEFI per se is not a different soft error upset mechanism, but rather a term that specifies a certain system response [. . .] to a particle induced upset [. . .].“ [33]

The definition of SEU uses the term soft error, defined as „[. . .] [a]n erroneous output signal from a latch or memory cell that can be corrected by performing one or more normal functions of the device containing the latch or memory cell. [. . .] As commonly used, the term refers to an error caused by radiation or electromagnetic pulses and not to an error associated with a physical defect introduced during the manufacturing process.“ [41] The rate at which soft errors occur is the soft error rate (SER) [41] frequently measured in failures in time (FITs) [33].

Now with the response terminology in place, we can further define the intrinsic quantities of our device by cross sections. The term SEU cross-section characterizes the SEU sensitivity of a circuit as a function of the operating conditions [41]. The SEU cross-section is „[. . .] an intrinsic parameter of a chip/circuit that specifies its response to a particle species (e.g. neutron, proton, pion, heavy ion, etc.).“ [41]

Table 2.1: SEU rates in flip-flops at nominal supply voltage under JESD89A-conform testing conditions from multiple sources [45–47].

Process node nm^{-1}	Voltage V^{-1}	Alpha SER FIT·bit^{-1}	Neutron SER FIT·bit^{-1}
7	0.75	0.42	3.73
10	0.75	0.64	6.11
14	0.8	8.69	19.34
32	1.2	45	
90	1.2	28	15

Contrary the SER is „[. . .] a measure of a chip's response to a particular type of radiation environment. Its value varies from one location to another, depending on the radiation environment that is present.“ [41] With SER and SEU cross-section defined, we can finally link a radiation environment to the radiation-induced faults of our device by Equation (2.5). For „thin“ [33] devices the following equation holds:

$$\mathrm{SER} = \int_E \sigma(E) \times \dot{\Psi} \, \mathrm{d}E \ll 1 \tag{2.5}$$

Depending on process technology, supply voltage, cell layout, circuit topology, circuit state, and dynamic circuit behavior, particles contribute differently to the total SER of the device. Technology scaling results in ever smaller critical charge Q_{crit}, but the charge collection volume in FinFETs decreases as well [44]. For a 7 nm bulk FinFET process technology, alpha particles can't generate sufficient charge for SEUs in the small volume below the fin [45]. A comparison of alpha particles and high-energy neutron SER for different technology nodes is presented in Table 2.1. These values are hard to compare, as testing conditions and circuit layouts differ. Moreover, most reported results in the literature are normalized to protect intellectual property (IP).

Other consequences or radiation to semiconductors are total ionizing dose (TID) effects, which refer to the accumulation of charge traps in silicon dioxide. If radiation-induced charge is trapped in the gate oxide of a MOSFET, the threshold voltage shifts, and with enough trapped charge it is impossible to turn off the MOSFET [48]. Furthermore, alpha particles, x-rays, and secondary particles from neutrons are sources of micro-doses, which can already effect process corner sub-10 nm FinFETs [49]. Affected transistors switch slower, which can result in timing errors in the considered circuit.

While detailed descriptions of fault physics are essential for our understanding of the real world, it is still necessary to come up with good models of these faults for practical design- and development tasks. Very complex analytical models, striving to capture nearly all effects of the physical reality, are limited in use by the required computational power to solve them. Instead, picking the right abstraction helps to acquire models which are easy to work with. Where technology computer aided design (TCAD) simulations benefit from detailed models to describe radiation events, circuit-level simulations gain from simpler current source models, which capture the circuit's response to radiation events in useful, circuit-related terms. For example, a circuit node struck by an alpha

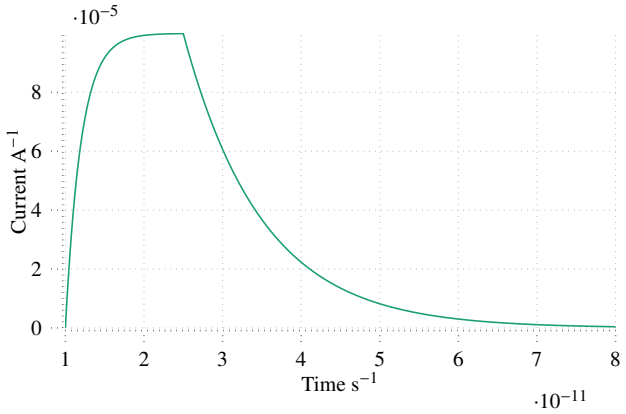


Figure 2.10: Double exponential current model for alpha particle strike at a circuit node.

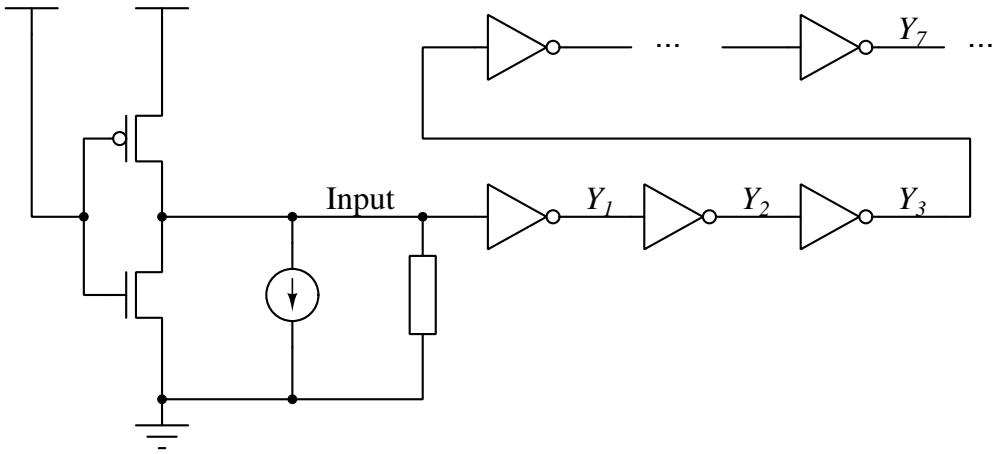


Figure 2.11: Circuit diagram of double exponential current pulse generator and inverter chain

particle can be modeled with a double-exponential current source [50–52], as shown in Figure 2.10. Even higher abstraction levels, like register transfer (RT) level for digital logic, might model the resulting voltage spike as a simple pulse.

Regarding the double-exponential current source model in Equation (2.6),

$$I(t) = \begin{cases} 0 & t < t_r \\ I_p \left(1 - \mathrm{e}^{\frac{-(t-t_r)}{\tau_r}}\right) & t_r < t < t_f \\ I_p \left(\mathrm{e}^{\frac{-(t-t_f)}{\tau_f}} - \mathrm{e}^{\frac{-(t-t_r)}{\tau_r}}\right) & t > t_f \end{cases} \tag{2.6}$$

the parameters are the onset of the rising current t_r, the onset of the falling current t_f, the peak current I_p, rise time constant τ_r, and fall time constant τ_f. Suitable parameters are extracted from TCAD simulations, and further refinement of the model by two parallel double-exponential current sources to consider prompt charge collection and sustained charge collection increase its accuracy [52].

We can investigate the propagation of current-induced voltage transients, or SETs, as shown in Figures 2.12 to 2.14, to discuss their response models and

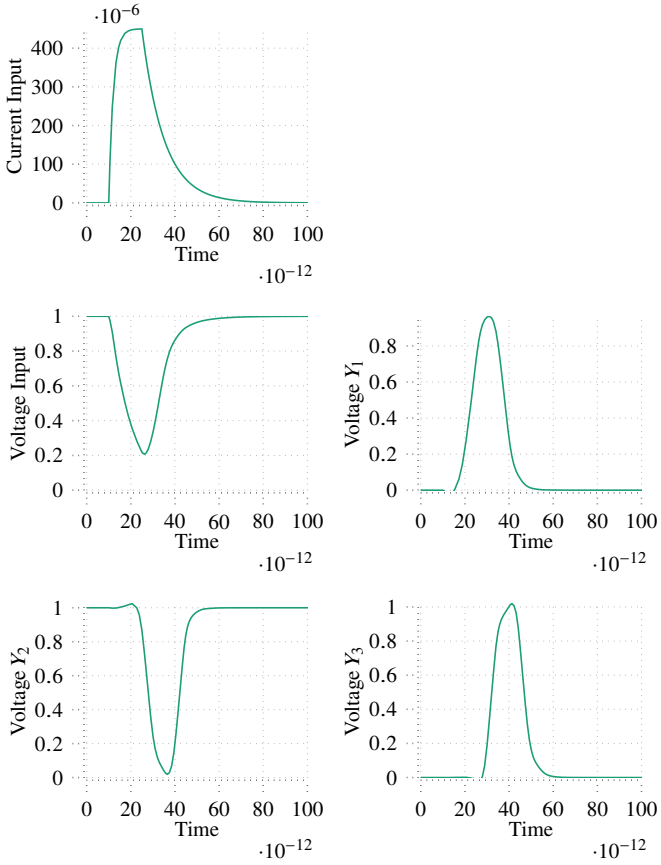


Figure 2.12: Double-exponential current pulse in inverter chain. The pulse's parameters according to Equation (2.6) are $I_p = 450\,\mu A$, $\tau_r = 2\,ps$, $\tau_f = 10\,ps$, $t_r = 10\,ps$, and $t_f = 25\,ps$. The voltage transient propagates through the inverter chain, while getting slightly narrower with each inverter passage.

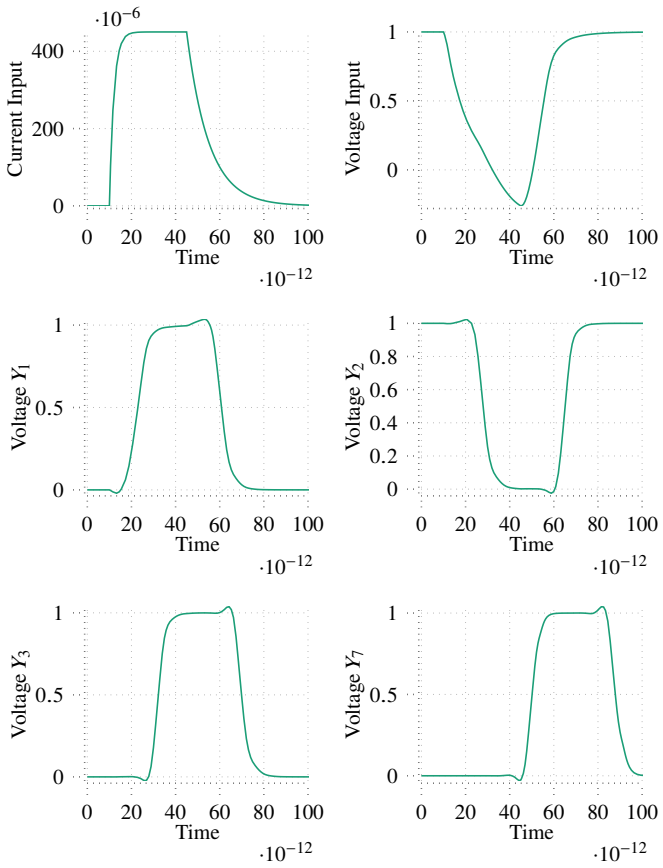


Figure 2.13: Wide double-exponential current pulse in inverter chain. The pulse's parameters according to Equation (2.6) are $I_p = 450\,\mu A$, $\tau_r = 2\,ps$, $\tau_f = 45\,ps$, $t_r = 10\,ps$, and $t_f = 25\,ps$. The voltage transient propagates through the inverter chain, and with each inverter the shape normalizes towards a pulse shape.

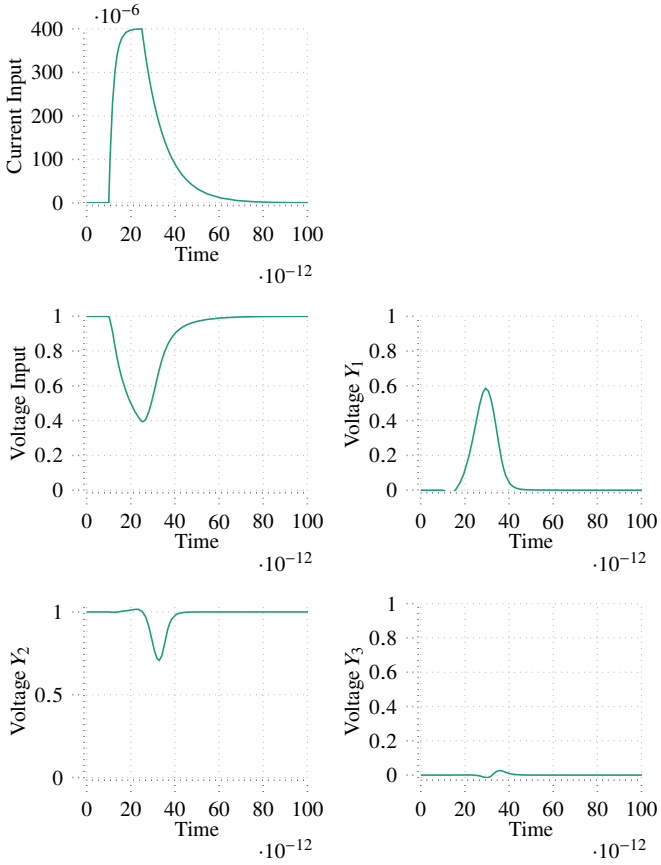


Figure 2.14: Low current double-exponential pulse in inverter chain. The pulse's parameters according to Equation (2.6) are $I_p = 400\,\mu A$, $\tau_r = 2\,ps$, $\tau_f = 25\,ps$, $t_r = 10\,ps$, and $t_f = 25\,ps$. With $50\,\mu A$ less peak current compared to Figure 2.12, the resulting voltage transient is damped with each inverter passage.

possible mitigations. The voltage transients in Figures 2.12 to 2.14 are generated by circuit simulation using the FreePDK 45 nm process design kit (PDK). The test circuit consists of a double-exponential current pulse generator followed by an inverter chain, as shown in Figure 2.11. Depending on the deposited charge, transients propagate or are attenuated. This allows us to differentiate their responses in terms of SET and SEU: A voltage transient propagating through the logic is a SET. If the SET reaches a latch and is sampled, it is indistinguishable from a SEU [33].

Moreover, we can reason about the masking of SETs, which happens due to the *1)* electrical (pulse attenuation), *2)* logical (denied propagation) and *3)* timing properties (occurrence not during setup- or hold time) of a circuit. It is sensible to consider SET in RT-level simulations with a simple rectangular pulse of desired width. Such pulses can be inserted during simulation at random circuit nodes, resulting in glitches in combinational logic and SEUs in storage elements.

The consequences vary according to the circuit, but a general higher-level description in terms of timing errors allows to consider faults of different origin, including the radiation-induced faults described so far. If we design our circuit with appropriate redundancy, we can cover errors from process variations, supply voltage variations, temperature, and aging similar to radiation, as described in the following chapters.

2.2 Fault tolerance techniques

Ensuring the continuous correct service of a system and thus making it reliable requires addressing faults during system design and operation. Faults are classified based on whether their duration is permanent or transient, their extend is local or distributed, and their value is determinate or indeterminate [9].

Once a fault is activated, the system deviates from its correct service state. If the deviation or error affects the delivered service a failure occurs [9]. Fault tolerance techniques are used to prevent system failure and differ in which class of fault they are able to tolerate.

In the following sections, most presented fault tolerance techniques assume hardware faults. While software faults are a reality, we exclude them here by assuming a perfect software design- and development process, as software faults are design faults [53].

2.2.1 Error detection

Error detection approaches allow to separate the task of error detection from error correction, which can result in cost-efficient fault tolerant systems. They can be broadly classified in four fields: *1)* Anomaly detection, in data [54] and behavior [55,56]; *2)* Dynamic verification or run-time checking of properties [57, 58]; *3)* Redundant execution, including lockstep, multi threading, and duplicate instructions [59,60]; and *4)* built-in self test (BIST) in hard- and software [61,62].

Anomaly detection in data can be implemented with error detection codes (EDCs) [63] exploiting spatial redundancy, such as parity [64]. For example, an arithmetic operation in a CPU generates the desired result and calculates a

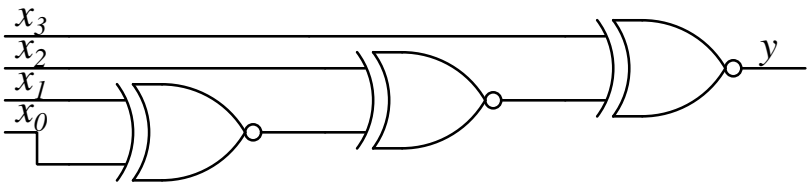


Figure 2.15: 4 bit parity generator

parity bit, which is stored next to the result. If the stored result is affected by a SBU, the parity changes, and this change is detectable by recalculation of the parity bit. Parity checking of data storage and transfer is a staple in fault-tolerant computing [65], as hardware implementations with XNOR logic have relatively small area cost. The odd parity bit of a data word is calculated by XNOR reduction in Figure 2.15, with the resulting parity bit set if the number of set bits in the data word is even. An error is detected if a XNOR reduction of the original data word and the parity bit produces a logical one.

Another spatial approach is dual modular redundancy (DMR) [66], where a module is duplicated and operated in parallel. If the results of the original and the duplicated module differ, which is detectable by XOR operations, an error is detected. Main drawback is the overhead, as duplication is costly in terms of area and power. There is no direct way in pure DMR to tell which module produced the error, therefore no correction can take place. But if the signal distribution of the computed result is known, the more likely result can be chosen to implement probabilistic correction [67].

Self-checking logic design is a systematic approach to extend circuit synthesis with EDCs. Resulting self-checking circuits produce encoded output signals, which allows to detect errors by dedicated checker hardware [68]. EDC selection for self-checking logic needs to balance circuit overhead with error detection capabilities. A prominent example are Berger codes, which are systematic codes that count the number of ones or zeros in a data word to generate the check bits [69].

Dual-, or two-rail logic is another example of self-checking logic, as it produces encoded output values which can be checked. Dual-rail logic uses the smallest unordered code known to represent logic one and zero by two out of four possible two-bit combinations [70]. For example, Figure 2.16 shows a dual-rail XOR gate, used to compare DMR-protected circuits [71]. This allows to detect errors in the checker itself.

Anomaly detection in data is possible by temporal redundancy as well. Such approaches might sample a data signal multiple times and compare all samples to detect differences [72]. If the data signal should be stable, detected differences signal errors. A popular circuit-level implementation are Razor flip-flops. In Razor flip-flops, the data input of a flip-flop is additionally sampled by a latch controlled by a delayed clock [73], as shown in Figure 2.17.

Another approach using temporal redundancy for anomaly detection in data are transition detectors. Such detectors monitor a circuit node for changes, and report these changes to further error management circuitry. For example, monitoring the internal node of a latch for changes outside of a time window for expected change allows to detect errors. For illustration, the Razor2 circuit in

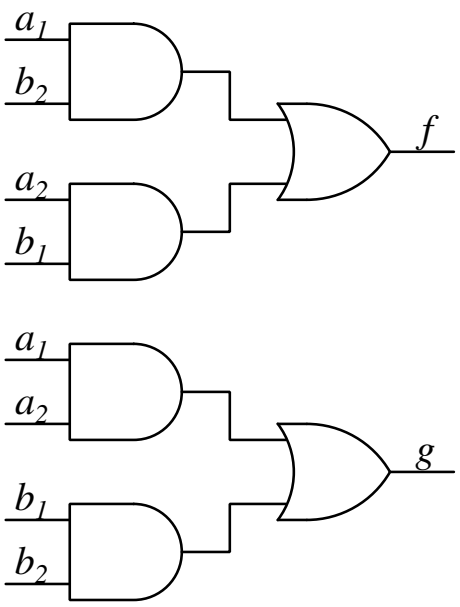


Figure 2.16: A self-checking checker using dual-rail encoded output signals [70]

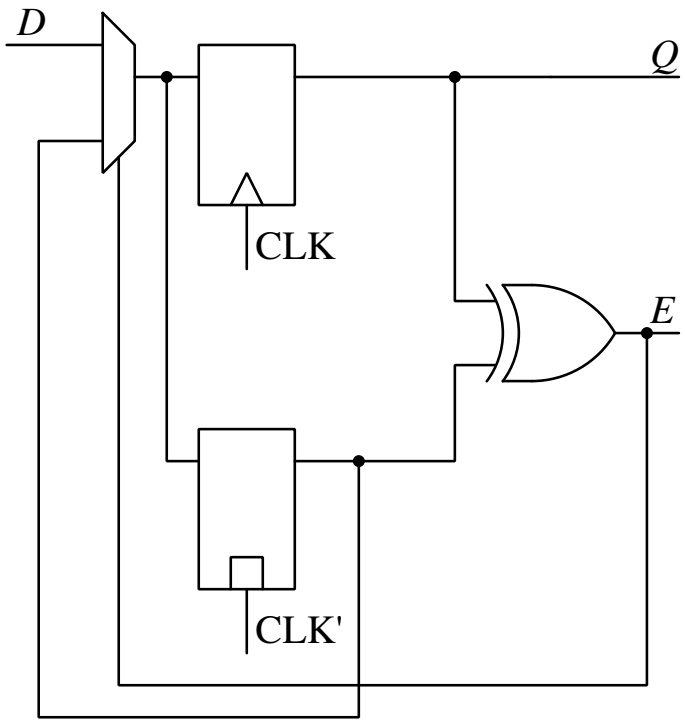


Figure 2.17: Razor flip-flops [73]

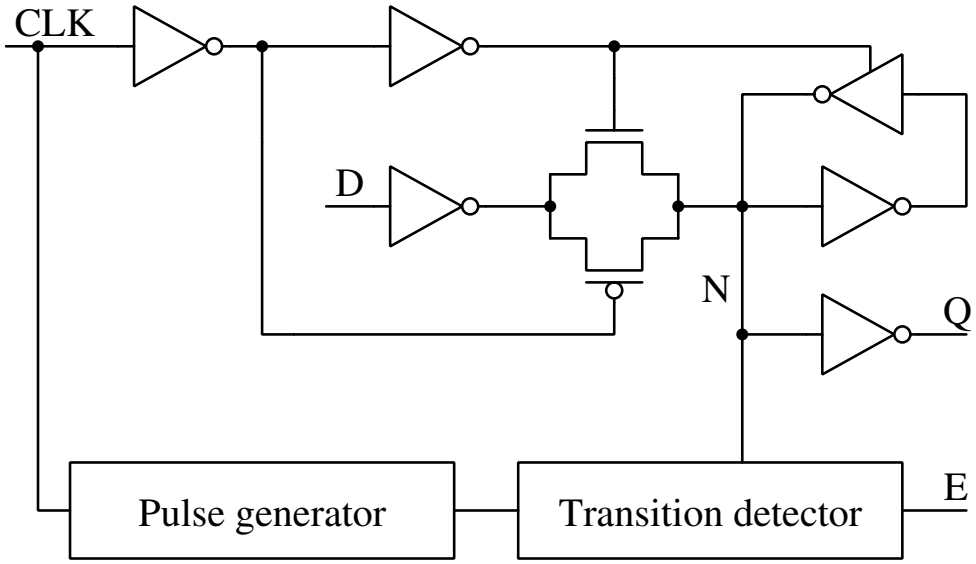


Figure 2.18: Razor2 [74]

Figure 2.18 resorts to this principles [74].

The transition detector in Razor2 consists of two sub-circuits shown in Figure 2.19. Transmission gates in Figure 2.19a are used as tunable delay elements to sample the internal latch node N at different points in time. These signals are evaluated by the transition detection circuit in Figure 2.19b, which signals detected errors until active-low reset. Shortly after the rising edge of the clock the detection clock generator in Figure 2.19c disables the transition detector, to allow capturing of regular, error-free signals.

Error detection based on temporal redundancy allows to trade performance for reliability and energy savings [75], but nearly every proposed temporal redundancy error detection circuit exploits dynamic CMOS design features, which are not available for COTS FPGA implementations of CPS.

Another branch of error detection approaches resorts to anomaly detection in behavior. Watchdogs processors are prominent examples, as they observe a system concurrently and compare the system's behavior to expected behavior. Observations of memory access, control flow, or computed results provides the watchdog processor with information about the system's actual behavior [76].

Other behavior anomaly detectors are instruction counters to detect extraordinary high activity, branch counters to detect hanging execution by heuristics or with dedicated hardware [77], or range-based invariants on integer function return values, loads, and stores by constant upper- and lower bounds [56]. In pipelined CPUs confidence branch predictors can be leveraged to detect incorrect control flow with high probability, as well as memory access and alignment exceptions as part of the CPU's instruction set architecture (ISA) to detect errors [55].

Despite anomaly detection in data and behavior, dynamic verification, or run-time checking of properties, can detect errors as well. For a von Neumann CPU core, these properties are forward progress through the control flow graph of the program binary, correct computation results, preservation of the program's dataflow graph, and correct interaction with the memory system [57]. In pipelined CPUs, the commit of results to memory can be verified [58]. Dynamic verification

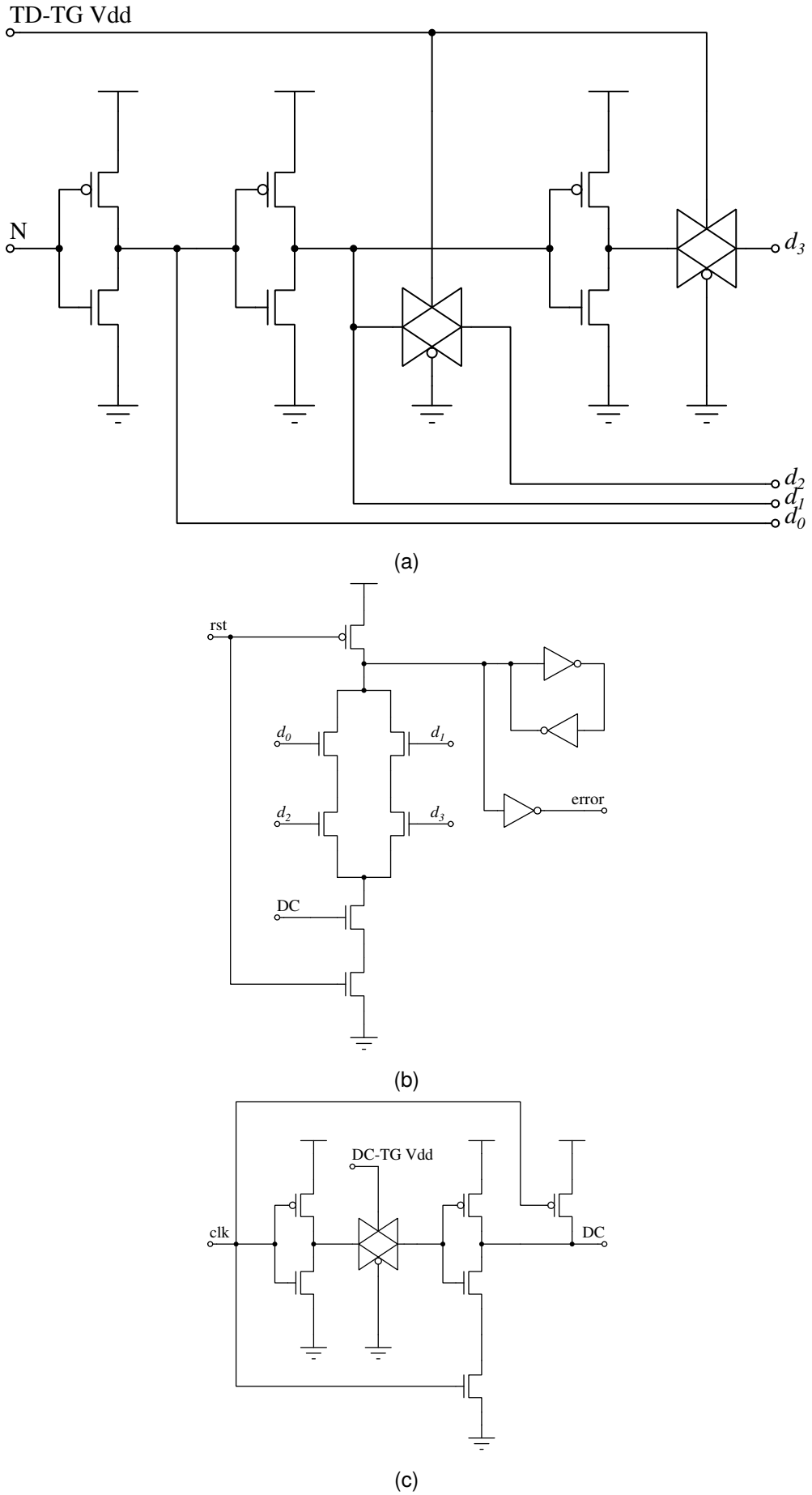


Figure 2.19: Razor2 circuit-level implementation [74]

approaches are attractive because they don't require redundant execution, but they incur hardware overheads to verify the validity of the properties which are assumed to be true in error-free operation.

Still, complementary approaches resorting to redundant execution are sensible for CPS design as some of them don't require hardware modifications. In the spatial domain, redundant execution is possible by two cores operating in tight lockstep to compare their results after every instruction; and in the temporal domain multithreading can provide the required redundancy for error detection [59].

Software solutions for error detection mostly resort to temporal redundancy by duplicating the execution of instructions [60], procedures [78], or whole programs [79]. Another approach to software error detection is to generate two versions of a program with the same functionality, which are executed with different input data. Due to the systematic program generation the results are comparable and errors are detectable [80, 81]. Executable assertions introduce additional conditional checks in the program to test the plausibility of intermediate data, resulting in the detection of errors during program execution [82, 83]. Error correction is usually implemented on a higher architectural or software level [84].

In conclusion, error detection approaches are diverse, but not all are suitable for COTS-based CPS design. Software-based and FPGA-compatible approaches are favored, as further explained in the following chapters.

2.2.2 Error and fault masking

Masking techniques are fault tolerance techniques which transparently prevent failures by masking errors. Modular redundancy with voting [85] is a popular masking technique [86–88] due to its transparency and applicability on several levels.

Triple modular redundancy (TMR) with imperfect voting as introduced by von Neumann [85] is a possibility to increase the reliability of a system by introducing redundancy. In its simplest form a triplication of the unreliable component is combined with a majority voter to transparently correct a false result from a failing component.

We construct a TMR network for a unreliable component by triplicating the component and feeding the output of each component into a majority organ. This majority organ is a combinatorial circuit that propagates the result of a majority vote of three input signals. Figure 2.20 depicts the described TMR network for an arbitrary component M. Given the probability of failure ϵ for the majority organ, we are interested in the failure probability of the TMR network. If each input line of the majority organ has a probability of carrying the wrong signal η_i, a general upper bound for the TMR network failure probability ρ is given as $\rho = \epsilon + \eta_0 + \eta_1 + \eta_2$ [85]. Under the assumptions of statistical independence for wrong signals on the input lines and a high probability for identical signals on the input lines we can derive a lower bound for ρ. For this we formulate the probability of a majority of wrong input signals θ_N with the probabilities of *up to i* of N wrong input signals $\theta_{i,N}$. Let ${\eta_i}^C = 1 - \eta_i$ be the complementary event

probability for a wrong signal on a specific input line.

$$\theta_{1,N} = \sum_{i=0}^{N-1} \eta_i \prod_{i \neq j, j=0}^{N-1} \eta_j^C \tag{2.7}$$

$$\theta_{1,3} = \eta_0\eta_1^C\eta_2^C + \eta_1\eta_0^C\eta_2^C + \eta_2\eta_0^C\eta_1^C \tag{2.8}$$

$$\theta_{2,3} = \theta_{1,3} + \eta_0\eta_1\eta_2^C + \eta_2\eta_1\eta_0^C + \eta_0\eta_2\eta_1^C \tag{2.9}$$

$$\theta_{3,3} = \theta_{2,3} + \eta_0\eta_1\eta_2 \tag{2.10}$$

$$\theta_3 = \theta_{3,3} - \theta_{1,3} \tag{2.11}$$

$$= \eta_0\eta_1 + \eta_2\eta_1 + \eta_0\eta_2 - 2\eta_0\eta_1\eta_2 \tag{2.12}$$

$$\rho' = (1-\epsilon)\theta_3 + \epsilon(1-\theta_3) \tag{2.13}$$

$$= \epsilon + (1-2\epsilon)\theta_3 \tag{2.14}$$

If our whole system — without TMR — has a reliability of R_0, and we assume equal reliability of each module within, the reliability of a module — without TMR — is $R_M = R_0^{1/m}$. For an arbitrary module with equal probabilities $\eta_i = \eta = 1 - R_M$ the lower failure bound is $\epsilon + (1-2\epsilon)(3\eta^2 - 2\eta^3)$. Assuming system failure if a single TMR-protected module fails, we can derive the reliability of the whole TMR-protected system as a product of the module reliabilities, visualized in Figure 2.21:

$$R = \left[1 - \left(\epsilon + (1-2\epsilon)(3\eta^2 - 2\eta^3)\right)\right]^m \tag{2.15}$$

Therefore voter reliability is paramount for such TMR-protected systems. While other TMR voting configurations are known [90] to improve voting reliability, TMR still falls short in case of MBUs or SELs: MBUs, or more generally common-mode failures, invalidate the assumption of independent errors on input signals, and SELs can destroy the possibility to vote. Nevertheless other spatial masking techniques like NAND multiplexing [91, 92] or quadded logic [93] provide a worse ratio of reliability to area overhead making TMR the dominant choice for FPGA designs [94] due to large logic paths and few voting elements [95].

We can see TMR more generally as a binary linear repetition code with a fast and easy to implement decoder [96]. Separate error correcting codes are common for memory protection or self-checking designs [72]. Circuit and layout hardening techniques [97–99] mask errors as well but are not applicable for high performance COTS-based solutions. But the major drawback of masking techniques is the amount of additional resources required to implement and operate the redundancy. Especially for CPS, which suffer from limited energy budgets, the additional power consumption of the redundancy is a fundamental problem.

2.2.3 Fault detection and diagnosis

Fault detection techniques allow to identify faulty modules by model checking and are used for debugging purposes [100, 101]. It is possible to find an over-approximation of the faulty gates in a circuit, given a set of input assignments for which the circuit misbehaves and corresponding correct output values, using Boolean satisfiability (SAT)-based localization techniques [102]. The exact set of

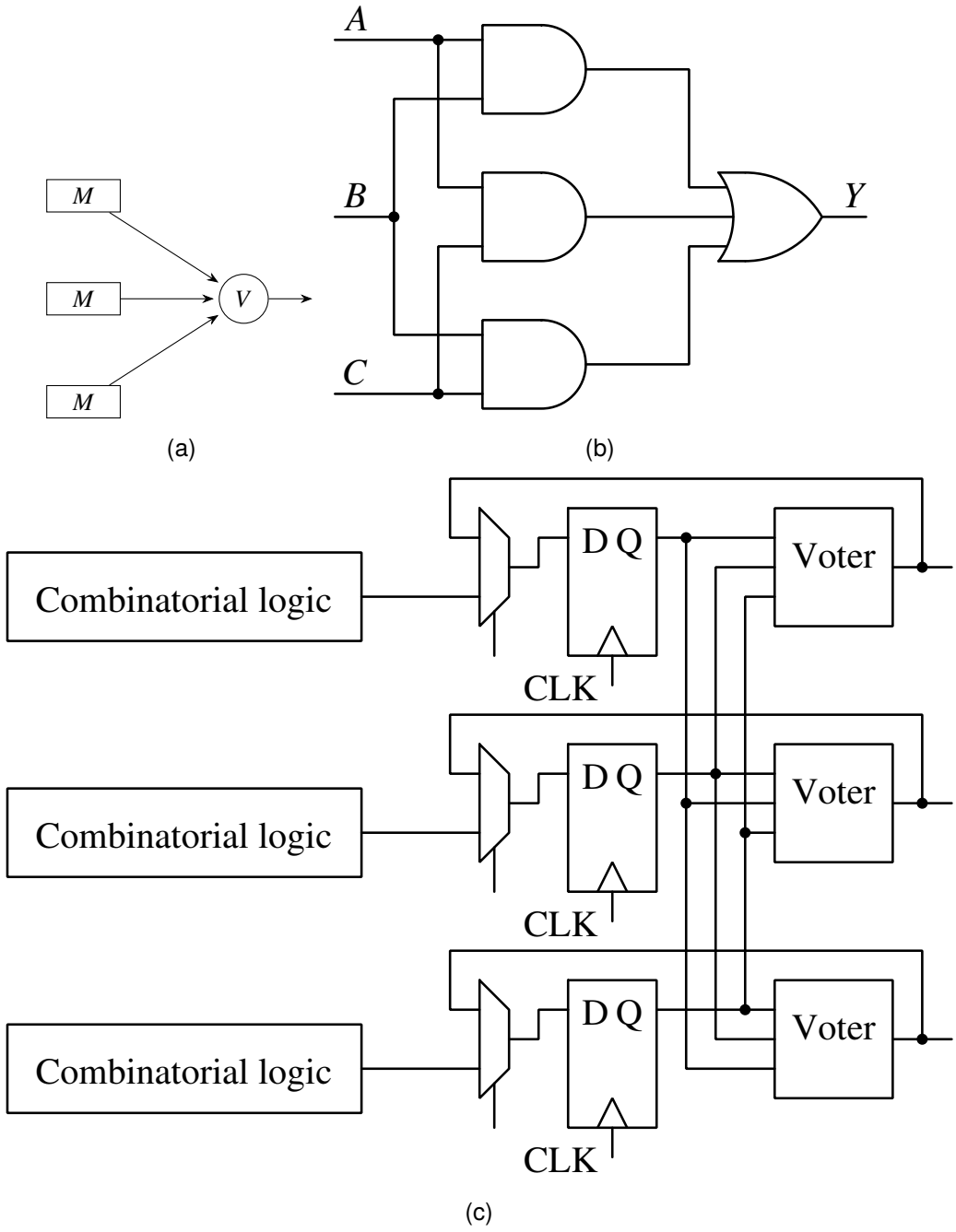


Figure 2.20: TMR network with unperfect voting V for an arbitrary component M [86] in Figure 2.20a, and implementation of TMR voting circuit, or majority organ, using standard gates in Figure 2.20b. Figure 2.20c shows the TMR setup recommended by the European Cooperation for Space Standardization (ECSS) [89].

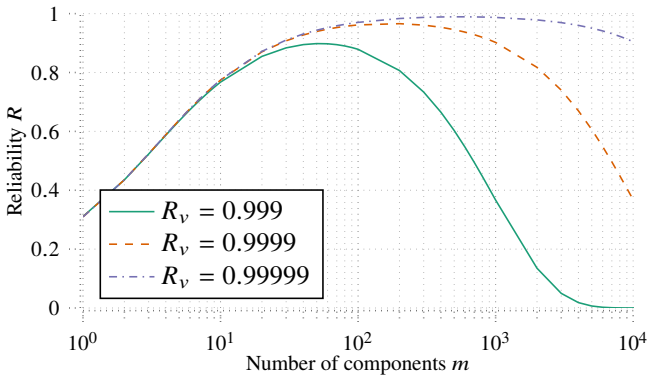


Figure 2.21: Reliability of TMR with unperfect voting over number of components m in a system. Each component has a reliability of $R_0 = 0.37$. Voter reliability is paramount for system reliability.

faulty gates is obtainable by solving a quantified Boolean satisfiability (QSAT) problem by iterative refinement of an initial over-approximation using a SAT oracle [103].

SAT-based methods are popular due to their run-time performance compared to directly solving QSAT problems [104]. Fault detection is a powerful tool in debugging but requires a white box model of the fault-affected system and a working white box reference model. This is in contrast to diagnosis, where only a black box model of an fault-affected system is available besides the working white box reference model [105]. Nevertheless both approaches are not suitable for online operation in face of limited computational resources and missing white box reference models for COTS components.

2.2.4 Containment, monitoring, and state recovery

Containment techniques are used to prevent errors from propagating through the system [15]. Typically boundaries are established around the lowest level replaceable modules to confine faults and errors, which could be processors, memory, or interconnects [106]. Once established they are used to detect, correct, or block errors at the modules interfaces.

Defining sensible fault- and error containment regions is necessary to ensure the assumption of component independence most protection techniques rely on [107]. Moreover the redundancy used to control and disable a module's output data and to suspect and check incoming data needs to be in a separate fault- and error containment region [108]. Reconfigurable containment regions, enabled by a software-controlled interconnect, allow to preserve the performance of multiprocessor architectures and dynamic selection of modular redundancy schemes [109].

Containment techniques are especially applicable for COTS components because they require control of interfaces and not necessarily control of component internals.

Monitoring of the system state or output signals allows to identify modules that are in error. If such modules are detected repair or reconfiguration techniques can be used to remove or isolate the modules or to redistribute the workload to

healthy modules [110, 111].

Predefined test points [112] or dedicated heart beat lines expose the internal state of modules for gathering and exploitation of health data. Depending on the availability of test-points and access to health information monitoring is a suitable approach for COTS products.

State recovery techniques permit to reach a system state without errors once errors are detected. Checkpointing and rollback [113, 114] or single instruction retry techniques are state recovery techniques with application for processor designs on the architectural- and micro-architectural level. Recovery is triggered once errors are detected. The error checking interval can be as tight as in lock-step execution [115] or be expanded for performance gains [116].

Coprocessor-based detection and recovery augments a processor with a coprocessor that can execute one Turing-complete instruction to enable fault detection and recovery [117]. Watchdog triggered resets [76, 118] and memory scrubbing techniques allow to reach an error-free state as well. Most state recovery techniques are suitable for COTS-based CPSs, and they fit nicely with error detection approaches to implement system recovery.

2.3 Low-power design

This section explains the fundamentals of power dissipation in CMOS, and low-power circuit design. Low-power design is required to cope with ICs growing power dissipation per unit area. Power dissipation in CMOS is differentiated as either dynamic, due to logic transitions, or static, due to leakage. Dynamic power dissipation due to charging parasitic capacitances and current flow through channel resistance is proportional to the square supply voltage and average switched capacitance. Short circuit currents while both n- and p-networks of CMOS conduct contribute proportional to rise- and fall times of gates to dynamic power dissipation. Static power dissipation due to leakage increases with voltage scaling: Scaling the supply voltage requires to scale the threshold voltage to ensure sufficient noise margins, but leakage increases exponentially with threshold voltage reduction [119].

To minimize total power dissipation, while considering performance in terms of throughput or latency, low-power design techniques are used. Low power design can affect a system's design at all abstraction levels, from device- and circuit level up to architectural and system level, because potential power savings increase at each abstraction level [120]. At different abstraction levels, low-power design techniques may either reducing switching activity, switched capacitances, voltage, or leakage currents. The choice of sequential elements and clocking scheme, as well as different CMOS design styles like static, dynamic, or pass gate impact power dissipation.

Leakage currents may be reduced by resorting to multiple supply- and threshold voltages, power gating, and voltage scaling. Voltage scaling, dynamic or static, and near-threshold computing save power by exploitation of supply- and threshold voltage margins [121]. Switching activity can be reduced by logic synthesis, reducing transitions and parasitic capacitances by proper choice and composition of logic cells, or higher level approaches. Higher level approaches include low-

power codes for state machine vectors and communication, structural choices of arithmetic components, reducing memory access and sequencing by simpler data organization, exploitation of pipelining and parallelism to reduce the number of clock cycles needed for computation, power scheduling for dynamic power optimization, and reducing communication which increasingly dominates power consumption due to increasing interconnect capacitances [122].

Therefore systems that strive to combine dependable- and low-power design should emphasize local computation and reduce the need of communication for error detection. Moreover, new approaches need to be power aware, and ensure that they do not increase chip power density. For this, they need to integrate with current state of the art low-power approaches like power gating, voltage scaling, power scheduling, and near- or sub-threshold design.

2.4 Graceful degradation

Instead of operating at peak performance until a fault enforces a reconfiguration or leads to system failure, a system with graceful degradation accepts a performance decline for an extended operational lifetime [123]. The idea is to disable features of the system that are affected by faults instead of total withdrawal from the processing element.

Especially CPSs benefit from graceful degradation, as their maintenance and repair might be impossible or not desirable. Moreover, CPSs are typically used in long-life applications, which need graceful degradation in the presence of faults [124], and particularly COTS-based CPSs need graceful degradation [125] and fault tolerance [126].

Early analytical modeling of reliable systems considered graceful degradation as an alternative to maintenance [127]. By evaluation of all final system state probabilities after a period of use, and association of each final state with a performance index, graceful degradation can be understood as a probability to provide a given performance, or utility [128]. Such models allow to prepare for graceful degradation during design time, which can be leveraged during runtime to select a system configuration that degrades gracefully [129].

Another interpretation of graceful degradation is graceful degradation as a pattern, or general solution to a common problem [130]. This problem can arise in totally different applications, for example in control theory, where it is desirable that attacked monitoring systems report gracefully degrading sensor data [131], or in circuit design, where slack in logical paths is redistributed that the circuit degrades gracefully with acceptable error rates under supply voltage reduction to save power [132].

Further examples of the graceful degradation pattern are reducing the impact of faulty components in software component to hardware node mapping problems [133], adaptive routing algorithms for network-on-chips (NoCs) which consider routers partially available instead of fully available or out-of-service [134], or reacting gracefully to sudden changes in control problems such as degraded unmanned aerial vehicle (UAV) flight controls switching from navigation with global navigation satellite system (GNSS) to algorithms that don't need GNSS [135].

Proposed means to enable graceful degradation in SOCs are hardware recon-

figuration, workload adaption, and task re-mapping [136], or pair-and-swap of CPU cores in multicore systems to selectively disable faulty cores: All $2n$ cores are paired, and each pair executes in lockstep. If a pair disagrees, cores are swapped with another pair to identify the faulty core, and to test if the fault is permanent [137]. Another approach in SOCs is to formulate an optimization problem to maximize the product of performance, functionality, and energy consumption as QoS, and solve the optimization problem by a dedicated runtime manager [138], resulting in graceful degradation. In digital circuit design, constraining the impact of errors in adders to only a few digits results in gracefully degrading results by adjusting the dynamic range [139].

Graceful degradation in real-time scheduling problems is possible by adaptive fault tolerance scheduling which allocates redundancy to jobs from non-critical tasks such that deadlines are satisfied. Possible implementations select software-based temporal redundancy techniques, which require more time for calculation, aggravating the scheduling problem [140]. In mixed-criticality scheduling problems, providing low criticality tasks some service even after mode change gracefully degrades the system's QoS [141].

While these approaches are suitable for CPSs, COTS-based or deeply-embedded systems can lack the possibility or resources to implement them.

2.5 Real-time scheduling

This section motivates the mixed-criticality scheduling problem, shows our notation of mixed-criticality models, and presents dynamic scheduling algorithms to solve it.

To meet non-functional requirements of embedded systems in the avionics and automotive sector, components of different criticality, or required level of assurance against failure, are integrated onto a common hardware platform [142, 143], resulting in mixed-criticality systems [144].

For mixed-criticality real-time control systems, like on-board flight computers, a criticality is assigned to each task, or reoccurring computation. Computations, or jobs, need to finish prior to their deadline to meet the timing requirements of the system, and share the processor with other jobs. If job arrival times and their execution times are not known in advance, it is to decide during system run-time on a schedule when jobs are allowed to access the processor. This makes schedulability verification prior to system deployment a necessity. For schedulability verification, jobs' worst-case execution times (WCETs) are modeled as constant upper bounds, which are impossible or hard to derive in a safe and precise manner [145], and tend to increase with the criticality [146]. In specifying multiple estimates for a job's execution time, mixed-criticality scheduling approaches are able to provide different levels of assurance, increasing the chance to verify the schedulability of the system.

In the following section, we show our notation to describe a model of mixed-criticality scheduling, which formalizes the above description.

2.5.1 Notation

We use the standard dual-criticality task system model, where each task τ_i in a dual-criticality sporadic task set $\mathbb{T} = \{\tau_1, \ldots, \tau_n\}$ is characterized by

- a criticality $\chi_i \in \{L, H\}$;
- execution time budgets in both criticality modes c_i^L, c_i^H;
- a relative deadline d_i of the jobs of τ_i; and
- the minimum interarrival time, or period p between two jobs of τ_i.

For our considered implicit deadline dual-criticality task sets with independent, sporadic tasks execution time budgets, relative deadlines, and periods are related as follows:

$$\forall \tau_i \in \mathbb{T} : c_i^L \leq c_i^H \leq d_i = p_i \tag{2.16}$$

Each task τ_i generates an unbounded sequence of jobs. A job arrives at α_{ij} and requires γ_{ij} execution time. Job arrivals of sporadic tasks are separated at least by the task's period; in the worst case, every period a job is released. Given the execution time budget and period, we can define the task utilization as $u_i = c_i/p_i$.

Jobs need to finish execution prior to their absolute deadlines $D_{ij} = \alpha_{ij} + d_i$. The execution time of jobs from high criticality tasks can exceed c_i^L, but never c_i^H. Low criticality jobs are not allowed to execute longer than c_i^L. If every job of high criticality tasks can execute for γ_{ij} during $[\alpha_{ij}, D_{ij})$ the task system is mixed-criticality schedulable [147].

To guarantee that every job of high criticality tasks meets its deadline, classic dual-criticality schedulers separate the system operation in two modes: low- and high criticality mode. As long as no job from a high criticality task executes longer than c_i^L, the system stays in low criticality mode, and deadlines of jobs from low- and high criticality tasks are met. If a job from a high criticality task exceeds c_i^L without signaling completion, the scheduler switches to high criticality mode. In high criticality mode, the scheduler immediately stops the release of jobs from low criticality tasks, and drops all unfinished low criticality jobs. This allows to schedule for the common case according to the optimistic execution time budgets c_i^L, while guaranteeing correctness in the uncommon case according to the pessimistic execution time budgets c_i^H.

We can express the later using task utilizations by defining utilizations U for low- and high criticality tasks and their relation to the uniprocessor's supply of $\sigma = 1$ in both modes:

$$U_L^L = \sum_{i:\chi_i=L} c_i^L/p_i \tag{2.17}$$

$$U_H^L = \sum_{i:\chi_i=H} c_i^L/p_i \qquad U_H^H = \sum_{i:\chi_i=H} c_i^H/p_i \tag{2.18}$$

$$U_L^L + U_H^L \leq \sigma \qquad U_H^H \leq \sigma \tag{2.19}$$

For the reader's convenience we provide a list of symbols and abbreviations in Chapter 8 to summarize our notation.

