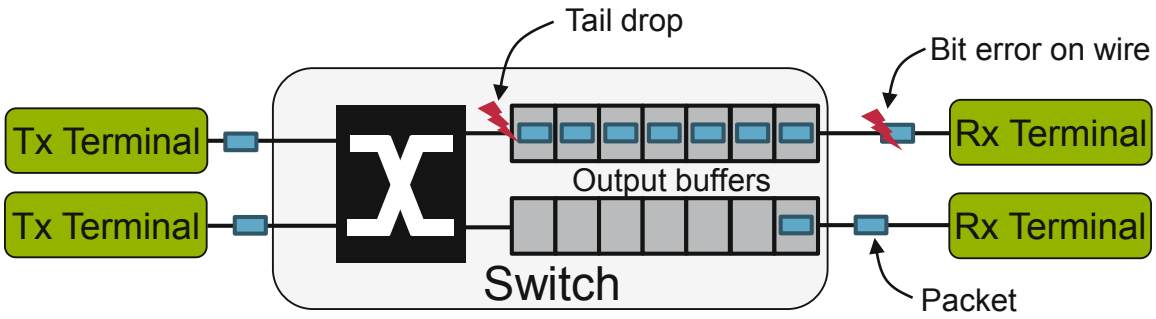
Philip Axer

Performance of Time-Critical Embedded Systems under the Influence of Errors and Error Handling Protocols



Cuvillier Verlag Göttingen
Internationaler wissenschaftlicher Fachverlag

Performance of Time-Critical Embedded Systems under the Influence of Errors and Error Handling Protocols

Performance of Time-Critical Embedded Systems under the Influence of Errors and Error Handling Protocols

Von der Fakultät für Elektrotechnik, Informationstechnik, Physik der Technischen Universität Carolo-Wilhelmina zu Braunschweig

zur Erlangung des Grades eines Doktors

der Ingenieurwissenschaften (Dr.-Ing.)

genehmigte Dissertation

von: Dipl.-Ing. Philip Axer

aus: Henstedt-Ulzburg

eingereicht am: 23.10.2015

mündliche Prüfung am: 30.11.2015

1. Referent: Prof. Dr.-Ing Rolf Ernst
2. Referent: Prof. Dr. rer nat Hermann Härtig
3. Referent: Apl. Prof. Dr.-Ing. Wael Adi (Vorsitzender)

Druckjahr: 2016

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.
1. Aufl. - Göttingen: Cuvillier, 2016
Zugl.: (TU) Braunschweig, Univ., Diss., 2016

Dissertation an der Technischen Universität Braunschweig,
Fakultät für Elektrotechnik, Informationstechnik, Physik

Nonnenstieg 8, 37075 Göttingen
Telefon: 0551-54724-0
Telefax: 0551-54724-21
www.cuvillier.de

1. Auflage, 2016
Gedruckt auf umweltfreundlichem, säurefreiem Papier
aus nachhaltiger Forstwirtschaft.

ISBN 978-3-7369-9197-2
eISBN 978-3-7369-8197-3

Acknowledgements

The research presented in this dissertation was a result of six years of intensive work at the Institute für Datentechnik und Kommunikationsnetze at the Technische Universität Braunschweig. I would like to express my sincere gratitude to the research group including the administrative staff and the workshop. Especially, I would like to thank the head of the group, my advisor, Prof. Ernst for making this possible for me. Prof. Ernst understands to connect academic research and industry like nobody else while still thinking one step ahead.

Also I would like to thank the entire research staff including all former colleagues, be it Post-Docs, Phd students as well as Bachelor and Master students. Without the numerous technical whiteboard discussions, coffee rounds, and late-night discussions this work would not have been possible. It were the people that made IDA a very special place for me.

I would like to express my special gratitude to my best friends who where always there whenever I needed them. Without the love and support of my family Ingeborg, Klaus, Steffen, Caroline my academic career and this work would never have been possible. Finally, I would like to thank Annemarie for putting joy and happiness in the time of the writing and beyond.

Kurzfassung

Sowohl Eingebettete Systeme im Allgemeinen, als auch Sicherheitskritische Systeme im Speziellen werden zunehmend komplexer. Hinzu kommt, dass aufgrund der Verkleinerung der Strukturbreite moderner Halbleiterprozesse die transiente Fehlerrate deutlich ansteigt. Daher kann nicht von einem fehlerfreien Betrieb von zukünftigen eingebetteten, sicherheitskritischen Systemen unter nominal Bedingungen ausgegangen werden.

Als Faustregel kann man zusammenfassen, dass die Schlüsselparameter im Entwurfsraum Performance, Preis und Zuverlässigkeit so gut wie immer widersprüchliche Entwurfsziele sind. Diese Arbeit zielt auf diesen Entwurfsraum ab, zeigt die Herausforderungen und diskutiert die Trade-Offs.

Von besonderem Interesse ist die Zuverlässigkeit unter Echzeitaspekten. Selbstverständlich gibt es Fehlerbehandlungsprotokolle, Fehlercodes und modulare Redundanz. Allerdings hat die Korrektur von Fehlern immer einen gewissen Einfluss auf das Zeitverhalten des gesamten Systems. Selbst, wenn ein Fehler korrigiert werden konnte, ist unklar, unter welchen Situationen das Zeitverhalten eingehalten wird. Dies kann zu der absurden Situation führen, dass ein Fehler in einem Fahrerassistenzsystem korrigiert werden kann, dennoch aber das Verpassen einer Deadline zu einem Systemfehler führt.

In dieser Arbeit stellen wir die ASTEROID Plattform vor, die im Rahmen einer Kooperation der TU Braunschweig mit der TU Dresden entstanden ist. Diese Plattform ist speziell im Hinblick auf Echtzeitaspekte, Performance, Zuverlässigkeit und damit einhergehend Sicherheit entworfen worden. ASTEROID unterscheidet sich von anderen MPSoC Plattformen durch seinen Cross-Layer Fehlerbehandlungsansatz. Die eigentliche Hardware-

plattform implementiert nur das absolute Minimum an Fehlertoleranz, um das darüber geschaltete Betriebssystem zu unterstützen. Dieses übernimmt dann die eigentliche Redundanz und erlaubt damit eine flexible Mischung von redundanten und nicht-redundanten Anwendungen.

In dieser Arbeit wird die Plattform in Bezug auf die Echtzeitperformanz unter Fehlern in einer kompositionellen Weise untersucht. Dafür werden Fehlereffekte in der on-chip und off-chip Kommunikation sowie Fehler im eigentlichen Rechenkern selbst betrachtet.

Der wissenschaftliche Beitrag dieser Arbeit liegt zum einen in einer generalisierten kompositionellen Performanz Analyse, die zudem Fehlereffekte berücksichtigt. Zum Anderen werden Ende-zu-Ende Protokolle und redundante Anwendungen modelliert und in Bezug auf ihre Echtzeitfähigkeit untersucht. Für viele der genutzten Verfahren wird auch eine Zuverlässigkeitsabschätzung des Echtzeitverhaltens bei einem gegebenen Fehlermodell durchgeführt.

Abstract

As for the entire embedded-systems domain, the complexity of safety-critical systems is growing rapidly. Additionally, the rate of errors in such devices also increases for instance due to silicon shrinking. Hence, error-free operation under in-specification operating conditions cannot be assumed for next-generation safety-critical devices.

As a rule of thumb the key design parameters for such systems performance, price and reliability are almost always contradicting design goals. This work addresses the related design space, highlights the challenges and discusses the trade-offs.

Of unique interest is the reliability under real-time aspects. Naturally, there are error-handling protocols, error-correcting codes, and modular redundancy available. However, the effect of errors always has an influence on system timing. Even if an error is handled and corrected, it remains unclear under which situations timing requirements are met. This leads to the absurd situation that a device such as an advanced driver assistance system produces correct data even under errors but fails to deliver service because hard deadlines are missed.

We present the ASTEROID architecture as a next-generation high-performance, real-time platform which addresses reliability and thus safety aspects. ASTEROID differs from other MPSoC platforms in its cross-layer error handling approach. The hardware implements the bare minimum to support the operating system with support for redundant computing, allowing the software to flexibly schedule tasks for redundant or regular execution. This architecture was joint work between TU Braunschweig and TU Dresden. In this work, we present the hardware architecture and discuss the real-time performance under errors in a compositional way.

Therefore, we consider errors in communication (be it on-chip as well as off-chip) and errors in the processing core itself.

The scientific contributions are first to extend compositional performance analysis (CPA) also by covering error effects, second to cover end-to-end error protocols with CPA, third to provide execution models and analysis for redundant execution and finally to bound the likelihood of timing violations in communication and computation under a given error model.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Embedded systems have penetrated our daily life without many of us noticing, by that blurring what we mean by an embedded system. Our day-to-day routine gets into contact with computer systems in all aspects, and some of these daily encounters depend on the correctness of embedded computers. Modern, public transportation systems offer a fully automated, unattended train operation which is capable of handling starting, stopping, door operation as well as emergency situations. Similarly, modern cars are equipped with semi-automatic driver assistance features and it is only a matter of time until autonomous driving will be the common case.

Medical devices such as wearable health technology are predicted to revolutionize medical care. Gadgets for medical therapy, sports, or just every-day fitness are capable to track brainwaves, heart rate, blood glucose level, sleep pattern, and more. Wearable devices cannot only be used to monitor and track but also to regulate for instance inject medication or stimulate nerve cells. It is predicted that by end of 2016 more than 100 million wearable medical devices are sold per year. The market for fitness related products will reach 80 million units by then.

In the context of embedded systems, *safety critical systems* play an important role in medical care, commercial aircraft, nuclear power, and weapons [153]. There are many different definitions of what safety critical precisely means. A customary meaning is given in [153] which encompasses

> "systems whose failure might endanger human life, lead to substantial economic loss or cause extensive environmental damage."

This is consistent with most readers intuition, which account railway signaling systems, flight control systems as well as steer by wire as safety critical. However, a more general definition is also given by [153] which matches the one of [13] which is based on the notion of consequences.

> "If the failure of a system could lead to consequences that are determined to be unacceptable, then the system is safety-critical."

Traditionally, safety critical systems were closed, self-contained computers systems with very limited interface to its environment. This includes systems such as the Ariane 5 rocket of which a crash can result in a financial loss of more than US$ 370 million (Cluster spacecraft incident) as well as the Boeing 777 which is equipped with several computerized systems which replaced most of the traditional mechanical and hydraulic equipment. A report by the National Transportation Safety Board to the Federal Aviation Administration (FAA) [235] describes serious problems with the glass cockpit displays which replaced the traditional analog dials and gauges. These problems have led to at least 50 in-flight incidents, some of these causing the pilots to panic due to blank displays and lost communication. Such a failure can result in the death of hundreds of passengers.

However, recently a new specimen of non-traditional safety-critical systems has emerged. Such systems are not directly linked to catastrophic hazards, but may indirectly cause them. Nowadays, the cellular phone network does not only provide a convenient way to communicate with each other, but is also the backbone for emergency service (i.e. 112/911). In most countries the cellular network serves a dual use: it is used to signal an emergency to authorities as well by the authorities themselves, mainly to coordinate the operation. The importance of the cellular infrastructure for the greater public good (saving lives, preventing fires, etc.) elevates the former convenience technology to a safety-critical level. Other non-traditional sectors include banking, (non-nuclear) electricity generation, management of water systems (i.e. desalination).

1.2 The Role of Safety Standards

In the last years, we saw a strong trend towards standardization of the entire safety life cycle. Traditional quality assurance and process management guidelines such as ISO 15504 / SPICE [139] or ISO 9001 [138] are not suitable for the development process of safety critical systems.

This is already known from the conservative avionics industry, in which software must be developed and tested according to the domain specific standard DO-178b [224] and hardware components according to DO-254 [223]. The final aircraft will only achieve FAA approval (Type Certificate), if the

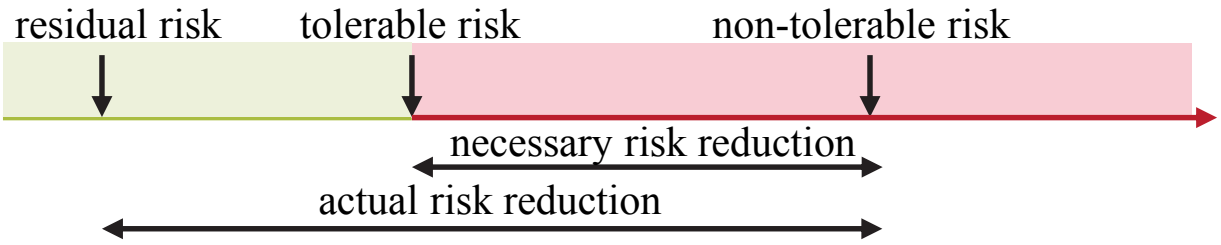


Figure 1.1: If the risk is not tolerable, additional measures such as fault-tolerance must be applied.

rules and processes of the required standards are obeyed. A similar process is compulsory for industrial plants such as power plants and heavy machinery [135].

Interestingly, such standards are rather new to the automotive domain and were not introduced prior to 2011. This has two reasons: Firstly because the consequences of a car crash are mostly considered as benign compared to a plane crash and secondly, because automotive manufacturers were keen to provide very high quality products to prevent liability issues. This changed with the introduction of the ISO 26262 [136] which is loosely based on its industrial counterpart IEC 61508. However, there are some differences. IEC 61508 is targeted towards equipment produced in low quantities, where ISO 26262 addresses volume production of the automotive market. Since then, industry puts a tremendous effort into developing a safety culture around their products.

The concept around ISO 26262 is based on risk and, as previously explained, safety is defined as the absence of unreasonable risk. Although the concept of risk seems to be very obvious, it is rather complicated to asses and systematically biased by the limitation of the human mind. The human mind tends to apply simple heuristics when risk is assessed. People are bias towards recent news and experiences which leads to a cognitive bias towards these events. This is called *availability heuristic* [277]. In order to systematically assess risk, the combination of likelihood of occurrence and the severity of the harm of a hazard must be considered. The risk is tolerable if society can accept it and safety standards guide the designer to determine and quantify the acceptance.

Figure 1.1 shows the typical case for a safety critical system. After a particular function is evaluated according to the guidelines dictated by the standard, it is evident that the risk is non-tolerable. This can be the case if standard implementations such as commercial of the shelf (COTS) hardware or software are too error prone. Thus, the actual risk which emanates from the function must be reduced by applying further measures covered for instance by using a different technology or fault tolerance

approaches. Any deployed function which is integrated in a larger system obviously still exhibits a residual risk - but safety standards guarantee that this risk remains below a tolerable threshold.

Now, it is interesting to know the additional effort required to be compliant with the state of the art safety standards, their methods and processes [244].

- Generally, risk dictates effort.
- Comparison with reference products is required.
- Assessment of known information and data must be carried out.
- Additional research is required for novel features with high risk which do not origin from previously used ancestor technology.

When designing a traditional safety critical system, the entire system context must be known. This includes the platform architecture, deployed software modules and their interaction as well as the physical boundary conditions such as worst-case environmental conditions (i.e. vibration, temperature and other stress).

1.3 Development Process for Safety-Critical Systems

To handle and master a successful safety critical embedded system, an appropriate development process is mandatory [123]. The automotive industry, especially in Germany, typically applies the V-Model [2, 247, 136].

The V-Model separates the design and specification from implementation and testing as shown in Figure 1.2. Safety standards such as the ISO 26262 have refined the V-Model and incorporated the safety requirements and safety verification into the process. This ensures traceable level of design complexity and intrinsically produces the required assurance level required by the certification agency.

Contrarily to the standard V-Model as described in [2], the V-Model as used in ISO 26262 starts with a safety assessment as a starting point. Here safety functions are identified, the risk is assessed and a high level functional safety concept is produced. The safety concept is formalized as a safety requirements specification which is later used for the functional safety assessment, to validate whether the final system satisfies all safety concerns. This typically involves a Fault Tree Analysis (FTA) [134], which is a top-down failure analysis that reveals the root cause for undesirable or catastrophic events which can be linked to the system under design [171].

In the system design step, the system architecture is specified and broken down into components with specified interfaces. This includes the hardware architecture such as communication and processing platforms