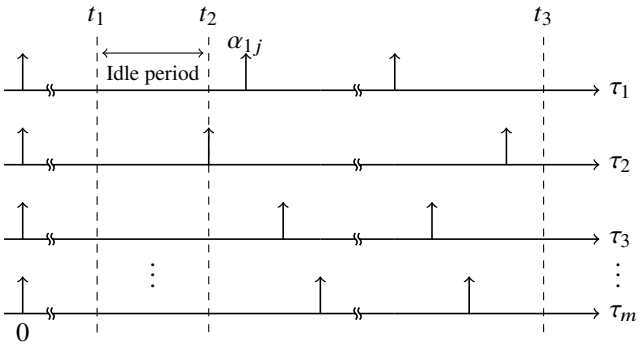


Figure 2.22: Deadline miss following a processor idle period. Each rising arrow is a job arrival, or request, from the task τ_i mentioned on the right. Time progresses from left to right. The idle period closest to the deadline miss at t_3 is between t_1 and t_2.

2.5.2 Earliest deadline first

EDF is a dynamic scheduling approach for single criticality task systems running on a uniprocessor. As the algorithm predates mixed-criticality scheduling, there is no notion of criticalities involved. The following proofs show that EDF is optimal, that is, it can provide the total processor speed to the task's jobs. This is reflected in the relation between the maximum requestable execution time of a task within a time interval, or computation demand, and the processor's supply $\sigma = 1$, which is relevant for the schedulability of earliest deadline first with virtual deadlines (EDF-VD) in both low- and high criticality mode.

To introduce and proof the relation between computation demand and the processor's supply, it is required to proof that there is no processor idle time prior to a deadline miss. Once this property of EDF scheduling is proven, we can proof the feasibility of EDF scheduling as long as the computation demand is below the processor's supply $\sigma = 1$

Proposition 1. *„ When the* [. . .] [*earliest deadline first*] *scheduling algorithm is used to schedule a set of tasks on a processor, there is no processor idle time prior to* [. . .] [*a deadline miss*]. *“* [148]

Proof. The proof is by contradiction, with the proposition that there is processor idle time prior to a deadline miss, as shown in Figure 2.22. The idle period closest to the deadline miss is between t_1 and t_2, and the deadline miss occurs at t_3. The arrival time of jobs, or requests of task τ_i, are denoted by α_{ij}. If from t_2 on all requests of task τ_1 are shifted to happen earlier, such that request α_{1j} starts at t_2, and there was no processor idle time between t_2 and t_3, there will be no processor idle time between t_2 and t_3 if request α_{1j} is shifted.

Repeating the argument for all tasks, letting all requests start at t_2, there will be a deadline miss with no processor idle period prior to it. This contradicts the initial proposition that there is processor idle time prior to a deadline miss, proving Proposition 1. □

The next proof shows that EDF scheduling is feasible if the requested computation (demand) is below the processor's maximum possible computation (supply).

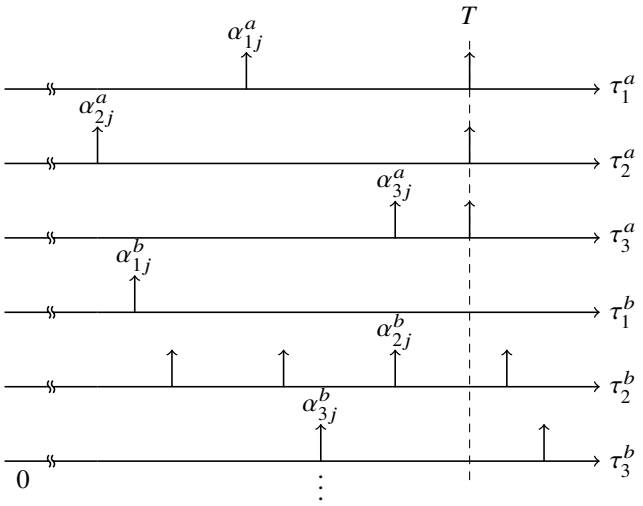


Figure 2.23: Job arrival scenario used in proof of Proposition 2 with implied deadline miss at T.

Proposition 2. *„ For a given set of m tasks, the* [. . .] [*earliest deadline first*] *scheduling algorithm is feasible if and only if* [. . .] [$c_1/p_1+c_2/p_2+\cdots+c_m/p_m \leq 1$]. *“* [148]

Proof. The proof is split up in two parts, to show the necessity followed by the sufficiency.

To show the necessity, the total computation demand is not allowed to exceed the available processor time. If a task releases jobs which execute for c_i time units, its computation demand in the interval $[0, t]$ is $(c_i/p_i)t$. The computation demand of a task set with m tasks is $d(t) = t\sum_{i=1}^{m} c_i/p_i$. If $d(t) > t$, there is clearly no feasible scheduling algorithm.

The sufficiency is shown by contradiction. The *initial proposition* is that task set's utilization $U = \sum_{i=1}^{m} c_i/p_i$ is below or equal to $\sigma = 1$, *and yet* the scheduling algorithm is not feasible, due to an deadline miss in the interval $(t, p_1p_2\cdots p_m)$. According to Proposition 1 there is a $t = T$ $(0 \leq T \leq p_1p_2\cdots p_m)$ at which there is a deadline miss with no processor idle time between $t = 0$ and $t = T$. Figure 2.23 shows the requests of a task set over time, where the corresponding deadlines of $\alpha^a_{1j}, \alpha^a_{2j}, \ldots$ are at T, and $\alpha^b_{1j}, \alpha^b_{2j}, \ldots$ have their deadline beyond T.

In the *first case*, only requests with deadline at T are carried out before T. Each task has $\lfloor T/p_i \rfloor$ requests in the interval $(0, T)$. Only a integral number of requests take place up to T, because p_i is the period between two requests by definition in periodic task systems, and the minimum period (worst case) between two requests in sporadic task systems [149]. The total demand of the task set is $d(T) = \sum_{i=1}^{m} \lfloor T/p_i \rfloor c_i$, and since there is no processor idle period the demand has to be greater that T, $d(T) > T$, for a deadline miss to happen.

Since $x \geq \lfloor x \rfloor \forall x$, we can rewrite the demand as $d(T) = \sum_{i=1}^{m} (T/p_i)c_i$, which is still larger 1. Normalizing by T yields $d(T)/T = \sum_{i=1}^{m} (c_i/p_i) = U > 1$, which contradicts the initial proposition.

In the *second case*, some α^b_{ij} requests are carried out before T. Having a deadline miss at T implies an interval $T' \leq t \leq T$ where only requests with a deadline at or before T are executed. Together they imply that one or more α^b_{ij}

requests are executed until $t = T'$. This implies that all requests initiated before T' with deadlines $\leq T$ are fulfilled before T'.

The computation demand within $T' \leq t \leq T$ is less than or equal to $\sum_{i=1}^{m} \lfloor (T - T')/p_i \rfloor c_i$. To cause a deadline miss, the demand must exceed $T - T'$, which implies an utilization above 1. This contradicts the initial proposition and proofs the sufficiency of Proposition 2. □

2.5.3 Earliest deadline first with allowance

Earliest deadline first with allowance (EDF-Allowance) is a EDF [148] scheduling approach which can tolerate up to k execution time budget overruns within a sliding time window W without deadline violations by leveraging static slack [150]. For each task, the allowance is the time beyond a job's overrun which can be tolerated without compromising the deadline of any job.

Proposition 3 ([150])**.** *With a fault model $^k/_W$, which allows up to k jobs to exceed their WCET over sliding window $W \leq \min_i p_i$, a set of sporadic tasks $\mathbb{T} = \{\tau_1, \ldots, \tau_n\}$ indexed by increasing period and scheduled with EDF, a set $sp(i,k)$ of at most $k-1$ tasks but τ_i having the smallest periods in $\mathbb{T}$, and $l_i(k,t)$ as the set of $k-1$ tasks, excluding τ_i, with the highest value of $1 + \left\lfloor {}^{(t-d_j)}/_{p_j} \right\rfloor$, the maximum allowance A_i^k for any faulty task τ_i is the minimum value of A_i^k satisfying the following equation*:

$$
\begin{aligned}
P &= \operatorname{lcm}_i p_i \\
U^* &= \sum_{\tau_j \in \tau_i \cup sp(i,k)} \frac{A_i^k}{p_j} \\
U + U^* &\leq 1 \\
X_{i,k-1}(t) &= \sum_{d_j \leq t \wedge \tau_j \in l_i(k,t)} \left(1 + \left\lfloor \frac{t - d_j}{p_j} \right\rfloor \right) \\
t &\geq h(t) + \left(1 + \left\lfloor \frac{t - d_i}{p_i} \right\rfloor \right) A_i^k + X_{i,k-1}(t) A_i^k \\
\forall t \in \mathcal{S} &= \cup \left\{ k p_i + d_i, k \in \mathbb{N} \right\} \cap \{ l | \forall l \in [d_i, U^* P) \}
\end{aligned}
$$

Proof. See [150]. □

The intuition behind Proposition 3 is best explained with $h(t)$, which describes the work at each point in time if each task releases jobs with its period synchronously, and is known as demand bound function (DBF) [151]. To derive $h(t)$ we can sum up the product of maximum computation c_i and the maximum number of requests $r(i,t)$ until time t for all tasks: $h(t) = \sum_i c_i r(i,t)$. The maximum number of requests until t is either zero or $k+1$ when $k p_i + d_i \leq t$ is satisfiable. The largest integer satisfying the former equation is $k = \lfloor {}^{t-d_i}/_{p_i} \rfloor$. This allows to define $r(i,t) = \max \left\{ 0, \lfloor \frac{t-d_i}{p_i} \rfloor \right\} + 1$.

The set $\mathcal{S}$ contains all check-worthy time points where $h(t) \leq t$ needs to be evaluated. The check-worthy time points are the absolute deadlines for minimum spaced requests, which describe the worst case.

With allowance, $h(t)$ needs to get extended. The term $\left(1 + \left\lfloor \frac{t-d_i}{p_i} \right\rfloor\right)$ describes the maximum number of requests of task τ_i. The sum in $X_{i,k-1}(t)$ is a sum over the maximum number of requests of the tasks in the set $l_i(k,t)$, with the maximum number of requests in t. All requests are weighted with the allowance to contribute to $h(t)$, which contains products of computation and requests.

2.5.4 Earliest deadline first with virtual deadlines

EDF-VD is a preemptive uniprocessor dynamic scheduling approach for mixed-criticality task systems. It is an extension of EDF scheduling to mixed-criticality task systems by introducing earlier, virtual deadlines. As in EDF, the priority of a job is defined by its deadline [148]. The closer the deadline, the higher the priority of the job. The job with the highest priority is granted access to the processor, and can preempt a currently running job, which returns to the scheduler's job queue.

In contrast to EDF scheduling, EDF-VD introduces the concepts of criticalities and modes, which separates the system's run-time into two modes: In the first mode, jobs from all tasks are scheduled. Once a job from a high criticality task requires more computation than anticipated ($\gamma_{ij} > c_i^L$), the system changes to the second mode, where only jobs from high criticality tasks are scheduled.

To guarantee that all high criticality jobs have enough time to finish prior to their deadlines, EDF-VD introduces earlier, virtual deadlines which reserve time necessary for a successful mode change. These earlier deadlines increase the priority of the task's jobs. Once the switch to high criticality mode is triggered, jobs from high criticality tasks are scheduled with their original deadline.

We can interpret the scheduling in both modes as regular EDF scheduling with different sets of tasks. If both modes satisfy the constraints of regular EDF scheduling, with the transitional phase taken care of by earlier deadlines, the system is guaranteed to never miss a deadline [147].

In this section we reproduce the proof for Proposition 4 from Baruah [147] to present how earlier deadlines can be found analytically, which we extend upon in our approach.

Proposition 4 (EDF-VD schedulability conditions [147])**.** *Given an implicit-deadline task system τ, if either*

$$\sum_{l=1}^{K} U_l(l) \leq 1 \tag{2.20}$$

or, for some k $(1 \leq k < K)$, the following condition holds:

$$1 - \sum_{l=1}^{k} U_l(l) > 0 \quad \textit{and} \quad \frac{\sum_{l=k+1}^{K} U_l(k)}{1 - \sum_{l=1}^{k} U_l(l)} \leq \frac{1 - \sum_{l=k+1}^{K} U_l(l)}{\sum_{l=1}^{k} U_l(l)} \tag{2.21}$$

, then τ can be correctly scheduled by EDF-VD.

For the reader's convenience we reproduce the demonstration for dual-criticality task systems by setting $K = 2$, and refer to the criticality levels by their symbolic names as introduced with our notation.

Proposition 5. *Given an implicit-deadline task system τ, if either*

$$U_L^L + U_H^H \leq 1 \tag{2.22}$$

or the following condition holds:

$$1 - U_L^L > 0 \quad \textit{and} \quad \frac{U_H^L}{1 - U_L^L} \leq \frac{1 - U_H^H}{U_L^L} \tag{2.23}$$

, then τ can be correctly scheduled by EDF-VD.

We now differentiate two cases: *1)* EDF is feasible; and *2)* EDF is not feasible. In the first case, the system can be scheduled with a worst case reservation. In the second case, there is an $x \leq 1$ that

$$U_L^L + \frac{U_H^L}{x} \leq 1 \tag{2.24}$$

and

$$xU_L^L + U_H^H \leq 1 \tag{2.25}$$

hold and EDF-VD is a correct scheduling policy for τ.

Definition 6. Virtual utilization is the utilization of the task system considering deadline scaling, that is, the effective utilization encountered by the scheduler:

$$\hat{U} = \sum_{i:\chi_i=L} \frac{c_i}{d_i} + \sum_{i:\chi_i=H} \frac{c_i^L}{xd_i} \tag{2.26}$$

Lemma 7. *Virtual utilization in low criticality mode is bounded.*

Proof. For an implicit deadline task system we can substitute deadlines with periods:

$$\hat{U} = \sum_{i:\chi_i=L} \frac{c_i}{p_i} + \sum_{i:\chi_i=H} \frac{c_i^L}{xp_i} \tag{2.27}$$

Using Equations (2.17) and (2.18) to separate in worst case reservation of low criticality tasks, and high criticality tasks executing in low mode, uniformly scaled to reserve time with $x < 1$:

$$\hat{U} = U_L^L + \frac{U_H^L}{x} \tag{2.28}$$

Therefore the virtual utilization in low criticality mode is bounded, and if $\hat{U} \leq 1$, correctness is ensured. □

Lemma 8. *Virtual utilization in high criticality mode is bounded.*

Proof. Assume by contradiction a deadline miss at t_f. I is a minimal set of jobs released by τ with deadline miss at t_f, first job arrival at time zero, and every proper subset of I would be schedulable by EDF-VD. Further assume that Equations (2.24) and (2.25) hold.

The job missing the deadline is of high criticality, because at t_f the system is in high mode. Let t^* denote the time where high behavior is first exhibited.

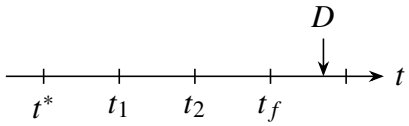


Figure 2.24: Visualization of Lemma 9. If a job with deadline $D > t_f$ receives execution between $[t_1, t_2)$, then there are no jobs pending during $[t_1, t_2)$ with deadline at most t_f, else they would be executed instead.

Lemma 9. *All jobs receiving execution in $[t_v, t_f)$ have deadlines at most t_f.*

Proof. If a job with deadline $D > t_f$ as in Figure 2.24 receives execution between t^* and t_f, for example in interval $[t_1, t_2)$, this would mean that during $[t_1, t_2)$ there are no jobs pending with deadline at most t_f, else they would be executed instead.

Accordingly the set of jobs with arrival time at least t_2 need to miss a deadline at t_f, but this set is a proper subset of I because jobs prior to t_2 are excluded, and as a proper subset it should be schedulable by EDF-VD, else we contradict the assumed minimality of I. Therefore all jobs receiving execution in $[t^*, t_f)$ have deadlines at most t_f. □

Lemma 10. *The cumulative execution requirement $\eta_i(t_f)$ for any* low *criticality task τ_i until t_f is bounded*:

$$\eta_i(t_f) \leq (a_{h0} + x(t_f - a_{h0}))u_i \tag{2.29}$$

Proof. No job J_{ij} of τ_i, which is of low criticality, will receive execution time after t^* by design, where high criticality behavior is observed for the first time.

Let J_{h0} denote the job with the earliest arrival time a_{h0} of all jobs executing in $[t^*, t_f)$. Because J_{h0} executes in $[t^*, t_f)$, it is by definition a high criticality job. Each low criticality job executing after the arrival a_{h0} must have a deadline no longer than the absolute virtual deadline of J_{h0}, $\hat{D}_{h0} = a_{h0} + x(D_{h0} - a_{h0})$, else the EDF scheduler would select J_{h0} for execution.

By Lemma 9 the real deadline of J_{h0} is $D_{h0} \leq t_f$, therefore no job with deadline greater than $a_{h0} + x(t_f - a_{h0})$ executes after a_{h0}.

Suppose a job with deadline greater than $a_{h0} + x(t_f - a_{h0})$ executed some time before a_{h0}, with t_l as the last time it executed. This means at this time there were no jobs with deadlines smaller or equal to $a_{h0} + x(t_f - a_{h0})$ awaiting execution. Hence, the set of jobs with arrival time $a_{ij} \geq t_l$ in I would miss a deadline, contradicting the assumed minimality of I. This implies there are at most $(a_{h0} + x(t_f - a_{h0}))/p_i$ jobs of τ_i until t_f with execution requirement c_i; bounding the total execution requirement with help of the durations derived

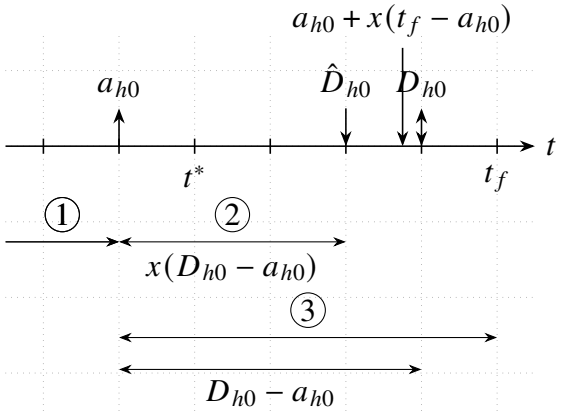


Figure 2.25: Visualization of Lemma 10. The relationship between the circled durations 1, 2, and 3 is used in Equation (2.30) to derive a bound $\eta_i(t_f)$ for the cumulative execution requirement until t_f.

above as shown in Figure 2.25:

$$\underbrace{\frac{a_{h0}}{p_i}c_i}_{①} + \underbrace{\frac{x(D_{h0} - a_{h0})}{p_i}c_i}_{②} \leq \underbrace{\frac{a_{h0}}{p_i}c_i}_{①} + \underbrace{\frac{x(t_f - a_{h0})}{p_i}c_i}_{③} \tag{2.30}$$

$$\eta_i(t_f) \leq \frac{a_{h0}}{p_i}c_i + \frac{x(t_f - a_{h0})}{p_i}c_i \tag{2.31}$$

$$\eta_i(t_f) \leq (a_{h0} + x(t_f - a_{h0}))u_i \tag{2.32}$$

Therefore for any *low* criticality task τ_i its cumulative execution requirement until t_f is bounded by Equation (2.32). □

Lemma 11. *The cumulative execution requirement $\eta_i(t_f)$ for any* high *criticality task τ_i until t_f is bounded*:

$$\eta_i(t_f) \leq \frac{a_{h0}}{x}u_i^L + x(t_f - a_{h0})u_i^H \tag{2.33}$$

Proof. To bound the cumulative execution requirement of *high* criticality tasks, differentiate two cases: *A)* Task τ_i does not release a job at or after a_{h0}; and *B)* Task τ_i releases one or more jobs at or after a_{h0}.

In Case *A)* each job of τ_i has a virtual deadline $\hat{D} \leq a_{h0} + x(t_f - a_{h0})$, otherwise the assumed minimality of I is contradicted. Hence, there are at most $(a_{h0} + x(t_f - a_{h0}))/p_i$ of them.

Since from all jobs executing in $[t^*, t_f)$ J_{h0} is the job with the earliest arrival time a_{h0}, and the jobs considered in this case are *not* released at or after a_{h0}, they do not execute in $[t^*, t_f)$, limiting their execution requirement to $c \leq c_i^L$, because high criticality behavior is observed at t^* first. Before a_{h0} jobs from τ_i

have a maximum demand of c_i^L, because $t^* > a_{h0}$.

$$\hat{D}_i = xD_i \leq a_{h0} + x(t_f - a_{h0}) \tag{2.34}$$

$$D_i \leq \frac{a_{h0}}{x} + t_f - a_{h0} \tag{2.35}$$

$$D_i \frac{c_i^L}{p_i} \leq \left(\frac{a_{h0}}{x} + t_f - a_{h0}\right) \frac{c_i^L}{p_i} \tag{2.36}$$

$$\leq \frac{a_{h0}}{x} u_i^L + (t_f - a_{h0}) u_i^H \tag{2.37}$$

The bound differentiates between the duration until a_{h0}, where low mode utilization applies, and the duration $[a_{h0}, t_f)$, which is upper bounded with high mode utilization.

In Case *B)* $a_{i0} \geq a_{h0}$ denotes the first release of a job from τ_i. Previous jobs did not execute in $[t^*, t_f)$ by definition of a_{i0} and a_{h0}. Therefore until a_{i0} the execution requirement is bounded by c_i^L, and after that by c_i^H. Hence it holds

$$a_{h0} u_i^L + (a_{i0} - a_{h0}) u_i^H + (t_f - a_{i0}) u_i^H \tag{2.38}$$

$$= a_{h0} u_i^L + (t_f - a_{h0}) u_i^H \tag{2.39}$$

$$\leq \frac{a_{h0}}{x} u_i^L + (t_f - a_{h0}) u_i^H \tag{2.40}$$

The resulting upper bound is identical to Case *A)*, and the cumulative execution requirement $\eta_i(t_f)$ for any high criticality task τ_i until t_f is bounded: $\eta_i(t_f) \leq \frac{a_{h0}}{x} u_i^L + (t_f - a_{h0}) u_i^H$. □

This concludes the proof for Lemma 8 that the virtual utilization in high criticality mode is bounded. □

Lemma 12. *The bounded execution requirement of all tasks $\eta(t_f)$ shows that the assumed deadline miss contradicts Equation* (2.25)*.*

Proof. By summing the execution requirement $\eta_i(t_f)$ over all tasks, the resulting duration needs to exceed t_f for a deadline miss:

$$\eta(t_f) = \sum_{i:\chi_i = L} \eta_i(t_f) + \sum_{i:\chi_i = H} \eta_i(t_f) \tag{2.41}$$

$$\leq \sum_{i:\chi_i = L} (a_{h0} + x(t_f - a_{h0})) u_i + \tag{2.42}$$

$$\sum_{i:\chi_i = H} \frac{a_{h0}}{x} u_i^L + (t_f - a_{h0}) u_i^H$$

$$= \sum_{i:\chi_i = L} a_{h0} u_i + x(t_f - a_{h0}) u_i + \tag{2.43}$$

$$\sum_{i:\chi_i = H} \frac{a_{h0}}{x} u_i^L + (t_f - a_{h0}) u_i^H$$

We can now start to rearrange terms and replace the sums over tasks with the same criticality with our convenient notation:

$$\eta(t_f) = a_{h0}\left(\sum_{i:\chi_i=L} u_i + \sum_{i:\chi_i=H} \frac{u_i^L}{x}\right) + \tag{2.44}$$

$$(t_f - a_{h0})\left(\sum_{i:\chi_i=L} x u_i + \sum_{i:\chi_i=H} u_i^H\right)$$

$$= a_{h0}\left(U_L + \frac{U_H^L}{x}\right) + (t_f - a_{h0})\left(xU_L + U_H^H\right) \tag{2.45}$$

$$\leq a_{h0} + (t_f - a_{h0})\left(xU_L + U_H^H\right) \tag{2.46}$$

Assumed deadline miss implies $\eta(t_f) > t_f$:

$$a_{h0} + (t_f - a_{h0})\left(xU_L + U_H^H\right) > t_f \tag{2.47}$$

$$xU_L + U_H^H > 1 \tag{2.48}$$

This contradicts Equation (2.25). □

Lemma 13. *If Equation* (2.23) *holds, then Equations* (2.24) *and* (2.25) *hold, and any x where*

$$\frac{U_H^L}{1 - U_L^L} \leq x \leq \frac{1 - U_H^H}{U_L^L}$$

suffices as a deadline scaling parameter.

Proof. Rewriting Equations (2.24) and (2.25) shows that a valid x exists:

$$U_L + \frac{U_H^L}{x} \leq 1 \tag{2.49}$$

$$xU_L + U_H^L \leq x \tag{2.50}$$

$$U_H^L \leq x - xU_L \tag{2.51}$$

$$\leq x\,(1 - U_L) \tag{2.52}$$

$$\frac{U_H^L}{1 - U_L} \leq x \tag{2.53}$$

Because $1 - U_L$ is always > 0, x is valid.

$$xU_L + U_H^H \leq 1 \tag{2.54}$$

$$xU_L + \leq 1 - U_H^H \tag{2.55}$$

$$x \leq \frac{1 - U_H^H}{U_L} \tag{2.56}$$

Because U_L is always > 0, x is valid. □

Now we can prove Proposition 5 to show that EDF-VD is feasible for τ:

Proof. By Lemmas 7, 8, 12 and 13 Proposition 5 holds. Therefore EDF-VD is feasible for τ. □

2.5.5 Earliest deadline first with non-uniform virtual deadlines

Given a task set, mixed-criticality scheduling can increase the chance to find a valid schedule. We differentiate EDF-based mixed-criticality approaches by how they construct their virtual deadline scaling factors, and if they result in uniform- or non-uniform virtual deadline scaling factors. We reproduce the original earliest deadline first with non-uniform virtual deadlines (EDF-NUVD) schedulability proposition [147] here for the reader, as our approach earliest deadline first with improved virtual deadlines (EDF-IVD) is closely related.

Proposition 14 (EDF-NUVD schedulability [147])**.** *Let* τ *be a* [...] [*dual-criticality*] *task* [...] [*set*] *and let* $0 < x_i < 1$*, for each* [...] [*high criticality task*]*. If*

$$U_L^L + \sum_{i:\chi_i=H} u_i^L / x_i \leq 1 \tag{2.57}$$

$$\sum_{i:\chi_i=H} u_i^H / (1 - x_i) \leq 1 \tag{2.58}$$

then τ *is schedulable by EDF-NUVD.*

Proof. No deadline is missed by EDF-NUVD in low criticality mode if Equation (2.57) holds. The first time where a job from a high criticality task exceeds its low criticality execution time budget is at t^*. Consider a job J_{ij} of a high criticality task τ_i that is active at t^*. The job arrives at α_{ij}, and has a absolute deadline at $D_{ij} = \alpha_{ij} + d_i$. Before t^*, J_{ij} is EDF-scheduled according to its virtual absolute deadline $\hat{D}_{ij} = \alpha_{ij} + x_i d_i$.

In low criticality mode, all jobs would meet their virtual relative deadlines. Since J_{ij} is still active at t^*, its earliest virtual absolute deadline is at the first overrun time: $\hat{D}_{ij} \geq t^*$. Therefore the duration between the first overrun time and absolute deadline is larger or equal to the duration between absolute deadline and virtual absolute deadline, as shown in Figure 2.26:

$$D_{ij} - t^* \geq D_{ij} - \hat{D}_{ij} \tag{2.59}$$

The worst case assumption is that each high criticality task has an active job with execution time equal to its execution time budget in high criticality mode, and the execution needs to take place during $D_{ij} - \hat{D}_{ij}$. Therefore Equation (2.58) regards the task set after t^* as a single-criticality task set under the worst case assumption, where each high-criticality task τ_i has period $D_{ij} - \hat{D}_{ij} = d_i(1 - x_i)$ and increased task utilization $u_i^H / (1 - x_i)$. If the sum over all increased task utilizations is below the supply of one, this single-criticality task set is schedulable.

□

In essence, by virtual relative deadlines we can reserve time for execution in high criticality mode, but we need to find a reservation which is still schedulable. If we choose very small virtual deadline scaling factors, we get earlier virtual relative deadlines and therefore higher task utilizations in low criticality mode according to Equation (2.57), and minimal task utilizations in high criticality mode under worst case assumptions according to Equation (2.58). Very large virtual deadline scaling factors result in nearly no reservation in low criticality

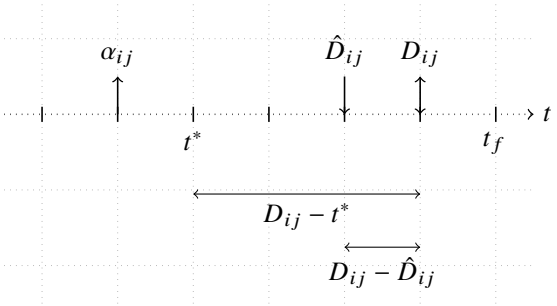


Figure 2.26: Visualization of Proposition 14.

mode, and maximum task utilizations in high criticality mode under worst case assumptions.

2.6 Conclusion

This chapter provided the preliminaries for CPS design, covering a taxonomy of fault-tolerant systems, sources of errors, and fault-tolerance techniques, followed by low-power design fundamentals, a motivation for graceful degradation, and dynamic real-time scheduling.

The taxonomy of fault-tolerant systems allows us to categorize our CPS design approach by concurrent error detection, compensation, and reinitialization to implement a system recovery fault tolerance strategy. One major source of errors in ICs is radiation, which we traced from ionizing radiation to unwanted changes in data state. By modeling the radiation environment and the circuit's susceptibility to radiation-induced faults separately, we could differentiate between different SEEs, discuss their implications, and come up with simulation models.

Our survey of error detection identified anomaly detection in data and behavior, dynamic verification, and redundant execution as major categories for CPS-suitable error detection approaches. For COTS-based CPSs, we concluded that software-based and FPGA-compatible approaches are favorable. Contrary, we refrained from masking techniques, as the additional power consumption is at odds with the limited energy budgets of CPSs. Moreover, fault detection and diagnosis are not suitable for COTS-based CPS due to missing white box reference models.

Instead, containment techniques are especially applicable for COTS-based CPS because they require control of interfaces instead of component internals, and state recovery techniques complement the favorable error detection approaches to implement system recovery in our envisioned CPSs. Our review of the fundamentals of low-power design highlighted that the selected fault tolerance techniques need to integrate with state of the art low-power approaches like power gating or voltage scaling, and we identified graceful degradation as a pattern to enable the integration of fault-tolerant and low-power design.

At last, we motivated and introduced the mixed-criticality scheduling problem, which arises if we combine fault-tolerant and real-time CPS design. Resorting to the preliminaries of this chapter, the next chapter presents our approach to CPS design, which combines fault-tolerant and low-power design with real-time guarantees.

3 System architecture

This chapter describes the system architecture of our proposed CPSs which integrates techniques of dependable- and low-power design with real-time guarantees.

In our system, each processing node consists of a CPU with dedicated memory. All homogeneous processing nodes are independent of each other, and each processing node runs its own real-time operating system (RTOS), with the mixed-criticality application distributed among them. We assume single errors, where no further error manifests during handling of the initial error, and that errors are translatable to time.

The following sections describe in detail how we distribute a mixed-criticality application over processing nodes, followed by our implementation of error detection. Next, we explain the dynamic priority scheduling of jobs within a processing element, and how the scheduler's decisions are influenced by errors. The last section links the error behavior within our architecture to the mixed-criticality scheduling model which provides the real-time guarantees.

3.1 Partitioning of multicore system

Modern COTS-based fault-tolerant systems are attractive because of the additional performance they provide, but their certification is a challenge [152]. One way to build fault-tolerant multicore systems is to treat cores as redundant module [153], or their submodules [154]. Instead of this spatial approach to fault tolerance we can resort to temporal redundancy in our multicore system if we partition our application [155], which transforms the multicore problem into multiple unicore problems solvable with one of our scheduling approaches earliest deadline first with virtual deadlines for single errors (EDF-VD-SE) or earliest deadline first with non-uniform virtual deadlines for single errors (EDF-NUVD-SE).

Both scheduling approaches consider mixed-criticality applications, where tasks have different criticalities. The vital core functionality of the CPS is implemented by tasks of the highest criticality. Often it is desirable to implement further functionality of lower criticality. Both high- and low criticality tasks compete for CPU time, which requires a schedule to ensure that all tasks are served prior to their deadline. This schedule is generated during system operation by the scheduler.

Prior to system operation, tasks are distributed over the processing nodes in our architecture. As there are multiple processing nodes, we can view the problem of distributing tasks to processing nodes in two ways: Treating the set of processing nodes as a single processing element with a supply of $\sigma \leq \sum_i \sigma_i$, or as a set of distinct processing elements, each with their own supply. The former approach falls short, as the complexity of a distributed RTOS is beyond

the capabilities of the targeted resource-constrained CPSs. But the later approach is feasible, as the distribution of tasks and schedulability verification can be done at design time, resulting in a fully static distribution during system run time.

We face a multiprocessor real-time scheduling problem, which consists of two parts [156]: *1)* how to allocate tasks to processors; and *2)* when and in what order jobs should execute. As stated in the previous paragraph, we prefer a static allocation at design time, without task migration to different processing nodes. Such partitioned scheduling approaches have several advantages: Overruns affect only a single processing element compared to global scheduling approaches, where an overrun can have consequences for all processing elements. Moreover, there are no task- or job migration costs, and once partitioned the multiprocessor scheduling problem reduces to multiple uniprocessor problems [156].

Still, global scheduling approaches with job migration between processing elements have some advantages over partitioned scheduling: As long as a processor is idle, no preemption is necessary, which reduces the context switch overhead, and spare capacity or slack can be used globally [156]. If task sets can change during system operation, global scheduling is beneficial as no reallocation of tasks is necessary. For our resource-constrained CPSs, the advantages of partitioned scheduling outweigh the advantages of global scheduling.

The task allocation problem in partitioned scheduling is typically formulated as a constrained optimization problem, which can be solved with graph theoretic, integer programming, or approximation methods [157]. For example, we can formulate the allocation problem as a variant of the bin packing problem where we want to allocate tasks equally by their utilization over a fixed number of processing elements. In the analogy of packing parts with fixed volumes in bins, the parts are tasks and their volume is the utilization. If we consider the number of processing elements fixed, we get a multiway number partitioning problem, where the fixed number of bins are the number of processing elements. Using greedy number partitioning, we distribute tasks by their utilization on processing elements such that the sum of utilizations for each processing element are nearly equal. One approximate algorithm to solve a multiway number partitioning problem by greedy number partitioning is longest-processing-time-first (LPT), shown in Algorithm 1.

Algorithm 1 LPT for multiway number partitioning [158].

Require: Set of numbers $\mathcal{S}$, Positive integer m
Ensure: Partition of $\mathcal{S}$ into m subsets
1: $S \leftarrow$ CreateEmptySubsets(m)
2: $O \leftarrow$ SortDescending($\mathcal{S}$)
3: **for all** $n \in O$ **do**
4: $\quad s \leftarrow$ GetSubsetWithSmallestSum(S)
5: $\quad s \leftarrow s \cup n$
6: **end for**
7: **return** S

We distribute tasks according to our model, which we presented in Section 2.5.1, by their utilization. It is important to note that our model is not tied to any particular distribution algorithm. Instead, our model provides the utilization

values, which are necessary for task distribution by any suitable approach. Therefore we compare three approaches: Random distribution, a heuristic approach, and bin packing. Our goal is to distribute all n_H high criticality tasks over n_P processing elements, such that the amount of possible low criticality work is maximized. Our heuristic approach sorts high criticality tasks by their descending utilization, and starts by distributing the task with the highest utilization to the first processing element. The task with second highest utilization is distributed to the second processing element. Once the last processing element got assigned a task, the next task is assigned to the first processing element, looping and continuing the distribution as described until all high criticality tasks are distributed.

Next, we solve for each processing element the schedulability conditions according to one of our approaches EDF-VD-SE or EDF-NUVD-SE. If for any processing element the schedulability conditions are not satisfiable, the system is not schedulable, and the designer needs to lower the number of high criticality tasks, or add more processing elements. Else, our approaches return the maximum supported utilization of low criticality tasks for each processing element.

Similar to the high criticality tasks we can now sort and distribute the low criticality tasks, but we sort the processing elements by their descending maximum supported utilization. This simple heuristic results in an assignment of the low criticality tasks with the highest utilization to the processing element with the maximum supported low criticality utilization. The corresponding pseudo code is shown in Algorithm 2.

As an example, let us consider the distribution of $\mathbb{T}_H = \{\tau_1, \tau_2, \tau_3, \tau_4\}$and $\mathbb{T}_L = \{\tau_5, \tau_6\}$, both already sorted according to Algorithm 2, to $n_p = 3$ processing elements. After line 8, we get $p_1 = \{\tau_1, \tau_4\}$, $p_2 = \{\tau_2\}$, and $p_3 = \{\tau_3\}$. The low criticality tasks τ_5, τ_6 are distributed to p_3 and p_2, as they feel less demand compared to p_1.

We can evaluate the heuristic approach by generating random task sets, distribute them, and count how many task sets distributions result in schedulable task sets for all cores in the multiprocessor system-on-chip (MPSOC). For this we implemented the distribution heuristic of Algorithm 2 as part of our framework, which we further describe in Chapter 4. We generate random task sets with the `UUnifast` algorithm [159], using a parameterization which represents avionics systems with periods between $p_l = 25$ and $p_u = 1000$ [160]. Moreover, we consider the impact of error detection within the execution time budgets, as explained in detail in Section 3.2, by selecting a pessimism [161] of $z = 2$. Figure 3.1 shows how many task sets result in schedulable task sets for all processing elements. With two processing nodes in the MPSOC only a fraction of task sets are schedulable in the depicted utilization range. If we increase the number of processing nodes, the heuristic has more choices and can achieve better results, especially for task sets with higher utilization. Furthermore, increasing the number of processing nodes provides more supply, which increases the chance to schedule task sets in the depicted high utilization range.

While the simple heuristic in Algorithm 2 can result in schedulable systems, the load is distributed non-uniform. If this is problematic due to aging or thermal constraints, a uniform distribution is recommended. One way to achieve near-uniform distribution is solving the bin packing problem mentioned earlier.

Algorithm 2 Distribution of mixed-criticality tasks to processing elements.

Require: Task set $\mathbb{T}$, Set of processing elements P
Ensure: Schedulability of $\mathbb{T}$ on n_P processing elements

$\mathbb{T}_H \leftarrow \{\tau_i \mid \forall \chi_i = H\}$
$T_H \leftarrow \text{ToList}(\mathbb{T}_H)$
$T_H \leftarrow \text{SortDescendingByUtilization}(T_H)$ ▷ In place sort
while $T_H \neq \emptyset$ **do** ▷ Distribute high criticality tasks
    for all $p \in P$ **do**
        $p \leftarrow \text{Head}(T_H)$
    end for
end while
$\mathbb{U} \leftarrow \emptyset$
for all $p \in P$ **do**
    if $s \leftarrow \text{IsSchedulable}(p)$ **then**
        $U_L^{\max} \leftarrow \text{GetMaximumLowtaskUtilization}(p)$
        $\mathbb{U} = \mathbb{U} \cup (p, U_L^{\max})$ ▷ Add tuple to $\mathbb{U}$
    else
        return False ▷ Not schedulable
    end if
end for
$U \leftarrow \text{ToList}(\mathbb{U})$
$U \leftarrow \text{SortDescendingByUtilization}(U)$ ▷ In place sort
$\mathbb{T}_L \leftarrow \{\tau_i \mid \forall \chi_i = L\}$
$T_L \leftarrow \text{ToList}(\mathbb{T}_L)$
$T_L \leftarrow \text{SortDescendingByUtilization}(T_L)$
$i \leftarrow 0$
while $T_L \neq \emptyset$ **do** ▷ Distribute low criticality tasks
    $(p, u) \leftarrow U[i]$
    $p \leftarrow \text{Head}(T_L)$
    $i \leftarrow i + 1 \mod n_p$
end while
return P ▷ Each $p \in P$ contains list of associated tasks

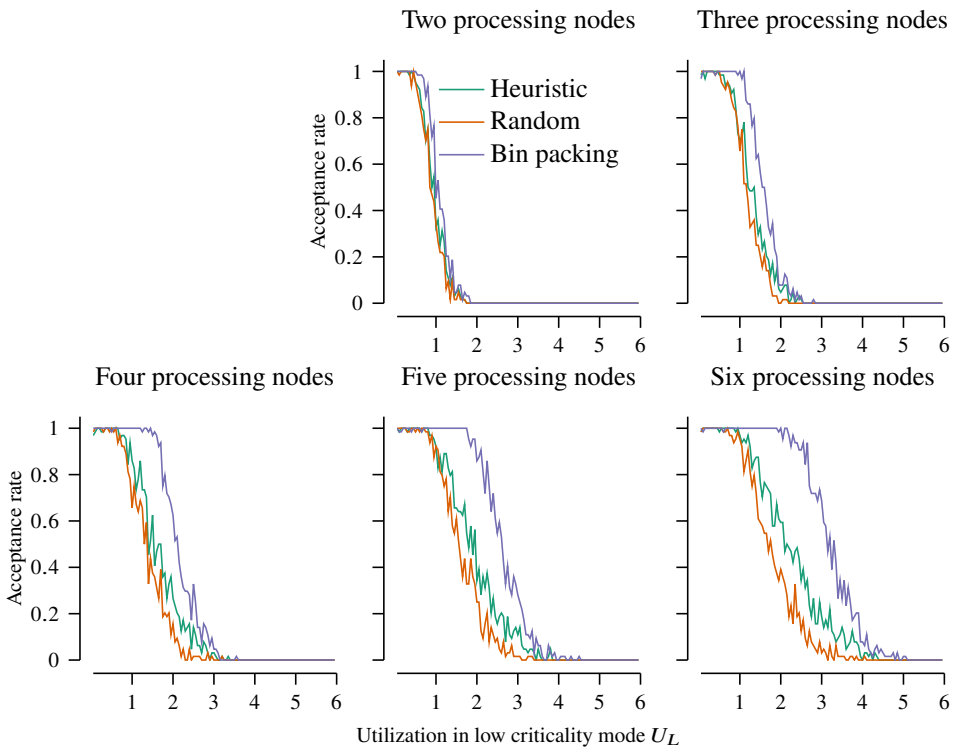


Figure 3.1: Acceptance rates of random task distribution and distribution according to Algorithm 2 and bin packing for different MPSOCs. Increasing number of processing elements in the MPSOC allows to schedule more task sets in the utilization range between 1.5 and 6. Moreover, the heuristic results in up to 0.289 better acceptance compared to random allocation. Still, bin packing results in best acceptance rates.

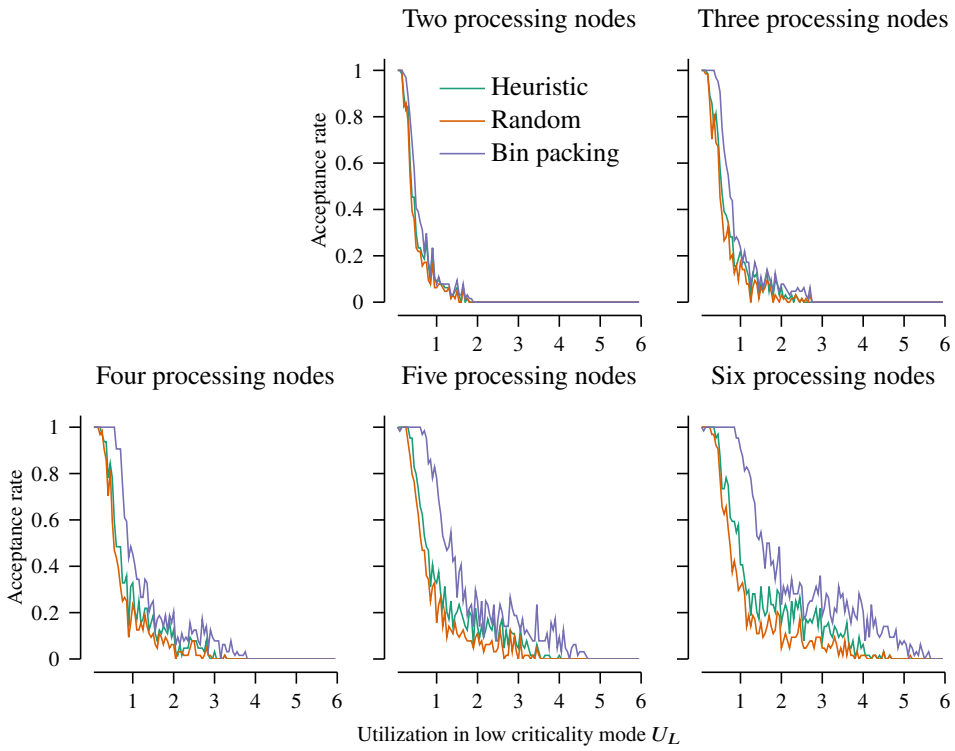


Figure 3.2: Acceptance rates of random task distribution and distribution according to Algorithm 2 and bin packing for different MPSOCs with increased pessimism.

We pack tasks by their utilization as volume, where utilization is the sum of utilization in low- and high criticality mode. While the summed utilization has no direct correspondance with the utilization during system operation, it seems like a good indicator to guide the allocation algorithm, as seen in Figure 3.1. Moreover, we observe different results for task sets with increased pessimism of $z = 5$ in Figure 3.2. While bin packing still provides the best results, the overall acceptance rates are lower, decline earlier, and are more random for higher utilizations. Therefore lower pessimism is beneficial to achieve more schedulable systems, and still bin packing results in best acceptance rates.

3.2 Error detection in hard- and software

To implement a system recover fault tolerance strategy, we need error detection capabilities. Error detection in our envisioned CPS makes use of temporal redundancy, as the power consumption of spatial redundancy techniques like DMR is prohibitive for energy-constrained CPS. Two approaches are especially attractive: software-based error detection by duplicated instructions, and circuit-based error detection by static CMOS error detection sequentials (EDSs).

Both error detection techniques fit well in our CPS design, as they are compatible with COTS components. Software-based error detection by duplicated instructions can be implemented without modification of RTOS kernel sources by post-processing the compiled application. Circuit-based error detection with static CMOS EDSs is challenging, as most EDSs are custom designs and target application-

specific integrated circuits (ASICs), but we developed a fully static CMOS EDS suitable for FPGA implementations [162].

Due to our demand of real-time guarantees, exploiting temporal redundancy for error detection might seem odd at first. The additional time for computation with error detection increases the system's utilization. This in turn increases the difficulty to guarantee always a valid schedule. But the turn-around time to check if a valid schedule is always possible is near zero, as there are either analytical solutions or very fast numerical approaches to generate an answer *during system design*.

3.2.1 Software-based error detection

In software-implemented hardware fault tolerance (SIHFT), pure software approaches are available for control flow checking and detection of transient errors in the data path. These approaches work on different granularity levels, from applications over procedures down to individual instructions. Instruction-level error detection can be implemented by redundant multithreading (RMT), or thread-local error detection. Most RMT approaches require custom hardware [163] or spare processing cores [164], which are unsuitable assumptions for our targeted systems. Therefore thread-local error detection is preferable in our case.

The general idea is to execute a piece of code multiple times, followed by a comparison of the generated result. For example, an arithmetic function might be calculated several times, and the numerical results are compared. If results differ, an error occurred. This approach is suitable for transient errors, where at least one calculation result is not corrupted. In case of permanent faults, the error detection can fail if all calculated results are identical and wrong. Moreover, precautions for loading- and storing values into memory are necessary to avoid state corruptions. Approaches like near zero silent data corruption (nZDC) protect storage operations by reloading and checking, and loads by duplication and comparison [165]. Besides data, the control flow of the application needs to be protected and functionality preserved, especially with approaches that reformulate control flow to add redundancy. One example is triplication of instruction sequences and majority voting of results, where the partitioning of machine registers into three sets and execution of three independent instruction sequences allows to detect and re-execute exception-raising instructions [166]. To avoid single points of failure, checking or majority voting can be intervened between instruction sequences by post-store error checking, where the redundant sequences load the stored result of the main sequence for checking [167]. State-of-the-art approaches combine above techniques to protect both the data path and control flow by use of redundant registers for arithmetic operations, memory access, and the program counter [168]. Still, the overhead of all duplicated instructions extend the required computation time, and needs to be balanced with energy- and performance budgets. Therefore probabilistic approaches [169], which combine error detection based on symptoms like hardware exceptions, cache misses, and branch mispredictions with duplicated instructions, or selective procedure call duplication (SPCD) [78] are more in line with our targeted systems.

SPCD balances error detection capabilities with energy consumption. Instead of duplicating all instructions, only some procedures have their instructions

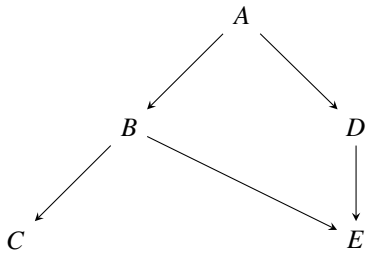


Figure 3.3: Example procedure call graph

duplicated. This reduces the amount of executed redundant instructions to save energy, but increases the error detection latency. Note there is an upper limit to the error detection latency, which is considered in the allocated additional time in the execution time budgets from our mixed-criticality application model.

Applying SPCD to a program requires knowledge about the calling behavior of the program and how to duplicate instructions. The calling behavior is modeled with a procedure call graph, which is a directed acyclic graph (DAG). Each node in the procedure call graph represents a procedure, and edges between nodes represent procedure call statements. For example, an edge from node A to node B indicates that procedure A calls procedure B, as shown in Figure 3.3. The top or start node of the procedure call graph is the main procedure of the program.

Instruction duplication is done according to the rules in Table 3.1. Both variables and constant are *identifiers*, and the apostrophe is used to differentiate between duplicated and original identifier, like in x and x' for the duplicated identifier. *Expressions* are either arithmetic or logic expressions with identifiers, and expr' indicates the duplicated expression expr An assignment *statement* is when an identifier gets assigned the result of an expression evaluation. Conditional statement, as shown in Table 3.1, determine the control flow of a program. They can't be duplicated directly, instead expressions in conditional statements are checked after the conditional statements with their duplicated expressions.

As an example let us consider the program in Algorithm 3. By partial duplication, using the rules in Table 3.1, we can arrive at the protected code in Algorithm 4 [78]. To select the necessary amount of duplication, within an error detection latency constraint λ, Algorithm 5 is used. The algorithm operates in two phases: First, all leaf nodes are checked if duplicating their statements can satisfy the error detection latency constraint λ, where α is the required comparison time and λ_X the execution time of procedure X:

$$2\lambda_X + \alpha < \lambda \tag{3.1}$$

This check is recursively repeated for all parent nodes, and all procedures that satisfy Equation (3.1) are marked. In the second phase, starting from the root node, procedure calls to marked procedures are duplicated. The call graph is traversed from the root node, as shown in lines 7 and 8. The height of G is the maximal length of a directed path in G, and the depth of node X is the number of edges in the path from the root to the node. At the end of Algorithm 5, the resulting protected program satisfies Equation (3.1).