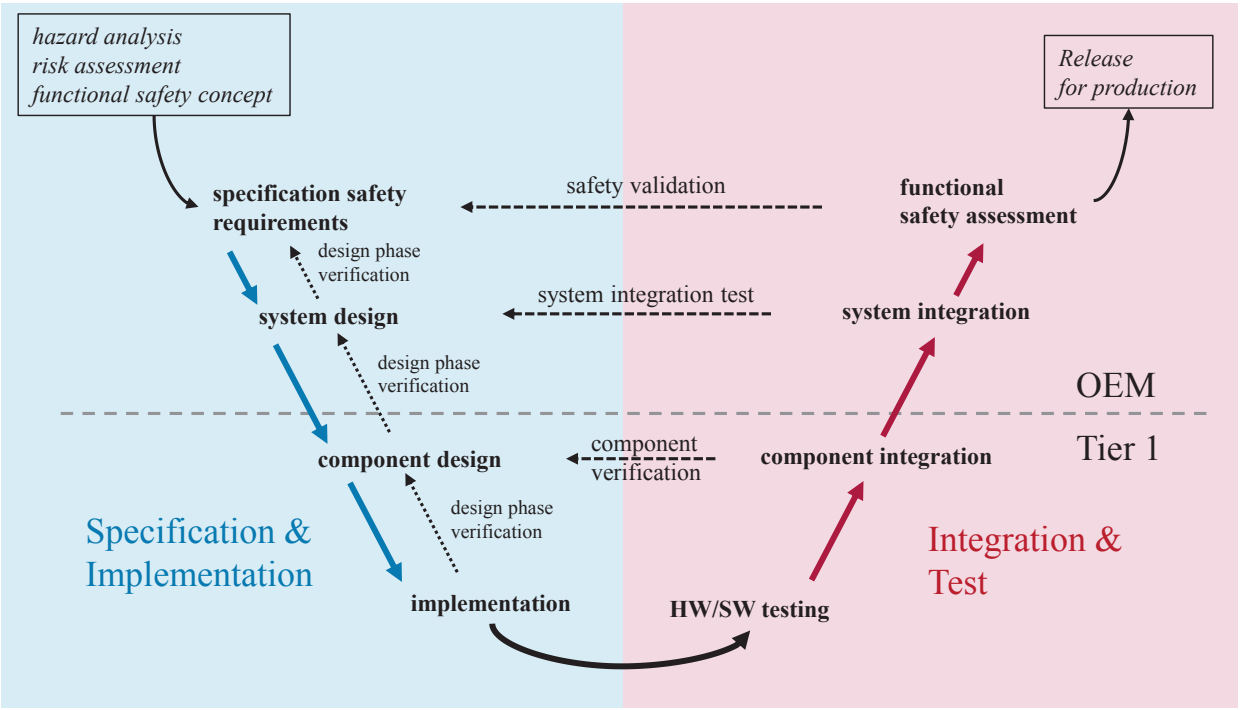


Figure 1.2: Simplified V-Model according to ISO 26262 [136].

as well as the high level software architecture. Here, safety standards recommend to capture a consistent set of requirements for instance by using Controlled Requirements Expression (CORE)[188]. Furthermore, tools, models and languages to reflect functional and non-functional behavior are strongly advised (e.g. MARTE[203], MATLAB / Simulink, AADL[238], SysML[204]).

In the component design process, individual components are broken down into function blocks which are later implemented by a programmer or hardware designer (bottom of Figure 1.2). For the hardware and software specification, ISO 26262 demands a continuous evaluation on the impact on safety. For instance, once the hardware platform is known, fault injection tests and further reliability tests should be carried out. Otherwise, there is an unknown risk of exceeding the reliability threshold and missing the safety goals. These failure tests are performed inline with test automation such as hardware-in-the-loop (HIL) tests, rest-bus simulations.

The right branch of the V-Model, Integration & Test, is responsible to verify and test the implemented functions and components against the specification. Naturally, this includes the error-free behavior as well as the service in case of errors. When the final safety validation step is completed successfully, the system can be released for production.

As shown in Figure 1.2, the component design and implementation is usually performed by the suppliers. To ease this transition step, the auto-

motive industry has standardized to automotive software and operating system interfaces in scope of the Automotive Open Systems Architecture (AUTOSAR) [12]. The concept of AUTOSAR is to focus on portability, composability and extendability where possible. Here AUTOSAR specifies a Runtime Environment which provides platform as well as communication abstraction for applications [116]. It implements well defined interfaces to connect external communication interfaces such as FLEXRAY, Controller Area Network, Ethernet and others.

AUTOSAR follows a component based design approach in which functions are encapsulated in AUTOSAR Software Components (SW-C). These components have well defined interfaces according to a standard description format. Software components are connected to a virtual function bus which abstracts the physical communication technology and allows application agnostic message passing. This allows an easy cut of system functionality into components without large overhead while maintaining a high degree of flexibility.

1.4 Trends

The industry impact of embedded systems has increased during the last decades and this trend is predicted to continue. The reason for this is that embedded computing and electronics are the main driver for features and the key for product differentiation. According to [217], the embedded systems market will reach a €1.5 trillion in revenue by 2015. The most important market segments measured by their compound annual growth rate (CAGR) are energy (45.4%), communications (13.2%), automotive (12.2%) as well as healthcare (11.4%). Interestingly the growth of consumer products is predicted to be only 6.2%. This highlights the importance of the special requirements and constraints of highly specialized domains with unique constraints such as low energy, low cost, ultra-high reliability, hard real-time under extreme environmental conditions. These market segments have to tackle the following new challenges to continue successfully their growth.

1.4.1 Architecture Complexity Challenge

There is a rapid technology advancement which enables the designer to add more and more features and functionality to the system. As a consequence, the size and complexity grows exponentially. This problem is likely getting worse, if the additional complexity cannot be conquered by compositional model-based design processes. Generally, there are two orthogonal dimensions to the complexity challenge: architecture as well as software complexity. The software complexity for a system in the automotive domain

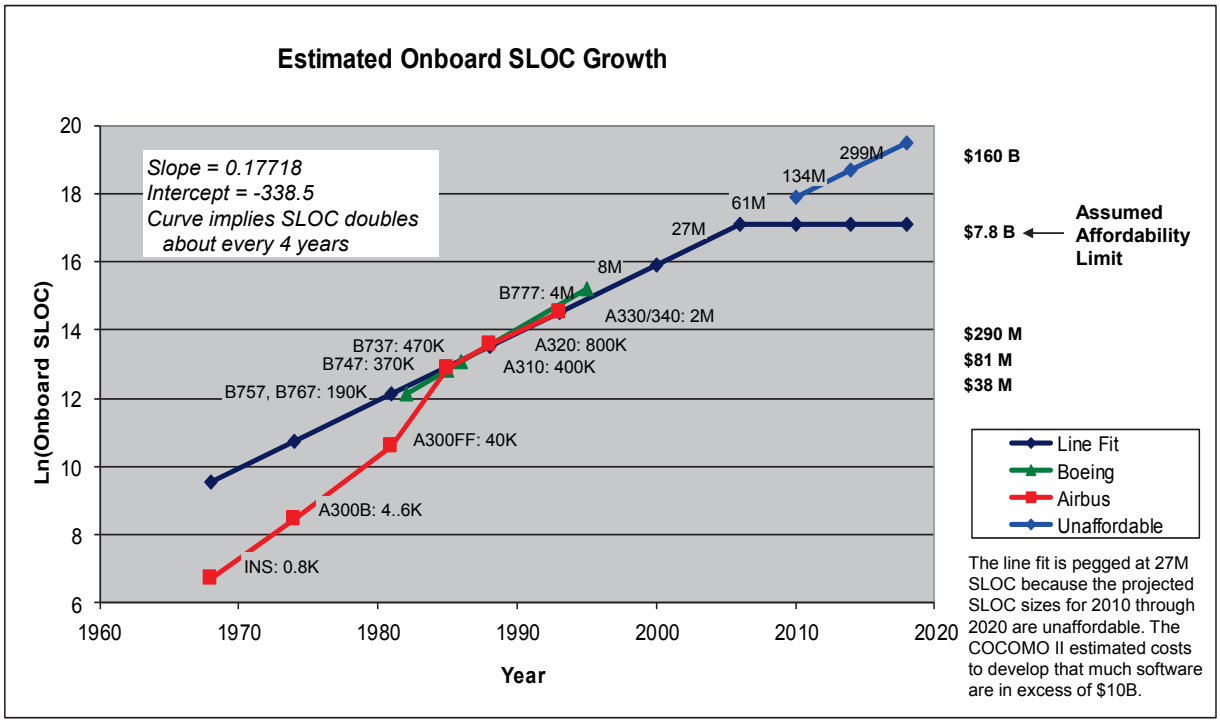


Figure 1.3: Amount of certified software code and the associated cost. (Source [285])

increased by two orders of magnitude (10^6 to 10^8 object instructions) in only 10 years which is comparable to the growth of the linux kernel during the same timespan [83]. A similar trend can be observed in the avionics industry [285], where the code size roughly doubles every year. It was estimated that by 2008 the associated costs including certification according to safety standards exceed a $ 7.8 billion threshold. This is assumed to be the affordability limit, software which exceeds 17 million lines of code is predicted to be uneconomical for aircraft designs. A modern A380 aircraft already has 100 million lines of code [287]. Handling this enormous complexity was only possible by applying formal methods such as model checking, model driven engineering on platform level as well as a compositional analysis on system level. Also standardized and modular software architectures ease the design process. Examples for such frameworks are AUTOSAR [12] used in the automotive domain as well as ARINC 653 [6] which is used in avionics.

Also the hardware platforms become more and more powerful. This advancement has boosted the data rate and processing performance required for today's and tomorrow's advanced driver assistance. Typical examples are in-vehicle navigation systems, adaptive cruise control and sophisticated camera-based precrash detection systems. The integration of multiple

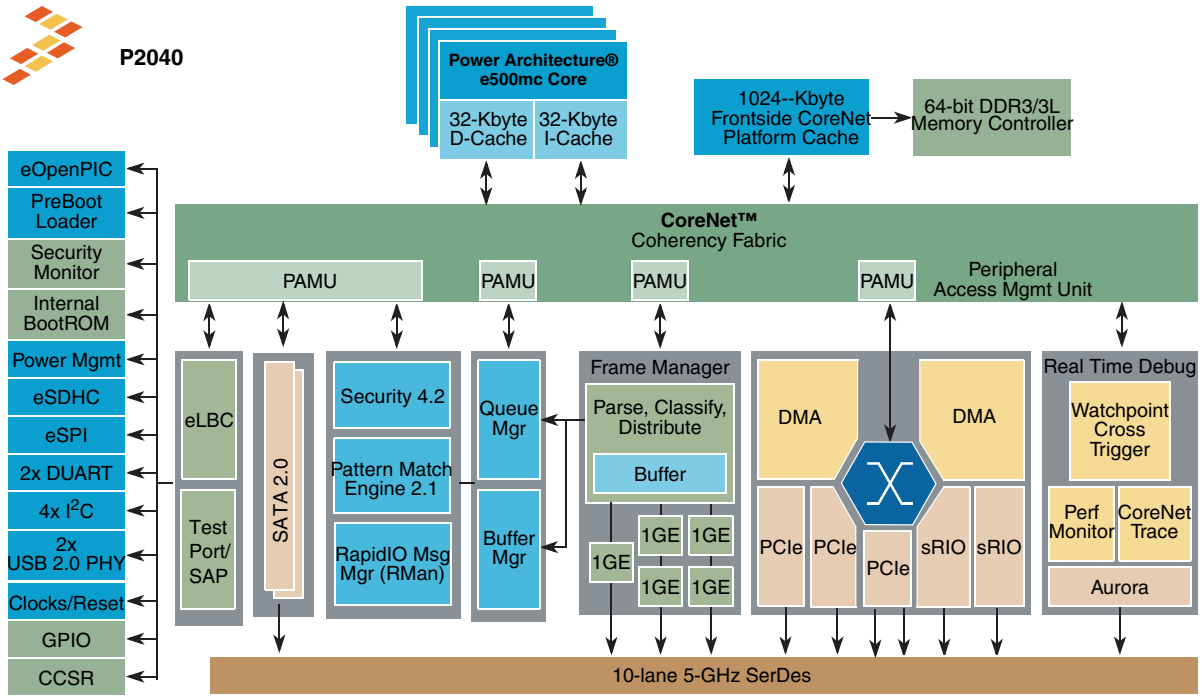


Figure 1.4: Block diagram of Freescale P2040 multi-core processor (Source [97])

processing elements allows to reduce the frequency, thus power and temperature and are an attractive design target for all computing domains. At first glance, this sounds promising as it enables the integration of tremendous complexity in the first place. The vast number of cores can potentially be used to integrate and partially isolate different functionality on such a platform. However, there is a downside: Processing elements found on multi- as well as many-core architectures share many common resources such as the communication infrastructure, caches, memory controllers and I/O ports. An example for a modern multi-core architecture is shown in Figure 1.4, a switch fabric connects all cores to shared DMA units, shared platform cache, a single DDR3 memory controller and various peripherals. This causes an easily overlooked entanglement of the timing and performance for the applications running on the platform [159]. Obviously, this inhibits a straight forward compositional consideration and leads to additional complexity during the verification stage. Compared to traditional architectures, the behavior of multi-core designs seems unpredictable and afflicted with complex to grasp timing anomalies. Therefore, traditional design processes are not applicable to multi- and many-core designs. Also, recent research has shown that the real-time performance of multi-core architectures does not necessarily outperform traditional single core designs [27] in all cases.

1.4.2 Cyber-Physical Systems Challenge

Most systems that we know today such as traffic control, health care, automotive safety, smart power grids, defense systems, environmental control and manufacturing have a tight coupling of computing and the networking infrastructure with physical processes. These systems are an integral part of the feedback loop where the physical processes affect computations and the other way around [169]. Sometime in 2008, the name cyber-physical system (CPS) was coined and serious research in this domain just started a couple of years ago.

In the physical world, the passage of time is inevitable and processes (e.g. mechanical, chemical) are concurrent by nature. Contrarily to the physical world, computation and communication models are intrinsically sequential and lack the proper abstraction. Timing and predictability was often neglected in computer science as pipeline design, caches and compiler design was tweaked to optimize the average-case performance ("make the common case fast").

A new level of abstraction must be found [26] which effectively combines computational models with models of the physical process to properly capture mutual dependencies. Here, traditional software component technologies failed as they are too software centristic. This includes operating system design, object oriented programming and service oriented architectures, because they abstract away important part of the system behavior (i.e. timing) as they try to focus only on the functional aspect of component design.

1.4.3 System of Systems Challenge

As discussed, new markets emerged such as smart electricity and water meters used for monitoring which will boost the sales of low-power, low-cost hardware. The next step is to combine embedded systems in a large scale global network of data and services. This leads to a new situation [60]: Systems of Systems (SoS) with a world of high computing density and drastically increased data rates and traffic volumes. There is no generally accepted definition for Systems of Systems. However, it is common ground that SoS are

> "themselves comprised of multiple autonomous embedded complex systems that can be diverse in technology, context, operation, geography and conceptual frame." [149].

An example for a typical SoS is the Coast Guard Deepwater Program [206] which is a 25 year program that connects recovery aircrafts, patrol boats, unmanned aerial vehicles with ground stations such as command, control and intelligence to replace almost all of today's US Coast Guard's

equipment. Other examples are FAA Air Traffic Management, Army Future Combat Systems, intelligent transport systems as well as Robotic Colonies.

In scope of the United States national space program new System of Systems engineering models and frameworks were proposed [65], which are now being adopted for non-defence related projects. These frameworks do not only account for the technological challenges but also consider the political, social and economic factors. SoS ultimately lead to heterogeneous, distributed architectures and it remains to be seen if such complex systems can be still be realized, validated and handled, if this trend continues.

1.4.4 Adaptability and Software Evolution Challenge

Today, a typical automotive vehicle design comprises many electronic control units each implementing distinct functionality (e.g. anti-lock braking unit, traction control system, emergency break assist). An upgrade of functionality is only possible through facelift upgrading or a completely new car design. Upgrading an deployed car is cumbersome and expensive: For example, the latest engine management configuration cannot be integrated without an expensive recall. On the other hand, customers have high expectations with respect to the in-vehicle infotainment system. They are used to the update cycle of entertainment products in the order of a few month. Google deploys major updates for their smartphone operating system Android every six to nine months where automotive entertainment software is never updated at all unless the customer decides to buy a new car, typically after five to six years.

Also, there is a paradigm shift towards software and network centric automotive design. New features in the automotive industry are mostly software driven and could be retrofitted into legacy cars. Such an “app store” opens up a totally new business model for OEMs and dealers.

However, the concept of software adaptability and evolution is not completely new [88]. But it has never been considered in the context of embedded as well as cyber physical system, where adaptability is inevitably linked with two conceptual problems: The *first challenge* is the competition of applications for resources. New applications share the same platform, this includes the communication infrastructure such as busses and switches as well as processors and memory. And the *second challenge* is the impact of platform and architecture change. If new hardware is added (i.e. a head up display is added to the system), other devices must be aware of the new functionality (i.e. for signal routing and configuration).

Both effects tightly couple legacy and new functionality. This is extremely problematic in domains where safety, security and availability are key constraints as such properties cannot easily be guaranteed after platform or software changes. Especially when new functionality cannot be trusted because it is developed by an unknown supplier. Novel mechanisms

which must be provided at design time must guarantee sufficient isolation, while not sacrificing flexibility and performance. Examples for such dynamic methods are load balancing, integration of quality of service and dynamic resource management [5]. Additionally these approaches must be integrated to provide feedback-based resource scheduling, middleware support for dynamic updates and new dynamic models which can be used for on-line verification. Otherwise integrity cannot be preserved and new subsystems cannot be admitted.

1.4.5 Resiliency Challenge

Continuous growth of complexity always reflects on the reliability of a system caused by nature of statistics. Interestingly, this is not a new phenomenon in computer technology. In the past, the cause for faults used to be the manufacturing and development process that impacted the quality of a product. This was tackled by testing the circuits and sorting out bad ones. Also the environment in which the device is operated affects the reliability. For instance, the soft-error rate increases with altitude. Future semiconductor devices will face new challenges [43, 36]:

- Transistor variability
- Device degradation
- Sensitivity to ionizing rays and particles

This leads to reliability problems of modern and future silicon devices which is illustrated in Figure 1.5. The graph shows the quality (i.e. speed grade) of a silicon gate over the time. Each dot represents an instance of the gate over time. After manufacturing, some devices are faster than others due to process variability. Thus, some gates are beyond the acceptable quality threshold (red area). Over time, aging effects lead to consistent decrease of performance. After some time, the gate operates out of the specified operating conditions. Also spontaneous, transient effects can occur (e.g. caused by negative-bias temperature instability). Generally, these effects are inherent to the silicon process and already existed in previous generations. However, in next generation devices these characteristics will appear much more pronounced.