In our mixed-criticality model, the error detection is considered within the execution time budgets. For example, Figure 3.4 shows the probability density

Original	Duplicated

x_0 = expr	x_0 = expr
	x_0' = expr′
	if $x_0 \neq x_0'$ **then**
	ErrorHandler
	end if

while expr **do**
stmt
end while
	while expr **do**
	if $\overline{\text{expr}}'$ **then**
	ErrorHandler
	end if
	stmt
	end while

if expr **then**
stmt1
else
stmt2
end if
	if expr **then**
	if $\overline{\text{expr}}'$ **then**
	ErrorHandler
	end if
	stmt1
	else
	if expr′ **then**
	ErrorHandler
	end if
	stmt2
	end if

for stmt1; expr; stmt2 **do**
stmt3
end for
	for stmt1; expr; stmt2 **do**
	if $\overline{\text{expr}}'$ **then**
	ErrorHandler
	end if
	stmt3
	end for

Table 3.1: Duplication rules of SPCD [78]

Algorithm 3 Unprotected example program [78]

```
1: procedure A
2:     a ← B(b)
3:     c ← c + a
4: end procedure
5: function B(b)
6:     d ← 2b
7:     return d
8: end function
```

Algorithm 4 Protected example program with SPCD [78]

```
1:  procedure A
2:      a ← B(b)
3:      a′ ← B(b’)                                  ▷ Duplicated call
4:      if a ≠ a′ then
5:          ErrorHandler()
6:      end if
7:      c ← c + a
8:      c′ ← c′ + a
9:      if c ≠ c′ then
10:         ErrorHandler()
11:     end if
12: end procedure
13: function B(b)
14:     d ← 2b
15:     return d
16: end function
```

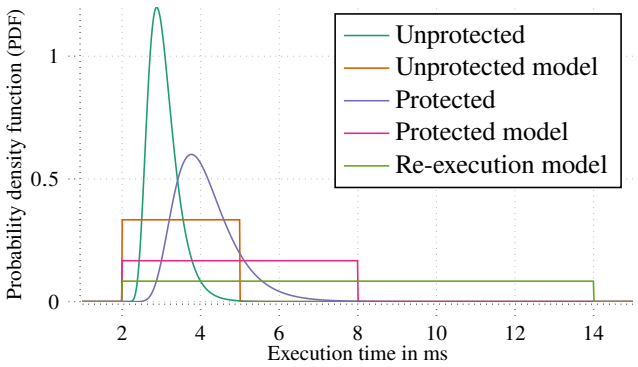


Figure 3.4: Model of execution times for a task with error detection.

function (PDF) of a task's execution time, labeled „Unprotected“. In this example, the execution time measurements follow a log-normal distribution. If duplicated instructions are used for error detection, the execution time measurements are expected to be more spread out, as shown by the log-normal PDF labeled „Protected“. We assume that both distributions can be approximated with simpler uniform distributions if we look at the worst-case behavior. This allows to identify suitable execution time budgets for both high- and low criticality mode in our mixed-criticality model: The protected task from the example is expected to execute within 2 ms to 8 ms in low criticality mode, and in case of error the re-execution could take another 6 ms. Selecting $c_i^L = 8\,\text{ms}$ and $c_i^H = 14\,\text{ms}$ would allow to represent the protected task in our mixed-criticality model.

3.2.2 Hardware-based error detection

If it turns out that the application of interest is not schedulable with software-based error detection by duplicated instructions, we can resort to circuit-based error detection, which signals errors within the next clock cycle. This prompt

Algorithm 5 SPCD algorithm [78]

Require: Program p
Ensure: Protected program

```
1: procedure CallDuplication(p)
2:     Build call graph G of program p
3:     for all leaf node procedure X in p do
4:         MarkIfLatencySatisfied(X)
5:     end for
6:     d ← 1
7:     while d ≤ the height of G do
8:         for all procedure X with depth of d do
9:             if X is unmarked then
10:                Duplicate statements
11:                if there is a procedure call calling procedure Y then
12:                    if Y is marked then
13:                        Duplicate procedure call statements
14:                    else
15:                        Duplicate arguments in the procedure call statement
16:                    end if
17:                end if
18:            end if
19:        end for
20:        d ← d + 1
21:    end while
22: end procedure
23: procedure MarkIfLatencySatisfied(X)
24:    if 2λ_X + α < λ then
25:        Mark procedure X
26:        for all parent node procedure Z do
27:            MarkIfLatencySatisfied(Z)
28:        end for
29:    end if
30: end procedure
```

signaling is near instant compared to the duration of error detection with software, and therefore needs no consideration in schedulability checks as the timer tick in an RTOS is in the order of 10 kHz, compared to typical clock cycles of 100 MHz.

Our temporal redundancy latch-based architecture (TRLA) EDS resorts to a *transition detector* to monitor the output node of a conventional latch, as displayed in the upper right corner of Figure 3.5. Late signal transitions during the transparent phase of the latch excite the transition detector which generates a pulse response that is captured by an SR-latch. The SR-latch is protected against SEUs by triplication and majority voting. The area requirements are alleviated by sharing of the protected SR-latch between multiple EDSs.

To enable optional correction by recomputation on the circuit level, [1] each EDS multiplexes between its input data and feedback from the output node, as shown in the upper left corner of Figure 3.5. An SET at the output node close to the sampling edge is flagged by the *late error detector* in the bottom right corner of Figure 3.5 and requires special treatment due to the propagation delay of the error correction logic.

The EDS is not exposed to increased metastability risks, compared to double-sampling and comparison EDS used in timing speculative designs, because it samples on the falling edge, which is not exposed to frequent data signal changes since the propagation delay t_{pd} of all paths feeding into the EDS are sufficiently short by design: $t_{pd} \leq T - t_{cq}$ with T as the cycle time and the clock-to-q delay of the latch t_{cq}. Compared to most EDSs heritage in timing speculative designs, where late signals are frequent by design and increase the risk for metastability [170], TRLA does not expose the sampling edge to frequent changes and therefore avoids a mean time between failures (MTBF) degradation.

The focus of this EDS is on mitigation of SEEs, therefore the pulse gating of the SR-latch is kept to a minimum, instead of increased pulse-widths leading to controlled time-borrowing. In this setup our EDS operates like a pulsed latch without further time-borrowing.

3.3 Earliest deadline first scheduler

Our dynamic priority mixed-criticality schedulers are based on traditional EDF schedulers. Their implementation requires only minor or no modification of the stock EDF schedulers available in modern RTOSs. While the scheduling algorithms are presented in detail in Chapters 5 and 6, their common architecture is explained here.

During system operation, our EDF-NUVD-SE or earliest deadline first with improved virtual deadlines for single errors (EDF-IVD-SE) scheduler is in one of three modes, as shown in Figure 3.6: *1)* initial low criticality mode; *2)* intermediate single error mode; and *3)* high criticality mode. Modes are switched as in EDF-VD-SE [161]: During the initial low criticality mode, all tasks are allowed to generate jobs. The jobs arrive at the scheduler's queue, and are EDF-scheduled according to their absolute deadlines or virtual absolute deadlines: Jobs from low criticality tasks are scheduled according to their absolute deadlines, and jobs from high criticality tasks are scheduled according to their virtual absolute

[1] Our CPS architecture resorts to software-level recomputation by default

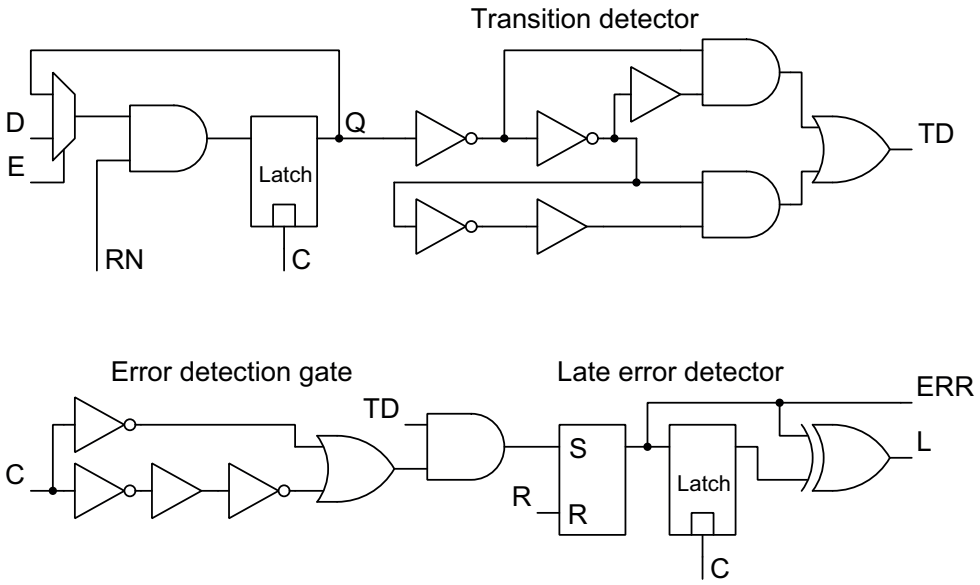


Figure 3.5: Static CMOS EDS with transition detector, error detection gate, and late error detector.

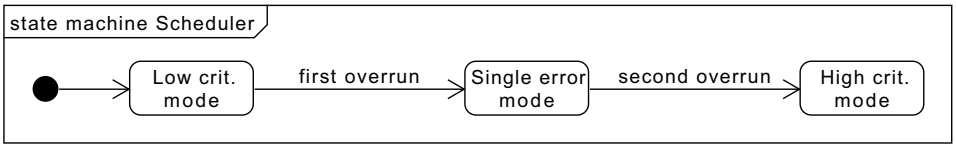


Figure 3.6: Unified modeling language (UML) state machine diagram of EDF-NUVD-SE mode switching.

deadlines. The mode- and task criticality dependent deadline selection is also shown in Algorithm 6.

As soon as a job is granted access to the CPU by the scheduler, as shown in Algorithm 7, it starts to execute, and the scheduler keeps track of the time since the job started. The scheduler is unknowledgeable of the time it takes for the job to actually finish, but the execution time budget is known. If the time since the job started is larger than the execution time budget, the scheduler kills the job if it is from a low criticality task, or switches to the intermediate single error mode if it is from a high criticality task.

During the intermediate single error mode, the scheduler operates as in the initial low criticality mode, but if a job from a high criticality task takes longer than the execution time budget, the system is switched to high criticality mode.

In high criticality mode, no jobs from low criticality tasks are generated, as shown in Algorithm 6, and jobs from high criticality tasks are scheduled according to their original deadline.

For illustration, we developed a EDF-scheduled application template for the Zephyr RTOS [171] as a baseline. The template is printed in full on Page 155 in the Appendix. Zephyr is especially interesting for CPS design as it is based on a small-footprint kernel, targeting resource-constrained embedded systems, and supports dynamic priority scheduling, as shown in Figure 3.10. In Figure 3.11

Algorithm 6 Mode-dependent CPU requests (jobs) from a sporadic task. If the task is ready, and the last job is at least p time units ago, a new job can be created where the absolute deadline is selected based on the task's criticality and the current system mode. For jobs from tasks with a high criticality level H, virtual absolute deadlines are used as long as the system is *not* in high criticality mode. Jobs from tasks with a low criticality level L use absolute deadlines, as long as the system is *not* in high criticality mode. In high criticality mode, no jobs from tasks with low criticality levels are generated. A related UML activity diagram is shown in Figure 3.8.

Require: Task τ_i
Ensure: Job J_{ij} with mode-dependent deadline

```
1:  loop
2:      if Task τ_i ready ∧ task period p_i passed then
3:          M ← GetCriticalityMode
4:          if χ_i = H then
5:              if M = H then
6:                  return Job with absolute deadline
7:              else
8:                  return Job with virtual absolute deadline
9:              end if
10:         else                                  ▷ Low criticality task
11:             if M = H then
12:                 return No job                 ▷ Dropped
13:             else
14:                 return Job with absolute deadline
15:             end if
16:         end if
17:     end if
18: end loop
```

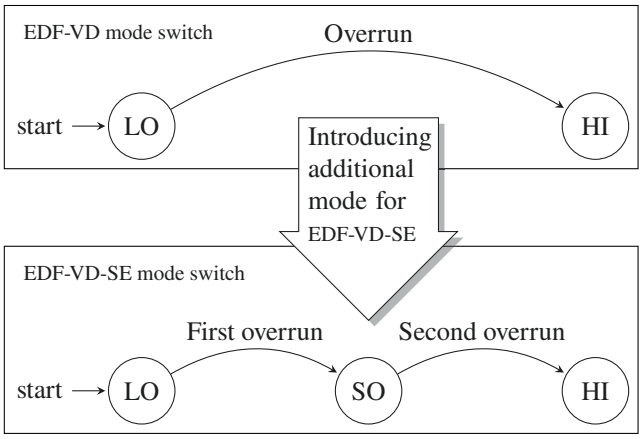


Figure 3.7: Mode switch in EDF-VD and EDF-VD-SE. With EDF-VD-SE we can tolerate a single overrun without abandoning low criticality tasks. With the second overrun, low criticality tasks are abandoned in favor of high criticality tasks. EDF-VD represents the mode switch behavior of classic dual-criticality mode switch schemes. Compared to classic dual-criticality mode switching, EDF-VD-SE has an intermediate mode labeled „SO“ (single overrun).

Algorithm 7 Scheduler grants access to CPU. Depending on the system criticality level and the task's criticality, jobs are allowed access to the CPU or not. Jobs from high criticality tasks can execute beyond c_i^L, and trigger a mode change. A related UML activity diagram is shown in Figure 3.9b.

Require: Jobs J_{ij} in priority queue sorted by earliest deadline
Ensure: Access to CPU for all jobs

```
1:  procedure RUN(Job of task τ_i)
2:      M ← GETCRITICALITYMODE
3:      if (χ_i = L) ∧ (M = H) then
4:          return                                  ▷ return without running
5:      end if
6:      while Job running time < c_i^L ∧ job not done do
7:          Run job J on CPU
8:      end while
9:      if Job not done ∧ χ_i = H then
10:         Increase system criticality mode
11:         Run job J on CPU
12:     end if
13: end procedure
14: loop
15:     J ← Job from priority queue with earliest deadline
16:     do parallel
17:         RUN(J)
18:         Preempt and return job J to queue if earlier deadline in queue
19:     end parallel by join
20: end loop
```

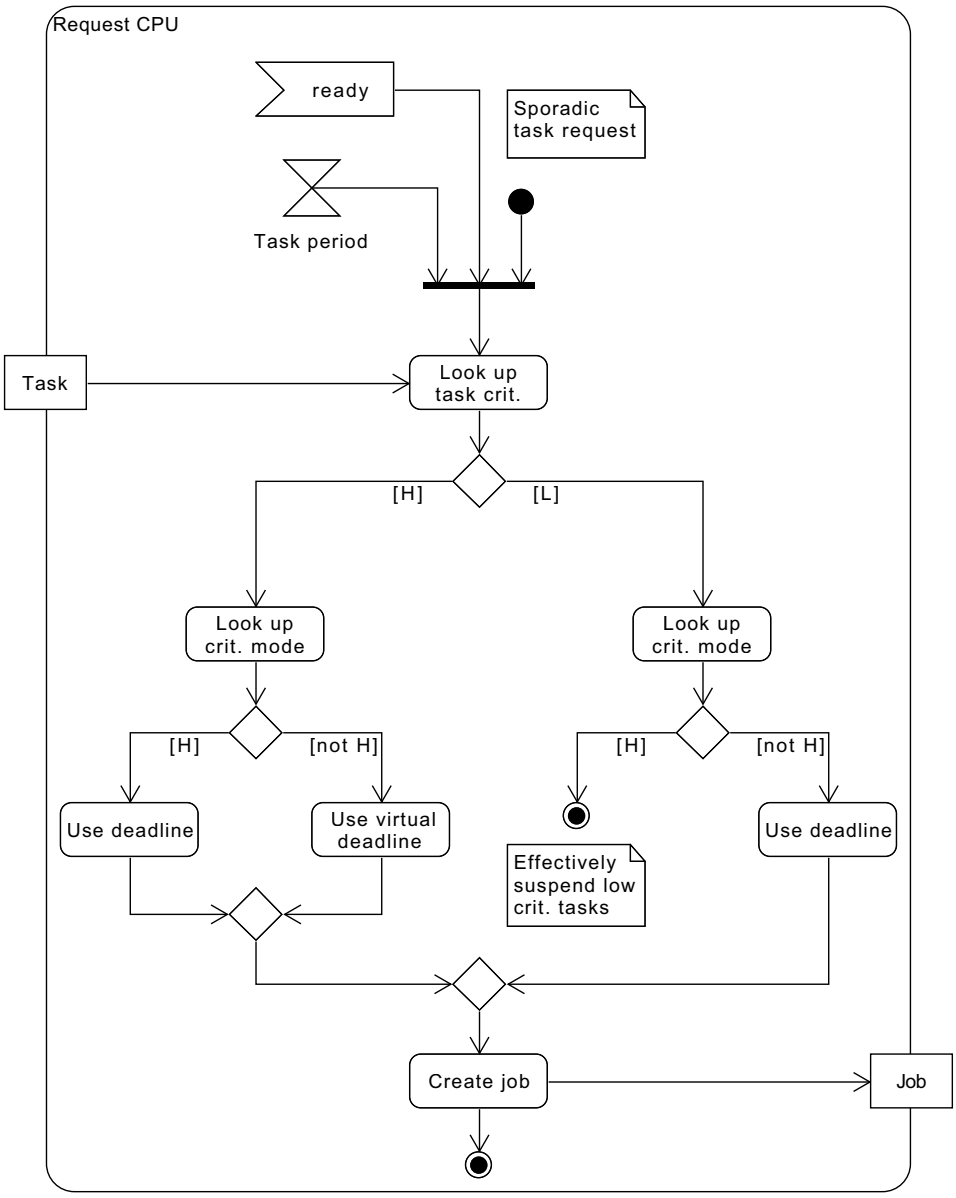
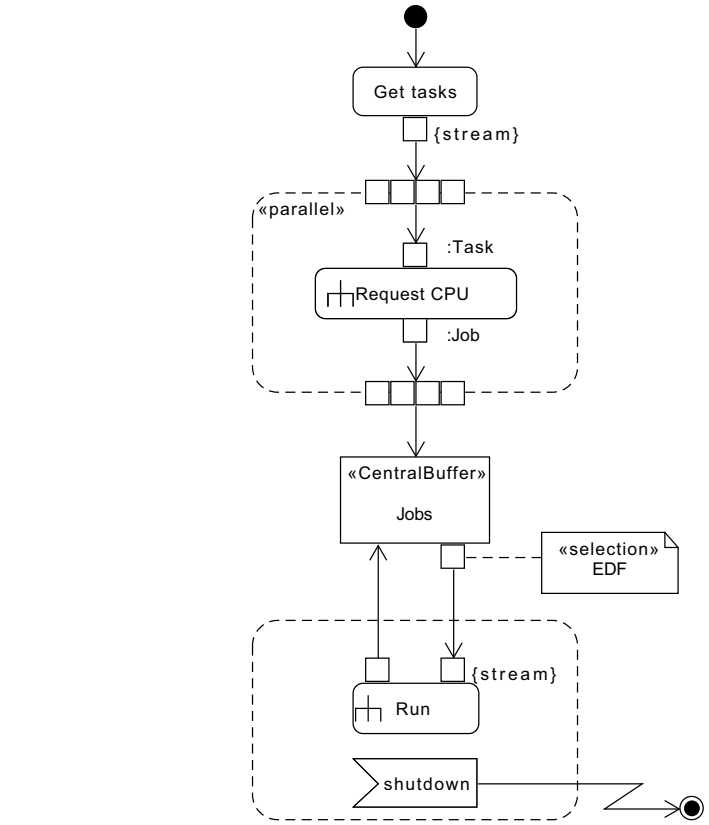
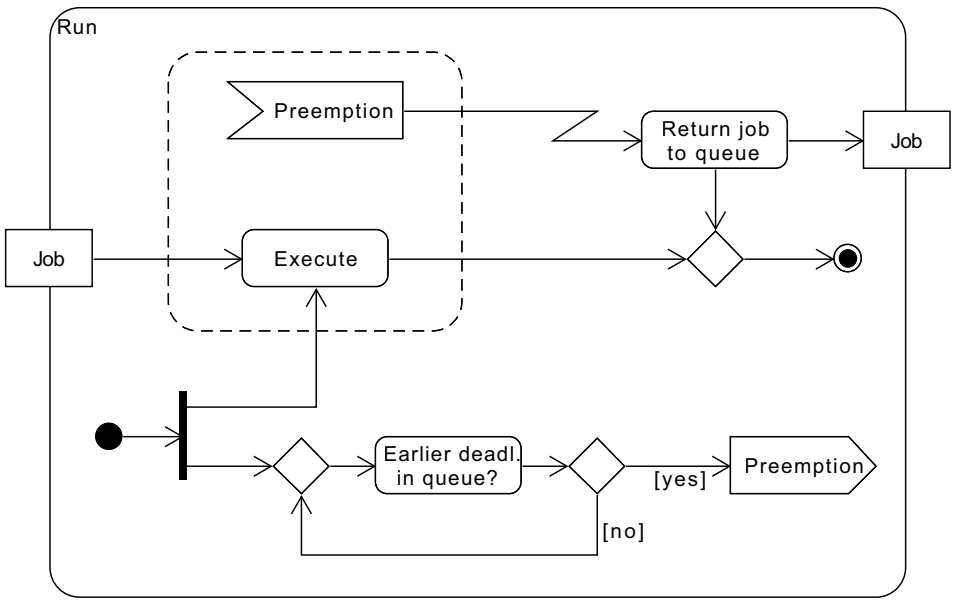


Figure 3.8: UML activity diagram of mode-dependent CPU requests (jobs) from a sporadic task. The acitivity diagram shows how and if a job for a task is created. Condition for task creation is that the task is ready, and that at least p time units passed since last job generation. The absolute deadline is selected based on the task's criticality and the current system mode. An algorithmic description in pseudo code is available in Algorithm 6.



(a)



(b)

Figure 3.9: UML activity diagram of the scheduler. Figure 3.9a shows the flow of jobs from creation to execution. The referenced activities „Request CPU“ and „Run“ are shown in Figures 3.8 and 3.9b.

we show how multiple criticalities and mode switching can be implemented within our template, without the need to modify kernel sources.

3.4 Errors modeled as worst-case execution times

Our approach targets emerging critical applications, where we expect errors, and yet the budget to deal with them is very limited compared to traditional critical applications. Such semi-critical applications, like wearable healthcare monitors, are required to be dependable, while being built from COTS components.

The key to dependable design with COTS components are error detection and correction. Both can be implemented in software by exploiting temporal redundancy. In our model, the time for error detection is part of the execution time budget in low criticality mode for high criticality tasks, and we consider error correction by recomputing wrong results in the WCET of high criticality tasks.

In EDF-VD-SE, we model the error detection and correction techniques used to ensure the service of high criticality tasks with execution time budgets. During error-free operation, protected high criticality tasks don't exceed their low mode execution time budget. Once errors are detected, the error correction requires additional time, which is reflected in the high mode execution time budget.

Execution time budget overrun is a powerful abstraction for transient errors in embedded systems because a lot of soft- and hardware errors manifest as delays or errors in timing. For example, a single energetic particle strike causing a voltage spike at a node in an integrated circuit can result in a SEU [41]. Such an SEU might flip a bit in a CPU register which holds a loop counter, or a calculated result gets corrupted and needs to be recalculated. These longer run-times are considered in the high mode execution time budget c_i^H. To model a task with error correction, we consider $c_i^H = z_i c_i^L$ as a multiple of c_i^L, where $z_i \in \{x \in \mathbb{Z} : x > 1\}$ captures the pessimism in estimating the task's WCET for error correction approaches where jobs are repeated.

3.5 Conclusion

This chapter introduced the architecture of our CPSs, starting with a top-down description of the enhanced MPSOC, and a layered view of the architecture, spanning from hard- to software. We presented how to distribute mixed-criticality applications on the processing nodes of our architecture in a static, design time and therefore CPS-friendly way by reducing the problem to multiple unicore scheduling problems.

Moreover, we explained how to combine error detection in hard- and software with real-time scheduling, to implement a system recover fault tolerance strategy for our CPS. The discussion included a description of a COTS-friendly EDS, which can mitigate SEEs, and motivated its use in our architecture. Next, we introduced our schedulers and how to implement them with an example of a modern RTOS, without the need to modify its kernel sources. Finally, we linked errors to our model of mixed-criticality applications.

Begin of code

```
/*
 * Return value same as e.g. memcmp
 * > 0 -> thread 1 priority  > thread 2 priority
 * = 0 -> thread 1 priority == thread 2 priority
 * < 0 -> thread 1 priority  < thread 2 priority
 * Do not rely on the actual value returned aside from the above.
 * (Again, like memcmp.)
 */
int32_t z_sched_prio_cmp(struct k_thread *thread_1,
        struct k_thread *thread_2)
{
        /* ‘prio‘ is <32b, so the below cannot overflow. */
        int32_t b1 = thread_1->base.prio;
        int32_t b2 = thread_2->base.prio;

        if (b1 != b2) {
                return b2 - b1;
        }

#ifdef CONFIG_SCHED_DEADLINE
        /* If we assume all deadlines live within the same "half" of
         * the 32 bit modulus space (this is a documented API rule),
         * then the latest deadline in the queue minus the earliest is
         * guaranteed to be (2’s complement) non-negative.  We can
         * leverage that to compare the values without having to check
         * the current time.
         */
        uint32_t d1 = thread_1->base.prio_deadline;
        uint32_t d2 = thread_2->base.prio_deadline;

        if (d1 != d2) {
                /* Sooner deadline means higher effective priority.
                 * Doing the calculation with unsigned types and casting
                 * to signed isn’t perfect, but at least reduces this
                 * from UB on overflow to impdef.
                 */
                return (int32_t) (d2 - d1);
        }
#endif
        return 0;
}
```

End of code

Figure 3.10: Zephyr RTOS scheduler source code snippet [172]. The function compares the priority, and optionally the dynamic priority, of two threads. If all threads have the same static priority, and `CONFIG_SCHED_DEADLINE` is defined, EDF scheduling is implemented.

Begin of code

```
/* [...] */

enum criticalitylevels { LO = 1, HI };
uint32_t system_mode = LO;

/* [...] */

const char *thread_a_id = "thread a";
uint32_t thread_a_criticality = LO;
uint32_t thread_a_work_us = 250000;
uint32_t thread_a_relative_deadline_us = 1000000; /* 1 second */

const char *thread_b_id = "thread b";
uint32_t thread_b_criticality = HI;
/* Need two estimates of worst-case execution time */
uint32_t thread_b_work_lo_us = 150000;
uint32_t thread_b_work_hi_us = 250000;
/* For high criticality tasks, two relative deadline are needed:
 * One for low criticality mode, and one for high criticality mode.
 */
uint32_t thread_b_relative_deadline_lo_us = 2800000;
uint32_t thread_b_relative_deadline_hi_us = 3000000;

/* [...] */

/* Function to spawn jobs has to understand criticalities to prevent
 * spawning low-criticality jobs in high criticality mode.
 * Moreover, the execution of new jobs is supervised by another
 * timer 'wcet_timer', which is stopped once the job finishes execution.
 * If 'wcet_timer' expires, we switch to the next higher criticality mode.
 */
void spawn_job_if_expired(struct k_timer *release_timer,
                          void (*spawn_function_pointer)(),
                          int period_seconds,
                          uint32_t job_criticality,
                          struct k_timer *wcet_timer)
{
        if (system_criticality > job_criticality) {
                return; /* drop job */
        }
        if (k_timer_status_get(release_timer) > 0) {
                /* Timer expired, call void function without argument to
                 * spawn new job.
                 * If the timer expired multiple times, we still start just
                 * one job, implicityly skipping jobs in between.
                 */
                printk("Timer expired\n");
                /* Add code here to use 'wcet_timer', which is stopped upon
                 * thread/job completion, and increase system criticality
                 * level if timer expires.
                 */
                (*spawn_function_pointer)();
        } else if (k_timer_remaining_get(release_timer) == 0) {
                /* Timer stopped; this is the default state after
                 * timer initialization.  Starting the timer here,
                 * hoping all timers are more or less started in sync.
                 */
                printk("Timer started\n");
                k_timer_start(release_timer, K_SECONDS(period_seconds),
                              K_SECONDS(period_seconds));
        } /* else timer is still running */
}

/* [...] */
```

End of code

Figure 3.11: Zephyr EDF template source code snippet indicating where support for criticalities can be implemented without changing the kernel's scheduler. Moreover, this snippet shows where mode switching and WCET tracking is done without modification of kernel sources.

The following chapters present the tooling and scheduling approaches EDF-VD-SE and EDF-NUVD-SE, which are the missing parts to complete our CPS design approach.

4 Developed software and tools

We developed several tools to conduct the scheduling simulation experiments. This chapter presents why and how we developed them in the first place, what other software solutions exists, and describes our experimental setup in detail.

4.1 Static verification

Many applications for CPSs are safety-critical, for example the braking system in a car. Such applications require that the system we develop to enable them are certified prior to operation: they need their correctness validated, or formally verified.

For hard real-time systems, formal verification that no deadline is missed is mandatory. Therefore, substantiating that a scheduling algorithms fulfills this requirement is necessary. The proof is split in two parts: A general part, which shows that the reasoning is sound as long as some constraints are valid, and the task set specific part, where we check if our task set fulfills the constraints of the general part.

For example, the proof for EDF-VD concludes its general part with a set of schedulability conditions as constraints. The task set specific part is to show that the task set fulfills the schedulability conditions. This requires to solve a set of inequalities, either analytical or numerical. We developed a framework which implements our mixed-criticality model and solvers for several schedulability conditions, which we present here.

The two main classes `RTask` and `Tasksystem` are shown in Figure 4.1. `RTask` describes a real-time task and defines all task- and simulation related parameters. Task parameters are period, relative deadline, and six execution time budgets. Simulation parameters are probabilities which influence the selection of a execution time budget, which are used by our simulator `Thready` to implement mixed-criticality simulations. Moreover, we specified some helper methods for consistent evaluation of derived properties like task utilizations or criticalitys.

The `Tasksystem` class is a consistent list of tasks with a javascript object notation (JSON) load- and store interface for de- and serialization. In our mixed-criticality notation `Tasksystem` implements a task set. Both `RTask` and `Tasksystem` are hashable, which is a necessary property for comparison of tasks and task sets. This allows to check if two task sets are identical, a question which arises during random generation of task systems if we want to refrain from working with duplicate task sets.

All schedulability checks refer to a task set, therefore we implemented them as class methods for `Tasksystem`. Moreover, we provide some methods as building blocks to apply the results from schedulability checks to the represented task set.

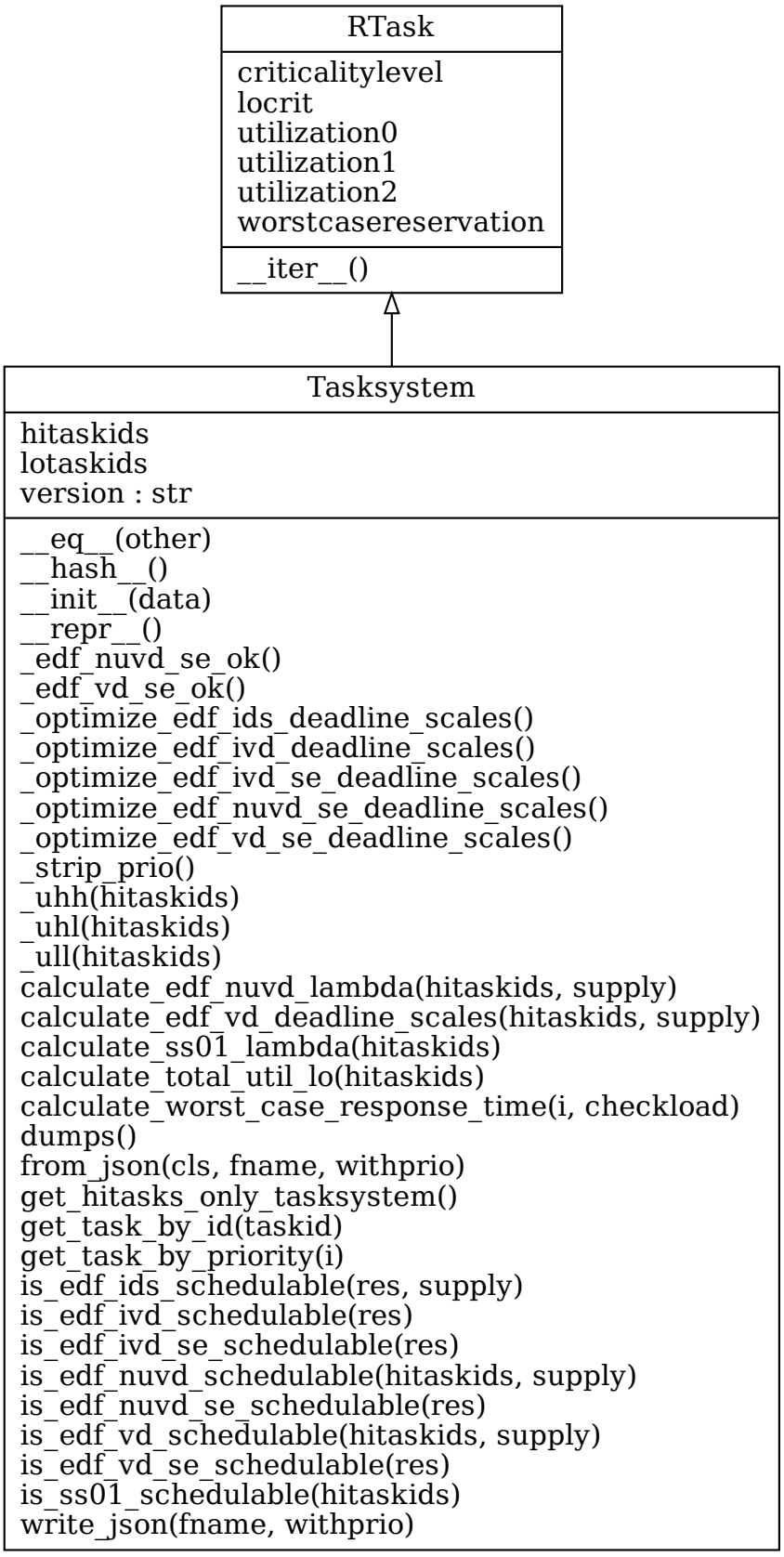


Figure 4.1: UML class diagram

The JSON interface is shared with out simulator `Thready`, and serializes a task set as an array of arrays: The task set is an array of tasks, and each task is an array of eleven numbers: *1)* An id number to identify the task by humans; *2)* the task's period; *3)* the task's relative deadline; *4)* six values for execution time budgets, interpreted in pairs as lower- and upper bound of uniform distributed execution times; *5)* two probabilities to simulate overrunning jobs by selection of execution time budget pair; and *6)* the parameter of the exponential distributed inter arrival time between jobs.

The following example shows a task set of three tasks, where task 0 has a period and relative deadline of 5, and computation is always uniformly drawn between 1 and 4.

Begin of code

```
[
        [0, 5, 5, 1,4, 0,0,  0,0, 1.0,0.0,  0.01],
        [1,20,20, 1,1, 2,4,  5,8, 0.9,0.09, 4.0],
        [2,20,20, 1,2, 3,4,  5,8, 0.9,0.09, 4.0]
]
```

End of code

The jobs of task 0 rise with the period, or to be precise, the probability of an arrival later than the minimum period is incredible low[1]. Task 1 has a period and relative deadline of 20, and the computation demand is 1 with a probability of 0.9, or between 2 and 4 with a probability of 0.09, or between 5 and 8 with a probability of 0.01. The jobs of task 1 rise on average 0.25 period after the minimum period.

The Python source code of both classes and their methods is found on Page 157.

4.2 Random task set generation

It is interesting to know which, or how many, task systems, or task sets, are deemed schedulable by a given algorithm. With a fixed set of task systems, the fraction of task systems deemed schedulable can be compared between algorithms as an indication of their practical applicability. If an algorithm is able to schedule a larger fraction of task systems, it seems more practical, because the likelihood of scheduling a specific task system increases. Comparing algorithms in this regard requires a large set of task systems, hence randomly-generated synthetic task systems are used.

The random generation is controlled by several parameters, which can affect results and bias the judgment about algorithms [159]. A major parameter is the task system's utilization U, because the difficulty increases with U. The distribution of U over all tasks in a system needs to be controlled to avoid any bias. It is favorable to have uniform distributed task utilization u_i; the `UUnifast` algorithm provides a controlled way of distributing U over n tasks in a system [159].

To calculate uniformly distributed task utilizations u_i in linear time the `UUnifast` algorithm splits U iteratively into n slices, as shown in Figure 4.3. Given a CDF $F_i(x)$ of the sum of i independent uniform distributed random variables and their sum limited to s, the algorithm draws a random value $x \leq s$

[1] For task 0 the probability would be 2.06×10^{-9}.

Figure 4.2: `UUnifast` algorithm [159]. The algorithm computes each task's utilization u_i iteratively by nested splitting of U between i tasks. Each iteration, a utilization u_i is calculated by inverse transform sampling a cumulative distribution function (CDF) of the sum of i uniform random variables limited to the iteratively decreasing remaining utilization s.

Require: Utilization U to distribute over n tasks
Ensure: Uniform distributed utilizations u_i
1: $u \leftarrow \emptyset$
2: $s \leftarrow U$
3: **for** $j \leftarrow [0, n-1) \cap \mathbb{Z}$ **do**
4: $\quad v \leftarrow \mathcal{U}(0,1)$
5: $\quad i \leftarrow n - (j+1)$
6: $\quad s_{\text{next}} = s \cdot v^{1/i}$
7: $\quad u \leftarrow u \cup \{s - s_{\text{next}}\}$
8: $\quad s \leftarrow s_{\text{next}}$
9: **end for**
10: $u \leftarrow u \cup \{s\}$

by inverse transform sampling from $F_i(x)$. The difference $s - x$ is the remaining utilization which needs to be split up between $n - 1$ tasks. This procedure, formalized in Figure 4.2, is repeated for the remaining $n - 1$ tasks.

The CDF $F_i(x)$ is derived from the PDF $f_i(x)$ of the sum of independent, uniformly distributed random variables in $[0, b]$:

$$f_i(x) = f_{\mathcal{U}_1+\mathcal{U}_2+\cdots+\mathcal{U}_i}(x) = \left(f_{\mathcal{U}_1} * f_{\mathcal{U}_2} * \cdots * f_{\mathcal{U}_i}\right)(x) \tag{4.1}$$

$$f_i(x) = \begin{cases} \frac{1}{b}x^{i-1} & x \in [0, b] \\ 0 & \text{else} \end{cases} \tag{4.2}$$

$$F_i(x) = \begin{cases} 0 & x \leq 0 \\ \left(\frac{x}{b}\right)^i & 0 < x \leq b \\ 1 & x > b \end{cases} \tag{4.3}$$

The sampling of random values from $F_i(x)$ requires the inverse function $F^{-1}(x)$, which is used in line six of the `UUnifast` algorithm in Figure 4.2 in case $0 < x \leq b$:

$$F_i\left(F^{-1}(x)\right) = x \tag{4.4}$$

$$\left(\frac{F^{-1}(x)}{b}\right)^i = x \tag{4.5}$$

$$F^{-1}(x) = b\sqrt[i]{x} = bu^{1/i} \tag{4.6}$$

As an example consider $U = 1$ and $n = 4$, which can result in $u = (0.243, 0.105, 0.399, 0.253)$ as shown in Table 4.1. The slices are iteratively

Table 4.1: Example of executing `UUnifast` algorithm for $U = 1$ and $n = 4$.

j	i	$s(j)$	$s_{\text{next}}(j)$	u_j
0	3	U	0.757	0.243
1	2	0.757	0.652	0.105
2	1	0.652	0.253	0.399
-	-	-	-	0.253

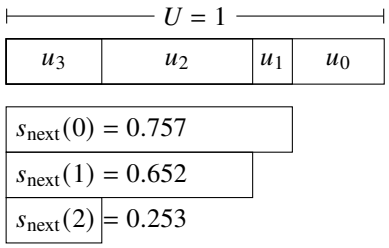


Figure 4.3: Example slicing of $U = 1$ by the `UUnifast` algorithm with $n = 4$ tasks.

calculated, where $v \in \mathcal{U}(0, 1)$ is a uniform distributed random variable:

$$s(0) = U \qquad s_{\text{next}}(0) = U \cdot v^{1/3} \tag{4.7}$$

$$s(1) = s_{\text{next}}(0) \qquad s_{\text{next}}(1) = s_{\text{next}}(0) \cdot v^{1/2} \tag{4.8}$$

$$s(2) = s_{\text{next}}(1) \qquad s_{\text{next}}(2) = s_{\text{next}}(1) \cdot v \tag{4.9}$$

The task's actual utilizations are the differences between two slices, except the last one:

$$u_0 = s(0) - s_{\text{next}}(0) \tag{4.10}$$

$$u_1 = s(1) - s_{\text{next}}(1) \tag{4.11}$$

$$u_2 = s(2) - s_{\text{next}}(2) \tag{4.12}$$

$$u_3 = s_{\text{next}}(2) \tag{4.13}$$

The remaining parameters in the task system generation are the limits of uniform random distributions for the task's period, deadline-to-period ratio, and factor between low- and high level utilization. Usually one can assume task periods within typical ranges, for example between $p_l = 200\,\text{ms}$ and $p_h = 5000\,\text{ms}$ depending on the kind of application. A uniform distribution $\mathcal{U}(p_l, p_u)$ is justified for general comparisons of algorithms, but other distributions can be considered if the applications of interest have a bias towards shorter or larger periods in a task system. For non-implicit-deadline tasks the deadline of each task is a multiple of the period, $d_i = \epsilon_i p_i$ with a deadline-to-period factor drawn uniformly $\epsilon_i \in \mathcal{U}(\epsilon_l, \epsilon_u)$. Implicit-deadline task systems are generated with $\epsilon_l = \epsilon_h = 1$. The execution time budget of each task is the product of the task's utilization u_i and period $p_i \in \mathcal{U}(p_l, p_u)$: $c_L = u_i p_i$.

For our experiments we generate synthetic task systems, and convert them into dual-criticality task systems. A uniformly distributed factor $z_i \in \mathcal{U}(z_l, z_u)$ determines the higher level execution time budget $c_H = z_i c_L$ [173], if by random

chance of $p = 0.5$ the task is of high criticality. The factor z_i describes the pessimism in estimating the task's WCET with c_H. Moreover, by selecting $z_l = z_h = z \in \{x \in \mathbb{Z} : x > 1\}$ we can model different software error correction approaches where jobs are repeated. Resulting dual-criticality task systems have different utilizations U_L, U_H in low- and high criticality mode, with $U_H > U_L$ due to increased pessimism in estimating the task's WCET. During random generation, the utilization in low criticality mode is chosen, which influences the utilization of low criticality tasks in low criticality mode U_L^L and of high criticality task in low criticality mode U_L^H:

$$U = U_L = U_L^L + U_H^L \tag{4.14}$$

This parameterized random generation approach is used to reproduce task system sets used in the literature and allows to investigate a schedulability test under explicit parameters and task system properties. We implemented random task system generators within our framework, which is further described in the last section of this chapter. The core functionality is found on Page 174 in the Appendix.

4.3 Fast scheduling simulator

Real-time systems, like on-board flight computers, have timing requirements which need to be satisfied. To meet these requirements, scheduled jobs need to finish before their deadline. If job arrival- and execution times are not known beforehand, the schedule in which jobs are granted access to the processor has to be created during runtime, according to a scheduling algorithm.

Such an algorithm is correct, if it can always create a schedule where each job receives enough processor time to finish before its deadline. Correctness is usually demonstrated for the worst case, defined by upper bounds on execution times [145], but for system performance the average case is of interest, which can be evaluated by simulations. For example, in mixed criticality scheduling, the time spend in low criticality mode is a key QoS metric, which depends on the probability of having a job executing longer than on average. Usually the assumed probabilities of such an event are very low, which requires to simulate the system for a long time. For such an investigation, simulator performance is of utmost importance to gather results in reasonable wall clock time, but available simulators fall short in this regard.

Although many simulators have been written, only few are readily available to researchers [174]. As with most scientific software, their value as an scientific artifact has not been recognized, resulting in lack of software preservation and reproducibility problems [175]. Moreover, most available simulators are hard to use in heterogeneous environments due to low portability, are hard to automate due to their user interfaces, and sacrifice simulation speed over features or ease of implementation. Although such feature-rich simulation environments are great to explore new algorithms, or to investigate the relative short critical instant of a task system, they don't allow to run long term simulations.

We fill this gap with `Thready`, which is the first portable, easy to automate, integrate, reproduce and understand open source simulator to address long term

simulations for sporadic task systems:

- `Thready` is easy to compile and distribute (Section 4.3.3), allowing to leverage the computational resources available in a heterogeneous laboratory computer environment (Section 4.3.4)
- `Thready`'s user interface is easy to understand and instrument, which fosters integration into scientific data analysis pipelines and reproducibility (Section 4.3.3)
- `Thready`'s three orders of magnitude speedup in latency allows to investigate QoS and average case performance metrics in long term simulations (Section 4.3.4)
- `Thready` allows to investigate systems by simulation which are infeasible with with current simulators, as demonstrated with a case study (Section 4.3.5)

4.3.1 Related software packages

Simulators are recognized as useful tools to teach, explore, and understand scheduling algorithms [176]. We differentiate three main approaches to simulators: Comprehensive simulation environments configured by the user, frameworks, or libraries, to construct new special purpose simulators from basic components, or single purpose programs simulating a specific aspect of a system or algorithm.

One of the early comprehensive simulators are the `STRESS` computer-aided software engineering (CASE) tools, which use a domain-specific language (DSL) to specify the simulated system [177]. Besides simulation of different algorithms, system configurations, and shared resources, `STRESS` allows to analyze and visualize simulation results. In comparison to other simulators, `STRESS` provides many features, as shown in Table 4.2, but is not publicly maintained as open source.

Another extensive simulator is `Cheddar`, which allows to simulate different hard- and software configurations, scheduling algorithms, supports shared resources, and provides extensive analysis and visualization capabilities [174]. Compared with `STRESS`, `Cheddar` is actively maintained and available as open source software. Written in Ada, `Cheddar` is well suited to simulate the critical instance of a task system to show the correctness of the scheduling algorithm, but inappropriate for long term simulations owing to its slow simulations speed.

A further comprehensive simulators is `Realtss`, which supports multiple scheduling algorithms, shared resources, tracing, and result evaluation [178]. The modular design, written in tool command language (TCL), would allow to extend the simulator, but `Realtss` is not publicly available as open source. In this regard, it shares the same fate as `schesim`, which featured hierarchical, uni- and multiprocessor scheduling, tracing and analysis [179], and is not available anymore. `schesim` allows the user to specify tasks via a JSON interface, but does not model errors. Similar to `Realtss` in features, but without support for shared resources and evaluation, is `RTSim`. `RTSim` targets education and

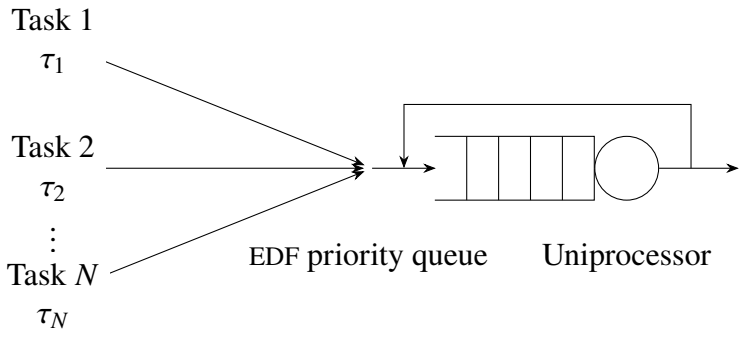


Figure 4.4: Simulation model of `Thready`. Each task τ_i generates jobs which are EDF scheduled on the uniprocessor. If a job with a higher priority (=earlier deadline) arrives, the current job on the uniprocessor is preempted and returns to the queue.

classroom usage [176], but does not consider errors in its model, which is a key feature of `Thready`.

Unlike comprehensive simulators, frameworks like `Fortissimo` or `rtlib` provide building blocks to develop tailored simulators for a specific system [180, 181]. Such tailored simulators can achieve better performance than comprehensive simulators, because they allow to reduce or disable unnecessary features. Their main drawback is the lack of interfaces, which need to be written by the user of such frameworks, else the resulting simulator is a single purpose program, where the problem definition is part of the simulator, without a possibility to modify it. Although single purpose programs have legitimate use cases [182], their creation and maintenance for many similar problems is a burden.

Contrary, `Thready` strives to combine the performance benefits of the framework approach with easy to instrument interfaces and support for diverse error models.

4.3.2 Thready model

`Thready` simulates preemptive EDF scheduling of a sporadic task system on a uniprocessor. The processor is the only shared resource, and there is no overhead of scheduler operation and context switch. All tasks in the simulated sporadic task set are independent of each other, and generate a sequence of jobs. Jobs signal their completion to the scheduler, which is not aware of the completion time a priori. If jobs are preempted, they return to the scheduler's queue, as shown in Figure 4.4. Time is discrete and time step resolution is defined by the user.

Sporadic task system and job generation

The simulator generates jobs according to the parameters of each task τ_i. Jobs are characterized by their arrival time α_{ij}, actual execution time γ_{ij}, and absolute deadline d_{ij}.