The size of a transistor will, if the trend continues, decrease further - even beyond todays (2014) 22 nm technology node. At this stage, various effects become noticeable [184]. The feature sizes will be so small that different dopant areas will be separated by only a few atom layers. This causes dopant fluctuation which comes from the discreteness of dopant atoms in the transistor channel. Thus, because the law of large numbers

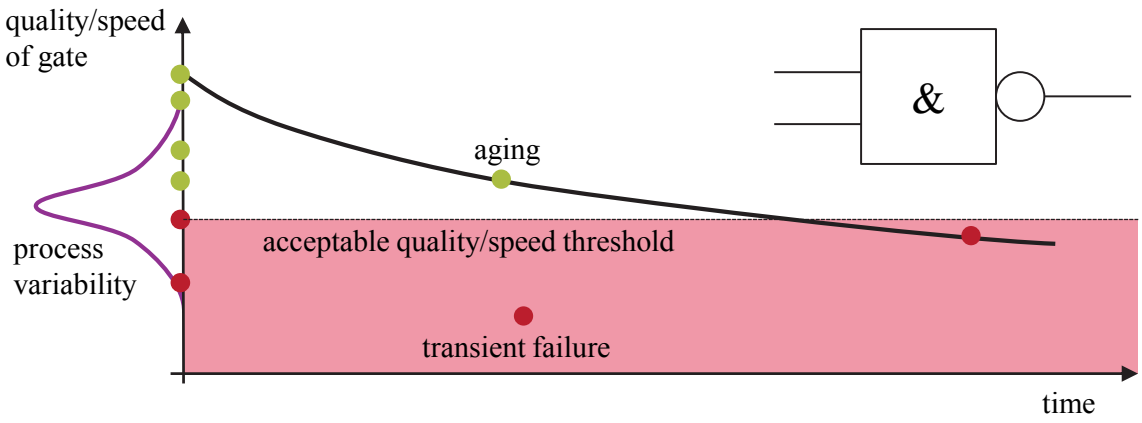


Figure 1.5: Failures over time. Shown are process variability and quality decay of a gate (NAND) over time (ageing). Transient effects lead to spontaneous failures [36].

does not apply anymore, single atomic defects will have more impact on a transistor than before.

A second source of variability is sub-wavelength lithography [289]. This is the reason for line edge roughness and other effects resulting in variation. The consequence is that each transistor will have its own electrical characteristics which will deviate drastically from the mean. Aside from these static causes for transistor variability that emerge during fabrication there are dynamic variations which occur during operation and vary over time. Different parts of a chip are utilized differently depending on the application that is executed. Thus, the heat flux will be different across the microprocessor die. This heat puts more demand into the power supply grid because the heat results in time-dependent, dynamic supply voltage drops and impacts the charge mobility. This impacts the path delay of the logic gates negatively. Additionally, the non-uniform heat distribution causes an application-dependent aging pattern and heterogeneous degradation.

The third challenge is the sensitivity to ionizing rays and particles. *Single event upsets (SEU)* happen, if ionizing rays strike through a transistor and change the logic state of parts of the circuitry [114]. Although this is not a new problem it gets more important if less charge (critical charge Q_{crit}) is needed to flip a bit. Interestingly, the error rate increases exponentially with decreased critical charge [275]. The new challenges need to be tackled in today's research and call for *resilient platforms*.

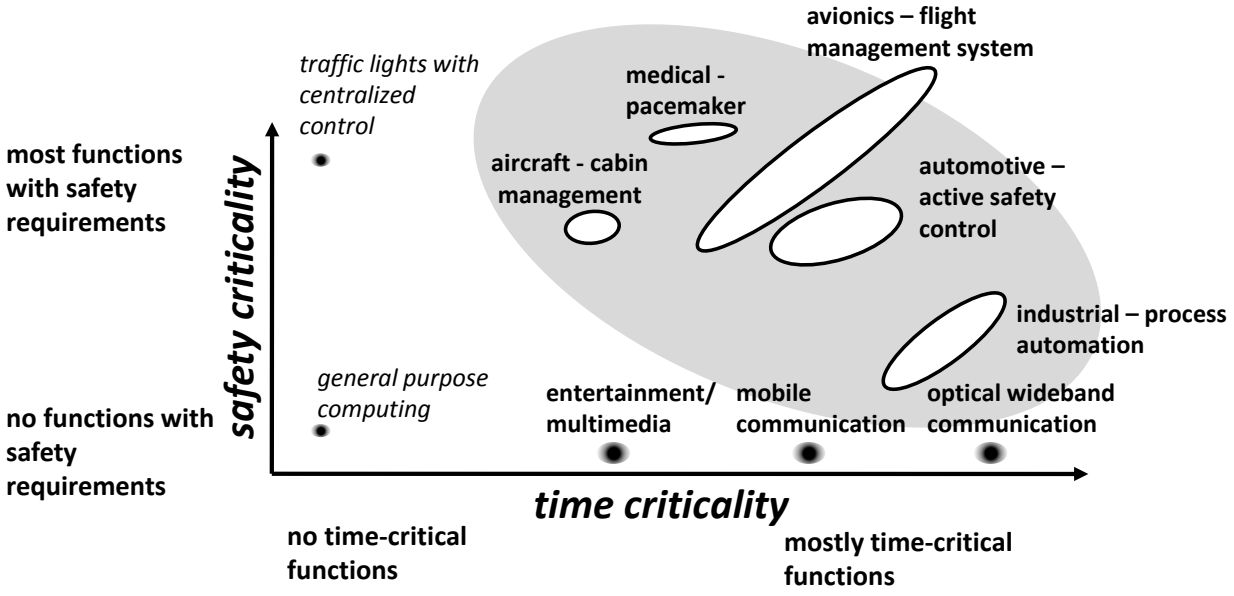


Figure 1.6: Two dimensions of mixed-criticality [14].

1.4.6 Mixed-Criticality Challenge

The previously mentioned advent of multi- and many-core processors offers the appealing possibility to efficiently consolidate different applications with a multitude of functionalities onto a single platform.

These applications have mixed-criticality requirements [281]. [28] defines a mixed-criticality platform as one that offers support for multiple functionalities, of which some will me “more important to the overall welfare” of the system than others. Obviously, they also recognized that the importance arises of the aforementioned certification problem, in which the correctness safety critical applications must be certified.

Some applications may have hard real-time constraints or are safety-critical whereas others are non-critical at all (e.g. best-effort entertainment). A design-space taxonomy and various examples of application types are depicted in Figure 1.6. The most interesting area is in the upper right corner (grey circle), here applications are time critical as well as safety critical.

An integration of those applications, requires special considerations in the design of MPSoC platforms and deployment of applications. Furthermore, increasing design costs will force MPSoC manufacturers to offer more flexible solutions that can target a wider range of applications. This means that the hardware support required by mixed-critical real-time applications must be flexible to be adapted for different applications. The challenging problem is to provide an effective, efficient, yet easy integration of mixed-criticality applications.

Also, computer and network systems must be considered to operate erroneous from time to time [41]. These errors can be single event upsets (cf. Resiliency Challenge) or bit errors on the communication medium such as Controller Area Network (CAN) [233], Ethernet (IEEE 802.3), or even the network on chip links. Here, it must be guaranteed that in case of transient error or retransmissions, critical communication can still deliver service according to its specification.

1.5 Integration and Verification of Mixed-Criticality Applications

The already complex timing behavior of a system which contains solely critical applications is usually certified by using formal analysis methods. For critical applications, the software behavior in terms on runtime, memory accesses as well as cache behavior can be analyzed by static program analysis [115]. This is possible, because critical software is designed and programmed such that loop iterations and call graph are known (no dynamic jumps) and further dynamic constructs which are offered by today's programming languages are prohibited (e.g. MISRA C is used [8]) However, for non-critical application, such assumptions cannot be made and analysis results cannot provided at all or are utterly conservative (cf. mixed-criticality challenge).

There is a broad body of approaches for formal performance analysis of systems. Here SymTA/S [120] can be used, which internally relies on the busy-window approach [172] for component analysis and event-model interfaces [232]. This approach supports a large variety of semantics such as register communication [89], various scheduling policies for field-busses [61], Ethernet [237, 74] as well as shared resource analysis for multicores [240]. A similar approach is Real-Time Calculus (RTC) [272] which is based on Network Calculus fundamentals [168] and used min-plus algebra to derive formal worst-case bounds.

The most important aspect, when integrating mixed-criticality applications is, that the integration strategy must provide sufficient isolation which is required by all major safety standards (e.g. [224, 135, 136]). Here, for instance the IEC 61508 explicitly states:

> "Where the software is to implement safety functions of different safety integrity levels, then all of the software shall be treated as belonging to the highest safety integrity level, unless adequate independence between the safety functions of the different safety integrity levels can be shown in the design."

Formal analysis, as typically used for safety critical system is not enough to provide sufficient proof of independence.

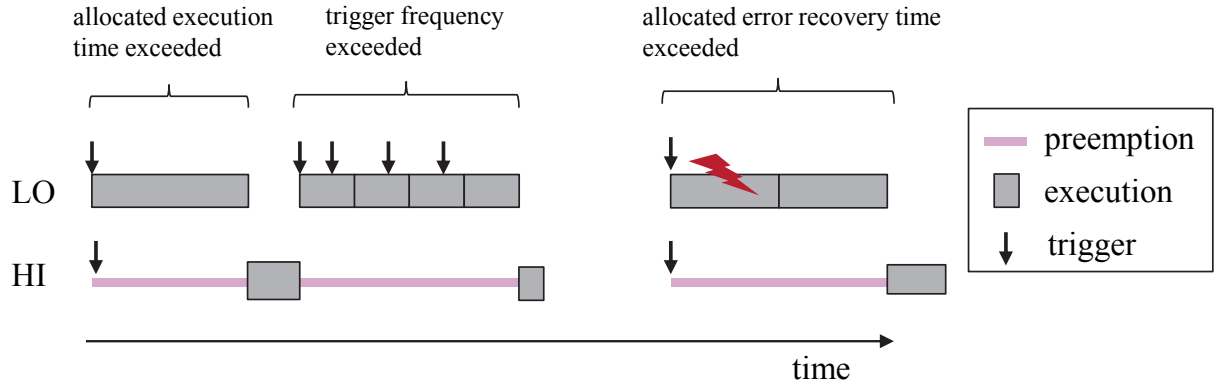


Figure 1.7: Typical timing problems, when low (LO) and high (HI) criticality applications are integrated. Execution time, trigger rate exceedance as well as error recovery overhead.

Figure 1.7 shows three typical timing problems which arise as a consequence of insufficient independence. A low criticality application (LO) is to be integrated together with a high criticality application (HI) on a single-core processor (assuming fixed-priority scheduling). A low criticality application can lead to starvation of the high criticality application for three reasons: Due to insufficient testing, the anticipated execution time of task LO could be exceeded which causes an unintended interference with task HI. Secondly, the arrival rate (e.g. frequency) of the trigger for task LO can exceed maximum specified rate. The third reason is introduced as the consequence of error recovery activities. This can be retransmissions on the communication medium or re-execution or a restart of task LO.

In general, for the integration of mixed-critical applications, the following properties of low critical software must be anticipated:

1. the *execution behavior* (time and memory) is unknown
2. the *interface specification* is incomplete
3. the behavior in case of *errors* is unpredictable

There exist two obvious solutions to a mixed-criticality integration. First, without considering isolation, the software stack can be considered holistically and certified according to the highest criticality (SIL lift-up effect [118]), as proposed by the standard. This approach is non-favorable as it implies very high certification costs and the overall system utilization is expected to be poor. This is due to the inherent over-provisioning in high critical software and its formal models.

Second, architectural isolation approaches can be applied to isolate software with respect to functional and non-functional behavior. ISO 26262

refers to this approach as “freedom from interference” [136]. To achieve this, isolation has to be enforced on two levels. Functionally, tasks must be isolated with respect to memory by using a hardware memory protection unit (MPU) or memory management unit (MMU). Additionally, worst-case timing interference must be bounded.

In this context, we can distinguish traditional approaches which follow a classical divide and conquer approach as well as recent research that permits a bounded interference between criticality levels. Traditional approaches focus on strict isolation, which was coined time orthogonalization in [242]. Such approaches prevent any dynamic run-time interference between applications of different criticality levels. If fixed-priority scheduling is used a criticality as priority assignment (CAPA) scheme can be used [199]. This suppresses any side effects on high criticality applications on single processor platforms, as long as no shared resources are involved. However, this may lead to poor responsiveness of low-critical applications[93]. This can be achieved by providing guaranteed service budgets on busses, processors and memory for instance by using strict time division multiple access with statically assigned time slots as used in the Time-Trigger Architecture approach [157, 201].

A similar approach is used by Time-Triggered Protocol [155], TT-Ethernet [156] as well as FlexRay [59]. Also avionic systems use this scheduling approach in their Integrated Modular Avionics (IMA) computer networks [6]. Such a TDMA scheme is also applicable to multicore platforms. For instance, [99] uses synchronized partitioned scheduling which guarantees that only memory accesses from one criticality level can occur. Goosens et al. [105] present a reconfigurable SDRAM memory controller that uses a TDMA scheme which is configurable with respect to slot sizes and bandwidth allocations which is suitable for mixed criticality integration.

Usually, the strict isolation is only required between tasks of different criticality levels, not between all tasks. Atacama [54], an Ethernet framework for mixed-critical communication, also uses time triggering for safety critical messages. Event-triggered messages are supported on a best-effort basis. Also Flexray [59] supports time triggering through the static segment as well as event-triggered messages through the dynamic segment.

This is similar to hierarchical scheduling as used for instance in virtual machines. Here a trusted hypervisor distributes a processing time budget to guest operating systems, which in turn use their internal scheduling policy. Virtual machines can distribute processing time according to a TDMA scheme (e.g. as done by ARINC 653 [6]), use round robin [253] or a server-based approach. Servers (e.g. deferrable server [263] and sporadic server [37]) were initially designed for scheduling sporadic workload in hard-real time environment [179], but can also be used to host mixed-

criticality workload. A server has an assigned budget which is replenished in regular time intervals.

Contrary to TDMA-based, strict isolation, more efficient integration approaches have been proposed recently. To enhance the efficiency of low critical, real-time applications, the typical worst-case analysis [219] can be used. Low-critical, real-time applications may miss their deadline from time to time, as long as bounds on the number (i.e. m out of n deadline hits) can be given. Also monitoring can be used to prevent unpredicted behavior [50]. In fact, monitors can be used twofold: Execution time overruns can be detected by using workload-based monitoring approaches [195]. AUTOSAR for instance proposes program flow checks, execution time monitors as well as hardware watchdogs [11]. Software interfaces can be monitored with respect to the activation pattern and stimulus consistency [128, 196, 165]. Contrary to monitoring, strict isolation schemes such as TDMA are not able to consume left-over processing time. Monitoring-based approaches typically allow a better average-case performance for low-criticality applications, as slack is recognized and efficiently distributed and consumed. Hence, more dynamic schemes seem to be very well suited for an efficient integration of workloads with such different requirements.

1.6 Concepts of Dependable Computing

As already motivated in the previous section, a key concern of a dependable system is, that it can justifiably be trusted. If the specification of a component does not satisfy the safety constraints as specified according to the safety concept (cf. ISO 26262 [136]) additional dependability measures must be employed.

When speaking about dependability, we first need to establish an adequate terminology [13]. An illustrative example of the relationship between the key concepts is given in Figure 1.8.

For the following definitions, we assume that the system under consideration consists of multiple components which communicate through well specified service interfaces (cf. V-Model, Section 1.3).

A system provides correct service when its interface adheres to its specification at all times. Otherwise we speak of a service failure or simply failure. This is when the service provided by the system interface deviates from its specification. A failure can only occur, if the system state is somehow altered and drifts from a valid state. This state deviation is called an error. Internally, errors can propagate from one component to another (active error), stay dormant or even vanish after some time. The ultimate cause of the error is the fault. This can be internal (e.g. a bit-flip in a register) or external such as an erroneous stimulus.

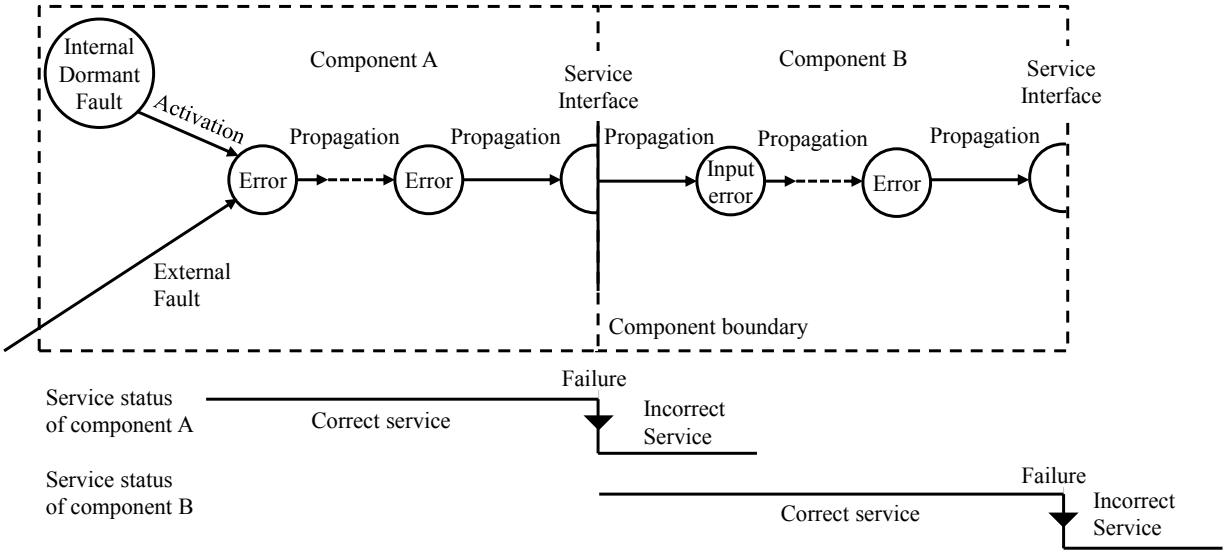


Figure 1.8: Error propagation: Fault, error, failure chain according to [13].