Task parameters are execution time distributions, minimum time between

Table 4.2: Comparison of simulator features

	Scheduling algorithms		Essential features			Ancillary features		
Name	Uniprocessor	Multiprocessor	Shared resources	Error model	Performance	Tracing	Evaluation	Task generation
`RTSim` [176]	✓	✓				✓		
`rtlib` [181]	✓	✓	✓			✓		
`Cheddar` [174]	✓	✓	✓			✓	✓	✓
`YAO-SIM` [182, 183]	✓						✓	
`Realtss` [178]	✓	✓	✓			✓	✓	
`schesim` [179]	✓	✓				✓	✓	✓
`STRESS` [177]	✓	✓	✓			✓	✓	
`Fortissimo` [180]						✓		✓
`Thready`	✓			✓	✓			

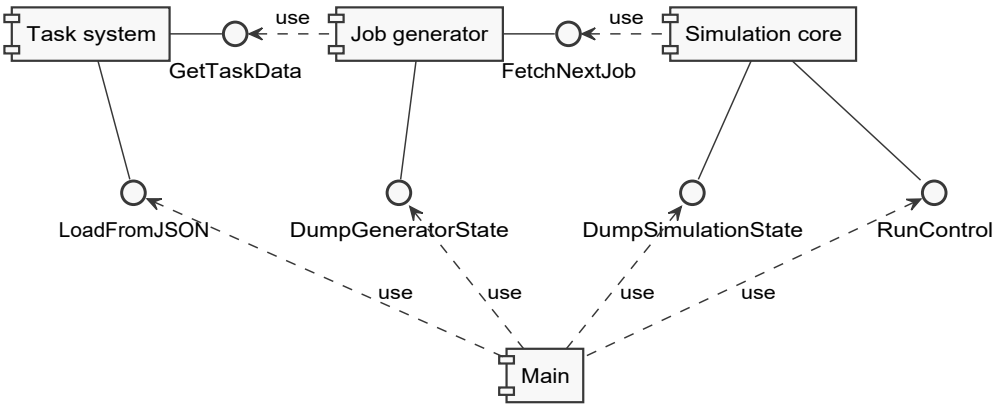


Figure 4.5: UML component diagram of `Thready`

two job arrivals p_i, and relative deadline d_i. Execution time distributions are uniform random distributions describing the execution time of a task's job. Moreover, execution time distributions may change with a given error probability to model errors, as described in Section 4.3.2. The minimum time between two job arrivals exhibits the worst case job arrival sequence in terms of processor demand. Depending on the parameterization of the exponential inter-arrival time distribution for the task, the average time between two arrivals tends to be larger.

Error model

For any job of a task the execution time distribution may change with a given error probability. This allows to model different kinds of errors. For example, in a mixed criticality system one may specify multiple execution time distributions for high criticality tasks. Then the error probability describes the probability to overrun a job's execution time budget. Another example are transient errors that force a job to restart [154]. The user's application therefore defines the meaning of error, and the simulator is agnostic to the cause of the error.

4.3.3 Interface and implementation

`Thready` consists of two major components as shown in Figure 4.5: A job generator, and the simulation core which consumes jobs. For each task, the job generator raises jobs according to the task's parameterization, and provides the jobs sorted in order of arrival time to the simulation core. Generation of the next job takes place as soon as the job is handed to the simulation core. The simulation core fetches jobs from the generator, puts them into his job queue, and schedules the jobs according to EDF on the uniprocessor.

User interface

The task systems are described in JSON. `Thready` follows common UNIX conventions in specifying command line parameters for configuration, and keeps quiet during normal use, except mentioning the final simulation result to stdout. If an error was simulated, `Thready` outputs information about the first job

that experienced an error. Once the simulation ends, `Thready` dumps the scheduler's job queue and current simulation time to a JSON file. This state dump can be investigated or used to continue the simulation. Moreover, the task systems are described in JSON as well, which allows to instrument `Thready` by any software capable of reading and writing JSON.

Job generator implementation

The job generator is responsible for creating a stream of jobs for each task according to the task's parameters. Every time the simulation core consumes a job of task τ_i, the job generator creates the next job for τ_i.

The on demand job creation keeps the memory footprint low, which increases performance by avoiding page misses and allows to run several simulations in parallel on a single machine. Contrary, generating and storing a full job trace slows down execution through page misses, and limits the possible simulation time to the available memory.

Jobs are sorted by arrival time, that is, the time the job enters the scheduler's queue, using a priority queue. This ensures correct order of job arrival from several tasks for the simulation core.

Simulation core implementation

The simulation core implements EDF scheduling by sorting arriving jobs by their deadline with a priority queue. The job with the earliest deadline is the job with the highest priority, and gets access to the uniprocessor. To speed up the simulation, time is progressed nonlinear by advancing from event to event. Events mark decision points where the EDF scheduler may switch the current running job. For example, if a new job arrives, the deadline of the currently running job is compared with the deadline of the arriving job. If the arriving job has an earlier deadline, the current job is preempted and returns to the scheduler's queue. Other examples are that a job finishes, or that a deadline is missed. Therefore, the simulation core can be described as an event loop, as shown in Figure 4.6, which traverses through all events until it returns. The return code of the event loop is evaluated to inform the user about the cause of the exit, and the current simulator state is dumped to a JSON file.

Distribution

The simulator is dedicated to the public domain, to foster access for the community in the spirit of early research software like SPICE, and to mitigate possible reproducibility issues. Motivated by performance `Thready` is written in modern C, and distributed in source form [184]. The small code base of 4599 lines of code (LOCs) and low complexity, as shown in Figure 4.7, makes it easy to integrate `Thready` into scientific data analysis pipelines.

4.3.4 Evaluation

`Thready` is written with simulation performance in mind to allow long term simulations in reasonable wall clock time. `Thready` allows to leverage the

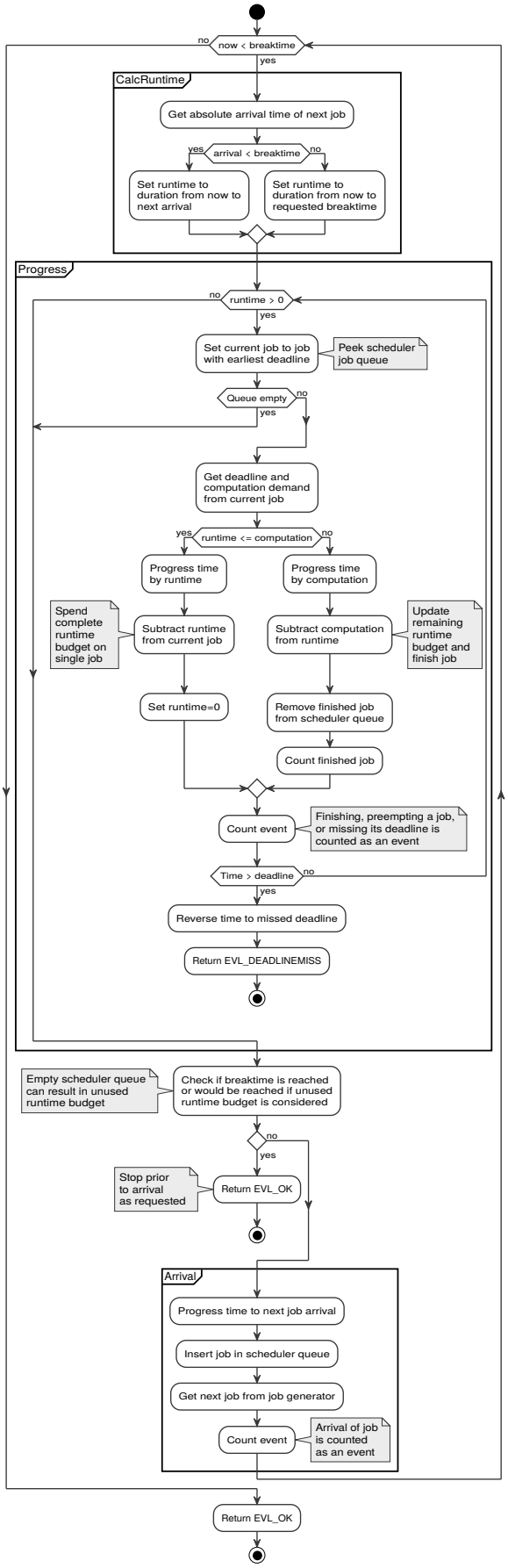


Figure 4.6: Flowchart of simulator core

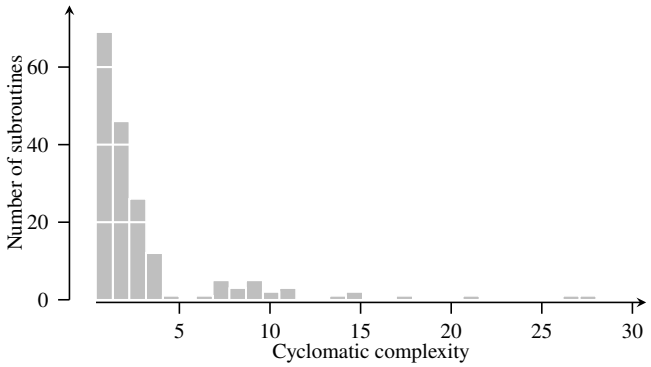


Figure 4.7: Histogram of subroutine cyclomatic complexity. Cyclomatic complexity corresponds to the number of independent paths through a piece of code, which increase with additional control flow statements, and should be as low as possible to ease unit testing and code maintenance. Values below 10 indicate easy to maintain code, showing that nearly every subroutine in `Thready`'s code base is of low complexity.

computation resources of a distributed, shared, and heterogeneous laboratory computer environment by portability and composability: `Thready` plays well together with other command line tools like GNU `Parallel` [185], which allows to run a plethora of simulations in parallel distributed on several physical machines.

`Thready` is trimmed down in size to fit in the level 1 (L1) cache of most low- and mid range desktop CPUs, which results in large performance gains and allows to run several simulation threads on a single machine without running into memory problems. This is especially beneficial on shared computers which do not employ any user quotas on memory usage, or provide any other guarantee for available resources.

In the following subsections, we quantify `Thready`'s advantages in simulation speed and memory usage, and compare our results with other simulators.

Memory usage

To quantify the memory consumption of `Thready` we profile its heap memory using Valgrind's massif [186]. We generate random task sets using the UUnifast algorithm, to ensure a uniform distributed utilization over all tasks [159]. To get results that are applicable to `Thready`'s prime use case, some tasks are allowed to release jobs that experience an error, and therefore take longer to execute.

Memory consumption peaks at ≈ 34.5 KiB before the actual simulation by reading and parsing the JSON task system file. During simulation memory demand is constant at ≈ 2.6 KiB due to the job generation approach described in Section 4.3.3, which allows `Thready` to run long term simulations. Moreover, this is irrespective of the number of tasks in the task system, as can be seen by the qualitative similar curves in Figure 4.8.

We selected `rtlib` to compare our results with other simulators, because `rtlib` can be considered as the fastest available simulator framework with

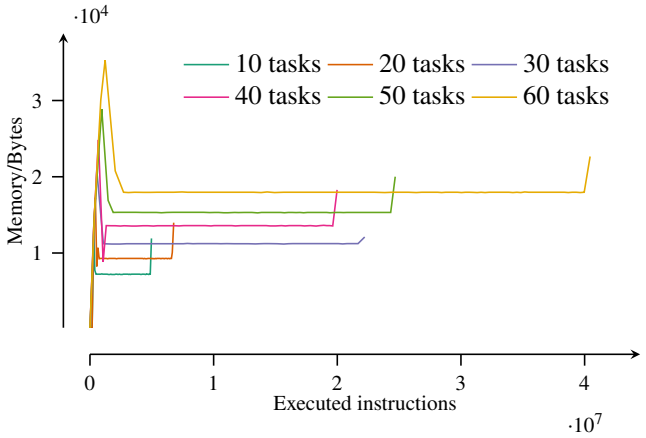


Figure 4.8: Heap memory of `Thready` during simulation of different task systems. Task systems of different size and fixed utilization of 0.7 are randomly created. Each simulation sees an increase in memory usage at the beginning and end of the simulation when JSON data is parsed or simulation results are serialized. Due to the job generation approach, the memory consumption stays constant during the actual simulation.

minimal memory consumption available. Moreover, its flexibility allows to construct a simulator similar in features to `Thready`. We disabled tracing for the `rtlib` based EDF simulator because it severely penalizes simulation speed and memory consumption, and to make the comparison as fair as possible. The `rtlib` based EDF simulator and `Thready` are used to simulate the same task system, which is given in Table 4.3, for a ten hour simulation in millisecond time step resolution. For this simulation, each task's relative deadline and period are equal. As an implicit deadline task system, it is EDF schedulable with c_1 as WCET [148]. Moreover, actual computation demand of jobs is uniformly distributed $\mathcal{U}(c_0, c_1)$, and there is no additional inter arrival time between two jobs of the same task.

The `rtlib` based EDF simulator allocates all required memory up front to the simulation, which results in a flat memory profile of ≈ 134.8 KiB reported by Valgrind's massif. We compare this value to `Thready`'s constant memory profile of ≈ 1.7 KiB starting after JSON parsing finishes and the simulation core starts to operate. Compared with the `rtlib` based EDF simulator, `Thready` requires ≈ 98.7 % less heap memory, which enables to run many simulation threads in parallel on a single machine. Moreover, the small memory footprint facilitates the simulation process, as shown in Section 4.3.4.

Performance

To quantify the simulation speed of `Thready` and the `rtlib` based EDF simulator we profile both with the LINUX tool perf. Both simulators run a long term simulation of the task system in Table 4.3 for 10 years simulated time in millisecond time step resolution. The system running the simulations is a low-end workstation, where the simulation threads are first in first out (FIFO) scheduled with highest priority to minimize the performance penalty of context

Table 4.3: Implicit deadline task system for comparing simulators, which is EDF schedulable with c_1 as WCET.

Task i	Period	c_0	c_1
1	10	2	4
2	30	1	3
3	40	1	4
4	10	1	2

Table 4.4: Detailed performance results

		`rtlib`	`Thready`
	Runtime/s	9.96×10^3	1.68
	Context switches	2.49×10^3	1
	CPU migrations	1	0
	Page faults	168	70
L1 data cache	Loads	24.30×10^{12}	4.11×10^9
	Misses	8.76×10^9	36.44×10^3

switches and CPU migrations. Detailed results averaged from eight repetition runs are presented in Table 4.4.

The `rtlib` based EDF simulator runs for 9959.96(44375) s to simulate the task system in Table 4.3 for 10 years. `Thready` runs for 1.6823(243) s to simulate the same, which is a speedup in latency of `Thready` with respect to the `rtlib` based EDF simulator of ≈ 5920. The three orders of magnitude improvement of `Thready` over `rtlib` in performance is facilitated by the small memory footprint of `Thready`, which avoids memory bottlenecks and leverages L1 data cache performance.

4.3.5 Case study

The advantages of `Thready` are demonstrated best with a case study. We investigate a dual-criticality task system for ten years under transient errors. The system, shown in Table 4.5, consists of one low criticality task, and two high criticality tasks. Jobs from the high criticality tasks are not allowed to miss their deadline. The system is scheduled similar to EDF-VD, where earlier, virtual deadlines are introduced to reserve time for a mode change which abandons all low criticality jobs.

Table 4.5: Case study implicit deadline task system

Task i	Period	c_0	c_1	c_2	c_3	c_4	c_5
0	5	1	4	–	–	–	–
1	20	1	1	2	4	5	8
2	20	1	2	3	4	5	8

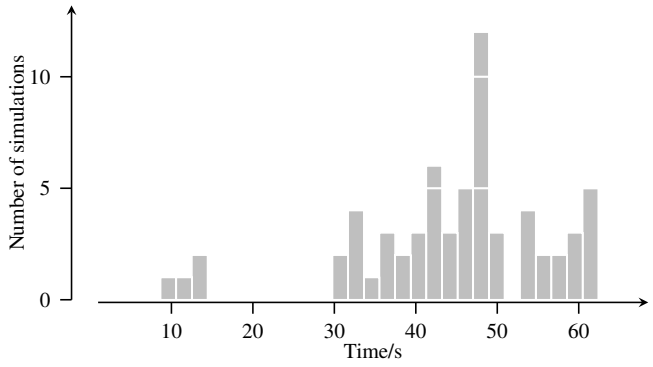


Figure 4.9: Histogram of simulation wall clock time. Measurements are gathered from 64 repetitions of a 10 year simulation of the task system in Table 4.5. Mean simulation wall clock time is 44.5 s.

Jobs from high criticality tasks may take longer to finish with a probability of $p = 0.05$, or may experience transient errors which require the job to restart with a probability of $p = 0.05$. In 90 % the job's execution time is uniformly drawn between c_0 and c_1. If a job may take longer, the job's execution time is uniformly drawn between c_2 and c_3. The transient error that requires the job to restart is modeled with a uniform execution time distribution $\mathcal{U}(c_4, c_5)$. `Thready`'s user interface allows to easily repeat the simulation experiment 64 times by running `Thready` with different random seeds using GNU `Parallel`. From all simulations, only two miss a virtual deadline, and no system violates its real deadline. The mean simulation wall clock time is 44.5 s, as indicated by the histogram in Figure 4.9.

Sophisticated long-term simulations under errors are possible due to `Thready`'s flexible error model, easy to instrument interface, and simulation performance: The three order of magnitude speedup in latency allows to simulate in 44.5 s what else would take $\approx$ 12.4 h.

4.3.6 Discussion

`Thready` combines the performance benefits from simulator framework approaches with easy to instrument interfaces and support for diverse error models. This frees the system designer from the burden of developing interfaces or single purpose programs, and fosters the development of reproducible scientific data analysis pipelines. Apparently `Thready` is a specialized and minimal tool in the tradition of UNIX: It solves one problem and solves it well [187]. Although other simulators provide more features like different scheduling algorithms, shared resources, or visualization, they lack in performance for long term simulations. Implementing such features for `Thready` requires to build further tools, which is easily enabled by `Thready`'s simple, short, and extensible code base.

4.3.7 Conclusion

`Thready` is the first portable and reproducible open source simulator to address long term simulations for sporadic task systems under errors. By `Thready`'s

Begin of code
```
import random
import math
import pandas as pd
import seaborn as sns
import numpy as np
sns.set_context("notebook")
sns.set_style("whitegrid")

from threadysequencer.tasksystem import (
    Tasksystem,
    TasksystemError,
    _generate_locrit_task,
    RTask,
)
from threadysequencer.generator import generate_uunifast_fraction
from threadysequencer.templates import BARUAH2012FIG2, HU2018
from threadysequencer.main import apply_virtual_deadline_scales

from tqdm.notebook import tqdm
```
End of code

Figure 4.10: Preamble for an acceptance rate experiment in a Jupyter notebook

three order of magnitude speedup in latency compared with the fastest state of the art simulator framework, system designers can investigate average case performance and QoS metrics, which facilitates understanding and better design decisions. Moreover, `Thready` is easy to integrate with other programs due to its interface, which enables sophisticated simulations, fosters integration into scientific data analysis pipelines, and encourages reproducibility for scheduling simulation experiments.

4.4 Reproducible simulation experiment infrastructure

Two kinds of experiments are of special interest for our work: *1)* Acceptance rates of random task sets; and *2)* long-term simulation experiments of random task sets to evaluate the effect of errors on QoS. Both should be executed in such a way that the results are reproducible, which allows to carry out equivalent tasks to reproduce the results [175]. To achieve this goal, we captured our workflow in Jupyter notebooks [188] and custom-written tools.

4.4.1 Acceptance rates

Acceptance rates are calculated in Jupyter notebooks with a Python 3.8 kernel. Each experiment has a dedicated notebook, which instruments our framework `threadysequencer` to generate random task sets, applies selected schedulability checks, and stores the results in a comma separated values (CSV) file. Figure 4.10 shows the preamble of an acceptance rate experiment notebook, which illustrates the used libraries and parts of our framework. Besides our framework we use a rather typical scientific Python stack with `numpy` and `pandas` for the collection of results, and `seaborn` for visualizations inside the notebook. The evaluation takes some time, mostly due to the random search for task sets, therefore we use `tqdm` to provide some feedback on the progression. The versions of all result-relevant Python libraries in use are listed in Table 4.6.

Table 4.6: Versions of used software packages

Software name	Version
pandas	1.1.2
numpy	1.17.3
scipy	1.5.2
sympy	1.6
jupyterlab	2.2.8
tabulate	0.8.7
Python	3.8.6
Linux kernel	4.18.0-305.3.1.el8.x86_64

4.4.2 Simulation experiments

Our simulation experiments evaluate the long-term behavior of task sets which are scheduled according to EDF-based mixed-criticality algorithms like EDF-VD, EDF-NUVD, and our single error tolerant variants. We are especially interested in the behavior of these algorithms under errors, and how errors influence the service for low-criticality tasks. Our framework provides the necessary functionality to implement the different mixed-criticality scheduling algorithms with `Thready`, by separating the simulation in distinct parts, one for each scheduler state.

The quantitative approach to these QoS simulations requires that we simulate a lot of random task sets to gather statistics about their behavior. Moreover, long runtimes demand a fast EDF simulation core. We use `Thready` for the later, and alleviate the problem of many simulations by parallelization.

Any random task set is simulated for a fixed number of repetitions, each with a different random seed. Furthermore, for each random seed all relevant scheduler modes need to be simulated. The number of required simulations with `Thready` is further increased by parameter combinations of the task set's utilization, composition of tasks, and the environment's error probabilities. For example, a utilization sweep from 0.65 to 1.0 in steps of 0.05 of 128 different task sets, each with 32 random traces, and two scheduler modes sums up to 57 344 simulations with `Thready`, and takes roughly 15.4 h until completion with 40 CPU cores[2].

A typical deployment of `threadysequencer` and `Thready` is shown in Figure 4.11. Both servers run the same version of `Thready`. The simulation experiment is controlled with our framework `threadysequencer` from a workstation, and distributes simulation on both servers. We use GNU `parallel` to transfer the generated task sets to the servers, initiate the simulation, and fetch the results. Each simulation result is assigned a unique identification number by calculating the hash of the task set and the following metadata: *1)* Simulated duration; *2)* Number of repetitions; *3)* Selected virtual deadline scales; *4)* Number of tasks in the task set; *5)* Running task set number from implicit fixed random seed; *6)* Task set utilization; and *7)* the used task set template which describes the task set's composition of tasks. The simulation

[2]Raw simulation time including time for random task system generation but without post-processing of results in Jupyter notebooks.

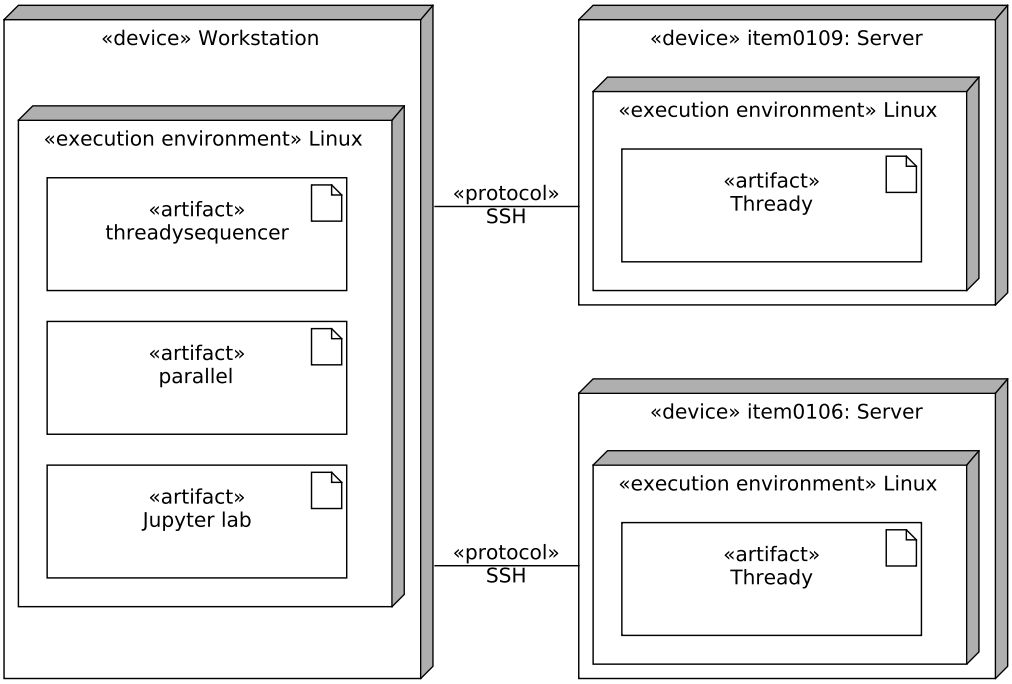


Figure 4.11: Deployment of `Thready` and our framework. Simulations are distributed over both servers by `threadysequencer` with help of GNU `parallel`.

results, as well as the randomly generated task sets themselves, are stored by `threadysequencer` in a hierarchical folder structure, sorted by modes and the unique simulation identification numbers. All stored files are plain text files and contain JSON-structured data.

Post-processing of the raw simulation results starts by collecting results in a convenient `pandas` data frame, followed by sanity checks to see if the simulation finished correctly. While the raw simulation data is authoritative, the data frame is easy to distribute, evaluate, and requires no knowledge about the hierarchical folder structure mentioned earlier. Furthermore, calculation of QoS figures and export to CSV files is implemented within Jupyter notebooks. Selected parts of the Python source code for `threadysequencer` are listed on Page 180 in the Appendix.

4.5 Conclusion

To investigate mixed-criticality scheduling algorithms we developed several tools. We motivated the need for static verification, and introduced the relevant parts of our framework which implements our mixed-criticality model. Moreover, we described how to generate random task sets for unbiased comparison of scheduling algorithms, and our simulator `Thready`, which enables long term simulations under errors and provides a three order of magnitude speedup in latency compared with the state of the art. We finished the chapter with a detailed description of our simulation setup, the deployment of our framework and `Thready`, and our scientific pipeline from simulation to reported statistics. In total, the developed framework enables the investigation of mixed-criticality

algorithms in a reproducible manner, which are presented in the following chapters.

5 Fault-tolerant scheduling with uniform deadline scales

For a preemptive uniprocessor, scheduling jobs from independent, sporadic, implicit deadline tasks, mixed-criticality scheduling approaches like EDF-VD guarantee that all high criticality tasks finish prior to their deadline, irrespective of the low criticality tasks [189]. This guarantee is achieved among other things by separating the run-time in modes, where change from the initial low criticality mode into the higher criticality mode, executing only high criticality tasks, is triggered by a high criticality job executing longer than accounted for. But abandoning low criticality tasks, or rather the service they implement, is not sensible [190, 191] if the system is expected to experience an actual mode change [192]. Especially in modern fault-tolerant hard real-time systems, where scheduling is not deterministic [193], mode change needs to be considered. Therefore holistic approaches need to go beyond scheduling guarantees for high criticality tasks, and strive to provide as much service as possible to low criticality tasks as well.

To extend the service of low criticality tasks, the change from low to high criticality mode should be delayed as long as possible [194]. Changing modes is tied to a particular event which is integral to the approach's schedulability check. For approaches like EDF-VD, the event is the overrun of a single high criticality job, which happens if the job's computation takes longer than anticipated by its low criticality budget.

We propose to reserve additional time during system design to accommodate for a single overrun, which results in a delayed or even prevented change into high criticality mode. With our approach, called EDF-VD-SE, we show

- how to find and reserve additional time by virtual deadlines using sequential least squares programming (SLSQP) (Section 5.2.2);
- that our adaptive approach results in a nearly constant acceptance rate of random task systems (Section 5.3.1); and
- how this approach improves the quality of service by effectively doubling the run-time on average in low criticality mode (Section 5.3.2).

5.1 Overview

EDF-VD is a preemptive uniprocessor dynamic scheduling approach for mixed-criticality task systems. It is an extension of EDF scheduling to mixed-criticality task systems by introducing earlier, virtual deadlines. As in EDF, the priority of a job is defined by its deadline [148]. The closer the deadline, the higher

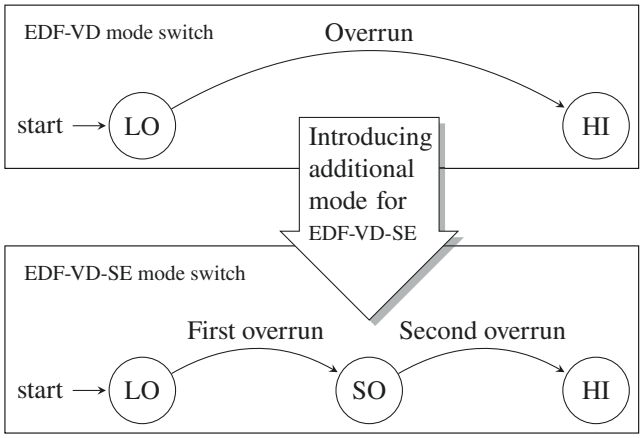


Figure 5.1: Mode switch in EDF-VD and EDF-VD-SE. With EDF-VD-SE we can tolerate a single overrun without abandoning low criticality tasks. With the second overrun, low criticality tasks are abandoned in favor of high criticality tasks. EDF-VD represents the mode switch behavior of classic dual-criticality mode switch schemes. Compared to classic dual-criticality mode switching, EDF-VD-SE has an intermediate mode labeled „SO“ (single overrun).

the priority of the job. The job with the highest priority is granted access to the processor, and can preempt a currently running job, which returns to the scheduler's job queue.

In contrast to EDF scheduling, EDF-VD introduces the concepts of criticalities and modes, which separates the system's run-time into two modes: In the first mode, jobs from all tasks are scheduled. Once a job from a high criticality task requires more computation than anticipated ($\gamma_{ij} > c_i^L$), the system changes to the second mode, where only jobs from high criticality tasks are scheduled.

To guarantee that all high criticality jobs have enough time to finish prior to their deadlines, EDF-VD introduces earlier, virtual deadlines which reserve time necessary for a successful mode change. These earlier deadlines increase the priority of the task's jobs. Once the switch to high criticality mode is triggered, jobs from high criticality tasks are scheduled with their original deadline.

We can interpret the scheduling in both modes as regular EDF scheduling with different sets of tasks. If both modes satisfy the constraints of regular EDF scheduling, with the transitional phase taken care of by earlier deadlines, the system is guaranteed to never miss a deadline [147].

The general idea of EDF-VD-SE is to delay the mode change in EDF-VD to high criticality mode as shown in Figure 3.7. The delay is achieved by tolerating a single overrun of execution time budget for a high criticality task, under the assumption that overruns are possible, but rare. In EDF-VD the scheduler would kill all low criticality tasks and serve only high criticality tasks after the first overrun. But with EDF-VD-SE, we can continue to service jobs from low criticality tasks, resulting in a better QoS for them.

With EDF-VD-SE, we consider both the schedulability of high criticality tasks, and the QoS for low criticality tasks during system operation. Compared to analysis-only approaches like EDF-VD, whose scope is the schedulability of high criticality tasks, EDF-VD-SE is motivated from a different perspective on the

problem: how to guarantee schedulability of high criticality tasks *and* improve the QoS for low criticality tasks. We deem this perspective more appropriate for safety-critical systems, and EDF-VD-SE as a holistic approach to solve the problem.

By reserving additional time during system design, EDF-VD-SE can tolerate an overrun of a high criticality task. The additional time is sourced from static slack, or by reducing the utilization of low criticality tasks. The question remains how much time do we need to reserve, and how to ensure that the approach works with the task system at hand. We adopt the idea of virtual, earlier deadlines as in EDF-VD to reserve time for the tolerated overrun. We formulate the question as an optimization problem and solve it with SLSQP to acquire a suitable virtual deadline scaling parameter and the maximum allowed utilization of low criticality tasks. Comparing with the task system at hand, we might find out that we can even add further low criticality tasks, which is valuable information for the system designer.

To derive a suitable virtual deadline scaling parameter for a task system with n high criticality tasks, we create for each high criticality task a virtual task system where the task's execution time budget in low criticality mode is equal to the budget in high criticality mode. If we can find a virtual deadline scaling parameter x suitable for all n virtual task systems, we can use this x to create the virtual deadlines $\hat{d}_i = xd_i$ in the original task system, and schedule the original task system with confidence that one overrun is tolerable.

5.2 Model

Our approach works well for modern fault-tolerant hard real-time systems build from COTS components. Ensuring the continuous correct service of such a system and thus making it reliable requires addressing faults during system design and operation. Faults are classified based on whether their duration is permanent or transient, their extend is local or distributed, and their value is determinate or indeterminate [195]. Once a fault is activated, the system deviates from its correct service state. This deviation is an *error*. If the error affects the delivered service a failure occurs [9]. Fault tolerance techniques are used to prevent system failure and differ in which class of fault they are able to tolerate.

Therefore our model, which we introduce in the following sections, can describe systems where error detection and correction is used to ensure the service of high criticality tasks, but not exclusively. First, we introduce the aspects of our model which are relevant during system operation, including mode change, application of virtual deadlines, and implications for the scheduler implementation (Section 5.2.1). Motivated by the system operation, we continue with the static aspects of our model which covers the schedulability check (Section 5.2.2).

5.2.1 System operation

The standard mode change scheme [147] [196] [192] for mode switched EDF scheduling of dual-criticality task systems separates the run-time in modes, with

the system starting in low criticality mode, where all jobs are scheduled according to their execution time budget c_i or c_i^L. Low criticality jobs are prevented to execute longer than c_i, but high criticality tasks can overrun their budget up to c_i^H. As soon as a high criticality tasks executes longer than c_i^L, the system changes into the high criticality mode, where only jobs from high criticality tasks are scheduled. Once the system enters high criticality mode, low criticality tasks, or rather the service they implement, are abandoned.

The mode change scheme in EDF-VD-SE is an expansion of the standard mode change scheme as shown in Figure 3.7. The additional intermediate mode between high- and low criticality mode accounts for the single tolerable error, and allows to continue servicing jobs from low criticality tasks. Once the second error is detected, the system changes into high criticality mode, which is identical to the standard mode change scheme.

In EDF-VD-SE the deadlines of high criticality jobs are scaled by x. When the system starts, jobs are EDF scheduled according to their earlier, virtual deadlines from the scaling. In case of a single error, which results in a job from a high criticality task τ_i to execute longer than c_i^L, the system records the error and continues to service all tasks. If a second error happens, all low criticality tasks and their jobs are removed from the system, and jobs from high criticality tasks are scheduled according to their original deadlines. Therefore the run-time operation is similar to EDF-VD except the additional flag to record the first error and to delay the mode change.

5.2.2 Schedulability conditions

To account for the unforeseen switch to high criticality mode, approaches like EDF-VD reserve processor capacity by introducing earlier, virtual deadlines $\hat{d}_i = xd_i$ for high criticality tasks. Therefore virtual deadlines should be chosen in such a way that they ensure schedulability during the mode switch, or transitional phase, under the assumptions of the implemented mode switch scheme. For a dual-criticality system, the necessary conditions to ensure steady-state schedulability under EDF-VD are [189]:

$$U_L^L + \frac{U_H^L}{x} \leq 1 \tag{5.1}$$

$$xU_L^L + U_H^H \leq 1 \tag{5.2}$$

In Equation (5.1) the term U_H^L/x accounts for the utilization increase of high criticality tasks from earlier virtual deadlines. We replace this term with $\hat{U}_H^L$, which reserves time for a *single* high criticality task τ_j in low criticality mode to tolerate an overrun without switching modes:

$$\hat{U}_H^L = \frac{c_j^H}{d_j} + \sum_{\substack{i \in \tau_H \\ i \neq j}} \frac{c_i^L}{xd_i} \tag{5.3}$$

The resulting schedulability condition for the low criticality mode considering τ_j is:

$$U_L^L + \frac{c_j^H}{d_j} + \sum_{\substack{i \in \tau_H \\ i \neq j}} \frac{c_i^L}{x d_i} \leq 1 \tag{5.4}$$

To account for a single error in *all* tasks we check if the task system is still schedulable if any high criticality task requires more time. This increases the number of equations we need to consider from two to $n + 1$ with n as the number of high criticality tasks in τ:

$$\forall j \quad U_L^L + \frac{c_j^H}{d_j} + \sum_{\substack{i \in \tau_H \\ i \neq j}} \frac{c_i^L}{x d_i} \leq 1 \tag{5.5}$$

The schedulability condition for the high criticality mode is unchanged.

In EDF-VD-SE, we seek a numerical solution of which deadline scaling parameter x of our task system with n high criticality tasks allows to schedule the task system while supporting the maximum amount of low criticality work U_L^L. Our vector of variables is $\boldsymbol{y} = \begin{bmatrix} x & U_L^L \end{bmatrix}$, our objective function is $f(\boldsymbol{y}) = U_L^L$, and we can formulate our search as an optimization problem, with the constraints stemming from the schedulability conditions in low- and high criticality mode for each high criticality task j:

$$\begin{aligned} \underset{\boldsymbol{y}}{\text{maximize}} \quad & U_L^L \\ \text{subject to} \quad \forall j \quad & 1 - U_L^L - u_j^H - \sum_{\substack{i \in \tau_H \\ i \neq j}} \frac{u_i^L}{x} \geq 0 \\ & 1 - x U_L^L - U_H^H \geq 0 \end{aligned} \tag{5.6}$$

The resulting optimization problem has a scalar objective function and $n + 1$ nonlinear inequality constraints, and can be solved by nonlinear programming (NLP). Because the objective function and constraints are twice continuously differentiable we can solve the problem with SLSQP [197, 198] to calculate the *maximum* utilization of low criticality tasks in low criticality mode U_L^m and working virtual deadline scaling parameter x. If the optimization succeeds, and U_L^L of the task system is below or equal to U_L^m, the task system is schedulable.

Example

As an example let us consider the task system in Table 5.1. The task system has two high criticality tasks τ_1, τ_2, and two low criticality tasks. As it stands, the utilization of low criticality tasks in low criticality mode is $U_L^L = 4/20$. Solving the optimization problem Equation (5.6) results in $U_L^m = 1/4$ and $x = 4/5$. Because $U_L^m \geq U_L^L$ the task system is schedulable by EDF-VD-SE.

The example's deadline scaling is visualized in Figure 5.2, which shows how earlier deadlines reserve time for the mode change. Despite earlier, virtual deadlines the period is not changed, and the mode is changed once one of the high criticality tasks executes longer than c_i^L.

Table 5.1: Example task system with two high criticality tasks and two low criticality tasks. The task system is schedulable with EDF-VD-SE, as demonstrated in Section 5.2.2. The virtual deadline scaling parameter for all high criticality tasks τ_j is $x = 4/5$.

j	c_i^L	c_i^H	$p_i = d_i$	u_i^L	u_i^H	xd_i
1	2	3	10	$4/20$	$3/10$	8
2	4	8	16	$5/20$	$5/10$	$64/5$
-	3	-	20	$3/20$	-	-
-	1	-	20	$1/20$	-	-

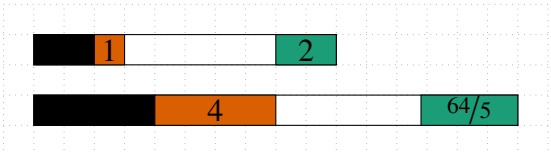


Figure 5.2: Visualization of both high criticality tasks from the example task system in Table 5.1. The length of each bar is equal to the task's period p_i and deadline d_i, with ratio of bar length to filled length as utilization. The black part of each bar is equal to the task's execution time budget in low criticality mode c_i^L. In high criticality mode, the execution time budget of each task is larger. The difference between low- and high criticality execution time budget is shown in red. Deadline scaling with x results in earlier deadlines, with the difference between original and virtual deadline indicated by the green part of the bar.

Modification of U_L^L

Solving the optimization problem Equation (5.6) can results in *1)* a schedulable task system $U_L^m \geq U_L^L$; or *2)* a non-schedulable task system $U_L^m < U_L^L$. Contrary to EDF-VD, we can use the additional information of U_L^m to modify our task system in both cases: *1)* $U_L^m \geq U_L^L$ allows us to add additional low criticality tasks to our task system until $U_L^L = U_L^m$; and *2)* $U_L^m < U_L^L$ guides us in removal of tasks until $U_L^L = U_L^m$. In both cases, a difference in utilization $\Delta U_L^L = U_L^m - U_L^L$ of zero indicates the maximum load for the processor. We investigate how modification of U_L^L by ΔU_L^L can help to improve the acceptance rate for EDF-VD-SE in Section 5.3.1.

5.3 Experiments

To understand the usefulness of EDF-VD-SE, we investigate how many task systems are actually schedulable with EDF-VD-SE. For this we generate a set of random task systems and apply the schedulability check described in Section 5.2.2, noting which task systems are schedulable and which not.

Moreover, we want to know the benefit in QoS, or additional system run-time beyond the first error. We use `Thready` [199] to simulate a large set of random task systems to investigate the run-time until the first- and second error.

5.3.1 Acceptance rate of `UUnifast` random task systems for different utilizations

The acceptance rate is the number of schedulable task systems divided by the total number of task systems. We generate random task systems to investigate how many can be scheduled by EDF-VD-SE in comparison to EDF-VD. One benefit of EDF-VD-SE is that we know the maximum allowed utilization by low criticality tasks after the schedulability check. Therefore we investigate the acceptance rate of EDF-VD-SE in two ways: First, we compare if the utilization from low criticality tasks in the random task system is below or equal to the maximum utilization allowed by EDF-VD-SE. If this is the case, we count the task system as schedulable. In the second investigation we use our knowledge about the maximum allowed utilization to modify the random task system, and report if it is schedulable after modification. While we do this, we take note of the low task utilization delta, which can be positive, if EDF-VD-SE allows to add further low criticality tasks to the system, or negative, if EDF-VD-SE requires to lower the utilization of low criticality tasks. Such a negative utilization delta can be interpreted as the cost to allow a single error.

In Figure 5.3, we compare the acceptance rate of EDF-VD-SE to EDF-VD, with and without modification of the random task system. Each data point is the result of calculating the acceptance rate of 1024 task systems for a specific utilization in low criticality mode $U_L = U_L^L + U_H^L$. We generate random task systems with specific U_L as described in Section 4.2, following a parameterization for task systems which are mostly schedulable by EDF-VD [189]: For each task periods are uniformly drawn between $p_l = 50$ and $p_u = 200$, and pessimism is uniformly

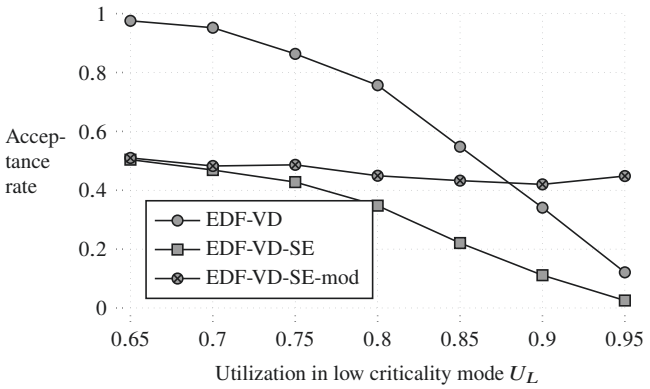


Figure 5.3: Acceptance rate of EDF-VD-SE and EDF-VD for `UUnifast` random task system. The data labeled „EDF-VD-SE-mod“ is for EDF-VD-SE with adjustment to the low criticality task utilization.

drawn between $z_l = 1$ and $z_u = 2$.

While EDF-VD shows a superior schedulability for task systems with utilization $U_T < 0.9$, EDF-VD-SE can use the knowledge about maximum low criticality task utilization to provide a near constant acceptance rate around 0.5. Moreover, the general decline of the acceptance rate is less emphasized for EDF-VD-SE without adaption compared to EDF-VD.

If we adjust our task systems by ΔU_L^L on average, as shown in Figure 5.4, we achieve a near constant acceptance rate. It is interesting to see how the average ΔU_L^L is positive for utilizations $U_L < 0.85$, and turns into a cost for task systems with higher utilization.

5.3.2 Quality of service comparison by mode switch time

Estimating the QoS improvement requires to simulate how task systems are actually scheduled on a uniprocessor. It is not sufficient to apply the schedulability check, which only ensures that the task system is schedulable. To investigate the increased service for low criticality tasks, we need to measure how long jobs from such tasks are scheduled, and when the system changes modes. The general idea is to simulate each task system multiple times until it switches to high criticality mode, to cover different job arrival patterns, and to analyze the scheduling response of the system.

These simulations are random by nature, and allow us to gather quantitative results. Random decisions during simulation are *1)* when jobs from a task arrive at the scheduler; *2)* how much computation the job requires; and *3)* does the job overrun its execution time budget if it is from a high criticality task. We control the randomness in these decisions in our simulation experiment by parameters, which we introduce prior to the actual simulation results.

Simulation parameters

Our simulation setup allows to change several parameters which influence the random decisions during simulation. The time between two jobs of the same sporadic task is at least its period, but by random chance it might be longer.

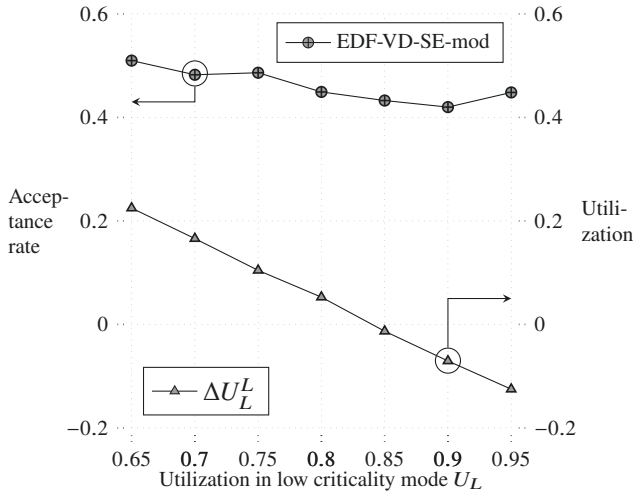


Figure 5.4: Acceptance rate for EDF-VD-SE with adjustment to the low criticality task utilization, described by the average ΔU_L^L. Arrows indicate the relevant axis for each data. A negative ΔU_L^L requires to decrease the utilization of low criticality tasks, and a positive value allows to add more.

We assume exponential distributed job inter-arrival times. The exponential distribution is parameterized with β, with the CDF as $F(x) = 1 - e^{-x/\beta}$ for $x \geq 0$. The time between two jobs is $p_i + e_i p_i$. We choose $\beta \ll 1$ to set the time between two jobs of the same task to its period p_i, to demonstrate our independence of dynamic slack time. In general, longer job inter-arrival times are beneficial for scheduling, because it creates dynamic slack time, which can be used to service jobs that else might have failed to meet their deadline. Our $\beta \ll 1$ is the worst case assumption in terms of dynamic slack, because it reduces the accumulation of dynamic slack to dynamic slack from unused computation budgets.

The environment is reflected in the probability p to have an error, which is observable by the scheduler as an overrun in computation when a high criticality job executes for $c_i > c_L$. With $p = 0.001$ we have a rather harsh environment, where it is not uncommon to have an error which results in an overrunning high criticality task. Each job's overrun probability is independent and equal to p. It is interesting to note that choosing a larger p is not influencing the fairness of the QoS comparison between EDF-VD and EDF-VD-SE. While a larger p increases the chance to have an error, which results in EDF-VD dropping the low criticality tasks while EDF-VD-SE can continue, it increases the chance to have a second error as well, which limits the QoS improvement of EDF-VD-SE over EDF-VD too.

The computation for each job from low criticality tasks is chosen uniformly between 1 and c_L. Jobs from high criticality tasks can overrun. If they overrun, the computation is chosen uniformly between $c_L + 1$ and c_H, else between 1 and c_L. This behavior is identical in both low criticality mode, and the intermediate „single overrun“ mode.

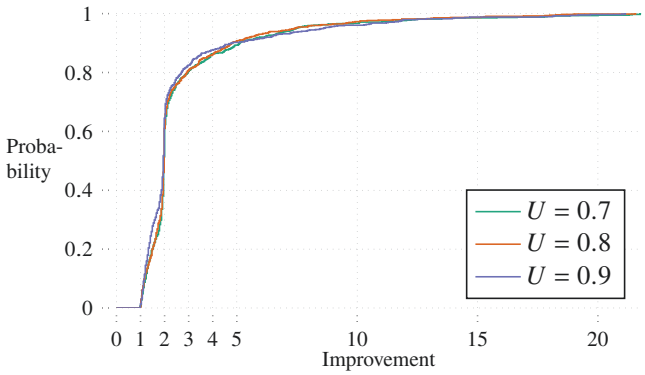


Figure 5.5: CDF of EDF-VD-SE QoS improvement. The QoS improvement is the ratio of time until the second error to first error: QoS = t_2/t_1. Task systems are `UUnifast` random generated with target utilizations of $U = 0.7$, $U = 0.8$, and $U = 0.9$.

Simulation results

We create dual criticality `UUnifast` random task systems with generation parameters similar to Baruah [189] which are schedulable by EDF-VD-SE as described in Section 5.2.2. We use `Thready` to simulate each EDF-VD-SE schedulable task system with a error probability of $p = 0.001$ for one hour with a millisecond time step resolution. Because EDF-VD-SE can tolerate a single error, which results in a single overrunning high criticality task, we record the system run-time up to the second overrun, where EDF-VD-SE would switch into high criticality mode and abandon all low criticality tasks. The prolonged run-time is the additional service for low criticality tasks. We calculate the QoS improvement as the ratio of time until the second error to first error: QoS = t_2/t_1. EDF-VD-SE either achieves the same or better QoS than EDF-VD, and Figure 5.5 show the distribution of all instances with QoS improvement, excluding improvements beyond three standard deviations. The steep incline of the CDF at an improvement of two is the result of a constant error probability p where on average an error happens every t_1 time steps, effectively doubling the run-time on average in low criticality mode with EDF-VD-SE.

The CDF of the QoS is independent of the actual value of p, because p influences both t_2 and t_1. Nevertheless, a higher value of p results in earlier t_2 and t_1, which is interesting to know for a specific task system in an actual application.

The run-time histogram in Figure 5.6 compares the distributions of system run-time until switch to high criticality mode between EDF-VD and EDF-VD-SE. For each approach, the same task systems with utilizations between $U = 0.65$ and $U = 0.95$ have been simulated. The distributions look like Log-normal, which is a result of the independent job overrun probability p, where the system run-time random variable is a multiplicative product of many independent random variables.

If we set the pessimism factor $z_l = z_h = z$, we can investigate how a controlled over-allocation of budgets influences the QoS. We select $z = 2$ to model software error correction approaches where jobs are repeated in case of detected errors. Larger values of z represent multiple recomputations and possibly majority

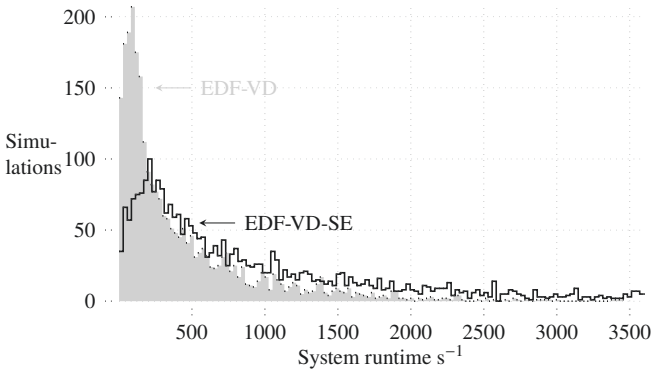


Figure 5.6: System run-time histogram of baseline EDF-VD and improved EDF-VD-SE.