In this context it is also important to differentiate between reliability and availability. Reliability expresses how long a system can operate without a failure. It is typically measured as mean time to (first) failure (MTTF). Where availability denotes the time during which a system is operational. In the context of safety critical system, reliability is typically the important metric, since the first failure must be considered to be catastrophic.

We can distinguish two service failure classes (failure domain). The first one is the content (data) failure, here the pure data information provided by the system is incorrect. If the data provided by the system is correct, but the result is delivered too late, we speak of a timing failure. Throughout this thesis, we focus on timing failures as they play an important role in hard real-time systems.

There are various types of and reasons for faults. If the design and implementation process is not carried out carefully the system contains design errors. This can be software bugs or even hardware implementation bugs (e.g. the infamous Pentium floating point bug). Another class are low-level hardware faults such as single event effects, stuck at errors caused by degraded transistors.

It is the scope of fault tolerance to avoid failures through error detection and recovery [13]. Without an adequate level of fault tolerance each component poses a single point of failure. This means, that if a component is erroneous, a single error will propagate and cause a system failure, leading to a function failure as observed by the user. This is only permissible to non-critical functions [170].

1.7 Summary and Contribution

In this chapter, we motivated the importance of cheap, reliable, high-performance platforms for future systems. The system engineer has to cope with problems from different domains (certification, integration, silicon reliabiltiy). Here, it is important that the right platform trade-offs are fixed early at design time.

The contribution of this thesis is twofold: We provide a MPSoC platform design which assumes inherently unreliable silicon components with highly increased soft-error rates, while providing guaranteed performance and reliability to the applications. This platform uses operating system supported redundant execution, as well as hardware assisted state comparison to boost performance compared to related work.

Furthermore, we present a modeling and analysis framework to evaluate the performance and reliability very early at design time, when no physical platform is available. This enables the system engineer to assess multiple platform trade-offs (such as area, voltage, reliability) by using abstract models and formal approaches. In particular, this thesis addresses the following items:

- *A Scalable Platform Approach* helps to unify multiple design philosophies. Today, there is no unified approach that can host non-critical applications as well as critical applications. Most systems are either targeted for the safety-critical domain or a best-effort environment. We provide ASTEROID which guarantees fault-tolerant execution where needed and falls back to best-effort execution, if possible. This fuses the advantages from both worlds.

- A *Generalization of CPA* is necessary for a broad consideration of errors across different error-handling protocols and arbitration schemes. We generalize the busy-period approach by stating a universal stopping condition which bounds the number of events which must be evaluated for a formal consideration.

- *Communication's Performance under Errors* can be vastly reduced. This includes on-chip communication as well as off-chip communication. Accurate modelling and analysis of error protocols for busses as well as point-to-point communcation is provided.

- *End-to-end Error Protocols* as used in switched networks such as NoCs or switched Ethernet were deemed inapplicable for real-time applications. Integrating such protocols into CPA helps us to provide performance bounds for error-free as well as error conditions.

- *Execution Models* for replicated workloads can also be adapted to high-performance problems. We show how replicated tasks are modelled

by Fork-Join task graphs and provide conservative approximations to capture timing.

- *Reliability of Redundant Execution*. Dynamic effects such as error handling and recovery can lead to transient load peaks. We provide a formal probability analysis to compute the likelihood of deadline misses caused by error handling overhead (i.e. reexecution). This direcly yiels a safety integrity level which can be used by the safety engineer to evaluate the safety goal.

1.8 Outline

This thesis is structured as follows. In Chapter 2, we introduce the concepts of fault-tolerant (processor) design which are used in the field of safety-critical embedded systems. We present ASTEROID, a flexible platform approach which combines hardware and software approaches to increase reliability. A flexible, dynamic platform comes with some uncertainties which are modeled and analyzed throughout this thesis.

In Chapter 3, we introduce a consistent view of the compositional performance analysis framework which is used to predict the timing of large-scale embedded systems. A homogeneous set of definitions is given which is applicable to a large range of scheduling problems.

Then we extend the resource analysis by considering various forms of error events. This includes errors in point-to-point or multi-master communication topologies (Chapter 4) which used a broadcasting ARQ scheme. This captures on-chip AMBA protocol as well as off-chip Controller Area Network.

The effect of end-to-end protocols used in switched networks (such as NoCs and Ethernet) is modeled in Chapter 5. Here, a data packet is sent only if previous packets are acknowledged. Contrary to bus-based approaches, the timing for end-to-end protocols also depends on round-trip-times which includes latencies on the return path.

Replicated execution increases the reliability by executing tasks on multiple cores. However, addional non-functional timing effects such as inter-core blocking contribute to the task's response time. In Chapter 6, we transform the problem of redundant task execution to a fork-join schedulability analysis. Also, we investigate the actual reliability improvement of a redundant execution since additional recovery time may lead to deadline misses.

Finally, we summarize this work and draw a conclusion in Chapter 7.

CHAPTER 2

Building Reliable Computer Systems

New sources of threats and their impact on component and system behavior were presented in previous chapters. In this chapter, we focus on mitigation strategies and their implementation aspects. In particular, the main focus lies on random fault events as the main source of errors.

Most components, small integrated circuits or larger ECUs, do not have a constant failure rate over time. The failure rate is typically time dependant and transitions through three phases: Early failure (burn-in) period, followed by a random failure (useful life) phase as well as wear-out phase as depicted in Figure 2.1. This *bathtub curve*, more precisely called a hazard function, is used in almost all practical reliability considerations such as to quantify and judge the duration of burn-in tests.

However, there is disagreement to which systems and components the bathtub cure can be applied. In this scope, [151] discusses these problems and reasons that most of these problems address early-life failures. Through the following chapters, unless stated otherwise, we assume components and systems in the constant failure rate period. Thus, external effects such as a radiation is known, device error-rates are available from manufacturers or suppliers and further sources of error can justifiably be excluded.

2.1 Traditional Fault-Tolerance Approaches

In all cases, fault tolerance is based on some form of redundancy to detect or even recover from errors. Redundancy can be implemented on all architectural levels: hardware, software, time or combinations thereof. There are three types of redundancy [81]:

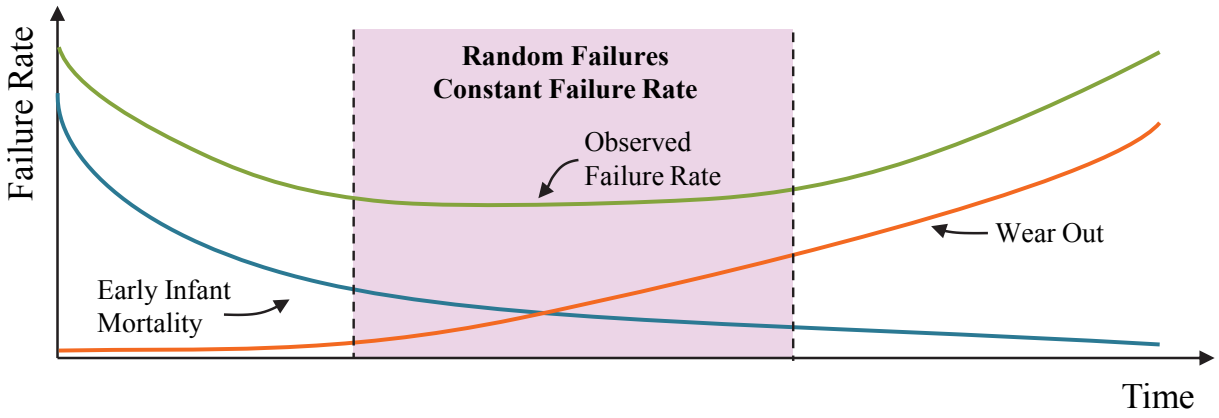


Figure 2.1: Bathtub curve: the observed failure rate over time resembles the shape of a bathtub.

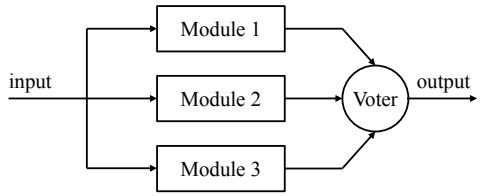
1. In passive (or static) redundancy, fault masking (e.g. majority voting) is used to prohibit error propagation, so further action of an operator or system is required.

2. Active (of dynamic) redundancy involves a two step process which consists of detection and recovery (e.g. detect the faulty component and replace it). A prominent example is acceptance tests combined with system reset.

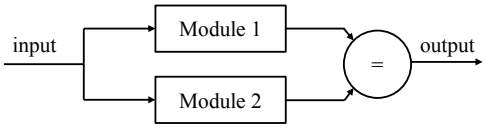
3. The hybrid is a combination of active and passive approaches.