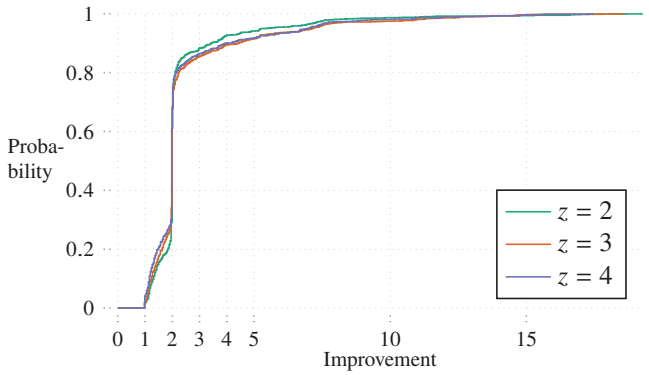


Figure 5.7: CDF of EDF-VD-SE QoS improvement for fixed pessimism factor z. The QoS improvement is the ratio of time until the second error to first error: QoS $= t_2/t_1$. Task systems are `UUnifast` random generated with target utilizations of $U = 0.7$.

voting of results to correct errors in software. As shown in Figures 5.7 to 5.9, the impact of the pessimism factor is nearly identical over all shown utilizations U. Looking at Figure 5.10 for $U = 0.7$, the pessimism factor emphasizes the improvement of EDF-VD-SE over EDF-VD, as more systems increase their run-time with increasing pessimism.

5.4 Discussion

EDF-VD-SE can tolerate a single job overrun without dropping jobs or missing deadlines, and represents an approach which makes a task system fail operational (FO) with a count of one [200]. Reserving additional time to tolerate an overrunning task successfully delays or prevents the change to high criticality mode. On average the QoS doubles, and some task systems manage to improve QoS by a factor of more than five. The static approach requires only minimal extension to the EDF-VD scheduler run-time to record the single overrun, which is beneficial for deeply embedded systems.

EDF-VD-SE is limited to handle a single error, multiple errors can cause utilization spikes which can lead to deadline violations depending on the currently

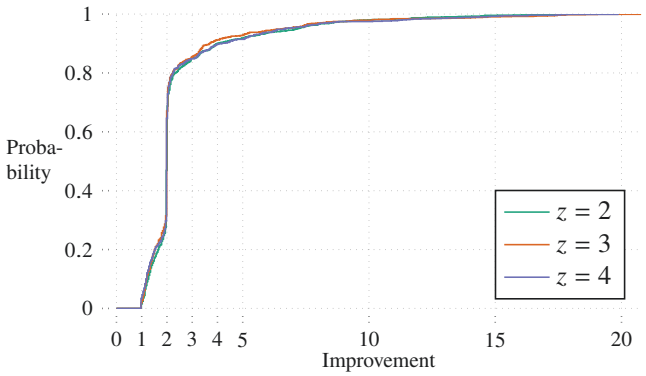


Figure 5.8: CDF of EDF-VD-SE QoS improvement for fixed pessimism factor z. The QoS improvement is the ratio of time until the second error to first error: QoS $= t_2/t_1$. Task systems are `UUnifast` random generated with target utilizations of $U = 0.8$.

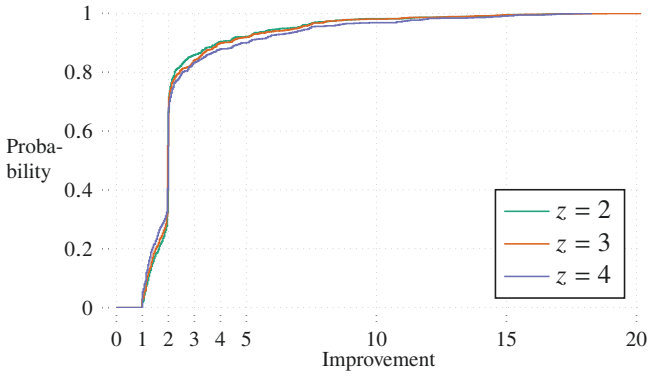


Figure 5.9: CDF of EDF-VD-SE QoS improvement for fixed pessimism factor z. The QoS improvement is the ratio of time until the second error to first error: QoS $= t_2/t_1$. Task systems are `UUnifast` random generated with target utilizations of $U = 0.9$.

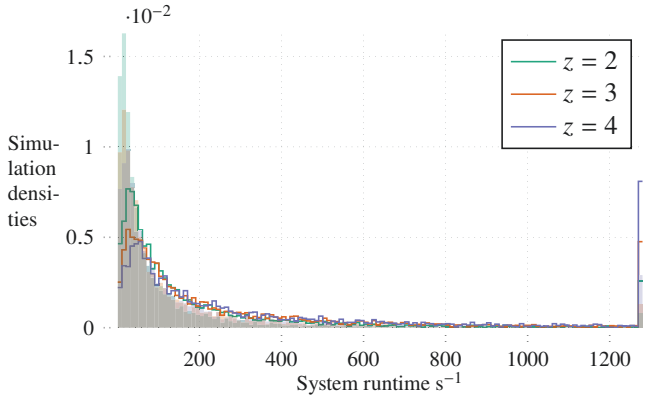


Figure 5.10: System run-time histogram of EDF-VD-SE for different pessimism factors z. Filled areas show the densities of EDF-VD for the corresponding EDF-VD-SE results of the same color. Bin size is 10 s, and the last bin contains the accumulated results of all simulations with run-times between 1280 s and 3600 s. Increased pessimism emphasizes the improvement of EDF-VD-SE over EDF-VD, as more systems increase their run-time. This is shown in the smaller spikes below 200 s, and increased number of simulations beyond 1280 s.

available dynamic slack. Nevertheless tolerating a single error already improves QoS, and multiple errors during system operation are rare if the error probability and system run-time are suitable for the application.

Compared to EDF-VD, the acceptance rate is fair considering that EDF-VD represents an optimal non-clairvoyant algorithm [147] which is not designed to tolerate a single error. Moreover, EDF-VD-SE allows the designer to choose virtual deadline scaling such that the utilization of low criticality tasks in low criticality mode is optimized.

5.5 Related work

Mixed criticality scheduling is a major aspect in modern fault-tolerant hard real-time systems which are build from COTS components to satisfy SWaP constraints [144, 201]. Scheduling for such systems is not deterministic due to the probabilistic nature of errors [193]. Therefore scheduling approaches can only strive to extend the system lifetime, or guarantee schedulability for a limited amount of errors in a given time window [194, 202], which is again only probabilistic.

Most dynamic priority scheduling approaches for fault-tolerant mixed criticality systems are related to, or based on, preemptive uniprocessor EDF scheduling. Preemptive uniprocessor EDF scheduling is a optimal solution which can schedule jobs from independent, sporadic, implicit deadline tasks [148, 149]. While fault-tolerance extensions of EDF exist [201, 203, 204], they do not support mixed criticality systems natively. The extension of EDF to mixed criticality systems by EDF-VD is an optimal non-clairvoyant algorithm [147] without explicit fault tolerance considerations, and the basis for EDF-VD-SE.

Regarding EDF-based scheduling approaches, fault tolerance can be achieved

by reserving additional time, by smarter mode changes, or by slack management. The elastic mixed criticality model with variable periods for low criticality tasks [205] or period scaling [206] provide a way of reducing the utilization to free additional time, which is exploited in EDF-VD-SE to tolerate a single fault. Reserving some time for an overrunning task [207] requires knowledge about the error probability of each task during system design, which is at best hard to estimate, and wrong estimates risk overloading the system. Contrary EDF-VD-SE is always safe for a single overrun, no matter the error probability, which can only reduce the additional service for low criticality tasks.

Assigning low criticality tasks a smaller execution time budget on higher levels to avoid a mode change [208], or letting a subset of low criticality tasks execute in high mode [209] can improve QoS for low criticality tasks, similar to EDF-VD-SE. Contrary to EDF-VD-SE, which is a static approach, service for low criticality tasks can be adapted dynamically by assuming independence of high criticality tasks and their mode switches [210]. Transitioning from high criticality mode back to low criticality mode [211] allows to improve QoS even further, but is not implemented in EDF-VD-SE yet.

An orthogonal approach is extensive dynamic slack monitoring [160], where opposed to EDF-VD-SE a slack-aware run time management is mandatory. This is a burden for deeply embedded systems, which lack additional computation resources to implement the slack-aware run time management.

5.6 Conclusion

Scheduling mixed-criticality fault-tolerant hard real-time systems needs to consider the QoS for low criticality tasks. Our static approach EDF-VD-SE reserves additional time to tolerate a single error without risking deadline violations while optimizing the amount of possible low criticality work, doubling on average the QoS for low criticality tasks. EDF-VD-SE requires no slack-aware run-time operation, which is beneficial for deeply embedded systems with limited computation resources, and no assumptions about error probabilities for safety guarantees. Therefore EDF-VD-SE is a viable and certification friendly approach to schedule tasks in modern deeply embedded mission and safety-critical systems.

6 Fault-tolerant scheduling with non-uniform deadline scales

To prevent catastrophic consequences, computer systems in critical applications have to tolerate faults. Accordingly, faults have to be considered during the design of such systems. A major consideration in real-time systems are faults that result in timing related errors. Their mitigation is of paramount importance to guarantee that all computations finish in time, prior to their deadline. But error mitigation capabilities are costly, as they require additional resources to provide the necessary redundancy. Therefore, it is beneficial to limit error mitigation to the highly critical functions of a system, and to provide lesser guarantees for functions of lower criticality.

In hard real-time systems, the scheduler is responsible to build a valid schedule such that all jobs finish before their deadline. In most applications, the schedulers are non-clairvoyant, and do not know a job's execution- or arrival time beforehand. To still guarantee a valid schedule, they need guarantees of worst-case execution times [145] and arrival times. These guarantees are captured in a task model. Mixed-criticality task models are the most relatable models for fault-tolerant hard real-time systems, as they allow to specify multiple estimates and therefore different levels of assurance for a job's execution time [191]. A task's criticality influences if and how a job can access the processor, and allows us to formulate different guarantees for tasks of different criticality.

Arguably the most popular mixed-criticality model, developed by Vestal [146] , differentiates between low- and high criticality tasks, but guarantees execution before their deadline only for jobs from tasks of high criticality. Jobs from low criticality tasks can execute as long as the jobs from high criticality tasks did not exceed their execution time budgets, else they are discarded. This lowers the quality of service for low criticality tasks in applications where we expect execution time budget overruns.

But lowering the quality of service is not always reasonable [190, 191], which requires consideration especially for modern fault-tolerant hard real-time systems. Instead, we can strive to provide continuous degraded service, or a graceful degradation, for the low criticality tasks. The provided degraded service is based on time redundancy sourced from spare processing time, or slack, which is the remaining time after computations of high criticality have been served. Typically, the slack is insufficient to execute all computations from all low criticality functionalities with their desired arrival rate. Therefore, current approaches match the demand from low criticality functionalities to available slack. This can be achieved by assigning low criticality tasks a smaller execution time budget on higher levels [208, 212] , or by variable [205] or scaled [206] periods. But selection of smaller execution time budgets is not possible if the budget is already at the minimum time required for the task, and partial execution

provides no benefit. Moreover, variable or scaled periods require a different task model, which assumes that low criticality tasks can provide their functionality with varied job frequency. Depending on the application, this assumption is not always valid: For example, in control applications, shorter sampling periods can decrease control performance and system stability [213]. Further possibilities are to dynamically adapt tasks independent of each other [210] , schedule only a subset of tasks in high criticality mode [209], or periodical dropping of low criticality jobs in high criticality mode [214]. While these approaches generate slack, they require more computational resources, due to their elaborate schedulers, compared to simpler mode-switched EDF schedulers. Moreover, the lack of resources in deeply embedded systems rules out dynamic slack monitoring approaches [160] , or to precompute proper schedules for all scenarios at design time [215]. Therefore, approaches which emphasize static design time decisions over elaborate dynamic decisions during system operation are preferable, but verification-friendly state-of-the-art approaches, which resort to static slack analysis during design time [150, 161] , suffer from low acceptance rates. Hence, there is a strong need for a mixed-criticality approach for industrial fault-tolerant systems which provides degraded service to low criticality tasks, is sensible for verification, and likely finds a solution for highly loaded systems.

This work introduces an approach with these desired properties to target mentioned systems in a straightforward way: We reserve additional time by virtual deadlines to accommodate for the first error, and to guarantee service to high criticality functionalities beyond the second error. By considering the system's work done before an error, we formulate tighter worst-case bounds, which allows us to find in a wider set of systems than previous suitable virtual deadline scaling factors. Furthermore, we formulate the schedulability conditions as an optimization problem, and solve them before system deployment, resulting in a static, verification friendly approach called EDF-IVD-SE. We show that

- EDF-IVD-SE is feasible (Section 6.2);
- EDF-IVD-SE is capable of tolerating a single error with affordable costs (Section 6.3); and
- EDF-IVD-SE doubles on average the QoS for low criticality tasks (Section 6.7.2).

Our affordable, fault-tolerant and verification-friendly approach achieves up to 1.56 better acceptance rate compared to EDF-Allowance, while doubling the QoS for low criticality tasks on average compared to optimal traditional mixed-criticality scheduling approaches like EDF-VD.

6.1 Problem overview

We want to use a fault-tolerant, real-time computer system in critical applications. Our problem is to verify that no deadline is missed *prior* to system operation, given a known scheduler, all timing-related properties of our application, and a model of faults and errors.

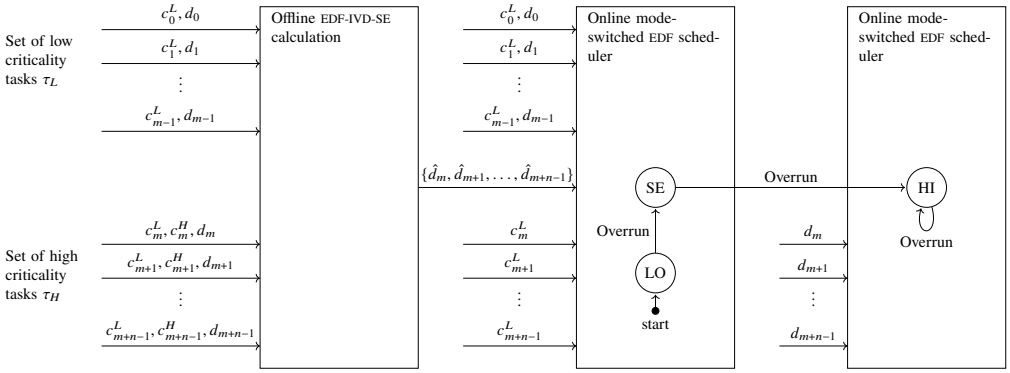


Figure 6.1: Overview of our approach with a task set $\mathbb{T} = \tau_L \cup \tau_H$ containing m low criticality tasks, and n high criticality tasks. Each block represents a process, which requires the information on the incoming arrows to generate the information on the outgoing arrow. The state machine diagram overlay on the right indicates when each process is active. During offline preparation, the application model is used to verify schedulability and to calculate the virtual relative deadlines $\hat{d}_i$ for all high criticality tasks by solving our optimization problem. The virtual relative deadlines are, besides schedulability verification, the key results from the offline calculations, which happen prior to online operation. During online operation, the mode-switched EDF scheduler uses the virtual relative deadlines to schedule jobs from high criticality tasks. Moreover, the system faces errors, which are considered in the application model as overruns of execution time budgets. Once the system is in high criticality mode, resembled by switching into state HI, jobs from high criticality tasks are scheduled by their original relative deadlines.

In our fault- and error model we consider that jobs from tasks of high criticality are protected to tolerate faults, but the protection costs additional execution time. If this additional execution time results in exceeding a task specific execution time limit, we have an error. Sometimes we refer to errors as overruns, if we want to stress the exceeding of task specific execution time limits, from a viewpoint of our application model.

A process-based overview of our EDF-IVD-SE approach is shown in Figure 6.1. Given the timing-related properties of our application model, we calculate how to account for errors during system operation by virtual, earlier deadlines. If a solution exists, we can deploy the system. During system operation, a mode-switched EDF scheduler selects which jobs are allowed to access the CPU. A detailed behavioral description of the mode-switched EDF scheduler is provided in Section 6.5.

The following Sections 6.2 to 6.4 show how to solve the initially stated problem, which is to verify that no deadline is missed considering errors during system operation.

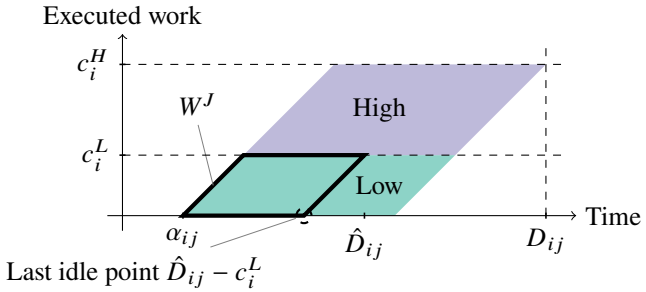


Figure 6.2: Visualization of Lemma 15 showing the job progression window W^J defined by the choice of a virtual absolute deadline $\hat{D}$. Job J_{ij} of task τ_i arrives at α_{ij}. Its fixed absolute deadline D_{ij} is $\alpha_{ij} + d_i$. Both execution time budgets are fixed as well, but the virtual absolute deadline is selected by the scheduling approach. The green shaded area below c_i^L and the purple shaded area between c_i^L and c_i^H indicate how the job can execute in low- and high criticality mode. The guarantee that in low criticality mode no deadline is missed defines the last idle point, which defines the window of possible progressions in executing a job. By selection of a virtual absolute deadline, the window can close or open fully to contain the whole green shaded area.

6.2 EDF-IVD: Reducing the pessimism in EDF-NUVD

The worst case assumption in Proposition 14 is very conservative, because it assumes that in high criticality mode each high criticality task needs to execute for c_i^H. This assumptions neglects that prior to the mode change some work has already finished. By taking this prior work into account, we derive tighter bounds for EDF-NUVD in this section.

Lemma 15 (Last idle point)**.** *The last point in time where a job in low criticality mode did not execute is at* $x_i D_{ij} - c_i^L$.

Proof. With a supply of $\sigma = 1$, the uniprocessor needs at least c_i^L time units to finish a job with execution demand of c_i^L, as shown in Figure 6.2. Jobs that never executed until later than c_i^L time units prior to their virtual absolute deadline would miss their virtual absolute deadline. Because no absolute deadlines or virtual absolute deadlines are missed by EDF-NUVD and EDF-NUVD-SE in low criticality mode if Equations (2.57) and (6.16) hold, the jobs can't miss their deadlines, therefore the last point in time where a job in low criticality mode did not execute is at $x_i D_{ij} - c_i^L$. □

With Lemma 15 we can derive an improved version of EDF-NUVD named EDF-IVD:

Proposition 16 (EDF-IVD schedulability)**.** *Let* $\mathbb{T}$ *be a dual-criticality implicit*

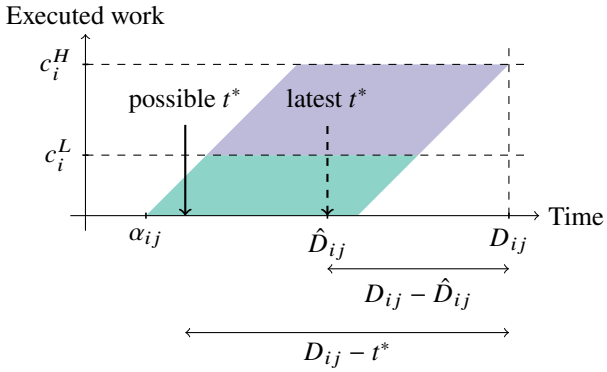


Figure 6.3: Visualization of Proposition 16. For the job to be active, the *latest* first overrun time t^* can be at the job's virtual absolute deadline $\hat{D}_i$. Whenever t^* is, the active job executed according to the shaded region below c_i^L during low criticality mode.

deadline task set and let $0 < x_i < 1$*, for each high criticality task. If*

$$U_L^L + \sum_{i:\chi_i=H} u_i^L/x_i \leq 1 \tag{6.1}$$

$$\sum_{i:\chi_i=H} u_i^H/(1 - x_i + u_i^L) \leq 1 \tag{6.2}$$

with mode switch from low- to high criticality mode at the first overrun time, then $\mathbb{T}$ *is schedulable by EDF-IVD.*

Proof. No deadline is missed by EDF-IVD in low criticality mode if Equation (6.1) holds. The first time where a job from a high criticality task exceeds its low criticality execution time budget is at t^*. Consider a job J_{ij} of a high criticality task τ_i that is active at t^*. The job arrives at α_{ij}, and has a absolute deadline at $D_{ij} = \alpha_{ij} + d_i$. Before t^*, J_{ij} is EDF-scheduled according to its virtual absolute deadline $\hat{D}_{ij} = \alpha_{ij} + x_i d_i$.

Since J_{ij} is still active at t^*, its earliest virtual absolute deadline is at the first overrun time: $\hat{D}_{ij} \geq t^*$. Therefore the duration between the first overrun time and absolute deadline is larger or equal to the duration between absolute deadline and virtual absolute deadline, as shown in Figure 6.3:

$$D_{ij} - t^* \geq D_{ij} - \hat{D}_{ij} \tag{6.3}$$

To derive the worst case assumption considering Lemma 15, we need to differentiate two cases: *1)* first overrun time is prior to last idle point; and *2)* first overrun time is in $[\hat{D}_{ij} - c_i^L, \hat{D}_{ij}]$. If the first case $t^* < \hat{D}_{ij} - c_i^L$ is true, then $D_{ij} - t^* > D_{ij} - (\hat{D}_{ij} - c_i^L)$. By switch to high criticality mode at t^*, the job J_{ij} of high criticality task τ_i has enough time left in this case to finish prior to its deadline.

In the second case, where t^* is in $[\hat{D}_{ij} - c_i^L, \hat{D}_{ij}]$, the job had to start executing prior to t^*, because Equation (6.1) holds. Still $t^* \geq \hat{D}_{ij} - c_i^L$, and $D_{ij} - t^* \leq D_{ij} - \hat{D}_{ij} + c_i^L$. But due to the low mode guarantee Equation (6.1)

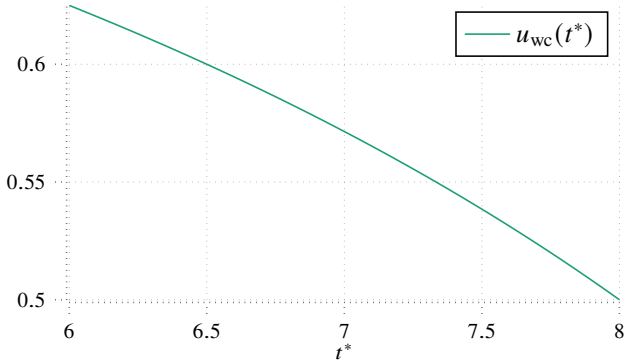


Figure 6.4: Evaluation of Equation (6.4) for example task and corresponding job with $c_i^L = 2, c_i^H = 5, D_{ij} = 14, \hat{D}_{ij} = 8$.

the job executed for $t^* - (\hat{D}_{ij} - c_i^L)$, and in the remaining time $D_{ij} - t^*$ only $c_i^H - t^* + \hat{D}_{ij} - c_i^L$ need to be worked on:

$$u_{\mathrm{wc}}\left(t^*\right) = \frac{c_i^H - \left(t^* - \left(\hat{D}_{ij} - c_i^L\right)\right)}{D_{ij} - t^*} \tag{6.4}$$

The derivative of Equation (6.4) with respect to t^* shows that Equation (6.4) has no extrema:

$$\frac{\mathrm{d}u_{\mathrm{wc}}}{\mathrm{d}t^*} = \frac{c_i^H - c_i^L}{\left(t^* - \hat{D}_{ij}\right)^2} \tag{6.5}$$

To identify the worst case utilization u_{wc} we look at both end points in the range $[\hat{D}_{ij} - c_i^L, \hat{D}_{ij}]$ for t^*, because Equation (6.4) declines with t^* and has no extrema for tasks according to Equation (2.16), as shown in Figure 6.4:

$$u_{\mathrm{wc}}\left(\hat{D}_{ij} - c_i^L\right) = \frac{c_i^H}{D_{ij} - \hat{D}_{ij} + c_i^L} \tag{6.6}$$

$$u_{\mathrm{wc}}\left(\hat{D}_{ij}\right) = \frac{c_i^H - \left(\hat{D}_{ij} - \left(\hat{D}_{ij} - c_i^L\right)\right)}{D_{ij} - \hat{D}_{ij}} \tag{6.7}$$

$$= \frac{c_i^H - c_i^L}{D_{ij} - \hat{D}_{ij}} \tag{6.8}$$

Given the utilization for both points, we still need to show which utilization is larger. A geometric interpretation is to identify the steeper slope, as shown in Figure 6.5.

Lemma 17. *The utilization described by Equation* (6.6) *is larger than the utilization described by Equation* (6.8):

$$u_{wc}\left(\hat{D}_{ij} - c_i^L\right) - u_{wc}\left(\hat{D}_{ij}\right) > 0$$

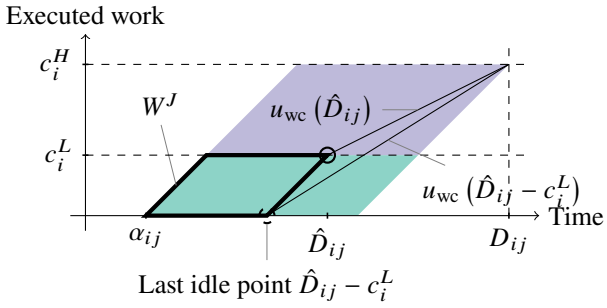


Figure 6.5: The steepest slope is observed at the last idle point, which corresponds to the largest utilization as described by Equation (6.6). Both slopes start at the boundary of the job progression window W^J. In this example, with $c_i^L = 2, c_i^H = 5, D_{ij} = 14, \hat{D}_{ij} = 8$, the slopes are $u_{\text{wc}}\left(\hat{D}_{ij} - c_i^L\right) = 5/8$ and $u_{\text{wc}}\left(\hat{D}_{ij}\right) = 4/8$.

Proof. We show that Equation (6.6) defines the worst case by contradiction, rewriting the difference as follows:

$$\frac{c_i^H}{D_{ij} - \hat{D}_{ij} + c_i^L} - \frac{c_i^H - c_i^L}{D_{ij} - \hat{D}_{ij}} \leq 0 \tag{6.9}$$

$$D_{ij} - \hat{D}_{ij} + \leq c_i^H - c_i^L \tag{6.10}$$

$$1 \leq \frac{c_i^H - c_i^L}{D_{ij} - \hat{D}_{ij}} \tag{6.11}$$

We can reason about the contradiction in two ways: *1)* If Equation (6.10) is true, the reserved time for high criticality work can be less than actual high criticality work, contradicting schedulability; or *2)* Due to the problem definition in Equation (2.16), where $c_i^L = c_i^H$ is possible, the difference in Equation (6.11) can become zero, leading to a contradiction. Therefore the utilization described by Equation (6.6) is greater than the utilization described by Equation (6.8). □

With the worst case utilization identified by Lemma 17, we can rewrite Equation (6.6) to get rid of absolute deadlines and virtual absolute deadlines:

$$u_{\text{wc}} = \frac{c_i^H}{D_{ij} - \hat{D}_{ij} + c_i^L} = \frac{c_i^H}{D_{ij}\left(1 - x_i + c_i^L / D_{ij}\right)} \tag{6.12}$$

Replacing the job's absolute deadline D_{ij} with $\alpha_{ij} + d_i$, and considering a synchronous release pattern of jobs from all tasks $\forall i\ \alpha_{i0} = 0$ yields:

$$u_{\text{wc}} = \frac{c_i^H}{d_i\left(1 - x_i + c_i^L / d_i\right)} = \frac{u_i^H}{1 - x_i + u_i^L} \tag{6.13}$$

Now Equation (6.2) regards the task set after t^* as a single-criticality task set under the worst case, where each high-criticality task τ_i has an increased task utilization as described by Equation (6.13). If the sum over all increased task utilizations is below the supply of one, this single-criticality task set is schedulable. □

6.3 Extension to single overrun tolerance

In our anticipated applications execution time budget overruns are expected [161]. As traditional dual-criticality scheduling discards low criticality tasks upon the first overrun, under the assumption that overruns are unlikely, we propose the following extensions for single overrun tolerance to provide better service to low criticality tasks.

First, we show how EDF-NUVD can be extended to tolerate a single overrun, which we call EDF-NUVD-SE, then we extend EDF-IVD to EDF-IVD-SE which can tolerate a single overrun as well.

6.3.1 Single error extension of EDF-NUVD

To derive the schedulability conditions for EDF-NUVD-SE we reserve additional time in low criticality mode to tolerate a single overrun. We increase the virtual utilization of high criticality tasks in low criticality mode $\hat{U}_H^L = \sum_{i:\chi_i=H} u_i^L/x_i$ by accounting for c_j^H instead of c_j^L:

$$\hat{U}_H^L = \sum_{i:\chi_i=H} u_i^L/x_i \tag{6.14}$$

$$\leq u_j^H/x_j + \sum_{\substack{i:\chi_i=H \\ i\neq j}} u_i^L/x_i = \tilde{U}_H^L \tag{6.15}$$

If we replace $\hat{U}_H^L$ with the increased virtual utilization $\tilde{U}_H^L$ in Equation (2.57), task j can use its high criticality execution time budget in low criticality mode. To allow any n_H high criticality tasks to overrun, we need to satisfy n_H low criticality mode equations, and the unchanged high criticality mode equation Equation (2.58):

Proposition 18 (EDF-NUVD-SE schedulability)**.** *Let $\mathbb{T}$ be a dual-criticality implicit deadline task set and let $0 < x_i < 1$, for each high criticality task. If*

$$\forall j \quad U_L^L + u_j^H/x_j + \sum_{\substack{i:\chi_i=H \\ i\neq j}} u_i^L/x_i \leq 1 \tag{6.16}$$

$$\sum_{i:\chi_i=H} u_i^H/(1-x_i) \leq 1 \tag{6.17}$$

with mode switch from low- to high criticality mode at the second overrun time, then $\mathbb{T}$ is schedulable by EDF-NUVD-SE.

Proof. If Equation (2.16) holds, then $u_j^H/x_j \geq u_j^L/x_j$ and therefore $\tilde{U}_H^L \geq \hat{U}_H^L$. No deadline is missed by EDF-NUVD-SE in low criticality mode until the first overrun at t^* if Equation (6.16) holds. The first overrun stems from a job J_{ij} of a high criticality task τ_i that is active at t^* with $\gamma_{ij} > c_j^L$. Since we reserved $u_i^H/x_i = c_i^H/(x_i d_i)$ and $\gamma_{ij} \leq c_i^H$ for any task, no deadline is missed until the second overrun time at $t^{\circledast}$.

Consider another job J_{kl} active at $t^{\circledast}$ which arrives at α_{kl} and has a absolute deadline D_{kl} and virtual absolute deadline $\hat{D}_{kl}$. Prior to $t^{\circledast}$, J_{kl} is EDF-scheduled

according to its virtual absolute deadline. Since J_{kl} is active at $t^{\circledast}$ the virtual absolute deadline is at or later than the second overrun time $\hat{D}_{kl} \geq t^{\circledast}$. Therefore the duration between absolute deadline and virtual absolute deadline is the minimum duration between absolute deadline and second overrun time:

$$D_{kl} - t^{\circledast} \geq D_{kl} - \hat{D}_{kl} \tag{6.18}$$

In the worst case each high criticality task has an active job with execution time equal to execution time budget in high criticality mode, and the execution needs to takes place during $D_{kl} - \hat{D}_{kl}$. Regarding the task set after the second overrun time as a single-criticality task set under the worst case assumption, results in Equation (6.17). If the worst case utilization is below the supply of one, this single-criticality task set is schedulable. Hence, EDF-NUVD-SE can schedule $\mathbb{T}$. □

6.3.2 Single error extension of EDF-IVD

Identical to EDF-NUVD-SE we reserve additional time in low criticality mode to tolerate a single overrun. We increase the virtual utilization of high criticality tasks in low criticality mode from $\hat{U}_H^L$ to $\tilde{U}_H^L$ by accounting for c_j^H instead of c_j^L as in Equation (6.15).

By replacing $\hat{U}_H^L$ with $\tilde{U}_H^L$ in Equation (6.1), task j can use its high criticality execution time budget in low criticality mode. Because it is unknown which task overruns, we need to consider that any of n_H high criticality tasks can overrun. This increases the number of low criticality mode equations to n_H:

Proposition 19 (EDF-IVD-SE schedulability)**.** *Let $\mathbb{T}$ be a dual-criticality implicit deadline task set according to Equation* (2.16)*, and let $0 < x_i < 1$, for each high criticality task. If*

$$\forall j \quad U_L^L + u_j^H/x_j + \sum_{\substack{i:\chi_i=H \\ i\neq j}} u_i^L/x_i \leq 1 \tag{6.19}$$

$$\sum_{i:\chi_i=H} u_i^H/(1 - x_i + u_i^L) \leq 1 \tag{6.20}$$

with mode switch from low- to high criticality mode at the second overrun time, then $\mathbb{T}$ is schedulable by EDF-IVD-SE.

Proof. As $u_j^H/x_j \geq u_j^L/x_j$ the increased virtual utilization of high criticality tasks in low criticality mode is larger or equal than their virtual utilization $\tilde{U}_H^L \geq \hat{U}_H^L$. If Equation (6.19) holds no deadline is missed by EDF-IVD-SE in low criticality mode until the first overrun at t^*. The first overrunning job J_{ij} is from a high criticality task τ_i that is active at t^* with $\gamma_{ij} > c_j^L$. Since we reserved $u_i^H/x_i = c_i^H/(x_i d_i)$ and $\gamma_{ij} \leq c_i^H$ for any task, no deadline is missed until the second overrun time at $t^{\circledast}$.

Consider another job J_{kl} active at $t^{\circledast}$ which arrives at α_{kl} and has a absolute deadline D_{kl} and virtual absolute deadline $\hat{D}_{kl}$. Prior to $t^{\circledast}$, J_{kl} is EDF-scheduled according to its virtual absolute deadline. Since J_{kl} is active at $t^{\circledast}$ the virtual absolute deadline is at or later than the second overrun time $\hat{D}_{kl} \geq t^{\circledast}$.

As in Proposition 16, the worst case according to Lemma 15 is where $t^{\circledast}$ is at $\hat{D}_{kl} - c_k^L$:

$$u_{\text{wc}} = \frac{c_k^H}{D_{kl} - \hat{D}_{kl} + c_k^L} = \frac{u_k^H}{1 - x_k + u_k^L} \tag{6.21}$$

Therefore Equation (6.20) considers that each task in the task set has an active job after the second overrun time as described in the worst case above, which can be successfully EDF-scheduled if the resulting utilization is below the uniprocessor's supply of one. □

6.4 Solving for virtual deadline scales

In this section we solve the schedulability conditions for EDF-NUVD-SE, EDF-IVD, and EDF-IVD-SE to get a solution for the virtual deadline scaling factors. With the virtual deadline scaling factors we can calculate in advance the virtual relative deadlines for all tasks, which are required by the scheduler during operation. Moreover, we are interested in a solution for the virtual deadline scaling factors which allows the maximum amount of low criticality work.

For each approach we formulate an optimization problem where the schedulability conditions are the constraints, and the low criticality work defines the objective function to maximize. For any task set with n_H high criticality tasks the vector of variables is $\boldsymbol{y} = \begin{bmatrix} x_0 & x_1 & \dots & x_{n_H-1} & U_L^L \end{bmatrix}$, and the objective function is $f(\boldsymbol{y}) = U_L^L$.

For EDF-NUVD-SE the schedulability conditions Equations (6.16) and (6.17) are the constraints in our optimization:

Optimization problem 20 (EDF-NUVD-SE).

$$\begin{aligned} \underset{\boldsymbol{y}}{\text{maximize}} \quad & U_L^L \\ \text{subject to} \quad \forall j \quad & 1 - U_L^L - u_j^H/x_j - \sum_{\substack{i:\chi_i=H \\ i\neq j}} u_i^L/x_i \geq 0 \\ & 1 - \sum_{i:\chi_i=H} u_i^H/(1-x_i) \geq 0 \end{aligned} \tag{6.22}$$

In EDF-IVD the constraints are Equations (6.1) and (6.2). Note that EDF-IVD requires fewer constraints than EDF-NUVD-SE or EDF-IVD-SE, as the low criticality schedulability condition is the same for all tasks:

Optimization problem 21 (EDF-IVD).

$$\begin{aligned} \underset{\boldsymbol{y}}{\text{maximize}} \quad & U_L^L \\ \text{subject to} \quad & 1 - U_L^L - \sum_{i:\chi_i=H} u_i^L/x_i \geq 0 \\ & 1 - \sum_{i:\chi_i=H} u_i^H/(1-x_i+u_i^L) \geq 0 \end{aligned} \tag{6.23}$$

Finally EDF-IVD-SE with its schedulability conditions Equations (6.1) and (6.2) results in the following optimization problem:

Optimization problem 22 (EDF-IVD-SE).

$$\begin{aligned} \underset{\boldsymbol{y}}{\text{maximize}} \quad & U_L^L \\ \text{subject to} \quad \forall j \quad & 1 - U_L^L - u_j^H / x_j - \sum_{\substack{i:\chi_i=H \\ i\neq j}} u_i^L / x_i \quad \geq 0 \\ & 1 - \sum_{i:\chi_i=H} u_i^H / (1 - x_i + u_i^L) \quad \geq 0 \end{aligned} \tag{6.24}$$

Both Optimization problems 20 and 22 have $n_H + 1$ nonlinear inequality constraints and a scalar objective function. Note Optimization problem 21 has two nonlinear inequality constraints and a scalar objective function, and all optimization problems are solvable by NLP.

We use SLSQP [197, 198] to solve Optimization problems 20 to 22, as all functions are twice continuously differentiable. On success, the resulting virtual deadline scaling factors and maximum utilization of low criticality tasks can be used to schedule the task set.

6.5 System operation

During system operation, our mode-switched EDF scheduler is in one of three modes, as shown in Figure 3.6: *1)* initial low criticality mode; *2)* intermediate single error mode; and *3)* high criticality mode. Modes are switched as explained in Section 3.3, summarized in a event-based description in Algorithm 8.

6.6 Case study

To exemplify the usefulness of EDF-IVD-SE we investigate a flight management system use case [216] shown in Table 6.1. While the task set is worst-case schedulable with EDF, it is not schedulable with EDF-Allowance: A worst-case reservation results in an utilization of $U = {}^{1993}/_{2000} < 1$, with nearly no static slack for any allowance.

But with EDF-IVD-SE the task set is schedulable: Solving the optimization problem, we find a solution with virtual deadline scaling factors as shown in Table 6.2 and maximum supported utilization in low criticality mode of ≈ 0.59. The task set's utilization in low criticality mode is 0.62, which guides us to lower the task set's utilization by ≈ 0.03. Depending on the application, it might be sensible to remove low criticality tasks, extend their period, or reduce their execution time budget. For this example, we reduce the execution time budget of task 9, 10 and 11, as shown in Table 6.3. Even the adjusted task set is not schedulable by EDF-Allowance, but successfully schedulable with EDF-IVD-SE.

Algorithm 8 Event-based description of mode-switched EDF scheduler used in EDF-IVD-SE. The scheduler operates in an endless loop, spanning line 29 to 44, to react to a stream of events e. Possible events are the arrival of a new job in the priority queue, the completion of the job currently running on the CPU, or the detection of a execution time budget overrun from a job of a high criticality task. Overruns are handled first, in line 33 to 36, by increasing the criticality mode of the system if it is not yet in high criticality mode. The response to other events is mode-dependent and handled in line 37 to 43. In low- and single error mode, all jobs can access the CPU according to their deadlines. In high criticality mode, jobs from tasks of low criticality are dropped, and only remaining jobs from high criticality tasks can access the CPU.

Require: Scheduling event e
Require: Jobs J_{ij} in priority queue sorted by earliest deadline
Ensure: Access to CPU for all jobs

1: **procedure** FULLSERVICE(Event e, Job J^Q, Job J^R)
2: **if** e = new job in queue **then**
3: **if** deadline of J^Q < deadline of J^R **then**
4: Preempt and move J^R to priority queue
5: Run job J^Q on CPU
6: **end if**
7: **else if** e = current job finishes **then**
8: Run job J^Q on CPU
9: **end if**
10: **end procedure**
11: **procedure** REDUCEDSERVICE(Event e, Job J^Q, Job J^R)
12: $\chi^Q \leftarrow$ LOOKUPCRITICALITY(J^Q)
13: **if** e = new job in queue **then**
14: **if** $\chi^Q = L$ **then**
15: drop J^Q
16: **else if** deadline of J^Q < deadline of J^R **then**
17: Preempt and move J^R to priority queue
18: Run job J^Q on CPU
19: **end if**
20: **else if** e = current job finishes **then**
21: **if** $\chi^Q = H$ **then**
22: Run job J^Q on CPU
23: **else**
24: drop J^Q
25: **end if**
26: **end if**
27: **end procedure**
28: **loop**
29: $M \leftarrow$ GETCRITICALITYMODE
30: $J^R \leftarrow$ Job currently running on CPU
31: $J^Q \leftarrow$ Job from priority queue
32: $e \leftarrow$ Next event
33: **if** e = detect overrun $\wedge$ $(M \neq H)$ **then**
34: Increase system criticality mode
35: **continue**
36: **end if**
37: **if** $M = L$ **then** ▷ Low criticality mode
38: FULLSERVICE(e, J^Q, J^R)
39: **else if** $M = S$ **then** ▷ Single error mode
40: FULLSERVICE(e, J^Q, J^R)
41: **else** ▷ High criticality mode
42: REDUCEDSERVICE(e, J^Q, J^R)
43: **end if**
44: **end loop**

Table 6.1: Flight management system use case [216]

Task i	p_i	d_i	c_i^L	c_i^H
1	5000	5000	10	20
2	200	200	10	20
3	1000	1000	10	20
4	1600	1600	10	20
5	100	100	10	20
6	1000	1000	10	20
7	1000	1000	10	20
8	1000	1000	20	–
9	1000	1000	200	–
10	1000	1000	200	–
11	1000	1000	200	–

Table 6.2: EDF-IVD-SE virtual deadline scales for task set in Table 6.1

Task i	x_i
1	0.603 009 38
2	0.631 890 57
3	0.607 817 98
4	0.605 562 71
5	0.749 381 33
6	0.607 817 98
7	0.607 817 98

Table 6.3: Adjusted flight management system use case

Task i	p_i	d_i	c_i^L	c_i^H
1	5000	5000	10	20
2	200	200	10	20
3	1000	1000	10	20
4	1600	1600	10	20
5	100	100	10	20
6	1000	1000	10	20
7	1000	1000	10	20
8	1000	1000	20	–
9	1000	1000	190	–
10	1000	1000	190	–
11	1000	1000	190	–

6.7 Experiments

In this section we quantify the amount of schedulable task systems with EDF-NUVD-SE, EDF-IVD-SE, and the benefit in QoS. To investigate the amount of schedulable task sets we generate random task sets and solve the corresponding optimization problem. We report the number of schedulable task sets over the total number of task sets, or *acceptance rate*, over increasing utilization in low criticality mode.

For QoS, we evaluate the additional time the system is operational after the first overrun time with `Thready` [199] by simulating a large set of random task sets.

6.7.1 Acceptance rate of `UUnifast` random task systems for different utilizations

Reporting the acceptance rate with random task sets requires to apply the schedulability check, and in case of EDF-NUVD-SE, EDF-IVD, and EDF-IVD-SE this includes solving the optimization problems presented in Section 6.4.

For our investigation we resort to random task sets, generated with the `UUnifast` algorithm [159], additional pessimism [161], and a mostly EDF-VD schedulable parameterization [189]: Periods are uniformly drawn between $p_l = 50$ and $p_u = 200$, and pessimism is uniformly drawn between $z_l = 1$ and $z_u = 2$. For each utilization in low criticality mode, we calculate the acceptance rate from 1024 task sets. Moreover, we compare the acceptance rates for a parameterization typically found in automotive and avionics systems with periods between $p_l = 25$ and $p_u = 1000$ [160].

In general, a higher acceptance rate is better, because it indicates a higher chance to accept a task set as schedulable. As the utilization in low criticality mode U_L increases, the difficulty to find acceptable virtual deadline scaling factors increases as well.

We compare the acceptance rate in Figure 6.6 for approaches without single error tolerance. All approaches decline in acceptance rate with increasing utilization in low criticality mode. It stands out that EDF-VD dominates both EDF-NUVD and EDF-IVD despite having only a single deadline scaling parameter. By intuition, approaches with individual deadline scaling parameters should be at least as powerful as uniform deadline scales. But the proof for EDF-VD feasibility [147] results in tighter bounds for the worst case, which increases the chance to deem a task set schedulable. Nevertheless, the acceptance rate of EDF-IVD is at least as good or better than the acceptance rate of EDF-NUVD. For the avionic- and automotive-like task sets in Figure 6.8 the trends are similar, although the absolute acceptance rates are lower.

Extending all three approaches to tolerate a single error drastically changes the acceptance rate as seen in Figure 6.7. For utilization up to 0.65 EDF-IVD-SE provides the best acceptance rate. Contrary, the uniform deadline scale of EDF-VD-SE provides less room for the numerical solver to find a solution. Similar to the approaches without single error tolerance, the acceptance rate trends for the avionic- and automotive-like task sets in Figure 6.9 are alike, while absolute acceptance rates are lower. Nevertheless, in terms of acceptance rate

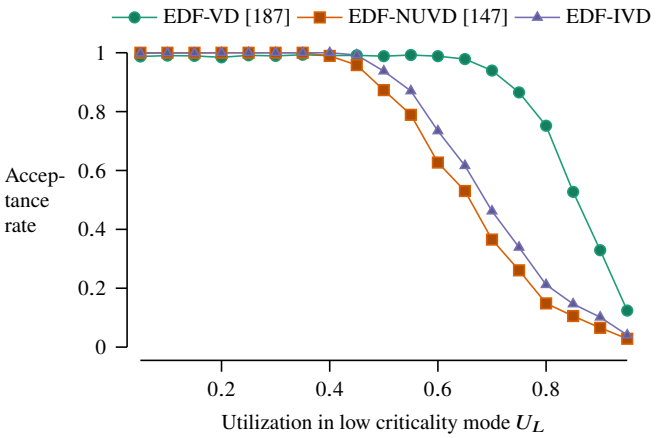


Figure 6.6: Acceptance rate for `UUnifast` random task system. A higher acceptance rate is better. With increasing utilization on low criticality mode U_L the difficulty to find acceptable deadline scales increases.

all approaches are sensitive to the composition of their task sets. Therefore, detailed descriptions about random task set generation [161] are a necessity for comparable results.

It is fundamental to note that the analytical solution for EDF-VD is optimal if errors are *not* considered, but with errors the analytical solution is not applicable anymore. Moreover, with just a single free parameter, the numerical solver has less opportunities to find a solution for EDF-VD-SE, compared to EDF-NUVD-SE and EDF-IVD-SE, resulting in reduced acceptance rates.

Comparing the acceptance rates of EDF-IVD to EDF-IVD-SE in Figure 6.10 for the same task sets highlights the cost of tolerating a single error in terms of acceptance rate. With a maximum cost of ≈ 0.146 EDF-IVD lends itself for single error extension.

In the same way we can compare the acceptance rates of EDF to EDF-Allowance. The maximum cost in terms of acceptance rate is ≈ 0.33, as shown in Figure 6.11, instead of ≈ 0.146 from our approach.

Moreover, utilizations of larger than 0.55 are increasingly difficult for EDF-Allowance, and the start of a decline in acceptance rate compared to EDF-IVD-SE. With up to ≈ 1.56 better acceptance rate EDF-IVD-SE can more likely schedule highly loaded task sets.

6.7.2 Quality of service comparison by mode switch time

Both EDF-IVD-SE and EDF-NUVD-SE are designed to tolerate a single overrun without discarding all low criticality tasks, prolonging the time when jobs from low criticality tasks can execute until the second overrun. This prolonged time results in additional service for low criticality tasks. Therefore, we define the QoS as the ratio of second overrun time to first overrun time: $t^{\circledast}/t^{*}$.

We investigate the QoS of EDF-IVD-SE and EDF-NUVD-SE by simulation with `Thready`, which supports EDF scheduling simulations of mixed-criticality task sets with fixed overrun probabilities. We select EDF-IVD-SE and EDF-NUVD-SE schedulable dual-criticality task sets which we generate with the `UUnifast`

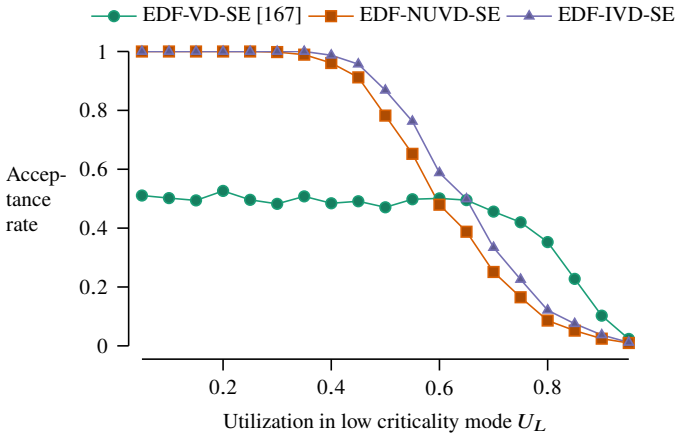


Figure 6.7: Acceptance rate of random task sets for approaches with single error tolerance.

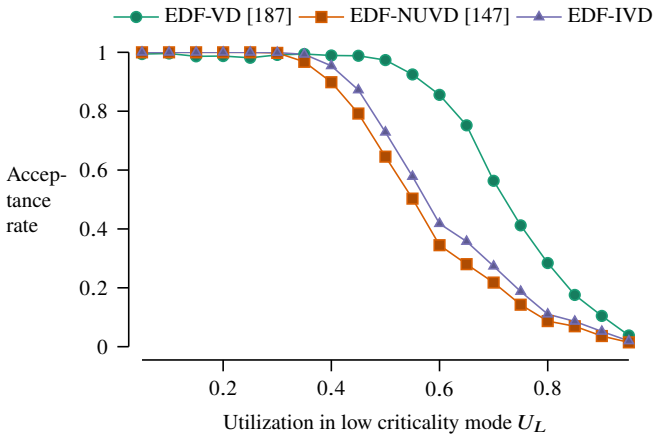


Figure 6.8: Acceptance rate of random automotive-like task sets for approaches without single error tolerance.

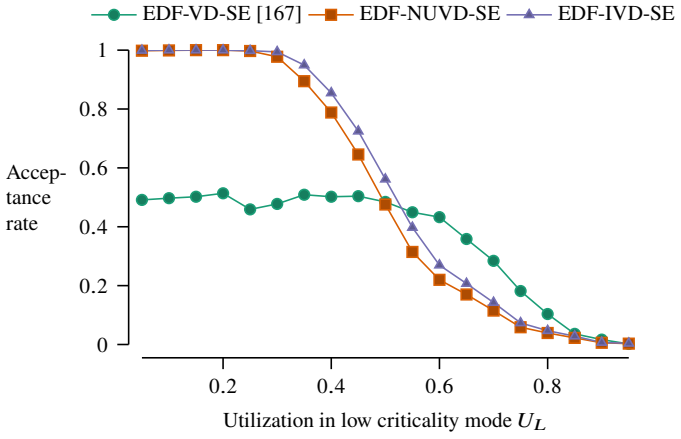


Figure 6.9: Acceptance rate of random automotive-like task sets for approaches with single error tolerance.

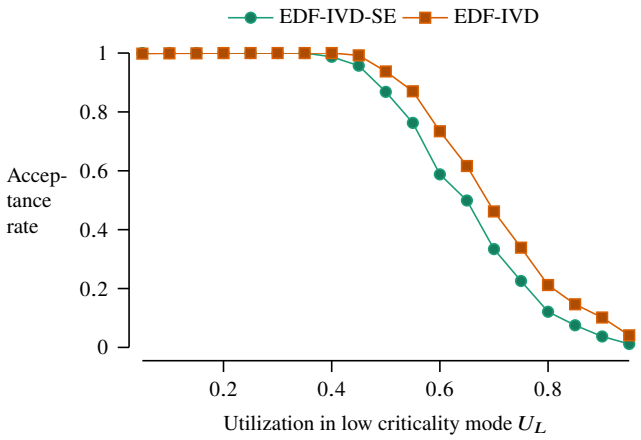


Figure 6.10: EDF-IVD-SE lends itself for single error extension. Plot shows the acceptance rate for `UUnifast` random task sets with EDF-VD-friendly parameterization [189]. The acceptance rate of EDF-IVD-SE is only slightly reduced compared to EDF-IVD. Therefore EDF-IVD-SE makes error tolerance affordable in most cases.

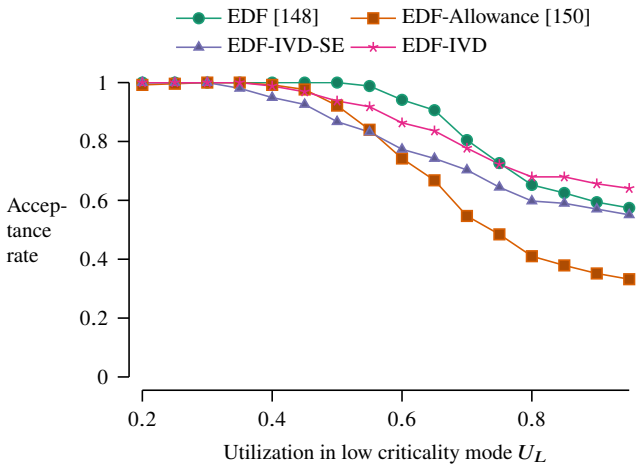


Figure 6.11: Acceptance rate of random task sets. The acceptance rate of EDF-IVD-SE is better than the allowance approach.

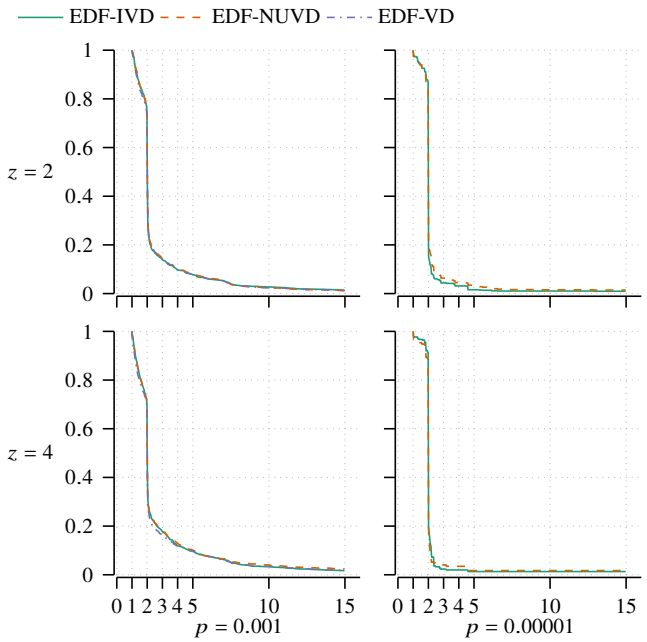


Figure 6.12: Complementary cumulative distribution function (CCDF), or survival function, of EDF-IVD-SE and EDF-NUVD-SE QoS for error probabilitys $p = 0.001$ and $p = 0.0001$. Rows differentiate pessimism, and columns differentiate error probability. In each plot, the ordinate abscissa Each plot shows the probability (ordinate) of a system to survive until $t^{\circledast}/t^{*}$ (abscissa).

algorithm and generation parameters as described in Section 6.7.1. The random task sets are simulated with millisecond time step resolution for one hour.

While all approaches can tolerate the first overrun deliberately, it is still interesting to quantify the probabilities to achieve at least a specific QoS.

In Figure 6.12 with an error probability of $p = 10^{-3}$ and pessimism of two EDF-NUVD-SE achieves at least a QoS of 1.82 with a probability of 0.8, or a QoS of 2.01 with a probability of 0.46. For the same scenario, EDF-IVD-SE achieves at least a QoS of 2.01 with a probability of 0.45, or a QoS of 1.85 with a probability of 0.8. Similar results are achieved for a pessimism of four, and error probability of 10^{-5}, with a QoS of 1.93 at a probability of 0.92. Therefore both EDF-IVD-SE and EDF-NUVD-SE double on average the QoS for low criticality tasks compared

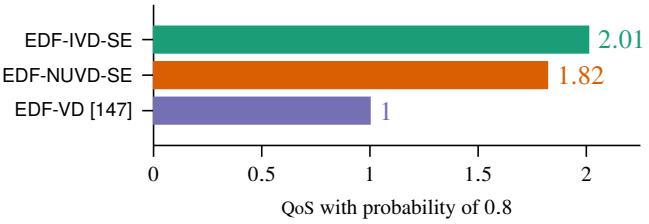


Figure 6.13: Achievable QoS for EDF-IVD-SE and EDF-NUVD-SE with probability of 0.8 compared to baseline EDF-VD. Both EDF-IVD-SE and EDF-NUVD-SE can tolerate a single error, and therefore double on average the QoS for low criticality tasks.

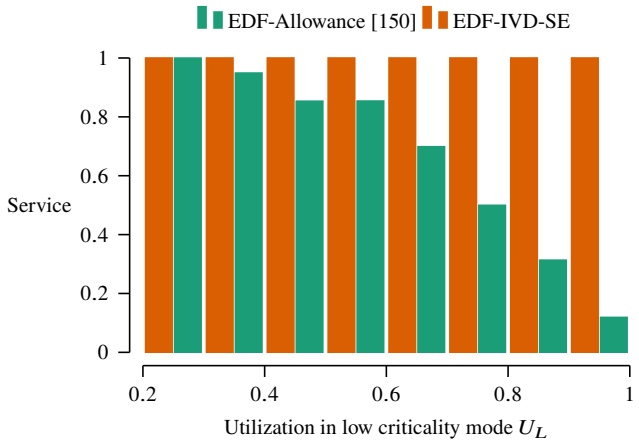


Figure 6.14: Time without deadline miss normalized to mission duration of EDF-IVD-SE and EDF-Allowance scheduled random task sets. While EDF-IVD-SE guarantees that the system survives the whole mission duration, EDF-Allowance can misses a deadline under bad circumstances. The likelihood that EDF-Allowance misses a deadline increases with utilization, and with the environment's error probability. Contrary, the service of EDF-IVD-SE is independent of the environment's error probability.

to baseline EDF-VD, as visualized in Figure 6.13.

From a global view, both EDF-NUVD-SE and EDF-IVD-SE provide the same overrun tolerance by design, and are similar in their capabilities beyond the first overrun. Nevertheless, EDF-IVD-SE is preferable, as it is more likely to find valid virtual deadline scaling factors for a given task set.

Furthermore, if we compare EDF-IVD-SE with EDF-Allowance, we immediately see that EDF-Allowance has no service guarantee, which results in rising likelihood of mission failures with rising utilization as shown in Figure 6.14. Contrary, EDF-IVD-SE has a service guarantee, which guarantees mission success.

6.8 Conclusion

Fault-tolerant real-time systems in emerging applications are not allowed to fail, but their budgets for error mitigation are minimal. We introduce and prove the feasibility of EDF-IVD-SE, our verification friendly mixed-criticality approach, which provides single error tolerance by design and guarantees that the most critical system functions are always operational.

By extensive simulation experiments we showed that EDF-IVD-SE has up to 1.56 better acceptance rate compared to similar state-of-the-art approaches, and how EDF-IVD-SE can provide full service to low criticality tasks for 1.93 times the original duration. Affordable and powerful, EDF-IVD-SE is a suitable mixed-criticality scheduling approach for fault-tolerant real-time systems in emerging critical applications.

7 Conclusion and outlook

Modern societies increasingly transfer responsibilities to computer systems which are integrated with the environment. Due to their responsibilities, such CPSs are required to be dependable, that is, they need to deliver their expected service, on time, even if faults and errors interfere with their operation. Moreover, their energy budget is very limited compared to traditional dependable computer systems, and therefore requires to resort to low-power design techniques. But the integration of dependable- and low-power design for CPSs results in conflicting requirements: How to guarantee dependability and real-time properties without spending excessive energy? Our solution, design for graceful degradation, combines temporal redundancy with real-time guarantees into a fault-tolerance strategy which is suitable for COTS-based CPSs. This allows to benefit from the reduced energy consumption and increased performance of COTS components in our CPS design approach, further empowering the integration of functions with different criticality with verifiable QoS guarantees.

7.1 Conclusion

By our taxonomy of fault-tolerant systems our approach makes use of concurrent error detection, compensation, and reinitialization to implement a system recovery fault tolerance strategy. This allows us to avoid service failures despite the presence of faults: We identify the presence of errors, compensate their consequences, and update our system's configuration accordingly. For error detection we proposed two approaches: a software- and a hardware-based approach. The software approach resorts to duplicated instructions and comparisons to detect errors, while the hardware approach monitors signal timings for their correctness. If we detect an error during the execution of a job, we compensate the consequences by rerunning the job. To account for the additional runtime, we reconfigure our system by switching to the next higher criticality mode. This mode switch ensures that deadlines of high criticality jobs are not violated.

We developed an fault- and error model, starting with the physics of radiation in semiconductors, and generalized the error model to cover errors from other domains but with the same behavior. While detailed models of fault physics help in understanding the cause of faults, they are unwieldy for practical design tasks beyond TCAD simulations. Therefore we derived analog- and digital circuit models: A double exponential current pulse is used to model particle strikes at circuit nodes, and its current-induced voltage transients were investigated by circuit simulations to motivate our digital circuit model of SETs and SEUs.

The integration of our fault-tolerance strategy with real-time scheduling gave rise to mixed-criticality scheduling problems, which we solved with our scheduling algorithms EDF-VD-SE and EDF-IVD-SE. In our fault-tolerance strategy,

errors are compensated by resorting to temporal redundancy. The additional time required to deal with them needs to be considered, particularly with regard to job deadlines in real-time systems. Accordingly we considered the impact of error compensation on the required execution time of jobs in our mixed-criticality model by adjusted execution time budgets. But traditional mixed-criticality scheduling approaches, stemming from their certification origins, assume errors to be unlikely. Once errors are expected during operation, QoS for low criticality tasks in traditional mixed-criticality scheduling approaches is low, because they are forever discarded during the first error compensation. Thus we introduced EDF-VD-SE, which delays or prevents the removal of low criticality tasks, doubling on average the QoS. We prove that EDF-VD-SE never misses a absolute deadline of high-criticality jobs, and formulate the search for suitable virtual relative deadlines as an optimization problem, which we solved with SLSQP.

With EDF-IVD-SE we developed a scheduling approach with similar error compensation capabilities and improved acceptance rates due to individual virtual deadline scaling parameters. We proved the feasibility of EDF-IVD-SE, and derived the accompanying optimization problem. Solving the optimization problems of EDF-VD-SE or EDF-IVD-SE does not only provide a solution for the deadline scaling parameters, but also maximizes the possible utilization of low criticality tasks in low criticality mode. This guides a system designer in selecting additional functionalities for integration while ensuring schedulability.

To implement our CPSs we resorted to MPSOCs and presented a heuristic algorithm to convert the arising multiprocessor scheduling problem into multiple uniprocessor scheduling problems. We developed a COTS-friendly EDS to bring concurrent error detection to FPGA-based MPSOCs, and showed how a modern RTOS can implement our approach without modification of kernel sources. Moreover, the RTOS is not required to keep track of dynamic slack, which is beneficial for deeply embedded systems with limited computation resources.

Our approach is certification friendly, as it is a static, design-time approach. We degrade gracefully by tolerating single errors, which doubles on average the QoS for low-criticality tasks. Furthermore, no assumptions about error probabilities are required.

Together, our approach combines desirable properties for CPS designs based on commodity hardware: The CPS is liberated from spending additional CPU cycles on dynamic slack management or complicated scheduling decisions, the designer is relieved from distributing tasks to processing elements and guided in selecting functions of lower criticality, while modifications of COTS-components are minimized. Moreover, without assumptions about available hardware features the selection of components is eased, allowing application- and cost optimal choices.

7.2 Outlook

Our approach considers dependable systems with limited energy budget for real-time applications. From this background, several decisions have been made to make our approach certification friendly. In line with this goal, the following directions for future work are foreseen:

Adaption for different task models

Our current task model, which is part of our mixed-criticality model, considers tasks with implicit deadlines, where relative deadlines are equal to periods. Depending on the application, other task models, where tasks can have a relative deadline smaller than their period, can be more appropriate. Adapting our approach to different task models would require a new proof of schedulability.

Idle tick detection

Currently the mode change in our system is unidirectional, and does not consider the actual load of the system. This results in pessimistic decisions regarding the mode change: Instead of reducing the system's criticality mode once the system is idle, the mode is not changed. With idle tick detection, the system's criticality mode could be decreased, which would allow to compensate further errors, resulting in further QoS improvements for low criticality tasks.

7.3 Summary

In summary, our approach solves a major design challenge for CPSs, and enables the design of low-power and fault-tolerant mixed-criticality real-time systems. These systems are in strong demand, from emerging healthcare applications to process- and environmental control, and facilitating their widespread adoption by reducing the barriers to build them is beneficial for society.

8 Glossary

Table 8.1: Notation

	Name	Description
σ	SEU cross-section	Number of events per unit fluence.
A	allowance	Tolerable time beyond overrun without compromising any deadlines
α	arrival time	Time where job enters the scheduler's queue
Q_{coll}	collected charge	
Q_{crit}	critical charge	
χ	criticality	Required level of assurance against failure
H	high criticality level	
L	low criticality level	
σ	cross section	intrinsic parameter of a chip/circuit that specifies its response to a particle species
D	absolute deadline	Time for a job to finish
$\hat{D}$	virtual absolute deadline	Virtual earlier time for a job to finish
d	relative deadline	Duration for a job to finish
$\hat{d}$	virtual relative deadline	Virtual shorter duration for a job to finish
ρ	density	
e	elementary positive charge	electric charge carried by a single proton or magnitude of the negative electric charge carried by a single electron
Ψ	energy fluence	radiant energy incident on a sphere
$\dot{R}$	energy flux	increment of radiant energy in a time interval
$\dot{\Psi}$	energy-fluence rate	increment of the energy fluence during a time interval
p	error probability	Probability for an error
γ	execution time	Time it takes to finish the computation without interruption

continued. . .

	Name	Description
c	execution time budget	Maximum execution time for a job which does not trigger a mode change
Φ	fluence	quotient of the number of particles incident on a sphere
$\dot{\Phi}$	fluence rate	increment of fluence over time interval
$\dot{N}$	flux	increment of the particle number over time interval
J	job	Computation
t^{zZz}	last idle point	latest time where a job from a high criticality task did not execute in low criticality mode
S/ρ	mass stopping power	
n_H	number of high criticality tasks	Number of tasks in task set with high criticality
t^*	first overrun time	First time where a job from a high criticality task exceeds its low criticality execution time budget
$t^{\circledast}$	second overrun time	Second time where a job from a high criticality task exceeds its low criticality execution time budget
p	period	Minimum interarrival time between two jobs of the same task
z	pessimism	Pessimism in estimating the task's WCET
θ_N	probability of majority of input lines wrong	
ϵ	probability of majority organ malfunction	
$\theta_{i,N}$	probability of up to i of N input lines wrong	
η_i	probability of wrong signal on input line, upper bound	
ρ	probability of wrong signal on majority gate output line	
R	radiant energy	energy of emitted, transferred, or received particles
L_Δ	restricted linear electronic stopping power	

continued. . .

	Name	Description
x	virtual deadline scaling factor	Factor multiplied with relative deadline to get virtual relative deadline
σ	supply	Computation resources provisioned by CPU
τ	task	Reoccuring computation
u	task utilization	Execution time budget over period
$\mathbb{T}$	task set	Independent reoccuring computations sharing the same processor
U	utilization	Sum of all task's execution time budget over period
$\hat{U}$	virtual utilization	Sum of all task's execution time budget over virtual relative deadline
$\tilde{U}$	increased virtual utilization	Sum of all task's execution time budget over virtual relative deadline
W^J	job progression window	Window of possible progressions in executing a job

Table 8.2: Abbreviations

ASIC	Application-specific integrated circuit
BIST	Built-in self test
CASE	Computer-aided software engineering
CCDF	Complementary cumulative distribution function
CDF	Cumulative distribution function
CLEAR	Cross-layer resilience evaluation framework
CMOS	Complementary metal-oxide-semiconductor
COTS	Commercial off-the-shelf
CPS	Cyber-physical system
CPU	Central processing unit
CSV	Comma separated values
DAG	Directed acyclic graph
DBF	Demand bound function
DMR	Dual modular redundancy
DSL	Domain-specific language
ECSS	European Cooperation for Space Standardization
EDC	Error detection code
EDF	Earliest deadline first
EDF-Allowance	Earliest deadline first with allowance
EDF-IVD	Earliest deadline first with improved virtual deadlines
EDF-IVD-SE	Earliest deadline first with improved virtual deadlines for single errors

continued. . .

EDF-NUVD	Earliest deadline first with non-uniform virtual deadlines
EDF-NUVD-SE	Earliest deadline first with non-uniform virtual deadlines for single errors
EDF-VD	Earliest deadline first with virtual deadlines
EDF-VD-SE	Earliest deadline first with virtual deadlines for single errors
EDS	Error detection sequential
FIFO	First in first out
FinFET	Fin field-effect transistor
FIT	Failure in time
FO	Fail operational
FPGA	Field programmable gate array
GCR	Galactic cosmic ray
GNSS	Global navigation satellite system
GNU	Gnu's not unix
IC	Integrated circuit
ICRU	International commission on radiation units and measurements
IoT	Internet-of-things
IP	Intellectual property
ISA	Instruction set architecture
JSON	Javascript object notation
L1	Level 1
LET	Linear energy transfer
LOC	Line of code
LPT	Longest-processing-time-first
MBU	Multiple-bit upset
MOSFET	Metal-oxide-semiconductor field-effect transistor
MPSOC	Multiprocessor system-on-chip
MTBF	Mean time between failures
NAND	Logical non-conjunction
NLP	Nonlinear programming
NoC	Network-on-chip
nZDC	Near zero silent data corruption
PDF	Probability density function
PDK	Process design kit
QoS	Quality of service
QSAT	Quantified Boolean satisfiability
RMT	Redundant multithreading
RT	Register transfer
RTOS	Real-time operating system
SAT	Boolean satisfiability
SBU	Single-bit upset
SEE	Single-event effect
SEFI	Single-event functional interrupt

continued. . .

SEL	Single-event latch-up
SER	Soft error rate
SET	Single-event transient
SEU	Single-event upset
SIHFT	Software-implemented hardware fault tolerance
SLSQP	Sequential least squares programming
SOC	System-on-chip
SPCD	Selective procedure call duplication
SWaP	Size, weight, and power
TCAD	Technology computer aided design
TCL	Tool command language
TID	Total ionizing dose
TMR	Triple modular redundancy
TRIUMF	Tri-university meson facility
TRLA	Temporal redundancy latch-based architecture
UAV	Unmanned aerial vehicle
UML	Unified modeling language
WCET	Worst-case execution time
XNOR	Exclusive non-disjunction
XOR	Exclusive disjunction

acceptance rate ratio of schedulable tasksets

accuracy proximity of measurement results to the true value (See: "precision")

active fault fault that causes an error (See: "error", "fault")

alpha particle two protons and two neutrons bound together similar to a helium nucleus (See: "atomic nucleus")

application (See: "computer program")

assembly instruction (See: "instruction")

atomic nucleus very dense central region of an atom

availability readiness of a system for correct service (See: "delivered service", "dependability")

call graph represents the calling behavior of a computer program (See: "procedure", "function", "program")

compensation error handling approach that leverages redundancy to mask errors (See: "error handling", "error")

computer program sequence of instructions that can be interpreted and executed by a computer (See: "program", "instruction", "procedure")

confidentiality absence of unauthorized modifications and repairs (See: "dependability")

degraded mode only offers a subset of needed service (See: "delivered service")

delivered service behavior of a system as it is perceived by its users, or sequence of the system's external state (See: "user", "system behavior")

dependability the ability to avoid service failures that are more frequent and more severe than is acceptable (See: "delivered service", "service failure")

dormant fault fault that does not cause an error (See: "error", "fault")

dose measure of energy deposited in matter by ionizing radiation per unit mass

electronvolt the amount of kinetic energy gained or lost by a single electron accelerating from rest through an electric potential difference of one volt in vacuum

entity distinct existence as an individual unit, e.g. hardware, software, humans

environment external entities interacting with a given system (See: "entity", "system")

error deviation from correct service state (See: "state", "failure")

error detection identification of present errors (See: "state", "error")

error handling fault tolerance approach which removes errors from the system state by any combination of rollback, rollforward, and compensation (See: "fault tolerance", "error", "compensation")

failure (See: "service failure")

fault cause of an error (See: "error")

fault handling keeps faults from being activated again (See: "error handling", "fault tolerance")

fault tolerance avoiding service failures in the presence of faults (See: "delivered service", "fault")

flip-flop circuit with two stable states

function (impure) callable instruction sequence which returns values (See: "instruction sequence", "procedure")

functional specification describes system function in terms of functionality and performance (See: "system function")

graceful degradation acceptance of a performance decline for an extended operational lifetime (See: "delivered service", "degraded mode", "pattern")

hard error Irreversible change in operation typically associated with permanent damage to one or more elements of a device or circuit.

instruction coded data that is interpreted by a computer as a command to perform an operation (See: "instruction sequence", "assembly instruction")

instruction sequence sequence of logical related instructions (See: "procedure")

integrity absence of improper system alterations (See: "dependability")

maintainability ability to receive modifications and repairs (See: "dependability")

mission duration Duration where a system is expected to work without failure.

neutron Nucleon with no electric charge.

nucleus (See: "atomic nucleus")

p-n junction boundary between p- and n-type semiconductor material

pattern general repeatable solution to a commonly occuring problem

precision degree to which reproducible measurements under unchanged conditions show the same result (See: "accuracy")

procedure reusable instruction sequence which is part of a larger computer program (See: "instruction sequence", "function")

program (See: "computer program")

reinitialization updating the system's record of faults and configuration during fault handling (See: "fault tolerance", "fault handling")

reliability continuity of correct service (See: "delivered service", "dependability")

routine (See: "procedure")

safety absence of harmful consequences (See: "dependability")

service (See: "delivered service")

service failure event that occurs when the delivered service deviates from correct service, or transition to incorrect service due to an error (See: "delivered service", "error")

service interface part of system's boundary where service delivery takes place (See: "delivered service", "system boundary")

soft error Correctable, erroneous output signal from a latch or memory cell.

state mode or condition of being (See: "total state")

state of the art highest level of development at a given time

subroutine (See: "procedure")

system entity that interacts with other entities and the physical world with its natural phenomena (See: "entity")

system behavior what the system does to implement its system function described by a sequence of states (See: "system function")

system boundary common frontier between system and its environment (See: "system", "environment")

system function what the system is intended to do

system recovery fault tolerance strategy where error handling is followed by fault handling (See: "error handling", "fault tolerance")

system structure enables the system to generate behavior (See: "system behavior")

systematic code encoding where data and check bits are separate (See: "unordered code")

task system (See: "task set")

terrestrial of or relating to the earth or its inhabitants

thread related sequence of instructions or actions within a program that runs at least in part independent of other actions within the program (See: "task", "job")

threshold voltage minimum gate-to-source voltage needed to create conducting path

total state a system's combined state of computation, communication, stored information, interconnection, and physical condition (See: "state", "system")

unordered code encoding where data and check bits are intertwined (See: "systematic code")

user system that receives service from another system

validation assurance that a system meets the needs of its stakeholders (See: "verification")

verification evaluation whether or not a system complies with a regulation or implements its specification (See: "validation")

x-ray high-energy electromagnetic radiation

Bibliography

[1] M. Duranton, K. De Bosschere, C. Gamrat, J. Maebe, H. Munk, and O. Zendra, "The HiPEAC vision 2017," tech. rep., 2017.

[2] E. A. Lee, "Cyber physical systems: Design challenges," in *11th IEEE Int. Symp. Object Oriented Real-Time Distrib. Comput.*, pp. 363–369, 2008.

[3] S. Olson, *Grand Challenges for Engineering: Imperatives, Prospects, and Priorities*. Washington, D.C.: National Academies Press, 2016.

[4] R. R. Rajkumar, I. Lee, L. Sha, and J. Stankovic, "Cyber-physical systems: The next computing revolution," in *Proc. 47th Des. Automat. Conf.*, pp. 731–736, ACM Press, 2010.

[5] G. Hackmann, W. Guo, G. Yan, Z. Sun, C. Lu, and S. Dyke, "Cyber-physical codesign of distributed structural health monitoring with wireless sensor networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, pp. 63–72, Jan. 2014.

[6] Z. Wang, H. Song, D. W. Watkins, K. G. Ong, P. Xue, Q. Yang, and X. Shi, "Cyber-physical systems for water sustainability: Challenges and opportunities," *IEEE Commun. Mag.*, vol. 53, pp. 216–222, May 2015.

[7] M. Wolf and D. Serpanos, "Safety and security in cyber-physical systems and internet-of-things systems," *Proc. IEEE*, vol. 106, pp. 9–20, Jan. 2018.

[8] J. C. Knight, "Issues of software reliability in medical systems," in *Proc. 3rd Annu. IEEE Symp. Comput.-Based Medical Syst.*, pp. 153–160, 1990.

[9] A. Avižienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Trans. Depend. Sec. Comput.*, vol. 1, pp. 11–33, Jan. 2004.

[10] A. Avižienis, "A visit to the jungle of terminology," in *Proc. 47th Annu. IEEE / IFIP Int. Conf. Dependable Syst. Netw. Workshops*, pp. 149–152, 2017.

[11] Y. Pu, C. Shi, G. Samson, D. Park, K. Easton, R. Beraha, A. Newham, M. Lin, V. Rangan, K. Chatha, D. Butterfield, and R. Attar, "A 9-mm2 ultra-low-power highly integrated 28-nm CMOS SoC for internet of things," *IEEE J. Solid-State Circuits*, vol. 53, no. 3, pp. 936–948, 2018.

[12] R. Bertran, P. Bose, D. Brooks, J. Burns, A. Buyuktosunoglu, N. Chandramoorthy, E. Cheng, M. Cochet, S. Eldridge, D. Friedman, H. Jacobson, R. Joshi, S. Mitra, R. Montoye, A. Paidimarri, P. Parida, K. Skadron,

M. Stan, K. Swaminathan, A. Vega, S. Venkataramani, C. Vezyrtzis, G. Y. Wei, J. D. Wellman, and M. Ziegler, "Very low voltage (VLV) design," in *IEEE 35th Int. Conf. Comput. Des.*, pp. 601–604, 2017.

[13] R. G. Dreslinski, M. Wieckowski, D. Blaauw, D. Sylvester, and T. Mudge, "Near-threshold computing: Reclaiming moore's law through energy efficient integrated circuits," *Proc. IEEE*, vol. 98, pp. 253–266, Feb. 2010.

[14] P. Basu, P. Pandey, A. Bal, C. Rajamanikkam, K. Chakraborty, and S. Roy, "TITAN: Uncovering the paradigm shift in security vulnerability at near-threshold computing," *IEEE Trans. Emerg. Topics Comput.*, vol. 6750, no. c, 2018.

[15] V. P. Nelson, "Fault-tolerant computing: Fundamental concepts," *Computer*, vol. 23, no. 7, pp. 19–25, 1990.

[16] V. Narayanan and Y. Xie, "Reliability concerns in embedded system designs," *IEEE Trans. Comput.*, vol. 39, no. 1, pp. 118–120, 2006.

[17] J. P. Loftus, Jr., C. Teixeira, and D. Kirkpatrick, *Space Mission Analysis and Design*, ch. Launch Systems, pp. 719–744. Kluwer, 2005.

[18] M. Cinque, D. Cotroneo, Z. Kalbarczyk, and R. K. Iyer, "How do mobile phones fail? a failure data analysis of symbian OS smart phones," in *37th. Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2007.

[19] L. Ceze, M. D. Hill, and T. F. Wenisch, "Arch2030: A vision of computer architecture research over the next 15 years," tech. rep., Computing Community Consortium Catalyst, 2017.

[20] J. Dean, D. Patterson, and C. Young, "A new golden age in computer architecture: Empowering the machine-learning revolution," *IEEE Micro*, vol. 38, no. 2, pp. 21–29, 2018.

[21] B. Bohnenstiel, A. Stillmaker, J. J. Pimentel, T. Andreas, B. Liu, A. T. Tran, E. Adeagbo, and B. M. Baas, "Kilocore: A 32-nm 1000-processor computational array," *IEEE J. Solid-State Circuits*, vol. 52, no. 4, 2017.

[22] S. Davidson, S. Xie, C. Torng, K. Al-Hawai, A. Rovinski, T. Ajayi, L. Vega, C. Zhao, R. Zhao, and S. Dai, "The celerity open-source 511-core RISC-V tiered accelerator fabric: Fast architectures and design methodologies for fast chips," *IEEE Micro*, vol. 38, no. 2, pp. 30–41, 2018.

[23] D. Bertozzi, L. Benini, and G. De Micheli, "Error control schemes for on-chip communication links: The energy-reliability tradeoff," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 24, no. 6, pp. 818–831, 2005.

[24] B. Shim and N. R. Shanbhag, "Energy-efficient soft error-tolerant digital signal processing," *IEEE Trans. VLSI Syst.*, vol. 14, no. 4, pp. 336–348, 2006.

[25] N. R. Shanbhag, "Reliable and efficient system-on-chip design," *IEEE Trans. Comput.*, vol. 37, no. 3, pp. 42–50, 2004.

[26] F. S. Alghareb, R. A. Ashraf, A. Alzahrani, and R. F. DeMara, "Energy and delay tradeoffs of soft-error masking for 16-nm finfet logic paths: Survey and impact of process variation in the near-threshold region," *IEEE Tran. Circ. Syst. Express Briefs*, vol. 64, no. 6, 2017.

[27] F. S. Alghareb, M. Lin, and R. F. DeMara, "Soft error effect tolerant temporal self-voting checkers: Energy vs. resilience tradeoffs," in *IEEE Annu. Symp. VLSI*, pp. 571–576, 2016.

[28] E. Cheng, S. Mirkhani, L. G. Szafaryn, C.-Y. Cher, H. Cho, K. Skadron, M. R. Stan, K. Lilja, J. A. Abraham, P. Bose, and S. Mitra, "CLEAR: Cross-layer exploration for architecting resilience: Combining hardware and software techniques to tolerate soft errors in processor cores," in *Proc. 53rd Annu. Des. Autom. Conf.*, ACM, June 2016.

[29] S. Schnell, "Ten simple rules for a computational biologist's laboratory notebook," *PLOS Comp. Bio.*, vol. 11, no. 9, pp. 1–5, 2015.

[30] T. J. Vision, "Open data and the social contract of scientific publishing," *BioScience*, vol. 60, no. 5, pp. 330–331, 2010.

[31] Institute of Electrical and Electronics Engineers, "IEEE code of ethics," tech. rep., IEEE, 2018.

[32] S. Miller and M. J. Selgelid, "Ethical and philosophical consideration of the dual-use dilemma in the biological sciences," *Science and Engineering Ethics*, vol. 13, no. 4, pp. 523–580, 2007.

[33] N. Seifert, "Radiation-induced soft errors: A chip-level modeling perspective," *Foundation and Trends in Electronic Design Automation*, vol. 4, pp. 99–221, Feb. 2010.

[34] G. Hubert, L. Artola, and D. Regis, "Impact of scaling on the soft error sensitivity of bulk, FDSOI and FinFET technologies due to atmospheric radiation," *Integration, the VLSI Journal*, vol. 50, pp. 39–47, 2015.

[35] J. E. Martin, *Physics for Radiation Protection*. Wiley, Mar. 2013.

[36] R. C. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies," *IEEE Trans. Device Mater. Rel.*, vol. 5, pp. 305–316, Sept. 2005.

[37] S. M. Seltzer, D. T. Bartlett, D. T. Burns, G. Dietze, H.-G. Menzel, H. G. Paretzke, and A. Wambersie, "ICRU report No. 85: Fundamental quantities and units for ionizing radiation," *Journal of the ICRU*, vol. 11, Oct. 2011. Revised.

[38] K. G. McKay and K. B. McAfee, "Electron multiplication in silicon and germanium," *Phys. Rev.*, vol. 91, Sept. 1953.

[39] Consultative Committee for Units, "The international system of units (SI)," tech. rep., Bureau International des Poids et Mesures, 2019.

[40] TRIUMF, "TRIUMF's proton- and neutron irradiation facilities." `https://www.triumf.ca/sites/default/files/PIFNIF-05.jpg`, Feb. 2021.

[41] JEDEC Solid State Technology Association, "Measurement and reporting of alpha particle and terrestrial cosmic ray-induced soft errors in semiconductor devices," JEDEC Standard JESD89A, JEDEC Solid State Technology Association, Arlington, VA, USA, Oct. 2006.

[42] N. Battezzati, L. Sterpone, and M. Violante, *Reconfigurable Field Programmable Gate Arrays for Mission-Critical Applications*. Springer, 2011.

[43] J. R. Srour and J. M. McGarrity, "Radiation effects on microelectronics in space," *Proc. IEEE*, vol. 76, no. 11, pp. 1443–1469, 1988.

[44] A. Calomarde, A. Rubio, F. Moll, and F. Gamiz, "Active radiation-hardening strategy in bulk FinFETs," *IEEE Access*, vol. 8, pp. 201441–201449, 2020.

[45] T. Uemura, B. Chung, J. Jo, M. Kim, D. Lee, G. Kim, S. Lee, T. Song, H. Rhee, B. Lee, and J. Choi, "Soft-error susceptibility in flip-flop in EUV 7 nm bulk-FinFET technology," in *IEEE Int. Reliability Physics Symp.*, Mar. 2021.

[46] T. Heijmen, P. Roche, G. Gasiot, and K. Forbes, "A comparative study on the soft-error rate of flip-flops from 90-nm production libraries," in *IEEE Int. Reliability Physics Symp.*, 2006.

[47] G. Gasiot, M. Glorieux, S. Clerc, D. Soussan, F. Abouzeid, and P. Roche, "Experimental soft error rate of several flip-flop designs representative of production chip in 32 nm CMOS technology," *IEEE Trans. Nucl. Sci.*, vol. 60, pp. 4226–4231, Dec. 2013.

[48] T. R. Oldham and F. B. McLean, "Total ionizing dose effects in MOS oxides and devices," *IEEE Trans. Nucl. Sci.*, vol. 50, pp. 483–499, June 2003.

[49] J.-W. Han, M. Meyyappan, and J. Kim, "Single event hard error due to terrestrial radiation," in *IEEE Int. Reliability Physics Symp.*, IEEE, Mar. 2021.

[50] G. C. Messenger, "Collection of charge on junction nodes from ion tracks," *IEEE Trans. Nucl. Sci.*, vol. 29, pp. 2024–2031, Dec. 1982.

[51] G. Srinivasan, P. Murley, and H. Tang, "Accurate, predictive modeling of soft error rate due to cosmic rays and chip alpha radiation," in *IEEE Int. Reliability Physics Symp.*, 1994.

[52] D. A. Black, W. H. Robinson, I. Z. Wilcox, D. B. Limbrick, and J. D. Black, "Modeling of single event transients with dual double-exponential current sources: Implications for logic cell characterization," *IEEE Trans. Nucl. Sci.*, vol. 62, pp. 1540–1549, Aug. 2015.

[53] I. Eusgeld, F. Fraikin, M. Rohr, F. Salfner, and U. Wappler, "Software reliability," in *Dependability Metrics: Advanced Lectures* (I. Eusgeld, F. C. Freiling, and R. Reussner, eds.), no. 4909 in Lecture Notes in Computer Science, ch. 2, pp. 104–125, Springer, 2008.

[54] P. Racunas, K. Constantinides, S. Manne, and S. S. Mukherjee, "Perturbation-based fault screening," in *IEEE Int. Symp. High Performance Comp. Arch.*, pp. 169–180, 2007.

[55] N. J. Wang and S. J. Patel, "Restore: Symptom-based soft error detection in microprocessors," *IEEE Trans. Depend. Sec. Comput.*, vol. 3, no. 3, pp. 188–201, 2006.

[56] M.-L. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou, "Understanding the propagation of hard errors to software and implications for resilient system design," *ACM SIGARCH Comp. Arch. News*, vol. 36, no. 1, pp. 265–276, 2008.

[57] A. Meixner, M. E. Bauer, and D. Sorin, "Argus: Low-cost, comprehensive error detection in simple cores," in *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, IEEE, 2007.

[58] T. M. Austin, "DIVA: A reliable substrate for deep submicron microarchitecture design," in *32nd Annu. Int. Symp. Microarchitecture*, pp. 196–207, 1999.

[59] D. J. Sorin, "Fault tolerant computer architecture," in *Synthesis Lectures on Computer Architecture* (M. D. Hill, ed.), Morgan & Claypool, 2009.

[60] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Error detection by duplicated instructions in super-scalar processors," *IEEE Trans. Rel.*, vol. 51, no. 1, pp. 63–75, 2002.

[61] G. Hetherington, T. Fryars, N. Tamarapalli, M. Kassab, A. Hassan, and J. Rajski, "Logic bist for large industrial designs: Real issues and case studies," in *Int. Test Conf.*, pp. 358–367, 1999.

[62] M. Psarakis, D. Gizopoulos, E. Sanchez, and M. Sonza Reorda, "Microprocessor software-based self-testing," *IEEE Des. Test. Comput.*, pp. 4–18, 2010.

[63] S. Mitra and E. J. McCluskey, "Which concurrent error detection scheme to choose?," in *ITC Int. Test Conf.*, pp. 985–994, 2000.

[64] G. Aydos and G. Fey, "Exploiting error detection latency for parity-based soft error detection," in *IEEE 19th Int. Symp. Des. Diagnostics Electron. Circuits Syst.*, 2016.

[65] A. Avižienis, "Design of fault-tolerant computers," in *Fall Joint Comput. Conf.*, pp. 733–743, 1967.

[66] D. Brière and P. Traverse, "AIRBUS a320/a330/a340 electrical flight controls: A family of fault-tolerant systems," in *23rd Int. Symp. Fault-Tolerant Comput.*, pp. 616–623, 1993.

[67] A. K. Nieuwland, S. Jasarevic, and G. Jerin, "Combinational logic soft error analysis and protection," in *12th IEEE Int. On-Line Testing Symp.*, IEEE, 2006.

[68] N. K. Jha and S.-J. Wang, "Design and synthesis of self-checking VLSI circuits," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 12, no. 6, 1993.

[69] J. M. Berger, "A note on error detection codes for asymmetric channels," *Information and Control*, vol. 4, pp. 68–73, Mar. 1961.

[70] S. Kundu and S. M. Reddy, "Embedded totally self-checking checkers: a practical design," *IEEE Des. Test. Comput.*, vol. 7, pp. 5–12, Aug. 1990.

[71] J. Johnson, W. Howes, M. Wirthlin, D. L. McMurtrey, M. Caffrey, P. Graham, and K. Morgan, "Using duplication with compare for on-line error detection in FPGA-based designs," in *IEEE Aerospace Conf.*, Mar. 2008.

[72] M. Nicolaidis, "Time redundancy based soft-error tolerance to rescue nanometer technologies," in *17th IEEE VLSI Test Symp.*, (Washington, DC, USA), pp. 86–94, IEEE Computer Society, Apr. 1999.

[73] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge, "Razor: A low-power pipeline based on circuit-level timing speculation," in *Proc. 36th Annu. IEEE / ACM Int. Symp. Microarchitecture*, pp. 7–18, IEEE, Dec. 2003.

[74] S. Das, C. Tokunaga, S. Pant, W.-H. Ma, S. Kalaiselvan, K. Lai, D. M. Bull, and D. T. Blaauw, "RazorII: In situ error detection and correction for PVT and SER tolerance," *IEEE J. Solid-State Circuits*, vol. 44, pp. 32–48, Jan. 2009.

[75] M. Nicolaidis and M. Dimopoulos, "Advanced double-sampling architectures," in *22nd IEEE Int. Symp. On-Line Testing Robust Syst. Des.* [217].

[76] A. Mahmood and E. J. McCluskey, "Concurrent error detection using watchdog processors - a survey," *IEEE Trans. Comput.*, vol. 37, no. 2, pp. 160–174, 1988.

[77] N. Nakka, G. P. Saggese, Z. Kalbarczyk, and R. K. Iyer, "An architectural framework for detecting process hangs/crashes," in *5th European Dependabe Comp. Conf.*, pp. 103–121, Springer, 2005.

[78] N. Oh and E. J. McCluskey, "Error detection by selective procedure call duplication for low energy consumption," *IEEE Trans. Rel.*, vol. 51, no. 4, pp. 392–402, 2002.

[79] S. K. Reinhardt and S. S. Mukherjee, "Transient fault detection via simultaneous multithreading," in *Proc. 27th Annu. Int. Symp. Comput. Architecture*, 2000.

[80] H. Engel, "Data flow transformations to detect results which are corrupted by hardware faults," in *Proc. IEEE High-Assurance Syst. Eng. Workshop*, 1996.

[81] N. Oh, S. Mitra, and E. J. McCluskey, "ED4I: Error detection by diverse data and duplicated instructions," *IEEE Trans. Comput.*, vol. 51, no. 2, p. 180, 2002.

[82] M. Hiller, "Executable assertions for detecting data errors in embedded control systems," in *Proc. Int. Conf. Dependable Syst. Netw.*, 2000.

[83] J. Vinter, J. Aidemark, P. Folkesson, and J. Karlsson, "Reducing critical failures for control algorithms using executable assertions and best effort recovery," in *Int. Conf. Dependable Syst. Netw.*, 2001.

[84] E. Koser and W. Stechele, "Tackling long duration transients in sequential logic," in *22nd IEEE Int. Symp. On-Line Testing Robust Syst. Des.* [217], pp. 137–142.

[85] J. von Neumann, "Probabilistic logics and the synthesis of reliable organisms from unreliable components," in *Automata Studies* (C. E. Shannon and J. McCarthy, eds.), vol. 34 of *Annals of Mathematics Studies*, pp. 43–98, Princeton University Press, 1956.

[86] R. E. Lyons and W. Vanderkulk, "The use of triple-modular redundancy to improve computer reliability," *IBM J. Res. Dev.*, vol. 6, pp. 200–209, Apr. 1962.

[87] T. Imagawa, H. Tsutsui, H. Ochi, and T. Sato, "A cost-effective selective TMR for heterogeneous coarse-grained reconfigurable architectures based on DFG-level vulnerability analysis," in *Proc. Conf. Des., Automat. Test Europe*, pp. 701–706, 2013.

[88] L. T. Clark, D. W. Patterson, R. C., and K. E. Holbert, "An embedded microprocessor radiation hardened by microarchitecture and circuits," *IEEE Trans. Comput.*, vol. 65, no. 2, pp. 382–395, 2016.

[89] ECSS Secretariat, "Space product assurance: Techniques for radiation effects mitigation in ASICs and FPGAs handbook," Handbook ECSS-Q-HB-60-02A, European Cooperation for Space Standardization, Sept. 2016.

[90] S. Mitra and E. J. McCluskey, "Word-voter: A new voter design for triple modular redundant systems," in *18th IEEE VLSI Test Symp.*, pp. 465–470, IEEE, 2000.

[91] K. Nikolic, A. Sadek, and M. Forshaw, "Architectures for reliable computing with unreliable nanodevices," in *Proc. 1st IEEE Conf. Nanotechnology*, pp. 254–259, 2001.

[92] D. Bhaduri, S. Shukla, P. Graham, and M. Gokhale, "Comparing reliability-redundancy tradeoffs for two von neumann multiplexing architectures," *IEEE Trans. Nanotechnol.*, vol. 6, no. 3, pp. 265–279, 2007.

[93] W. H. Pierce, *Failure-Tolerant Computer Design*. Academic Press, 1965.

[94] K. S. Morgan, D. L. McMurtrey, B. H. Pratt, and M. J. Wirthlin, "A comparison of TMR with alternative fault-tolerant design techniques for FPGAs," *IEEE Trans. Nucl. Sci.*, vol. 54, no. 6, pp. 2065–2072, 2007.

[95] A. E. Barbour, "Solutions to the minimization problem of fault-tolerant logic circuits," *IEEE Trans. Comput.*, vol. 41, pp. 429–443, 1992.

[96] W. J. van Gils, "A triple modular redundancy technique providing multiple-bit error protection without using extra redundancy," *IEEE Trans. Comput.*, vol. C-35, pp. 623–631, July 1986.

[97] T. Calin, M. Nicolaidis, and R. Velazco, "Upset hardened design for submicron CMOS technology," *IEEE Trans. Nucl. Sci.*, 1996.

[98] R. Naseer and J. Draper, "The DF-dice storage element for immunity to soft errors," in *48th Midwest Symp. Circuits Syst.*, pp. 303–306, IEEE, 2005.

[99] H. Wang, M. Li, X. Dai, S. Shi, L. Chen, and G. Guo, "Layout-based single event mitigation techniques for dynamic logic circuits," *J. Electron. Test.*, vol. 32, no. 1, pp. 97–103, 2016.

[100] G. Fey, S. Staber, R. Bloem, and R. Drechsler, "Automatic fault localization for property checking," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 27, pp. 1138–1149, 2008.

[101] A. Smith, A. Veneris, M. F. Ali, and A. Viglas, "Fault diagnosis and logic debugging using boolean satisfiability," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 24, no. 10, pp. 1606–1621, 2005.

[102] M. F. Ali, A. Veneris, S. Safarpour, M. Abadir, R. Drechsler, and A. Smith, "Debugging sequential circuits using boolean satisfiability," in *Proc. 5th Int. Workshop Microprocessor Test Verification*, 2004.

[103] H. Riener and G. Fey, "Exact diagnosis using boolean satisfiability," in *Proc. 35th Int. Conf. Comput.-Aided Des.*, p. 53, ACM, 2016.

[104] H. Riener and G. Fey, "Counterexample-guided diagnosis," in *Int. Verification Security Workshop*, 2016.

[105] J. Marques-Silva, M. Janota, A. Ignatiev, and A. Morgado, "Efficient model based diagnosis with maximum satisfiability," in *24th Int. Joint Conf. Artificial Intelligence*, vol. 15, pp. 1966–1972, 2015.

[106] A. L. Hopkins, Jr., T. B. Smith, III, and J. H. Lala, "FTMP: A highly reliable fault-tolerant multiprocessor for aircraft," *Proceedings of the IEEE*, vol. 66, no. 10, pp. 1221–1239, 1978.

[107] L. M. Kaufman, S. Bhide, and B. W. Johnson, "Modeling of common-mode failures in digital embedded systems," in *Annu. Rel. Maintainability Symp.*, 2000.

[108] H. Kopetz, "Fault containment and error detection in the time-triggered architecture," in *6th Int. Symp. Autonomous Decentralized Syst.*, 2003.

[109] N. Aggarwal, P. Ranganathan, N. P. Jouppi, and J. E. Smith, "Isolation in commodity multicore processors," *Computer*, vol. 40, no. 6, pp. 49–59, 2007.

[110] P. S. Morgan, "Enhancing the cassini mission through FP applications after launch," in *IEEE Aerospace Conf.*, pp. 1–20, 2016.

[111] S. Müller, M. Schölzel, and H. T. Vierhaus, "Towards a graceful degradable multicore-system by hierarchical handling of hard errors," in *21st Euromicro Int. Conf. Parallel, Distributed and Network-Based Process.*, pp. 302–309, IEEE, 2013.

[112] J. R. Fox, "Test-point condensation in the diagnosis of digital circuits," *Proc. IEEE*, vol. 124, no. 2, pp. 89–94, 1977.

[113] M. Prvulovic, Y. Zhang, and J. Torellas, "Revive: Cost-effective architectural support for rollback recovery in shared-memory multiprocessors," in *Proc. 29th Annu. Int. Symp. Comput. Architecture*, pp. 111–122, 2002.

[114] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood, "Safetynet: Improving the availability of shared memory multiprocessors with global checkpoint/recovery," in *Proc. 29th Annu. Int. Symp. Comput. Architecture*, 2002.

[115] T. J. Slegel, R. M. Averill III, and M. A. Check, "IBM's S/390 G5 microprocessor design," *IEEE Micro*, vol. 19, pp. 12–23, 1999.

[116] S. Gupta, A. Ansari, S. Feng, and S. Mahlke, "Adaptive online testing for efficient hard fault detection," in *IEEE Int. Conf. Comput. Des.*, pp. 343–349, 2009.

[117] S. Ananthanarayan, S. Garg, and H. D. Patel, "Low cost permanent fault detection using ultra-reduced instruction set co-processors," in *Proc. Conf. Des., Automat. Test Europe*, pp. 933–938, 2013.

[118] A. Benso, S. Di Carlo, G. Di Natale, and P. Prinetto, "A watchdog processor to detect data and control flow errors," in *9th IEEE On-Line Testing Symp.*, pp. 144–148, IEEE, 2003.

[119] K. Roy and S. C. Prasad, *Low power CMOS VLSI circuit design*. Wiley, 2000.

[120] C. Piguet, *History of Low-Power Electronics*. CRC Press, 2006.

[121] D. Markovic, C. C. Wang, L. P. Alarcon, T.-T. Liu, and J. M. Rabaey, "Ultralow-power design in near-threshold region," *Proc. IEEE*, vol. 98, pp. 237–252, Feb. 2010.

[122] I. L. Markov, "Limits on fundamental limits to computation," *Nature*, vol. 512, pp. 147–154, Aug. 2014.

[123] M. A. Breuer, "Adaptive computers," *Information and Control*, vol. 11, no. 4, pp. 402–422, 1967.

[124] A. L. Benjamin and J. H. Lala, "Advanced fault tolerant computing for future manned space missions," in *AIAA / IEEE Digit. Avionics Syst. Conf.*, vol. 2, pp. 8–5, 1997.

[125] D. Ngo and M. Harris, "A reliable infrastructure based on COTS technology for affordable space application," in *IEEE Aerospace Conf.*, vol. 5, pp. 2435–2441, 2001.

[126] M. N. Lovellette, K. S. Wood, D. L. Wood, J. H. Beall, P. P. Shirvani, N. Oh, and E. J. McCluskey, "Strategies for fault-tolerant, space-based computing: Lessons learned from the ARGOS testbed," in *IEEE Aerospace Conf.*, 2002.

[127] B. R. Borgerson and R. F. Freitas, "A reliability model for gracefully degrading and standby-sparing systems," *IEEE Trans. Comput.*, vol. C-24, pp. 517–525, May 1975.

[128] V. Cherkassky and M. Malek, "A measure of graceful degradation in parallel-computer systems," *IEEE Trans. Rel.*, vol. 38, no. 1, pp. 76–81, 1989.

[129] M. Glass, M. Lukasiewycz, C. Haubelt, and J. Teich, "Incorporating graceful degradation into embedded system design," in *Proc. Conf. Des., Autom. & Test Europe*, pp. 320–323, 2009.

[130] S. Hukerikar and C. Engelmann, "Resilience design patterns: A structured approach to resilience at extreme scale," techreport ORNL/TM-2016/687, Oak Ridge National Laboratory, Oct. 2016.

[131] H. E. Garcia, W.-C. Lin, S. M. Meerkov, and M. T. Ravichandran, "Resilient monitoring systems: Architecture, design, and application to boiler/turbine plant," *IEEE Trans. Cybern.*, vol. 44, pp. 2010–2023, Nov. 2014.

[132] A. B. Kahng, S. Kang, R. Kumar, and J. Sartori, "Slack redistribution for graceful degradation under voltage overscaling," in *15th Asia South Pacific Des. Automat. Conf.*, pp. 825–831, Jan. 2010.

[133] K. Becker and S. Voss, "Analyzing graceful degradation for mixed critical fault-tolerant real-time systems," in *IEEE 18th Int. Symp. Real-Time Distrib. Comput.*, pp. 110–118, IEEE, Apr. 2015.

[134] A. Kohler, G. Schley, and M. Radetzki, "Fault tolerant network on chip switching with graceful performance degradation," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 29, pp. 883–896, June 2010.

[135] T. Layh and D. Gebre-Egziabher, "Design for graceful degradation and recovery from GNSS interruptions," *IEEE Aerosp. Electron. Syst. Mag.*, vol. 32, pp. 4–17, Sept. 2017.

[136] D. Theodoropoulos, D. Pnevmatikatos, S. Tzilis, and I. Sourdis, "The DeSyRe runtime support for fault-tolerant embedded MPSoCs," in *Int. Symp. Parallel Distributed Process. Appl.*, pp. 197–204, 2014.

[137] M. Imai, T. Nagai, and T. Nanya, "Pair and swap: An approach to graceful degradation for dependable chip multiprocessors," in *Int. Conf. Dependable Syst. Netw. Workshops*, pp. 119–124, IEEE, June 2010.

[138] S. Tzilis, I. Sourdis, V. Vasilikos, D. Rodopoulos, and D. Soudris, "Runtime management of adaptive MPSoCs for graceful degradation," in *Proc. Int. Conf. Compilers, Architectures & Synthesis Embedded Syst.*, 2016.

[139] G. C. Cardarilli, M. Ottavi, S. Pontarelli, M. Re, and A. Salsano, "Fault localization, error correction, and graceful degradation in radix 2 signed digit-based adders," *IEEE Trans. Comput.*, vol. 55, pp. 534–540, May 2006.

[140] O. González, H. Shrikumar, J. A. Stankovict, and K. Ramamritham, "Adaptive fault tolerance and graceful degradation under dynamic hard real-time scheduling," in *Real-Time Syst.*, pp. 79–89, IEEE Comput. Soc, 1997.

[141] Z. Guo, K. Yang, S. Vaidhun, S. Arefin, S. K. Das, and H. Xiong, "Uniprocessor mixed-criticality scheduling with graceful degradation by completion rate," in *IEEE Real-Time Syst. Symp.*, IEEE, Dec. 2018.

[142] C. B. Watkins and R. Walter, "Transitioning from federated avionics architectures to integrated modular avionics," in *IEEE / AIAA 26th Digit. Avionics Syst. Conf.*, IEEE, Oct. 2007.

[143] R. Obermaisser, C. E. Salloum, B. Huber, and H. Kopetz, "From a federated to an integrated automotive architecture," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, pp. 956–965, July 2009.

[144] A. Burns and R. I. Davis, "A survey of research into mixed criticality systems," *ACM Computing Surveys*, vol. 50, pp. 1–37, Nov. 2017.

[145] R. Wilhelm, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, P. Stenström, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, and R. Heckmann, "The worst-case execution-time problem: Overview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems*, vol. 7, pp. 1–53, Apr. 2008.

[146] S. Vestal, "Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance," in *28th IEEE Int. Real-Time Syst. Symp.*, IEEE, Dec. 2007.

[147] S. Baruah, V. Bonifaci, G. D'angelo, H. Li, A. Marchetti-Spaccamela, S. V. D. Ster, and L. Stougie, "Preemptive uniprocessor scheduling of mixed-criticality sporadic task systems," *Journal of the ACM*, vol. 62, pp. 1–33, May 2015.

[148] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM*, vol. 20, pp. 46–61, Jan. 1973.

[149] S. K. Baruah, A. K. Mok, and L. E. Rosier, "Preemptively scheduling hard-real-time sporadic tasks on one processor," in *Proc. 11th Real-Time Syst. Symp.*, IEEE, 1990.

[150] L. Bougueroua, L. George, and S. Midonnet, "Dealing with execution-overruns to improve the temporal robustness of real-time systems scheduled FP and EDF," in *Second Int. Conf. Syst.*, pp. 52–52, IEEE, 2007.

[151] L. George, P. Courbin, and Y. Sorel, "Job vs. portioned partitioning for the earliest deadline first semi-partitioned scheduling," *J. Syst. Archit.*, vol. 57, pp. 518–535, May 2011.

[152] P. J. Parkinson, "The challenges of developing embedded real-time aerospace applications on next generation multi-core processors," in *Aviation Electron. Europe*, 2016.

[153] C.-L. Chou and R. Marculescu, "FARM: Fault-aware resource management in NoC-based multiprocessor platforms," in *Proc. Conf. Des., Autom. & Test Europe*, 2011.

[154] R. Schmidt, R. Massoud, J. Raik, A. Garcia-Ortiz, and R. Drechsler, "Reliability improvements for multiprocessor systems by health-aware task scheduling," in *IEEE 24th Int. Symp. On-Line Testing Robust Syst. Des.*, pp. 247–250, 2018.

[155] T. Guillaumet, A. Sharma, E. Feron, M. Krishna, R. Narayan, P. Baufreton, F. Neumann, and E. Grolleau, "Using reconfigurable multi-core architectures for safety-critical embedded systems," in *IEEE/AIAA 35th Digital Avionics Syst. Conf.*, pp. 1–6, IEEE, 2016.

[156] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Computing Surveys*, vol. 43, pp. 1–44, Oct. 2011.

[157] J. A. Bannister and K. S. Trivedi, "Task allocation in fault-tolerant distributed systems," *Acta Inf.*, vol. 20, no. 3, pp. 261–281, 1983.

[158] R. L. Graham, "Bounds on multiprocessing timing anomalies," *SIAM Journal on Applied Mathematics*, vol. 17, pp. 416–429, Mar. 1969.

[159] E. Bini and G. C. Buttazzo, "Measuring the performance of schedulability tests," *Real-Time Systems*, vol. 30, pp. 129–154, May 2005.

[160] B. Hu, L. Thiele, P. Huang, K. Huang, C. Griesbeck, and A. Knoll, "FFOB: efficient online mode-switch procrastination in mixed-criticality systems," *Real-Time Systems*, Dec. 2018.

[161] R. Schmidt and A. García-Ortiz, "Service improvements in real-time uniprocessor scheduling with single errors," *IEEE Access*, vol. 9, pp. 43540–43550, 2021.

[162] R. Schmidt, A. Garcia-Ortiz, and G. Fey, "Temporal redundancy latch-based architecture for soft error mitigation," in *IEEE 23rd Int. Symp. On-Line Testing Robust System Des.*, pp. 240–243, IEEE, July 2017.

[163] K. Mitropoulou, V. Porpodas, and T. M. Jones, "COMET: communication-optimised multi-threaded error-detection technique," in *2016 International Conference on Compilers, Architectures and Synthesis for Embedded Systems, CASES 2016, Pittsburgh, Pennsylvania, USA, October 1-7, 2016*, pp. 7:1–7:10, ACM, 2016.

[164] H. So, M. Didehban, Y. Ko, A. Shrivastava, and K. Lee, "EXPERT: effective and flexible error protection by redundant multithreading," in *2018 Design, Automation & Test in Europe Conference & Exhibition, DATE 2018, Dresden, Germany, March 19-23, 2018* (J. Madsen and A. K. Coskun, eds.), pp. 533–538, IEEE, 2018.

[165] M. Didehban and A. Shrivastava, "nzdc: a compiler technique for near zero silent data corruption," in *Proceedings of the 53rd Annual Design Automation Conference, DAC 2016, Austin, TX, USA, June 5-9, 2016*, pp. 48:1–48:6, ACM, 2016.

[166] M. Didehban, A. Shrivastava, and S. R. D. Lokam, "NEMESIS: A software approach for computing in presence of soft errors," in *2017 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2017, Irvine, CA, USA, November 13-16, 2017* (S. Parameswaran, ed.), pp. 297–304, IEEE, 2017.

[167] H. So, M. Didehban, A. Shrivastava, and K. Lee, "A software-level redundant multithreading for soft/hard error detection and recovery," in *Design, Automation & Test in Europe Conference & Exhibition, DATE 2019, Florence, Italy, March 25-29, 2019* (J. Teich and F. Fummi, eds.), pp. 1559–1562, IEEE, 2019.

[168] H. So, M. Didehban, J. Jung, A. Shrivastava, and K. Lee, "CHITIN: A comprehensive in-thread instruction replication technique against transient faults," in *Design, Automation & Test in Europe Conference & Exhibition, DATE 2021, Grenoble, France, February 1-5, 2021*, pp. 1440–1445, IEEE, 2021.

[169] S. Feng, S. Gupta, A. Ansari, and S. A. Mahlke, "Shoestring: probabilistic soft error reliability on the cheap," in *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2010, Pittsburgh, Pennsylvania, USA, March 13-17, 2010* (J. C. Hoe and V. S. Adve, eds.), pp. 385–396, ACM, 2010.

[170] S. Beer, M. Cannizzaro, J. Cortadella, R. Ginosar, and L. Lavagno, "Metastability in better-than-worst-case designs," in *20th IEEE Int. Symp. Asynchronous Circ. and Syst.*, pp. 101–102, 2014.

[171] Zephyr Project, "Zephyr RTOS project homepage." `https://www.zephyrproject.org/`, 2021. Accessed 2021-07.

[172] Zephyr Project, "Zephr RTOS scheduler source code." `https://github.com/zephyrproject-rtos/zephyr/blob/main/kernel/sched.c`, July 2021. Commit 8b3f36c.

[173] N. Guan, P. Ekberg, M. Stigge, and W. Yi, "Improving the scheduling of certifiable mixed-criticality sporadic task systems," Tech. Rep. 2013-008, Department of Information Technology, Uppsala University, Apr. 2013.

[174] F. Singhoff, J. Legrand, L. Nana, and L. Marcé, "Cheddar: a flexible real time scheduling framework," in *ACM SIGAda Ada Letters*, vol. 24, pp. 1–8, ACM, 2004.

[175] P. Ivie and D. Thain, "Reproducibility in scientific computing," *ACM Computing Surveys*, vol. 51, pp. 1–36, July 2018.

[176] A. Manacero, Jr., M. B. Miola, and V. A. Nabuco, "Teaching real-time with a scheduler simulator," in *31st ASEE / IEEE Frontiers Education Conf.*, vol. 2, pp. 15–19, IEEE, 2001.

[177] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings, "STRESS: A simulator for hard real-time systems," *Software: Practice and Experience*, vol. 24, no. 6, pp. 543–564, 1994.

[178] A. Diaz, R. Batista, and O. Castro, "Realtss: a real-time scheduling simulator," in *4th Int. Conf. Electrical Electron. Eng.*, pp. 165–168, IEEE, 2007.

[179] Y. Matsubara, Y. Sano, S. Honda, and H. Takada, "An open-source flexible scheduling simulator for real-time applications," in *IEEE 15th Int. Symp. Object/Component/Service-Oriented Real-Time Distrib. Comput.*, pp. 16–22, IEEE, 2012.

[180] T. Kramp, M. Adrian, and R. Koster, "An open framework for real-time scheduling simulation," in *Int. Parallel Distrib. Process. Symp.*, pp. 766–772, Springer, 2000.

[181] G. Lipari, G. Lamastra, A. Casile, L. Palopoli, and C. Bartolini, "RTSIM: Real-time system simulator," 2009. `http://rtsim.sssup.it/`.

[182] G. Yao, G. Buttazzo, and M. Bertogna, "YAO-SIM: Yet another operating system simulator for real-time scheduling," 2011. `http://yaosim.sssup.it/`.

[183] G. Yao, G. Buttazzo, and M. Bertogna, "Feasibility analysis under fixed priority scheduling with limited preemptions," *Real-Time Systems*, vol. 47, no. 3, pp. 198–223, 2011.

[184] R. Schmidt, "thready - a lightweight and fast scheduling simulator," 2020. `http://doi.org/10.5281/zenodo.3601863`.

[185] O. Tange, "GNU parallel: The command-line power tool," *;login:*, vol. 36, no. 1, pp. 42–47, 2011.

[186] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *ACM Sigplan notices*, vol. 42, pp. 89–100, ACM, 2007.

[187] M. D. McIlroy, E. N. Pinson, and B. A. Tague, "UNIX time-sharing system: Foreword," *Bell System Technical Journal*, vol. 57, no. 6, pp. 1899–1904, 1978.

[188] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. Hamrick, J. Grout, S. Corlay, P. Ivanov, D. Avila, S. Abdalla, C. Willing, and Jupyter Development Team, "Jupyter notebooks — a publishing format for reproducible computational workflows," in *Proc. 20th Int. Conf. Electronic Publishing* (F. Loizides and B. Schmidt, eds.), IOS Press, 2016.

[189] S. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, S. van der Ster, and L. Stougie, "The preemptive uniprocessor scheduling of mixed-criticality implicit-deadline sporadic task systems," in *24th Euromicro Conf. Real-Time Syst.*, IEEE, July 2012.

[190] T. Fleming and A. Burns, "Incorporating the notion of importance into mixed criticality systems," in *Proc. 2nd Workshop Mixed Crit. Syst.*, pp. 33–38, 2014.

[191] S. Baruah and A. Burns, "Incorporating robustness and resilience into mixed-criticality scheduling theory," in *Proc. 22nd IEEE Int. Symp. Real-Time Comput.*, 2019.

[192] A. Burns and S. K. Baruah, "Towards a more practical model for mixed criticality systems," in *Proc. 1st Int. Workshop Mixed Crit. Syst.*, pp. 1–6, 2013.

[193] A. Burns, S. Punnekkat, L. Strigini, and D. R. Wright, "Probabilistic scheduling guarantees for fault-tolerant real-time systems," in *Dependable Comput. Crit. Appl. 7*, IEEE, 1999.

[194] F. Santy, L. George, P. Thierry, and J. Goossens, "Relaxing mixed-criticality scheduling strictness for task sets scheduled with FP," in *24th Euromicro Conf. Real-Time Syst.*, IEEE, July 2012.

[195] A. Avižienis, "Fault-tolerant systems," *IEEE Trans. Comput.*, vol. C-25, no. 12, pp. 1304–1312, 1976.

[196] P. Ekberg and W. Yi, "Bounding and shaping the demand of generalized mixed-criticality sporadic task systems," *Real-Time Systems*, vol. 50, pp. 48–86, June 2013.

[197] D. Kraft, "A software package for sequential quadratic programming," Tech. Rep. DFVLR-FB 88-28, Deutsche Forschungs- und Versuchsanstalt für Luft- und Raumfahrt, Oberpfaffenhofen, Germany, 1988.

[198] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nat. Methods*, vol. 17, pp. 261–272, 2020.

[199] R. Schmidt and A. García-Ortiz, "Thready: A fast scheduling simulator for real-time task systems," in *9th Int. Conf. Modern Circuits Syst. Technol.*, 2020.

[200] A. Burns, R. I. Davis, S. Baruah, and I. Bate, "Robust mixed-criticality systems," *IEEE Transactions on Computers*, vol. 67, pp. 1478–1491, Oct. 2018.

[201] A. Thekkilakattil, R. Dobrin, and S. Punnekkat, "Fault tolerant scheduling of mixed criticality real-time tasks under error bursts," *Procedia Comput. Sci.*, vol. 46, pp. 1148–1155, 2015.

[202] R. M. Pathan, "Fault-tolerant and real-time scheduling for mixed-criticality systems," *Real-Time Syst.*, 2014.

[203] Q. Han, L. Niu, G. Quan, S. Ren, and S. Ren, "Energy efficient fault-tolerant earliest deadline first scheduling for hard real-time systems," *Real-Time Systems*, vol. 50, pp. 592–619, Sept. 2014.

[204] H. Beitollahi, S. G. Miremadi, and G. Deconinck, "Fault-tolerant earliest-deadline-first scheduling algorithm," in *IEEE Int. Parallel Distrib. Process. Symp.*, IEEE, 2007.

[205] H. Su and D. Zhu, "An elastic mixed-criticality task model and its scheduling algorithm," in *Des., Automat. & Test Europe Conf. & Exhibition*, IEEE Conference Publications, 2013.

[206] P. Huang, G. Giannopoulou, N. Stoimenov, and L. Thiele, "Service adaptions for mixed-criticality systems," in *19th Asia South Pacific Des. Automat. Conf.*, IEEE, Jan. 2014.

[207] X. Zhao, Y. Wei, and W. Li, "The improved earliest deadline first with virtual deadlines mixed-criticality scheduling algorithm," in *IEEE Int. Symp. Parallel Distrib. Process. with Appl. IEEE Int. Conf. Ubiquitous Comput. Communications*, IEEE, Dec. 2017.

[208] D. Liu, J. Spasic, N. Guan, G. Chen, S. Liu, T. Stefanov, and W. Yi, "EDF-VD scheduling of mixed-criticality systems with degraded quality guarantees," in *IEEE Real-Time Syst. Symp.*, IEEE, Nov. 2016.

[209] K. Yang and Z. Guo, "EDF-based mixed-criticality scheduling with graceful degradation by bounded lateness," in *IEEE 25th Int. Conf. Embedded Real-Time Comput. Syst. Appl.*, IEEE, Aug. 2019.

[210] G. Chen, N. Guan, D. Liu, Q. He, K. Huang, T. Stefanov, and W. Yi, "Utilization-based scheduling of flexible mixed-criticality real-time tasks," *IEEE Transactions on Computers*, vol. 67, pp. 543–558, Apr. 2018.

[211] H. Su, N. Guan, and D. Zhu, "Service guarantee exploration for mixed-criticality systems," in *IEEE 20th Int. Conf. Embedded Real-Time Comput. Syst. Appl.*, IEEE, Aug. 2014.

[212] S. Baruah and P. Ekberg, "Graceful degradation in semi-clairvoyant scheduling," in *33rd Euromicro Conf. Real-Time Syst.*, pp. 9:1–9:21, 2021.

[213] A. Cervin, J. Eker, B. Bernhardsson, and K.-E. Årzén, "Feedback-feedforward scheduling of control tasks," *Real-Time Systems*, vol. 23, no. 1/2, pp. 25–53, 2002.

[214] B. Ranjbar, B. Safaei, A. Ejlali, and A. Kumar, "FANTOM: Fault tolerant task-drop aware scheduling for mixed-criticality systems," *IEEE Access*, vol. 8, pp. 187232–187248, 2020.

[215] B. Ranjbar, A. Hosseinghorban, M. Salehi, A. Ejlali, and A. Kumar, "Toward the design of fault-tolerance- and peak-power-aware multi-core mixed-criticality systems," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, 2021.

[216] P. Huang, H. Yang, and L. Thiele, "On the scheduling of fault-tolerant mixed-criticality systems," techreport 351, Computer Engineering and Networks Laboratory, ETH Zurich, 2013.

[217] *22nd IEEE Int. Symp. On-Line Testing and Robust Syst. Design*, July 2016.

Appendix

1 Zephyr EDF template

This template (`zephyr-edf-template.c`) shows how dynamic priority EDF scheduling of multiple tasks can be implemented within the Zephyr RTOS:

Begin of code

```
/*
 * Copyright (c) 2017 Linaro Limited
 * Modifications copyright (c) 2021 Robert Schmidt
 *
 * SPDX-License-Identifier: Apache-2.0
 */

#include <zephyr.h>
#include <sys/printk.h>

/* Size of stack area used by each thread */
#define STACKSIZE 1024

/* Scheduling priority used by each thread,
 * threads with same priotity are scheduled by their deadline */
#define PRIORITY 7

/* The actual fake work / sleeping we want to do */
void fixed_workload(uint32_t work_us)
{
        int64_t ticks = k_uptime_ticks();
        printk("Entering fixed_workload() at tick %" PRId64 " (%" PRId64
               " ms)\n",
               ticks, k_ticks_to_ms_floor64(ticks));

        k_busy_wait(work_us);

        ticks = k_uptime_ticks();
        printk("Leaving fixed_workload() at tick %" PRId64 " (%" PRId64
               " ms)\n",
               ticks, k_ticks_to_ms_floor64(ticks));
}

/* Some boilerplate to satisfy thread/job creation interface */
void fixed_workload_entry(void *work_us, void *thread_id, void *dummy3)
{
        int64_t ticks = k_uptime_ticks();
        printk("Entering fixed_workload_entry() at tick %" PRId64 " (%" PRId64
               " ms)\n",
               ticks, k_ticks_to_ms_floor64(ticks));
        /* Spawning a thread dynamicaly requires to have
         * these dummy arguments */
        ARG_UNUSED(dummy3);
        uint32_t c = *(uint32_t *)work_us;
        char *myname = *(char **)thread_id;
        printk("I am a job from %s\n", myname);
        fixed_workload(c);
}

/* Move thread spawning to main() function which is executed by the
 * main thread which initializes the kernel.
 * This allows to set the deadlines for threads required for EDF scheduling
 * within the same static priority.
 * In mixed-criticality terms, zephyr threads are jobs.
 */
```

```
56
57 K_THREAD_STACK_DEFINE(thread_a_stack, STACKSIZE);
58 struct k_thread thread_a_data;
59 k_tid_t thread_a_tid;
60
61 K_THREAD_STACK_DEFINE(thread_b_stack, STACKSIZE);
62 struct k_thread thread_b_data;
63 k_tid_t thread_b_tid;
64
65 /* Timer for period thread/job creation */
66 struct k_timer periodic_release_timer_thread_a;
67 struct k_timer periodic_release_timer_thread_b;
68
69 /* Here I define all relevant thread parameters in my model:
70  * - work time
71  * - relative deadline
72  * - an ID to illustrate who is executing via printk (for illustration only)
73  */
74 const char *thread_a_id = "thread a";
75 uint32_t thread_a_work_us = 250000;
76 uint32_t thread_a_relative_deadline_us = 1000000; /* 1 second */
77
78 const char *thread_b_id = "thread b";
79 uint32_t thread_b_work_us = 250000;
80 uint32_t thread_b_relative_deadline_us = 3000000; /* 3 seconds */
81
82 /* Spawns thread/job with 250ms fake work and deadline of 1s.
83  * Associates thread with thread data structure.
84  *
85  * spawn_thread_a_job -> thread_a_entry -> thread_a
86  */
87 void spawn_thread_a_job()
88 {
89         printk("Spawning a job for thread a\n");
90         thread_a_tid =
91                 k_thread_create(&thread_a_data, thread_a_stack,
92                                 K_THREAD_STACK_SIZEOF(thread_a_stack),
93                                 fixed_workload_entry, &thread_a_work_us,
94                                 &thread_a_id, NULL, PRIORITY, 0, K_NO_WAIT);
95         k_thread_deadline_set(
96                 thread_a_tid,
97                 k_us_to_cyc_floor32(thread_a_relative_deadline_us));
98 }
99
100 void spawn_thread_b_job()
101 {
102         printk("Spawning a job for thread b\n");
103         thread_b_tid =
104                 k_thread_create(&thread_b_data, thread_b_stack,
105                                 K_THREAD_STACK_SIZEOF(thread_b_stack),
106                                 fixed_workload_entry, &thread_b_work_us,
107                                 &thread_b_id, NULL, PRIORITY, 0, K_NO_WAIT);
108         k_thread_deadline_set(
109                 thread_b_tid,
110                 k_us_to_cyc_floor32(thread_b_relative_deadline_us));
111 }
112
113 /* Convenience function to handle the timer expiration and
114  * decide if we need to release the next job */
115 void spawn_job_if_expired(struct k_timer *release_timer,
116                           void (*spawn_function_pointer)(),
117                           int period_seconds)
118 {
119         if (k_timer_status_get(release_timer) > 0) {
120                 /* Timer expired, call void function without argument to
121                  * spawn new job.
122                  * If the timer expired multiple times, we still start just
123                  * one job, implicityly skipping jobs in between.
124                  */
125                 printk("Timer expired\n");
126                 (*spawn_function_pointer)();
127         } else if (k_timer_remaining_get(release_timer) == 0) {
128                 /* Timer stopped; this is the default state after
```

```
                 * timer initialization.  Starting the timer here,
                 * hoping all timers are more or less started in sync.
                 */
                printk("Timer started\n");
                k_timer_start(release_timer, K_SECONDS(period_seconds),
                              K_SECONDS(period_seconds));
        } /* else timer is still running */
}

void main(void)
{
        int64_t ticks = k_uptime_ticks();
        printk("Entering main() at tick %" PRId64 " (%" PRId64 " ms)\n",
               ticks, k_ticks_to_ms_floor64(ticks));

        /* Define a timer for each thread to spawn jobs with its expiry. */
        k_timer_init(&periodic_release_timer_thread_a, NULL, NULL);
        k_timer_start(&periodic_release_timer_thread_a, K_SECONDS(1),
                      K_SECONDS(1));

        k_timer_init(&periodic_release_timer_thread_b, NULL, NULL);
        k_timer_start(&periodic_release_timer_thread_b, K_SECONDS(3),
                      K_SECONDS(3));

        while (1) {
                spawn_job_if_expired(&periodic_release_timer_thread_a,
                                     &spawn_thread_a_job, 1);
                spawn_job_if_expired(&periodic_release_timer_thread_b,
                                     &spawn_thread_b_job, 3);
                /* While main thread sleeps 300ms, others can execute. */
                printk("Sleeping in main()\n");
                k_sleep(K_MSEC(300));
        }
        printk("I can never end here...");
}
```

End of code

2 Mixed-criticality framework

Fundamental building blocks are real-time tasks and task sets (`threadysequencer/tasksystem.py`).

Begin of code

```
import json
import copy
import math
import dataclasses
import numpy as np
import scipy.optimize as optimize
from functools import reduce
from dataclasses import dataclass
from fractions import Fraction

from tabulate import tabulate


class TasksystemError(Exception):
    """Indicate that a task system differs from expectations."""

    pass


class EnhancedJSONEncoder(json.JSONEncoder):
    def default(self, o):
        if dataclasses.is_dataclass(o):
            return dataclasses.astuple(o)
        return super().default(o)


@dataclass(order=False, frozen=True)
class RTask:
```

```
    """Defines the order of all task- and simulation related parameters.

    There is no meaningful default ordering for sorting tasks,
    instead one might use for example list.sort(key=lambda t: t.period).
    Once initialized, there is no meaning in changing the fields,
    therefore frozen is set to true. This allows to hash tasks.
    """

    taskid: int
    period: int
    relativedeadline: int
    computation0: int
    computation1: int
    computation2: int
    computation3: int
    computation4: int
    computation5: int
    """Probability to uniformly draw the computation demand between
    computation0 and computation1."""
    probability0: float
    """Probability to uniformly draw the computation demand between
    computation2 and computation3."""
    probability1: float
    """Defines exponential distributed inter arrival time between jobs."""
    beta: float
    """Priority for fixed priority scheduling for comparisions."""
    priority: int

    @property
    def utilization0(self):
        return Fraction(self.computation1, self.period)

    @property
    def utilization1(self):
        return Fraction(self.computation3, self.period)

    @property
    def utilization2(self):
        return Fraction(self.computation5, self.period)

    @property
    def locrit(self):
        return (
            (self.computation2 == 0)
            and (self.computation3 == 0)
            and (self.probability0 == 1.0)
            and (self.probability1 == 0.0)
        )

    @property
    def criticalitylevel(self):
        if self.locrit:
            return 1
        elif (
            (self.computation4 == 0)
            and (self.probability0 != 1.0)
            and (self.probability0 + self.probability1 == 1.0)
        ):
            return 2
        elif (
            self.computation4
            and self.computation5
            and (self.probability0 + self.probability1 < 1.0)
        ):
            return 3
        else:
            raise TasksystemError(
                "Could not identify criticality level of task; wrong input?"
            )

    @property
    def worstcasereservation(self):
        """Worst case reservation of a dual-criticality task excluding
```

```
102             possible errors.
103         """
104         if self.probability1 != 0.0:
105             return self.computation3
106         else:
107             return self.computation1
108 
109     def __iter__(self):
110         yield self.taskid
111         yield self.period
112         yield self.relativedeadline
113         yield self.computation0
114         yield self.computation1
115         yield self.computation2
116         yield self.computation3
117         yield self.computation4
118         yield self.computation5
119         yield self.probability0
120         yield self.probability1
121         yield self.beta
122         yield self.priority
123 
124 
125 def _generate_locrit_task(utilization):
126     """Generate single low criticality task for a given utilization.
127     """
128     taskid = -1
129     f = Fraction(utilization).limit_denominator(1000)
130     computation1, period = (f.numerator, f.denominator)
131     if computation1 == 0:
132         raise TasksystemError(
133             f"No sensible fraction for utilization of {utilization}!"
134         )
135     computation0 = 1
136     reldead = period
137     computation2 = 0
138     computation3 = 0
139     probability0 = 1.0
140     probability1 = 0.0
141     task = RTask(
142         taskid,
143         period,
144         reldead,
145         computation0,
146         computation1,
147         computation2,
148         computation3,
149         0,
150         0,  # computation4 and computation5 not used
151         probability0,
152         probability1,
153         0.001,  # dummy job trace generation parameter
154         1,  # priority not considered
155     )
156     return task
157 
158 
159 class Tasksystem(list):
160     """A consistent list of tasks with JSON load/store interface."""
161 
162     version = "0.8"
163 
164     def __init__(self, data, **kwds):
165         super().__init__(**kwds)
166         self.extend(data)
167 
168     def __eq__(self, other):
169         return hash(self) == hash(other)
170 
171     def __hash__(self):
172         me = [hash(task) for task in self.values()]
173         me = sorted(me)
174         return hash(tuple(me))
```

```
175
176     def __repr__(self):
177         return tabulate(
178             self,
179             headers=[
180                 "Id",
181                 "Period",
182                 "Deadline",
183                 "C0",
184                 "C1",
185                 "C2",
186                 "C3",
187                 "C4",
188                 "C5",
189                 "Prob0",
190                 "Prob1",
191                 "Beta",
192                 "Prio",
193             ],
194         )
195
196     @classmethod
197     def from_json(cls, fname, withprio=False):
198         if withprio:
199             with open(fname, "r") as fh:
200                 data = json.load(fh)
201             ts = [RTask(*v) for v in data]
202             return cls(ts)
203         else:
204             with open(fname, "r") as fh:
205                 data = json.load(fh)
206             ts = list()
207             for v in data:
208                 tmp = list(v)
209                 tmp.append(0)
210                 ts.append(RTask(*tmp))
211             return cls(ts)
212
213     def get_task_by_id(self, taskid):
214         for t in self:
215             if t.taskid == taskid:
216                 return t
217         raise TasksystemError("Unknown task id")
218
219     @property
220     def hitaskids(self):
221         return [t.taskid for t in self if not t.locrit]
222
223     @property
224     def lotaskids(self):
225         return [t.taskid for t in self if t.locrit]
226
227     def get_hitasks_only_tasksystem(self):
228         tmp = [t for t in self if not t.locrit]
229         return Tasksystem(tmp)
230
231     def _strip_prio(self):
232         noprio = list()
233         for t in self:
234             noprio.append(
235                 (
236                     t.taskid,
237                     t.period,
238                     t.relativedeadline,
239                     t.computation0,
240                     t.computation1,
241                     t.computation2,
242                     t.computation3,
243                     t.computation4,
244                     t.computation5,
245                     t.probability0,
246                     t.probability1,
247                     t.beta,
```

```
                )
            )
        return Tasksystem(noprio)

    def write_json(self, fname, withprio=False):
        if withprio:
            with open(fname, "w") as fh:
                json.dump(self, fh, cls=EnhancedJSONEncoder)
        else:
            noprio = self._strip_prio()
            with open(fname, "w") as fh:
                json.dump(noprio, fh, cls=EnhancedJSONEncoder)

    def _ull(self, hitaskids):
        """Total utilization of low tasks in low mode."""
        return sum(
            [
                task.utilization0
                for task in self
                if task.taskid not in hitaskids
            ]
        )

    def _uhh(self, hitaskids):
        """Total utilization of high tasks in high mode."""
        return sum(
            [task.utilization1 for task in self if task.taskid in hitaskids]
        )

    def _uhl(self, hitaskids):
        """Total utilization of high tasks in low mode."""
        return sum(
            [task.utilization0 for task in self if task.taskid in hitaskids]
        )

    def calculate_total_util_lo(self, hitaskids):
        return self._uhl(hitaskids) + self._ull(hitaskids)

    def dumps(self):
        return json.dumps(self, cls=EnhancedJSONEncoder)

    def calculate_ss01_lambda(self, hitaskids):  # Muller2016
        s12 = sum(
            [
                math.sqrt(
                    task.utilization0 * (task.utilization1-task.utilization0)
                )
                for task in self
                if task.taskid in hitaskids
            ]
        )  # Eq. 7
        uhh = self._uhh(hitaskids)
        if uhh > 1.0:
            return None, None
        ull = self._ull(hitaskids)
        uhl = self._uhl(hitaskids)
        try:
            lowlamb = s12 / (1.0 - uhh + uhl)  # Eq. 15
            hilamb = (1.0 - ull - uhl) / s12
        except ZeroDivisionError:
            return None, None
        if math.isclose(lowlamb, hilamb):
            return lowlamb, hilamb
        if lowlamb > hilamb:
            return None, None
        return lowlamb, hilamb

    def is_ss01_schedulable(self, hitaskids):
        """Schedulable with non-uniform deadline scaling as in Muller2016."""
        lowlamb, hilamb = self.calculate_ss01_lambda(hitaskids)
        static_condition_lo = self.calculate_total_util_lo(hitaskids) < 1.0
        return not (
            lowlamb is None and hilamb is None or not static_condition_lo
```

```
        )

    def calculate_edf_vd_deadline_scales(self, hitaskids, supply=1):
        """Calculate uniform deadline scaling as in Baruah2015."""
        uhh = self._uhh(hitaskids)
        ull = self._ull(hitaskids)
        uhl = self._uhl(hitaskids)
        if (uhh > supply) or (uhl + ull > supply):
            raise TasksystemError(
                "Tasksystem is not EDF-VD [Baruah2015] schedulable!"
            )
        xlow = uhl / (supply - ull)
        xhigh = (supply - uhh) / ull
        return xlow, xhigh

    def is_edf_vd_schedulable(self, hitaskids, supply=1):
        """Schedulable with uniform deadline scaling as in Baruah2015."""
        xlow, xhigh = self.calculate_edf_vd_deadline_scales(
            hitaskids, supply=supply
        )
        return xlow <= xhigh

    def calculate_edf_nuvd_lambda(
        self, hitaskids, supply=1.0
    ):  # Baruah2015 p 14:30
        s12 = sum(
            [
                math.sqrt(task.utilization0 * task.utilization1)
                for task in self
                if task.taskid in hitaskids
            ]
        )  # Theorem 6.2
        uhh = self._uhh(hitaskids)
        if uhh > supply:
            return None, None
        ull = self._ull(hitaskids)
        uhl = self._uhl(hitaskids)
        try:
            lowlamb = s12 / (supply - uhh)  # Eq. 31
            hilamb = (supply - ull - uhl) / s12
        except ZeroDivisionError:
            return None, None
        if math.isclose(lowlamb, hilamb):
            return lowlamb, hilamb
        if lowlamb > hilamb:
            return None, None
        return lowlamb, hilamb

    def is_edf_nuvd_schedulable(self, hitaskids, supply=1.0):
        lowlamb, hilamb = self.calculate_edf_nuvd_lambda(
            hitaskids, supply=supply
        )
        return not (lowlamb is None and hilamb is None)

    def get_task_by_priority(self, i):
        for t in self:
            if t.priority == i:
                return t
        raise TasksystemError("No task with priority %d found!" % i)

    def calculate_worst_case_response_time(self, i, checkload=False):
        """WCRT of equivalent worst case reservation task system.

        The worst case response time is calculated according to the
        recursive algorithm in Joseph1986, which is defined for
        a **non**-mixed-criticality task system!
        Therefore the worst case reservation task system equivalent is
        used, where the highest computation demand of a task is taken
        for the computation demand of the equivalent task system
        without criticality levels.

        Recursion limits are not checked for!
        """
```

```
394
395         def lcm(denominators):
396             """https://stackoverflow.com/a/49816058"""
397             return reduce(lambda a, b: a * b // math.gcd(a, b), denominators)
398
399         def loadcondition(tasksystem, i):
400             """Needs to be satisfied for a feasible priority assignment.
401             """
402             M = lcm([t.period for t in tasksystem if t.priority <= i])
403             print(M)
404             load = [
405                 M / t.period * t.worstcasereservation
406                 for t in tasksystem
407                 if t.priority < i
408             ]
409             print(load)
410             print(sum(load))
411             return sum(load) < M
412
413         def comp(tasksystem, t1, t2, i):
414             csum = 0.0
415             for j in range(1, i):
416                 t = tasksystem.get_task_by_priority(j)
417                 T = t.period
418                 C = t.worstcasereservation
419                 csum += (int(math.ceil(t2 / T)) - int(math.ceil(t1 / T))) * C
420             return int(csum)
421
422         def resp(tasksystem, t1, t2, i):
423             c = comp(tasksystem, t1, t2, i)
424             if c == 0:
425                 return t2
426             else:
427                 return resp(tasksystem, t2, t2 + c, i)
428
429         if checkload and not loadcondition(self, i):
430             raise TasksystemError(
431                 "Priority assignment not feasible, load condition not met!"
432             )
433         t = self.get_task_by_priority(i)
434         C = t.worstcasereservation
435         return resp(self, 0, C, i)
436
437     def is_edf_nuvd_se_schedulable(self, res=None):
438         """Schedulable if result from num. optimization and static check agree.
439         """
440         if res is None:
441             res = self._optimize_edf_nuvd_se_deadline_scales()
442         if res.success:
443             U11 = res.x[-1]
444             lotask = _generate_locrit_task(U11)
445             checkts = [t for t in self if not t.locrit]
446             checkts.append(lotask)
447             checkts = Tasksystem(checkts)
448             return checkts._edf_nuvd_se_ok()
449         return False
450
451     def _edf_nuvd_se_ok(self):
452         """Reuse EDF-NUVD check with adjusted supply to check if
453         task system is schedulable under EDF-NUVD-SE.
454         """
455         highcrittasks = [t for t in self if not t.locrit]
456         for t_j in highcrittasks:
457             reducedts = [t for t in self if t != t_j]
458             reducedts = Tasksystem(reducedts)
459             supply = 1.0 - float(t_j.utilization1)
460             if not reducedts.is_edf_nuvd_schedulable(
461                 reducedts.hitaskids, supply=supply
462             ):
463                 return False
464         return True
465
466     def _optimize_edf_nuvd_se_deadline_scales(self):
```

```
        """Maximize utilization of low criticality tasks in low criticality
        mode, calculating feasible deadline scalings.
        """

        def costfunc(x):
            """We want to maximize U_1(1).

            x: Numpy array of unknowns
            """
            return -x[-1]

        def costfuncderiv(x):
            """Derivative w.r.t. virtual deadline scaling parameters is zero.
            """
            a = np.zeros_like(x)
            a[-1] = -1  # dfdu11
            return a

        def generate_single_lowmode_constraint(ts, taskj):
            """Generates an inequality constraint function and deriviative
            for SLSQP solver.

            The constraint is for the low criticality mode and allows high
            criticality task j to overrun.

            taskj: Task id
            ts: Tasksystem
            """
            hitaskids = sorted(ts.hitaskids)
            u_taskj = float(ts.get_task_by_id(taskj).utilization1)
            # Create array of high crit task utilizations in low mode
            other_hitask_utils = np.zeros_like(hitaskids, dtype=np.float64)
            for i, taski in enumerate(hitaskids):
                if taski == taskj:
                    other_hitask_utils[i] = 0
                else:
                    u_i = float(ts.get_task_by_id(taski).utilization0)
                    other_hitask_utils[i] = u_i
            # For constraint derivative need array of all u_i^L and for
            # task j replaced by u_j^H
            all_mixed_hitask_utils = np.zeros_like(hitaskids, dtype=np.float64)
            for i, taski in enumerate(hitaskids):
                if taski == taskj:
                    all_mixed_hitask_utils[i] = u_taskj
                else:
                    u_i = float(ts.get_task_by_id(taski).utilization0)
                    all_mixed_hitask_utils[i] = u_i

            def _get_scale_from_id(taskids, tid):
                """Return index position of tid in taskids which is the same
                index in array of unknowns x
                """
                return taskids.index(tid)

            xjind = _get_scale_from_id(hitaskids, taskj)

            def constraintfun(x):
                """
                x: Array of unknowns, with x[-1] as U_L^L and x[:-1] as
                virtual deadline scales
                """
                # Sum over other_hitask_utils works because for task j we
                # entered 0 into it, effectively making this a sum over
                # all tasks except j.
                return np.array(
                    [
                        1
                        - x[-1]
                        - u_taskj / x[xjind]
                        - np.sum(other_hitask_utils / x[:-1])
                    ]
                )

```

```
            def constraintjac(x):
                """Need derivative w.r.t. all unknowns, as in order of x[]"""
                y = all_mixed_hitask_utils / (x[:-1] ** 2)
                # Concat [-1] as derivative of low mode constraint function
                # w.r.t. U_L^L; Notebook p.226
                return np.concatenate([y, [-1]])

            return constraintfun, constraintjac

        def generate_single_himode_constraint(ts, taskj):
            """Generates an inequality constraint function and deriviative
            for SLSQP solver.

            The constraint is for the high criticality mode and allows high
            criticality task j to overrun.
            """
            hitaskids = sorted(ts.hitaskids)
            u_taskj = ts.get_task_by_id(taskj).utilization1
            other_hitask_utils = np.zeros_like(hitaskids, dtype=np.float64)
            for i, taski in enumerate(hitaskids):
                if taski == taskj:
                    other_hitask_utils[i] = 0
                else:
                    u_i = ts.get_task_by_id(taski).utilization1
                    other_hitask_utils[i] = u_i
            all_high_hitask_utils = np.zeros_like(hitaskids, dtype=np.float64)
            for i, taski in enumerate(hitaskids):
                u_i = float(ts.get_task_by_id(taski).utilization1)
                all_high_hitask_utils[i] = u_i

            def constraintfun(x):
                return np.array(
                    [1 - np.sum(all_high_hitask_utils / (1 - x[:-1]))]
                )

            def constraintjac(x):
                y = (-1) * all_high_hitask_utils / ((1 - x[:-1]) ** 2)
                # Concat [0] as derivative of high mode constraint function
                # w.r.t. U_L^L; Notebook p. 227
                return np.concatenate([y, [0]])

            return constraintfun, constraintjac

        tshi = self.get_hitasks_only_tasksystem()
        if len(tshi) < 2:
            raise TasksystemError("Tasksystem lacks high criticality tasks!")
        cons = list()
        for taskj in sorted(tshi.hitaskids):
            constraintfun, constraintjac = generate_single_lowmode_constraint(
                tshi, taskj
            )
            cons.append(
                {"type": "ineq", "fun": constraintfun, "jac": constraintjac}
            )
            # himode constraints dazu
            constraintfun, constraintjac = generate_single_himode_constraint(
                tshi, taskj
            )
            cons.append(
                {"type": "ineq", "fun": constraintfun, "jac": constraintjac}
            )
        # Bounds for acceptable solution of virtual deadline scaling
        # parameters and U_1(1) as adopted from Alberto
        eps = 0.0001
        bounds = [(eps, 1 - eps)] * (len(tshi) + 1)
        initial_guess = [0.8 for _ in range(len(tshi) + 1)]
        maxiter = 500

        res = optimize.minimize(
            costfunc,
            initial_guess,
            jac=costfuncderiv,
            constraints=cons,
```

```
613             bounds=bounds,
614             method="SLSQP",
615             options={"disp": False, "maxiter": maxiter, "ftol": 1e-9},
616         )
617         return res
618 
619     def _optimize_edf_ivd_se_deadline_scales(self):
620         """Maximize utilization of low criticality tasks in low criticality
621         mode, calculating feasible deadline scalings.
622         """
623 
624         def costfunc(x):
625             """We want to maximize U_1(1).
626 
627             x: Numpy array of unknowns
628             """
629             return -x[-1]
630 
631         def costfuncderiv(x):
632             """Derivative w.r.t. all virtual deadline scaling parameters is zero.
633             """
634             a = np.zeros_like(x)
635             a[-1] = -1  # dfdu11
636             return a
637 
638         def generate_single_lowmode_constraint(ts, taskj):
639             """Generates an inequality constraint function and deriviative
640             for SLSQP solver.
641 
642             The constraint is for the low criticality mode and allows high
643             criticality task j to overrun.
644 
645             taskj: Task id
646             ts: Tasksystem
647             """
648             hitaskids = sorted(ts.hitaskids)
649             u_taskj = float(ts.get_task_by_id(taskj).utilization1)
650             # Create array of high crit task utilizations in low mode
651             other_hitask_utils = np.zeros_like(hitaskids, dtype=np.float64)
652             for i, taski in enumerate(hitaskids):
653                 if taski == taskj:
654                     other_hitask_utils[i] = 0
655                 else:
656                     u_i = float(ts.get_task_by_id(taski).utilization0)
657                     other_hitask_utils[i] = u_i
658             # For constraint derivative need array of all u_i^L and for
659             # task j replaced by u_j^H
660             all_mixed_hitask_utils = np.zeros_like(hitaskids, dtype=np.float64)
661             for i, taski in enumerate(hitaskids):
662                 if taski == taskj:
663                     all_mixed_hitask_utils[i] = u_taskj
664                 else:
665                     u_i = float(ts.get_task_by_id(taski).utilization0)
666                     all_mixed_hitask_utils[i] = u_i
667 
668             def _get_scale_from_id(taskids, tid):
669                 """Return index position of tid in taskids which is the same
670                 index in array of unknowns x
671                 """
672                 return taskids.index(tid)
673 
674             xjind = _get_scale_from_id(hitaskids, taskj)
675 
676             def constraintfun(x):
677                 """
678                 x: Array of unknowns, with x[-1] as U_L^L and x[:-1] as
679                 virtual deadline scales
680                 """
681                 # Sum over other_hitask_utils works because for task j we
682                 # entered 0 into it, effectively making this a sum over
683                 # all tasks except j.
684                 return np.array(
685                     [
```

```
                        1
                        - x[-1]
                        - u_taskj / x[xjind]
                        - np.sum(other_hitask_utils / x[:-1])
                    ]
                )

            def constraintjac(x):
                """Need derivative w.r.t. all unknowns, as in order of x[]"""
                y = all_mixed_hitask_utils / (x[:-1] ** 2)
                # Concat [-1] as derivative of low mode constraint function
                # w.r.t. U_L^L; Notebook p.226
                return np.concatenate([y, [-1]])

            return constraintfun, constraintjac

        def generate_single_himode_constraint(ts, taskj):
            """Generates an inequality constraint function and deriviative
            for SLSQP solver.

            The constraint is for the high criticality mode and allows high
            criticality task j to overrun.
            """
            hitaskids = sorted(ts.hitaskids)
            u_taskj = ts.get_task_by_id(taskj).utilization1
            other_hitask_utils = np.zeros_like(hitaskids, dtype=np.float64)
            for i, taski in enumerate(hitaskids):
                if taski == taskj:
                    other_hitask_utils[i] = 0
                else:
                    u_i = ts.get_task_by_id(taski).utilization1
                    other_hitask_utils[i] = u_i
            all_high_hitask_utils = np.zeros_like(hitaskids, dtype=np.float64)
            for i, taski in enumerate(hitaskids):
                u_i = float(ts.get_task_by_id(taski).utilization1)
                all_high_hitask_utils[i] = u_i
            all_low_hitask_utils = np.zeros_like(hitaskids, dtype=np.float64)
            for i, taski in enumerate(hitaskids):
                u_i = float(ts.get_task_by_id(taski).utilization0)
                all_low_hitask_utils[i] = u_i

            def constraintfun(x):
                return np.array(
                    [
                        1
                        - np.sum(
                            all_high_hitask_utils
                            / (1 - x[:-1] + all_low_hitask_utils)
                        )
                    ]
                )

            def constraintjac(x):
                y = (
                    (-1)
                    * all_high_hitask_utils
                    / ((1 - x[:-1] + all_low_hitask_utils) ** 2)
                )
                # Concat [0] as derivative of high mode constraint function
                # w.r.t. U_L^L; Notebook p. 227
                return np.concatenate([y, [0]])

            return constraintfun, constraintjac

        tshi = self.get_hitasks_only_tasksystem()
        if len(tshi) < 2:
            raise TasksystemError("Tasksystem lacks high criticality tasks!")
        cons = list()
        for taskj in sorted(tshi.hitaskids):
            constraintfun, constraintjac = generate_single_lowmode_constraint(
                tshi, taskj
            )
            cons.append(
```

```
                {"type": "ineq", "fun": constraintfun, "jac": constraintjac}
            )
            # himode constraints dazu
            constraintfun, constraintjac = generate_single_himode_constraint(
                tshi, taskj
            )
            cons.append(
                {"type": "ineq", "fun": constraintfun, "jac": constraintjac}
            )
        # Bounds for acceptable solution of virtual deadline scaling
        # parameters and U_1(1) as adopted from Alberto
        eps = 0.0001
        bounds = [(eps, 1 - eps)] * (len(tshi) + 1)
        initial_guess = [0.8 for _ in range(len(tshi) + 1)]
        maxiter = 500

        res = optimize.minimize(
            costfunc,
            initial_guess,
            jac=costfuncderiv,
            constraints=cons,
            bounds=bounds,
            method="SLSQP",
            options={"disp": False, "maxiter": maxiter, "ftol": 1e-9},
        )
        return res

    def _is_edf_ivd_se_schedulable_strict(self, res=None, supply=1.0):
        """Schedulable if numerical result exists.
        """
        if res is None:
            res = self._optimize_edf_ivd_se_deadline_scales()
        uhh = float(self._uhh(self.hitaskids))
        ull = float(self._ull(self.hitaskids))
        if res.success and (res.x[-1] >= ull) and (uhh <= supply):
            return True
        return False

    def is_edf_ivd_schedulable(self, res=None):
        """Schedulable if numerical result exists.
        """
        if res is None:
            res = self._optimize_edf_ivd_deadline_scales()
        return res.success  # success looks like a boolean

    def is_edf_ivd_se_schedulable(self, res=None):
        """Schedulable if numerical result exists.
        """
        if res is None:
            res = self._optimize_edf_ivd_se_deadline_scales()
        return res.success  # success looks like a boolean

    def _optimize_edf_ivd_deadline_scales(self):
        """Maximize utilization of low criticality tasks in low criticality
        mode, calculating feasible deadline scalings for EDF-IVD.
        """

        def costfunc(x):
            """We want to maximize U_1(1), or the utilization of all
            low crit. tasks in low mode.

            x: Numpy array of unknowns
            """
            return -x[-1]

        def costfuncderiv(x):
            """Derivative w.r.t. all virtual deadline scaling parameters
            is zero.
            """
            a = np.zeros_like(x)
            a[-1] = -1  # dfdu11
            return a

```

```
832         def generate_lowmode_constraint(ts):
833             """Generates an inequality constraint function and deriviative
834             for SLSQP solver.
835
836             The constraint is for the low criticality mode:
837
838             $1 - U_L^L - \sum_{i:\chi_i = H} u_i^L/x_i \geq 0$
839
840             ts: Tasksystem
841             """
842             hitaskids = sorted(ts.hitaskids)
843             # Create array of high crit task utilizations in low mode
844             # Remark regarding single error approaches: This separated u_j
845             # and all remaining u_i because u_j was treated differently.
846             hitask_utils = np.zeros_like(hitaskids, dtype=np.float64)
847             for i, taski in enumerate(hitaskids):
848                 u_i = float(ts.get_task_by_id(taski).utilization0)
849                 hitask_utils[i] = u_i
850
851             def constraintfun(x):
852                 """
853                 x: Array of unknowns, with x[-1] as U_L^L and x[:-1] as virtual
854                 deadline scales
855                 """
856                 return np.array([1 - x[-1] - np.sum(hitask_utils / x[:-1])])
857
858             def constraintjac(x):
859                 """Need derivative w.r.t. all unknowns, as in order of x[]"""
860                 # See tr09-edf-ivd-numerical-solution.pdf.xopp
861                 y = hitask_utils / (x[:-1] ** 2)
862                 # Concat [-1] as derivative of low mode constraint function
863                 # w.r.t. U_L^L
864                 return np.concatenate([y, [-1]])
865
866             return constraintfun, constraintjac
867
868         def generate_himode_constraint(ts):
869             """Generates an inequality constraint function and deriviative
870             for SLSQP solver.
871
872             The constraint is for the high criticality mode:
873
874             $1 - \sum_{i:\chi_i = H} u_i^H / (1-x_i+u_i^L) \geq 0$
875             """
876             hitaskids = sorted(ts.hitaskids)
877             hitask_utils_hi = np.zeros_like(hitaskids, dtype=np.float64)
878             hitask_utils_lo = np.zeros_like(hitaskids, dtype=np.float64)
879             for i, taski in enumerate(hitaskids):
880                 u_i = float(ts.get_task_by_id(taski).utilization0)
881                 hitask_utils_lo[i] = u_i
882                 u_i = float(ts.get_task_by_id(taski).utilization1)
883                 hitask_utils_hi[i] = u_i
884
885             def constraintfun(x):
886                 return np.array(
887                     [
888                         1
889                         - np.sum(
890                             hitask_utils_hi / (1 - x[:-1] + hitask_utils_lo)
891                         )
892                     ]
893                 )
894
895             def constraintjac(x):
896                 # See tr09-edf-ivd-numerical-solution.pdf.xopp
897                 y = (
898                     (-1)
899                     * hitask_utils_hi
900                     / ((1 - x[:-1] + hitask_utils_lo) ** 2)
901                 )
902                 # Concat [0] as derivative of high mode constraint function
903                 # w.r.t. U_L^L
904                 return np.concatenate([y, [0]])
```

```
            return constraintfun, constraintjac

        tshi = self.get_hitasks_only_tasksystem()
        if len(tshi) < 2:
            raise TasksystemError("Tasksystem lacks high criticality tasks!")
        cons = list()
        constraintfun, constraintjac = generate_lowmode_constraint(self)
        cons.append(
            {"type": "ineq", "fun": constraintfun, "jac": constraintjac}
        )
        constraintfun, constraintjac = generate_himode_constraint(self)
        cons.append(
            {"type": "ineq", "fun": constraintfun, "jac": constraintjac}
        )

        # Bounds for acceptable solution of virtual deadline scaling
        # parameters and U_1(1) as adopted from Alberto
        eps = 0.0001
        bounds = [(eps, 1 - eps)] * (len(tshi) + 1)
        initial_guess = [0.8 for _ in range(len(tshi) + 1)]
        maxiter = 500

        res = optimize.minimize(
            costfunc,
            initial_guess,
            jac=costfuncderiv,
            constraints=cons,
            bounds=bounds,
            method="SLSQP",
            options={"disp": False, "maxiter": maxiter, "ftol": 1e-9},
        )
        return res

    def _optimize_edf_ids_deadline_scales(self):
        """Maximize utilization of low criticality tasks in low criticality
        mode, calculating feasible deadline scalings for EDF-IDS.
        """

        def costfunc(x):
            """We want to maximize U_1(1), or the utilization of all low
            crit. tasks in low mode.

            x: Numpy array of unknowns
            """
            return -x[-1]

        def costfuncderiv(x):
            """Derivative w.r.t. all virtual deadline scaling parameters
            is zero.
            """
            a = np.zeros_like(x)
            a[-1] = -1  # dfdu11
            return a

        def generate_lowmode_constraint(ts):
            """Generates an inequality constraint function and deriviative
            for SLSQP solver.

            The constraint is for the low criticality mode:

            $1 - U_L^L - \sum_{i:\chi_i = H} u_i^L/x_i \geq 0$

            ts: Tasksystem
            """
            hitaskids = sorted(ts.hitaskids)
            # Create array of high crit task utilizations in low mode
            # Remark regarding single error approaches: This separated u_j
            # and all remaining u_i because u_j was treated differently.
            hitask_utils = np.zeros_like(hitaskids, dtype=np.float64)
            for i, taski in enumerate(hitaskids):
                u_i = float(ts.get_task_by_id(taski).utilization0)
                hitask_utils[i] = u_i
```

```
978
979             def constraintfun(x):
980                 """
981                 x: Array of unknowns, with x[-1] as U_L^L and x[:-1] as
982                 virtual deadline scales
983                 """
984                 return np.array([1 - x[-1] - np.sum(hitask_utils / x[:-1])])
985
986             def constraintjac(x):
987                 """Need derivative w.r.t. all unknowns, as in order of x[]"""
988                 # See tr09-edf-ivd-numerical-solution.pdf.xopp
989                 y = hitask_utils / (x[:-1] ** 2)
990                 # Concat [-1] as derivative of low mode constraint function
991                 # w.r.t. U_L^L
992                 return np.concatenate([y, [-1]])
993
994             return constraintfun, constraintjac
995
996         def generate_himode_constraints(ts):
997             """Generates multiple inequality constraint functions and
998             deriviatives for SLSQP solver.
999
1000            The constraint is for the high criticality mode, for all h
1001
1002            $1 - x_h - (u_h^H - u_h^L) - x_h \sum_{i:\chi_i = H, i \neq h}
1003            (u_i^H - u_i^L) / (x_i) \geq 0$
1004
1005            """
1006            hitaskids = sorted(ts.hitaskids)
1007            hitask_utils_hi = np.zeros_like(hitaskids, dtype=np.float64)
1008            hitask_utils_lo = np.zeros_like(hitaskids, dtype=np.float64)
1009            for i, taski in enumerate(hitaskids):
1010                u_i = float(ts.get_task_by_id(taski).utilization0)
1011                hitask_utils_lo[i] = u_i
1012                u_i = float(ts.get_task_by_id(taski).utilization1)
1013                hitask_utils_hi[i] = u_i
1014            hitask_utils_diff = hitask_utils_hi - hitask_utils_lo
1015
1016            static_diffs = list()
1017            for xindex in range(len(hitaskids)):
1018                # To generate the sum over hitasks except for task h set the
1019                # difference at the index for task h to zero.
1020                diff_wo_h = copy.deepcopy(hitask_utils_diff)
1021                diff_wo_h[xindex] = 0.0
1022                static_diffs.append(diff_wo_h)
1023            static_diffs = tuple(static_diffs)
1024
1025            def constraintjacgeneral(xindex):
1026                """Returns function which depends on x and returns an np
1027                array, xindex defines the context"""
1028
1029                def constraintjac(x):
1030                    y = x[xindex] * static_diffs[xindex] / x[:-1] ** 2
1031                    y[xindex] = -1 - np.sum(static_diffs[xindex] / x[:-1])
1032                    return np.concatenate([y, [0]])
1033
1034                return constraintjac
1035
1036            constraintfuns = list()
1037            constraintjacs = list()
1038            for xindex in range(len(hitaskids)):
1039                constraintfuns.append(
1040                    lambda x: np.array(
1041                        [
1042                            1
1043                            - x[xindex]
1044                            - (
1045                                hitask_utils_hi[xindex]
1046                                - hitask_utils_lo[xindex]
1047                            )
1048                            - x[xindex] * np.sum(static_diffs[xindex] / x[:-1])
1049                        ]
1050                    )
```

```
                )

                # for each task h i need an np array with functions for
                # derrivatives to each x, where x != xh is different to the
                # derivative for xh, and I need the derivative wrt ULL which is
                # zero.
                constraintjacs.append(constraintjacgeneral(xindex))

            return constraintfuns, constraintjacs

        tshi = self.get_hitasks_only_tasksystem()
        if len(tshi) < 2:
            raise TasksystemError("Tasksystem lacks high criticality tasks!")
        cons = list()
        constraintfun, constraintjac = generate_lowmode_constraint(self)
        cons.append(
            {"type": "ineq", "fun": constraintfun, "jac": constraintjac}
        )
        constraintfuns, constraintjacs = generate_himode_constraints(self)
        for cf, cj in zip(constraintfuns, constraintjacs):
            cons.append({"type": "ineq", "fun": cf, "jac": cj})

        eps = 0.0001
        bounds = [(eps, 1 - eps)] * (len(tshi) + 1)
        initial_guess = [0.8 for _ in range(len(tshi) + 1)]
        maxiter = 500

        res = optimize.minimize(
            costfunc,
            initial_guess,
            jac=costfuncderiv,
            constraints=cons,
            bounds=bounds,
            method="SLSQP",
            options={"disp": False, "maxiter": maxiter, "ftol": 1e-9},
        )
        return res

    def is_edf_ids_schedulable(self, res=None, supply=1.0):
        """Schedulable if numerical result exists.
        """
        if res is None:
            res = self._optimize_edf_ids_deadline_scales()
        uhh = float(self._uhh(self.hitaskids))
        ull = float(self._ull(self.hitaskids))
        if res.success and (res.x[-1] >= ull) and (uhh <= supply):
            return True
        return False

    def _optimize_edf_vd_se_deadline_scales(self):
        """Maximize utilization of low criticality tasks in low criticality
        mode, calculating feasible deadline scalings.
        """

        def costfunc(x):
            """We want to maximize U_1(1).

            x: Numpy array of unknowns
            """
            return -x[-1]

        def costfuncderiv(x):
            """Derivative w.r.t. all virtual deadline scaling parameters
            is zero.
            """
            a = np.zeros_like(x)
            a[-1] = -1  # dfdu11
            return a

        def generate_single_lowmode_constraint(ts, taskj):
            """Generates an inequality constraint function and deriviative
            for SLSQP solver.

```

```
            The constraint is for the low criticality mode and allows high
            criticality task j to overrun.
            """
            hitaskids = sorted(ts.hitaskids)
            u_taskj = float(ts.get_task_by_id(taskj).utilization1)
            other_hitask_utils = np.zeros_like(hitaskids, dtype=np.float64)
            for i, taski in enumerate(hitaskids):
                if taski == taskj:
                    other_hitask_utils[i] = 0
                else:
                    u_i = float(ts.get_task_by_id(taski).utilization0)
                    other_hitask_utils[i] = u_i

            def constraintfun(x):
                return np.array(
                    [1 - x[-1] - u_taskj - np.sum(other_hitask_utils / x[0])]
                )

            def constraintjac(x):
                y = np.sum(other_hitask_utils) / (x[0] ** 2)
                return np.array([y, -1])

            return constraintfun, constraintjac

        def generate_single_himode_constraint(ts, taskj):
            """Generates an inequality constraint function and deriviative
            for SLSQP solver.

            The constraint is for the high criticality mode and allows high
            criticality task j to overrun.
            """
            hitaskids = sorted(ts.hitaskids)
            u_taskj = ts.get_task_by_id(taskj).utilization1
            hitask_utils = np.zeros_like(hitaskids, dtype=np.float64)
            for i, taski in enumerate(hitaskids):
                u_i = ts.get_task_by_id(taski).utilization1
                hitask_utils[i] = u_i

            def constraintfun(x):
                return np.array([1 - x[0] * x[1] - np.sum(hitask_utils)])

            def constraintjac(x):
                return np.array([-x[1], -x[0]])

            return constraintfun, constraintjac

        tshi = self.get_hitasks_only_tasksystem()
        if len(tshi) < 2:
            raise TasksystemError("Tasksystem lacks high criticality tasks!")
        cons = list()
        for taskj in sorted(tshi.hitaskids):
            constraintfun, constraintjac = generate_single_lowmode_constraint(
                tshi, taskj
            )
            cons.append(
                {"type": "ineq", "fun": constraintfun, "jac": constraintjac}
            )
            # Add himode constraint
            # TODO: Move out of loop, constraint is not dependent on task
            # and one constraint for high mode is enough
            constraintfun, constraintjac = generate_single_himode_constraint(
                tshi, taskj
            )
            cons.append(
                {"type": "ineq", "fun": constraintfun, "jac": constraintjac}
            )
        eps = 0.0001
        bounds = [(eps, 1 - eps)] * 2
        initial_guess = [0.8, 0.8]
        maxiter = 500

        res = optimize.minimize(
            costfunc,
```

```
            initial_guess,
            jac=costfuncderiv,
            constraints=cons,
            bounds=bounds,
            method="SLSQP",
            options={"disp": False, "maxiter": maxiter, "ftol": 1e-9},
        )
        return res

    def is_edf_vd_se_schedulable(self, res=None):
        """Schedulable if result from numerical optimization and static check
        agree.
        """
        if res is None:
            res = self._optimize_edf_vd_se_deadline_scales()
        if res.success:
            U11 = res.x[-1]
            lotask = _generate_locrit_task(U11)
            checkts = [t for t in self if not t.locrit]
            checkts.append(lotask)
            checkts = Tasksystem(checkts)
            return checkts._edf_vd_se_ok()
        return False

    def _edf_vd_se_ok(self):
        """Reuse EDF-NUVD check with adjusted supply to check if task system
        is schedulable under EDF-NUVD-SE.
        """
        highcrittasks = [t for t in self if not t.locrit]
        for t_j in highcrittasks:
            reducedts = [t for t in self if t != t_j]
            reducedts = Tasksystem(reducedts)
            supply = 1.0 - float(t_j.utilization1)
            if not reducedts.is_edf_vd_schedulable(
                reducedts.hitaskids, supply=supply
            ):
                return False
        return True
```

End of code

3 Reproducible simulation experiment infrastructure

Random task sets generation (`threadysequencer/generator.py`):

Begin of code

```
"""Random task and task system generation."""
import random
import math
from fractions import Fraction
from .tasksystem import RTask, Tasksystem, TasksystemError
from .templates import BARUAH2012FIG2


class TaskGenerationError(Exception):
    pass


def uunifast(num_bins, maxutil):
    """Distribute utilization uniformly over number of bins.

    Parameters
    ----------
    num_bins : int
        Number of bins
    maxutil : float
        Utilization to distribute over number of bins


    Returns
    -------
    tuple
```

```
        A n-tuple of utilizations
    """
    utils = list()
    sumu = maxutil
    for i in range(num_bins - 1):
        r = random.uniform(0.0, 1.0)
        nextsumu = sumu * r ** (1.0 / (num_bins - (i + 1)))
        utils.append(sumu - nextsumu)
        sumu = nextsumu
    utils.append(sumu)
    return tuple(utils)


def to_int(f):
    return int(math.floor(f))


def uniformint(a, b):
    return to_int(random.uniform(a, b))


def generate_task(parameters, taskid, utilization, random_lowcrit=False):
    """Generate a single random RTask.

    Generate a single random task similar to [Guan2013].

    Parameters
    ----------
    parameters : dict
        A dictionary of distribution parameters and probabilities
    taskid : int
        ID of task to generate
    utilization : float
        Target utilization for the task
    random_lowcrit: Boolean
        Make the task a low criticality task with a probability of 0.5

    Returns
    -------
    task, delta_u : tuple
        Task object and utilization difference of task to requested one
    """
    assert utilization > 0.0
    period = uniformint(parameters["p_l"], parameters["p_u"])
    computation0 = 1
    computation1 = to_int(utilization * period)
    dtpr = random.uniform(parameters["dtpr_l"], parameters["dtpr_u"])
    reldead = to_int(period * dtpr)
    delta_c = utilization * period - computation1
    delta_u = delta_c / period

    computation2 = computation1 + 1
    z = random.uniform(parameters["z_l"], parameters["z_u"])
    computation3 = to_int(z * computation1)

    probability0 = parameters["probability0"]
    probability1 = parameters["probability1"]

    if random_lowcrit and random.choice((True, False)):
        # True means we have a low criticality task
        computation2 = 0
        computation3 = 0
        probability0 = 1.0
        probability1 = 0.0

    task = RTask(
        taskid,
        period,
        reldead,
        computation0,
        computation1,
        computation2,
        computation3,
```

```
        0,
        0,  # computation4 and computation5 not used
        probability0,
        probability1,
        parameters["beta"],
        1,  # priority not considered
    )
    return task, delta_u


def generate_fraction_task(
    parameters, taskid, utilization, random_lowcrit=False
):
    """Generate a single random RTask.

    Generate a single implicit deadline random task.

    Parameters
    ----------
    parameters : dict
        A dictionary of distribution parameters and probabilities
    taskid : int
        ID of task to generate
    utilization : float
        Target utilization for the task
    random_lowcrit: Boolean
        Make the task a low criticality task with a probability of 0.5

    Returns
    -------
    task:
        Task object
    """
    assert utilization > 0.0
    f = Fraction(utilization).limit_denominator(1000)
    computation1, period = (f.numerator, f.denominator)
    if computation1 == 0:
        raise TaskGenerationError(
            f"No sensible fraction for utilization of {utilization}!"
        )
    computation0 = 1
    reldead = period

    computation2 = computation1 + 1
    z = random.uniform(parameters["z_l"], parameters["z_u"])
    computation3 = to_int(z * computation2)

    probability0 = parameters["probability0"]
    probability1 = parameters["probability1"]

    if random_lowcrit and random.choice((True, False)):
        # True means we have a low criticality task
        computation2 = 0
        computation3 = 0
        probability0 = 1.0
        probability1 = 0.0

    task = RTask(
        taskid,
        period,
        reldead,
        computation0,
        computation1,
        computation2,
        computation3,
        0,
        0,  # computation4 and computation5 not used
        probability0,
        probability1,
        parameters["beta"],
        1,  # priority not considered
    )
    return task
```

```
173
174
175 def generate_uunifast_fraction(parameters, num_tasks, maxutil):
176     """Generate random implicit deadline task system according to parameters.
177
178     Utilization in low mode $U_1(1)$ is uniformly distributed over
179     requested number of tasks. Each tasks's utilization is converted
180     to a rational number which allows to run the system on a simulator
181     like 'thready' which resorts to integer arithmetic.
182
183     Parameters
184     ----------
185     parameters : dict
186         Parameter dictionary of distribution limits
187     num_tasks : int
188         Requested number of tasks in created task system
189     maxutil : float
190         Requested utilization of task system in low mode $U_1(1)$
191
192
193     Returns
194     -------
195     Tuple
196         A random task system and a list of low criticality task ids.
197     """
198     utils = uunifast(num_tasks, maxutil)
199     ts = list()
200     lowcrit_tasks = list()
201     for i, u in enumerate(utils):
202         try:
203             task = generate_fraction_task(
204                 parameters, i, u, random_lowcrit=True
205             )
206         except TaskGenerationError:
207             continue
208         if task.locrit:
209             lowcrit_tasks.append(task.taskid)
210         ts.append(task)
211     return Tasksystem(ts), tuple(lowcrit_tasks)
212
213
214 def generate_uunifastint(parameters, num_tasks, maxutil):
215     """Generate random task system according to parameters.
216
217     Utilization in low mode $U_1(1)$ is uniformly distributed over
218     requested number of tasks. Each tasks's utilization is converted
219     to a rational number which allows to run the system on a simulator
220     like 'thready' which resorts to integer arithmetic.
221
222     Parameters
223     ----------
224     parameters : dict
225         Parameter dictionary of distribution limits
226     num_tasks : int
227         Requested number of tasks in created task system
228     maxutil : float
229         Requested utilization of task system in low mode $U_1(1)$
230
231
232     Returns
233     -------
234     Tuple
235         A random task system, the number of missing tasks, and a list
236         of low criticality task ids.
237     """
238     utils = uunifast(num_tasks, maxutil)
239     ts = list()
240     delta_u_accum = 0.0
241     lowcrit_tasks = list()
242     for i, u in enumerate(utils):
243         task, delta_u = generate_task(parameters, i, u, random_lowcrit=True)
244         if (task.computation1 >= task.computation0) and (
245             task.computation3 >= task.computation2
```

```
        ):
            if task.locrit:
                lowcrit_tasks.append(task.taskid)
            delta_u_accum += delta_u
            ts.append(task)
        else:
            delta_u_accum += u
    task, delta_u = generate_task(parameters, -1, delta_u_accum)
    ts.append(task)
    missing = num_tasks - len(ts)
    return Tasksystem(ts), missing, tuple(lowcrit_tasks)


def is_worstcase_schedulable(ts):
    """Check if we can use a worst case reservation and EDF.
    """
    hti = ts.hitaskids
    return (ts._uhh(hti) + ts._ull(hti)) <= 1


def no_scales_necessary(ts):
    """Check if task system works with single overrun and all virtual deadline
    scales set to one.

    This function is used to filter randomly generated task systems.
    """
    Ull0rig = float(ts._ull(ts.hitaskids))
    hitaskids = sorted(ts.hitaskids)
    for taskj in hitaskids:
        u_taskj = float(ts.get_task_by_id(taskj).utilization1)
        ULO = Ull0rig + u_taskj
        UHI = u_taskj
        for i, taski in enumerate(hitaskids):
            if taski == taskj:
                u_i = 0
            else:
                u_i = float(ts.get_task_by_id(taski).utilization1)
            ULO += u_i
            UHI += u_i
    return (ULO <= 1) and (UHI <= 1)


def generate_nice_ts(util, trials=128, template=BARUAH2012FIG2):
    """Generate a random task system which is not trivial and contains at
    least two high criticality tasks.
    """
    generation_trials = 0
    while generation_trials < trials:
        ts, _ = generate_uunifast_fraction(
            template, random.randint(3, 32), util
        )
        hitaskids = ts.hitaskids
        if (
            len(hitaskids) < 2
            or is_worstcase_schedulable(ts)
            or no_scales_necessary(ts)
        ):  # or not ts.is_edf_nuvd_schedulable(hitaskids))):

            generation_trials += 1
            continue
        else:
            # now i have a task system i can optimize for
            return ts
    raise TasksystemError("Could not generate random task system")
```

End of code

Collection of parameterizations for different kinds of random task sets found in the literature (`threadysequencer/templates.py`):

Begin of code

```
BARUAH2012FIG2 = {
    "p_l": 50,
    "p_u": 200,
```

```
    "dtpr_l": 1.0,
    "dtpr_u": 1.0,
    "z_l": 1,
    "z_u": 2,
    "probability0": 0.95,
    "probability1": 0.05,
    "beta": 0.001,
}

BARUAH2012FIG2LOWPROB = {
    "p_l": 50,
    "p_u": 200,
    "dtpr_l": 1.0,
    "dtpr_u": 1.0,
    "z_l": 1,
    "z_u": 2,
    "probability0": 0.999,
    "probability1": 0.001,
    "beta": 10.0,
}

Z2P3 = {
    "p_l": 50,
    "p_u": 200,
    "dtpr_l": 1.0,
    "dtpr_u": 1.0,
    "z_l": 2,
    "z_u": 2,
    "probability0": 0.999,
    "probability1": 0.001,
    "beta": 0.0001,
}

Z3P3 = {
    "p_l": 50,
    "p_u": 200,
    "dtpr_l": 1.0,
    "dtpr_u": 1.0,
    "z_l": 3,
    "z_u": 3,
    "probability0": 0.999,
    "probability1": 0.001,
    "beta": 0.0001,
}

Z4P3 = {
    "p_l": 50,
    "p_u": 200,
    "dtpr_l": 1.0,
    "dtpr_u": 1.0,
    "z_l": 4,
    "z_u": 4,
    "probability0": 0.999,
    "probability1": 0.001,
    "beta": 0.0001,
}

Z2P5 = {
    "p_l": 50,
    "p_u": 200,
    "dtpr_l": 1.0,
    "dtpr_u": 1.0,
    "z_l": 2,
    "z_u": 2,
    "probability0": 0.99999,
    "probability1": 0.00001,
    "beta": 0.0001,
}

Z3P5 = {
    "p_l": 50,
    "p_u": 200,
    "dtpr_l": 1.0,
```

```
    "dtpr_u": 1.0,
    "z_l": 3,
    "z_u": 3,
    "probability0": 0.99999,
    "probability1": 0.00001,
    "beta": 0.0001,
}

Z4P5 = {
    "p_l": 50,
    "p_u": 200,
    "dtpr_l": 1.0,
    "dtpr_u": 1.0,
    "z_l": 4,
    "z_u": 4,
    "probability0": 0.99999,
    "probability1": 0.00001,
    "beta": 0.0001,
}

HU2018 = {
    "p_l": 25,
    "p_u": 1000,
    "dtpr_l": 1.0,
    "dtpr_u": 1.0,
    "z_l": 2,
    "z_u": 2,
    "probability0": 0.9999,
    "probability1": 0.0001,
    "beta": 1.0,
}
```

End of code

Distribution and collection of simulation results (`threadysequencer/main.py`):

Begin of code

```
import json
import os
import datetime
import hashlib

# import logging
import random
import math
import copy
import subprocess
import shlex
import shutil
import sys
import pandas as pd
from pathlib import Path

from numpy import arange

from threadysequencer.tasksystem import (
    Tasksystem,
    TasksystemError,
    _generate_locrit_task,
    RTask,
)
from threadysequencer.item import get_callstring, SERVERFILETEMPLATE
from threadysequencer.rawutils import (
    get_simulation_end,
    get_deadlinemiss,
    get_overrunning_taskid,
)
from threadysequencer.templates import (
    BARUAH2012FIG2,
    BARUAH2012FIG2LOWPROB,
    HU2018,
    Z2P3,
    Z3P3,
    Z4P3,
```

```
    Z2P5,
    Z3P5,
    Z4P5,
)
from threadysequencer.generator import generate_uunifast_fraction
from threadysequencer.generator import generate_nice_ts


random.seed(42)


def progress(count, total, status=""):
    """A simple progress indicator."""
    percents = round(100.0 * count / float(total), 1)
    sys.stdout.write("%s%s %s\r" % (percents, "%", status))
    sys.stdout.flush()


def generate_hashstring(metadata):
    return hashlib.blake2b(metadata.encode("utf-8"), digest_size=20).hexdigest()


def tseq_main():
    import argparse

    parser = argparse.ArgumentParser(
        description="Two mode simulation with switch on virtual deadline miss"
    )
    parser.add_argument(
        "tslofname", help="full path to first mode task system json file"
    )
    parser.add_argument(
        "tshifname", help="full path to second mode task system json file"
    )
    parser.add_argument("simtime", type=int, help="time to simulate")
    parser.add_argument(
        "repetitions",
        type=int,
        help="number of simulations with different random traces",
    )
    parser.add_argument(
        "lojobids", type=int, nargs="+", help="jobs removed between modes"
    )
    args = parser.parse_args()

    today = datetime.date.today()
    tasksystemlo = Tasksystem.from_json(args.tslofname)
    tasksystemhi = Tasksystem.from_json(args.tshifname)

    # Find out the difference between virtual and real deadlines
    deadlinediff = dict()
    for task in tasksystemhi:
        lotasks = [t for t in tasksystemlo if t.taskid == task.taskid]
        try:
            assert lotasks[0].relativedeadline < task.relativedeadline
            deadlinediff[task.taskid] = (
                task.relativedeadline - lotasks[0].relativedeadline
            )
        except IndexError:
            raise TasksystemError(
                "Task from high mode task system not found!"
            )

    # Create the result folders
    metadata = (
        tasksystemlo.dumps()
        + tasksystemhi.dumps()
        + args.tslofname
        + args.tshifname
        + str(args.simtime)
        + str(args.repetitions)
        + str(args.lojobids)
    )
```

```
    hash = generate_hashstring(metadata)
    dirname = today.isoformat() + "-" + hash
    dirpath = Path(dirname)
    dirpath.mkdir()
    lodir = dirpath / "lomode"
    lodir.mkdir()
    hidir = dirpath / "himode"
    hidir.mkdir()

    # Prepare and run low mode simulation
    callstring = get_callstring(
        args.tslofname,
        simname=dirname,
        simtime=args.simtime,
        repetitions=args.repetitions,
    )
    call = shlex.split(callstring)
    shutil.copyfile(args.tslofname, lodir / args.tslofname)
    with open(lodir / "server", "w") as fh:
        fh.write(SERVERFILETEMPLATE)

    print("Starting LO mode simulations...")
    with subprocess.Popen(
        call, stdout=subprocess.PIPE, universal_newlines=True, cwd=str(lodir)
    ) as proc:
        i = 0
        for line in proc.stdout:
            # If intermediate output is worthwhile to analyze, do it here.
            if get_deadlinemiss(line) is not None:
                i += 1
            progress(i, args.repetitions, status="LO mode")

    # Remove all lo jobs from the state dump to continue without them
    for fpath in lodir.glob("*_dump.json"):
        with open(fpath, "r") as fhandle:
            dump = json.load(fhandle)
        hijobs = [job for job in dump["jobs"] if job[0] not in args.lojobids]
        # Adjust deadlines to original deadlines
        for i, job in enumerate(hijobs):
            taskid = job[0]
            try:
                hijobs[i][2] = hijobs[i][2] + deadlinediff[taskid]
            except KeyError:
                raise KeyError("Job id mismatch")
        dump["jobs"] = hijobs
        with open(hidir / fpath.name, "w") as fhandle:
            json.dump(dump, fhandle)

    # Run hi mode simulations
    shutil.copyfile(args.tshifname, hidir / args.tshifname)
    with open(hidir / "server", "w") as fh:
        fh.write(SERVERFILETEMPLATE)
    callstring = get_callstring(
        args.tshifname,
        simname=dirname,
        simtime=args.simtime,
        repetitions=args.repetitions,
        resume=True,
    )
    call = shlex.split(callstring)
    print("\nStarting HI mode simulations...")
    with subprocess.Popen(
        call, stdout=subprocess.PIPE, universal_newlines=True, cwd=str(hidir)
    ) as proc:
        i = 0
        for line in proc.stdout:
            # If intermediate output is worthwhile to analyze, do it here.
            if get_deadlinemiss(line) is not None:
                i += 1
            progress(i, args.repetitions, status="HI mode")
    print("\n")
```

```
def tqos_main():
    import argparse

    parser = argparse.ArgumentParser(
        description="Two mode simulation with switch on virtual deadline miss"
    )
    parser.add_argument("simtime", type=int, help="time to simulate")
    parser.add_argument(
        "repetitions",
        type=int,
        help="number of simulations with different random traces",
    )
    parser.add_argument(
        "tasksystems",
        type=int,
        help="number of task systems to generate for each utilization",
    )
    parser.add_argument("template",
        help="template for random task system generation")
    args = parser.parse_args()

    today = datetime.date.today()
    all_templates = {
        "BARUAH2012FIG2": BARUAH2012FIG2,
        "BARUAH2012FIG2LOWPROB": BARUAH2012FIG2LOWPROB,
        "HU2018": HU2018,
    }
    parameters = all_templates[args.template]
    num_tasks = 10
    utilrange = arange(0.65, 1.0, 0.05)
    num_utils = len(utilrange)
    num_modes = 2
    num_simulations = args.repetitions * args.tasksystems * num_utils * num_modes
    max_generation_trials = 10000000

    simid = 0
    for util in utilrange:
        tsid = 0
        generation_trials = 0
        while tsid < args.tasksystems:
            ts_orig, _ = generate_uunifast_fraction(parameters, num_tasks, util)
            if len(ts_orig) == 0:
                generation_trials += 1
                if generation_trials > max_generation_trials:
                    raise TasksystemError(f"Unable to find task system for {util}!")
                continue
            hitaskids = ts_orig.hitaskids
            lojobids = ts_orig.lotaskids
            if not ts_orig.is_ss01_schedulable(hitaskids):
                generation_trials += 1
                if generation_trials > max_generation_trials:
                    raise TasksystemError(f"Unable to find task system for {util}!")
                continue
            # This task system is schedulable
            lamblo, lambhi = ts_orig.calculate_ss01_lambda(hitaskids)
            lamb = random.uniform(lamblo, lambhi)

            # Create lo mode task system
            ts_lo = list()
            for t in ts_orig:
                if t.taskid in hitaskids:
                    try:
                        x = 1 / (
                            1 + lamb * math.sqrt(t.utilization1 / t.utilization0 - 1)
                        )
                    except ZeroDivisionError:
                        print("\n\n", ts_orig, "\n\n")
                        raise TasksystemError("Can't generate scaling!")
                    virtual_deadline = int(math.floor(x * t.relativedeadline))
                    task_vd = copy.deepcopy(t)
                    task_vd.relativedeadline = virtual_deadline
                    ts_lo.append(task_vd)
                else:
```

```
                ts_lo.append(t)
            tasksystemlo = Tasksystem(ts_lo)
            metadata = (
                str(args.simtime)
                + str(args.repetitions)
                + str(args.tasksystems)
                + str(lojobids)
                + str(util)
                + str(tsid)
                + str(parameters)
                + str(num_tasks)
            )
            hash = generate_hashstring(metadata)
            tslofname = today.isoformat() + "-" + hash + "-ss01-lo-ts.json"
            tasksystemlo.write_json(tslofname)
            tsorigfname = today.isoformat() + "-" + hash + "-ss01-orig-ts.json"
            ts_orig.write_json(tsorigfname)

            # Create hi mode task system
            ts_hi = [t for t in ts_orig if t.taskid in hitaskids]
            tasksystemhi = Tasksystem(ts_hi)
            tshifname = today.isoformat() + "-" + hash + "-ss01-hi-ts.json"
            tasksystemhi.write_json(tshifname)

            # Two mode simulation with parallelization of repetitions,
            # nearly identical to 'tseq_main()', refactoring possible.
            tasksystemlo = Tasksystem.from_json(tslofname)
            tasksystemhi = Tasksystem.from_json(tshifname)

            # Find out the difference between virtual and real deadlines
            deadlinediff = dict()
            for task in tasksystemhi:
                lotasks = [t for t in tasksystemlo if t.taskid == task.taskid]
                try:
                    assert lotasks[0].relativedeadline <= task.relativedeadline
                    deadlinediff[task.taskid] = (
                        task.relativedeadline - lotasks[0].relativedeadline
                    )
                except IndexError:
                    raise TasksystemError(
                        "Task from high mode task system not found!"
                    )

            dirname = today.isoformat() + "-" + hash
            dirpath = Path(dirname)
            try:
                dirpath.mkdir()
            except FileExistsError:
                print(
                    f"Tasksystem\n {ts_orig}\n with hash {hash} already exists.\n"
                )
                continue
            lodir = dirpath / "lomode"
            lodir.mkdir()
            hidir = dirpath / "himode"
            hidir.mkdir()

            # Prepare and run low mode simulation
            callstring = get_callstring(
                tslofname,
                simname=dirname,
                simtime=args.simtime,
                repetitions=args.repetitions,
            )
            call = shlex.split(callstring)
            shutil.copyfile(tslofname, lodir / tslofname)
            with open(lodir / "server", "w") as fh:
                fh.write(SERVERFILETEMPLATE)

            with subprocess.Popen(
                call, stdout=subprocess.PIPE, universal_newlines=True, cwd=str(lodir)
            ) as proc:
                i = 0
```

```
                for line in proc.stdout:
                    # If intermediate output is worthwhile to analyze, do it here.
                    if get_simulation_end(line):
                        simid += 1
                        i += 1
                    status = f"Util. {util:.2f} Tasksys. {tsid} Rep. {i} LO"
                    progress(simid, num_simulations, status=status)

            # Remove all lo jobs from the state dump to continue without them
            for fpath in lodir.glob("*_dump.json"):
                with open(fpath, "r") as fhandle:
                    dump = json.load(fhandle)
                hijobs = [job for job in dump["jobs"] if job[0] not in lojobids]
                # Adjust deadlines to original deadlines
                for i, job in enumerate(hijobs):
                    taskid = job[0]
                    try:
                        hijobs[i][2] = hijobs[i][2] + deadlinediff[taskid]
                    except KeyError:
                        raise KeyError("Job id mismatch")
                dump["jobs"] = hijobs
                with open(hidir / fpath.name, "w") as fhandle:
                    json.dump(dump, fhandle)

            # Run hi mode simulations
            shutil.copyfile(tshifname, hidir / tshifname)
            with open(hidir / "server", "w") as fh:
                fh.write(SERVERFILETEMPLATE)
            callstring = get_callstring(
                tshifname,
                simname=dirname,
                simtime=args.simtime,
                repetitions=args.repetitions,
                resume=True,
            )
            call = shlex.split(callstring)
            with subprocess.Popen(
                call, stdout=subprocess.PIPE, universal_newlines=True, cwd=str(hidir)
            ) as proc:
                i = 0
                for line in proc.stdout:
                    # If intermediate output is worthwhile to analyze, do it here.
                    if get_simulation_end(line):
                        i += 1
                        simid += 1
                    status = f"Util. {util:.2f} Tasksys. {tsid} Rep. {i} HI"
                    progress(simid, num_simulations, status=status)
            tsid += 1
    print("\n")


def apply_virtual_deadline_scales(ts_orig, scales, U11, individual_scales=False):
    """Scale the deadlines of high criticality tasks.

    ts_orig: original task system
    scales: virtual deadline scales sorted by increasing task id
    U11: maximum utilization of low crit tasks in low mode (from optimization)
    individual_scales: differentiate between approaches with a single deadline
    scaling parameter and individual (per high crit task) scaling parameters.

    Returns scaled task system.
    """
    # Create lo mode task system
    ts_lo = list()
    n = 0
    # Scale hi crit tasks
    ts_orig_bad_timestep = False
    for n, tid in enumerate(sorted(ts_orig.hitaskids)):
        t = ts_orig.get_task_by_id(tid)
        if individual_scales:
            x = scales[n]
        else:
            x = scales[0]
```

```
403
404         virtualdeadline = int(math.floor(t.relativedeadline * x))
405         if virtualdeadline <= t.computation2:
406             # The generated task system is a bad fit with the given time step size.
407             # A real world task system would use another timestep size for simulation.
408             # Try to find another
409             ts_orig_bad_timestep = True
410
411         task_vd = RTask(
412             t.taskid,
413             t.period,
414             virtualdeadline,
415             t.computation0,
416             t.computation1,
417             t.computation2,
418             t.computation3,
419             t.computation4,
420             t.computation5,
421             t.probability0,
422             t.probability1,
423             t.beta,
424             t.priority,
425         )
426         ts_lo.append(task_vd)
427     if ts_orig_bad_timestep:
428         raise TasksystemError("Bad timestep for tasksystem, use different tasksystem.")
429     singlelocrittask = _generate_locrit_task(U11)
430     ts_lo.append(singlelocrittask)
431     tasksystemlo = Tasksystem(ts_lo)
432     return tasksystemlo
433
434
435 def tqosse_main():
436     import argparse
437
438     parser = argparse.ArgumentParser(
439         description="Mode simulation of interesting task systems with switch on overrun"
440     )
441     parser.add_argument("simtime", type=int, help="time to simulate")
442     parser.add_argument(
443         "repetitions",
444         type=int,
445         help="number of simulations with different random traces",
446     )
447     parser.add_argument(
448         "tasksystems",
449         type=int,
450         help="number of task systems to generate for each utilization",
451     )
452     parser.add_argument("template", help="template for random task system generation")
453     parser.add_argument(
454         "scheduler", help="EDF-VD-SE, EDF-NUVD-SE, or EDF-IVD-SE"
455     )  # TODO: Allow both at once
456     args = parser.parse_args()
457     do_edfvdse = args.scheduler == "EDF-VD-SE"
458     do_edfnuvdse = args.scheduler == "EDF-NUVD-SE"
459     do_edfivdse = args.scheduler == "EDF-IVD-SE"
460     if not do_edfnuvdse and not do_edfvdse and not do_edfivdse:
461         raise NotImplementedError("Unknown approach command line argument!")
462
463     today = datetime.date.today()
464     all_templates = {
465         "BARUAH2012FIG2": BARUAH2012FIG2,
466         "BARUAH2012FIG2LOWPROB": BARUAH2012FIG2LOWPROB,
467         "HU2018": HU2018,
468         "Z2P3": Z2P3,
469         "Z3P3": Z3P3,
470         "Z4P3": Z4P3,
471         "Z2P5": Z2P5,
472         "Z3P5": Z3P5,
473         "Z4P5": Z4P5,
474     }
475     parameters = all_templates[args.template]
```

```
    num_tasks = 10
    utilrange = arange(0.65, 1.0, 0.05)
    num_utils = len(utilrange)
    num_modes = 2
    num_simulations = args.repetitions * args.tasksystems * num_utils * num_modes
    max_generation_trials = 10000000

    simid = 0
    generation_trials = 0
    for util in utilrange:
        tsid = 0
        while tsid < args.tasksystems:
            ts_orig = generate_nice_ts(util, template=parameters, trials=128)
            U11orig = float(ts_orig._ull(ts_orig.hitaskids))

            if do_edfnuvdse:
                optimizer = ts_orig._optimize_edf_nuvd_se_deadline_scales
                schedulability = ts_orig.is_edf_nuvd_se_schedulable
                individual_scales = True
                approachname = "edfnuvdse"
            elif do_edfivdse:
                optimizer = ts_orig._optimize_edf_ivd_se_deadline_scales
                schedulability = ts_orig.is_edf_ivd_se_schedulable
                individual_scales = True
                approachname = "edfivdse"
            elif do_edfvdse:
                optimizer = ts_orig._optimize_edf_vd_se_deadline_scales
                schedulability = ts_orig.is_edf_vd_se_schedulable
                individual_scales = False
                approachname = "edfvdse"
            else:
                raise NotImplementedError

            res = optimizer()
            # logging.debug(res.message)
            try:
                sched = schedulability(res=res)
            except TasksystemError:
                sched = False

            if not sched:
                # logging.debug("Tasksystem not schedulable:\n%s", ts_orig)
                generation_trials += 1
                sys.stdout.write(
                    "Random search %08d schedulable task systems\r" % generation_trials
                )
                sys.stdout.flush()
                if generation_trials > max_generation_trials:
                    raise TasksystemError(
                        "Exhausted generation trials for suitable task systems!"
                    )
                else:
                    continue

            print("\n")

            scales = res.x[:-1]
            U11 = res.x[-1]
            # positive if single error approach increases U11
            U11_delta = U11 - U11orig

            try:
                # logging.debug('Try virtual deadline scales\n%s', scales)
                tasksystemlo = apply_virtual_deadline_scales(
                    ts_orig, scales, U11, individual_scales=individual_scales
                )
            except TasksystemError:
                # logging.debug('Virtual deadline scales not suitable')
                continue

            metadata = (
                str(args.simtime)
                + str(args.repetitions)
```

```
                + str(args.tasksystems)
                + str(scales)
                + str(U11_delta)
                + str(tsid)
                + str(parameters)
                + str(num_tasks)
            )
            hash = generate_hashstring(metadata)
            tsprefix = today.isoformat() + "-" + hash
            tslofname = tsprefix + "-%s-lo-ts.json" % (approachname)
            tasksystemlo.write_json(tslofname)
            tsorigfname = tsprefix + "-%s-orig-ts.json" % (approachname)
            ts_orig.write_json(tsorigfname)

            # Create hi mode task system
            ts_hi = [t for t in ts_orig if t.taskid in ts_orig.hitaskids]
            tasksystemhi = Tasksystem(ts_hi)
            tshifname = tsprefix + "-%s-hi-ts.json" % (approachname)
            tasksystemhi.write_json(tshifname)

            # Two mode simulation with parallelization of repetitions,
            # nearly identical to 'tseq_main()', refactoring possible.
            tasksystemlo = Tasksystem.from_json(tslofname)
            tasksystemhi = Tasksystem.from_json(tshifname)

            # Find out the difference between virtual and real deadlines
            deadlinediff = dict()
            for task in tasksystemhi:
                lotasks = [t for t in tasksystemlo if t.taskid == task.taskid]
                try:
                    assert lotasks[0].relativedeadline <= task.relativedeadline
                    deadlinediff[task.taskid] = (
                        task.relativedeadline - lotasks[0].relativedeadline
                    )
                except IndexError:
                    raise TasksystemError(
                        "Task from high mode task system not found in low mode tasksystem!"
                    )

            dirname = tsprefix
            dirpath = Path(dirname)
            try:
                dirpath.mkdir()
            except FileExistsError:
                print(
                    f"Tasksystem\n {ts_orig}\n with hash {hash} already exists.\n"
                )
                continue
            lodir = dirpath / "1lomode"
            lodir.mkdir()
            sedir = dirpath / "2semode"
            sedir.mkdir()
            hidir = dirpath / "3himode"
            hidir.mkdir()

            # Prepare and run low mode simulation
            callstring = get_callstring(
                tslofname,
                simname=dirname,
                simtime=args.simtime,
                repetitions=args.repetitions,
                breakonoverrun=True,
                # allowfirst=True,  dont allow, because we need to adjust
                # the deadline of the overflowing job
            )
            call = shlex.split(callstring)
            shutil.copyfile(tslofname, lodir / tslofname)
            with open(lodir / "server", "w") as fh:
                fh.write(SERVERFILETEMPLATE)

            # Avoid complaints by perl if locale is not installed on remote
            my_env = {**os.environ, "LC_ALL": "C"}
            with subprocess.Popen(
```

```
                call,
                stdout=subprocess.PIPE,
                universal_newlines=True,
                cwd=str(lodir),
                env=my_env,
            ) as proc:
                i = 0
                for line in proc.stdout:
                    # If intermediate output is worthwhile to analyze,
                    # do it here.
                    if get_simulation_end(line):
                        simid += 1
                        i += 1
                    status = f"Util. {util:.2f} Tasksys. {tsid} Rep. {i} LO"
                    progress(simid, num_simulations, status=status)

            # Copy state dumps to working directory of next simulation,
            # adjusting the deadline of the overflowed job to its original
            # deadline.
            # Need to open simulation results csv file for messages
            # about overflowed job.
            losimresultsfname = tsprefix + ".csv"
            losimresults = pd.read_csv(lodir / losimresultsfname)
            for i in range(args.repetitions):
                taskidoverrun = get_overrunning_taskid(losimresults["Stdout"].iloc[i])
                fpath = lodir / f"{tsprefix}-{i}_dump.json"
                with open(fpath, "r") as fhandle:
                    dump = json.load(fhandle)
                # Adjust deadline of overrunning job to its original deadline
                jobsingleoriginal = list()
                for job in dump["jobs"]:
                    taskid = job[0]
                    if taskid == taskidoverrun:
                        try:
                            job[2] = job[2] + deadlinediff[taskid]
                        except KeyError:
                            raise KeyError("Job id mismatch")
                    jobsingleoriginal.append(job)

                dump["jobs"] = jobsingleoriginal
                with open(sedir / fpath.name, "w") as fhandle:
                    json.dump(dump, fhandle)

            shutil.copyfile(tslofname, sedir / tslofname)
            with open(sedir / "server", "w") as fh:
                fh.write(SERVERFILETEMPLATE)
            callstring = get_callstring(
                tslofname,
                simname=dirname,
                simtime=args.simtime,
                repetitions=args.repetitions,
                resume=True,
                resumelabel="se",
                breakonoverrun=True,
            )
            call = shlex.split(callstring)
            with subprocess.Popen(
                call,
                stdout=subprocess.PIPE,
                universal_newlines=True,
                cwd=str(sedir),
                env=my_env,
            ) as proc:
                i = 0
                for line in proc.stdout:
                    # If intermediate output is worthwhile to analyze,
                    # do it here.
                    if get_simulation_end(line):
                        i += 1
                        simid += 1
                    status = f"Util. {util:.2f} Tasksys. {tsid} Rep. {i} SE"
                    progress(simid, num_simulations, status=status)

```

```
695             tsid += 1
696     print("\n")
```
End of code