Absurdly, the added redundancy implies that a fault-tolerant system is in almost all cases less reliable (with respect to the mean time to failure) than a simplex (non-hardened) system. This is because a fault-tolerant system contains more parts (hardware or software) that can break. However, the difference is that a fault-tolerant system is aware of the failure and can act accordingly (e.g. graceful degradation) where the simplex counterpart will output garbage.

Throughout the following paragraphs we present and summarize the most common hardware as well as software redundancy approaches as well as their advantages and drawbacks. One of the most commonly used strategy is N-modular redundancy [170]. Here, multiple functionally equivalent modules perform the computation in parallel. A voter (or comparator) checks the results and forwards the correct data or asserts an error detection signal. Figure 2.2 shows triple modular redundancy (TMR) [181] which is capable to mask one erroneous module (passive redundancy) as well as dual modular redundancy (DMR) which can only detect errors (active redundancy). Thus, for DMR, additional recovery such as a restart or rollback mechanism [161, 265] is required. The voter always imposes a



(a) Triple modular redundancy: Three separate modules plus majority voting.



(b) Dual modular redundancy: two separate modules with a comparision stage.

Figure 2.2: Commonly used modular redundancy concepts.

single point of failure and its reliability must be some orders of magnitude more reliable compared to the modules which are voted on.

Dubrova [81] provides an exhaustive overview of voter design trade-offs as well as hot and cold standby approaches and their reliability performance compared to a simplex system. An example for hybrid redundancy is self-purging redundancy [180]. Here faulty modules are disconnected (purged) from the voting process, this allows to remove the faulty unit during operation without any downtime.

Communication infrastructure is typically protected by error correction or detection codes [175]. Using coding techniques in off-chip communication is common practice and error-detection codes (EDC) and error-correcting codes (ECC) are frequently applied together with automatic repeat request (ARQ) and forward error correction (FEC).

By adding redundancy to the transmitted data, single bit as well as burst errors can be detected (EDC) or even corrected (ECC). A typically used family are linear block codes (e.g. Hamming-Code, Reed-Muller-Code, BCH). There is a large design space with respect to error coding and the designer must trade-off computational complexity with error correction capability. Such codes only protect against signal integrity problems, caused by electro magnetic interference or other noise sources. If the physical wire is damaged, error coding is not sufficient. Flexray and AFDX offer a redundancy concept, where multiple spatial distributed wires are used (like DMR). Such a multi-channel concept is de-facto standard in avionics. The 777 Dreamliner, for instance, uses a triplex redundant bus architecture [291].

Fault tolerance can also be implemented in software. Multi-version programming [56], for instance, helps to prevent implementation and design errors. Independent development teams design functionally equivalent software components and software majority voting checks the results. This approach is described in DO-178 [224], but rarely used as programmers are likely to have a common misconception and make similar faults [152].

Also the added costs of two (or more) development teams are a strong disadvantage compared to the alternatives.

If transient hardware errors are to be detected by software, simple re-execution [215] can be used. In case spare processing power is available, such a software approach is cheaper and more flexible. The re-execution can be distributed on different physical resources (spatial distribution) or on a single resource using time multiplex. Such a mapping of redundant replicas to a multi-core system is a promising approach and will be discussed throughout the next chapters.

Simple error detection can be implemented as acceptance tests [280, 182, 262, 234]. Here the programmer adds assertions to the program which evaluate previously identified invariant conditions or plausibility properties at runtime. Such an approach could have prevented the Ariane 5 crash, caused by a variable overrun during type conversion [80].

Analogous to coding theory used for communication, redundancy can be added to data structures and the implemented algorithm. During the processing, the algorithm is applied to both, real data as well as redundancy information. After successful termination of the algorithm, a consistency check is applied as an error detection facility. This is known as Algorithm Based Fault Tolerance (ABFT) [129]. ABFT has successfully been applied to matrix multiplication, but also to Fourier transforms and matrix equation solvers.

Alternatively, assertion points and invariants can be identified and added by the compiler [145, 279]. A compiler-assisted appraoch uses basic block signatures. Here, each basic block is augmented with a checksum that is stored in the beginning and verified in the end to detect whether invalid control flow has occurred in between. The compiler can also dublicate computations to detect corrupted operands [229, 230], as well as duplication of conditional checks in order to detect invalid control flow branches [228, 40].

2.2 ASTEROID Approach

As motivated in the introduction, automotive as well as avionic OEMs are keen to integrate a set of vastly different applications onto one single platform. Typically, there are only a few safety-critical functions (active steering) and a lot of convenience, high performance functions (image-based road-sign detection). This platform may be unreliable and potentially offers multiple shared-resource pitfalls. First, let us consider how a fault tolerant architecture must look like if today's off-the-shelf architectures are used.

Multi-core systems comprise a set of independent processors which are connected to a communication fabric. This fabric connects the processors with main memory and peripherals. For instance, the previously presented

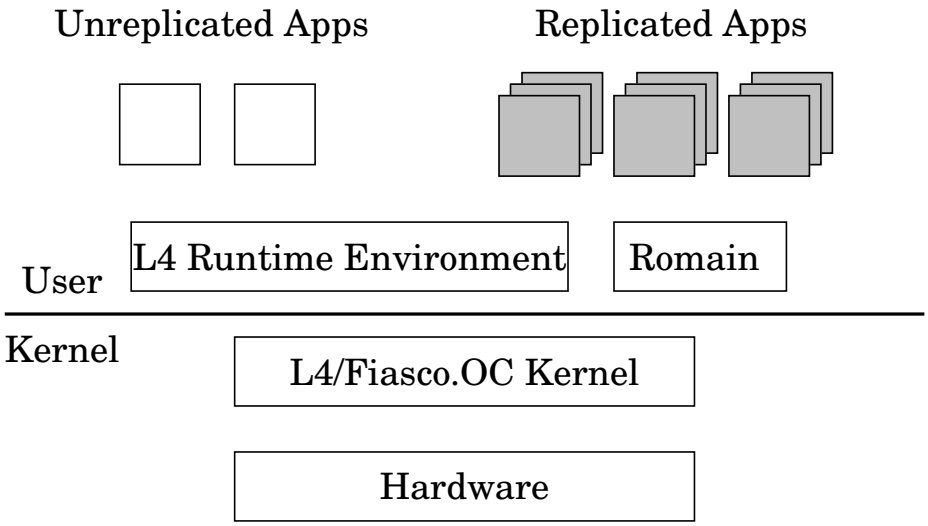


Figure 2.3: ASTEROID system architecture (Source [17]).

Freescale P2040 (cf. Figure 1.4) as well as the Infineon AURIX [132] implement such an architecture. These multi-core architectures offer the possibility to use lockstepping (DMR) for fault tolerance in safety-critical designs, which sounds promising.

However, future many-core chips such as the Intel Single-Chip Cloud Computer [125], TILERA's TILE-Gx [4] as well as the Integrated Dependable Architecture for Many-Cores (IDAMC) [187] have up to 100 cores.

If DMR is used in these architectures, a lot of processing power would be wasted. As motivated, only a few functions actually require the additional fault-tolerance. Best-effort applications would run in the protected environment unnecessarily. ASTEROID [17] addresses this problem by applying task-level redundancy to critical tasks only, where other applications are executed in a simplex fashion. Critical functions are duplicated (or even triplicated), whereas non-critical functions are not.

This replication approach was implemented by TU Dresden [75] as an operating system service running on top of the L4/Fiasco.OC microkernel [276, 112]. Figure 2.3 shows the final ASTEROID software platform. As in every microkernel system, the software is split into a privileged kernel as well as additional services such as device drivers which execute as user-level components. ASTERIOD adds a new component, *Romain* which implements task-level voting by redundant multithreading [75]. A major advantage of this approach is that the replicated application does not need any additional changes, such as additional operating system calls to control the redundancy. Almost all software can be executed reliable, as long as it is guaranteed that the execution of the kernel as well as Romain can be trusted. Similar to the trusted computing base [166], as known from the security domain, a *reliable computing base* must be established.

On startup, Romain spawns multiple replicas of the user-level application which execute independently. The voting process is orchestrated by the Romain master. It guarantees that the replicas receive the same data such as inter-process messages as well as memory-mapped I/O. In case of exceptions (e.g. illegal instructions, unmapped memory) the control flow is handed to the master process which externalizes a consistent state. Thus a functional deterministic execution behavior is enforced.

This works in the following fashion: If a processor interrupt is raised, such as a software trap, the master waits until all replicas have trapped. Then, the state is compared. Runtime overhead makes it infeasible to compare the entire process state, including main memory, only the processor registers are compared. In most cases, they contain parameters for system calls and other information which is sufficient for a fast voting process.

Once affected by a soft error, an application running on an non-fault-tolerant processor may fail in multiple ways:

1. No effect. Data corruption was masked by hardware or software.
2. Crash. An exception is raised (e.g. division by zero, illegal memory access) and the program crashes.
3. Silent Data Corruption. Data is altered but the program terminates.
4. Endless loop (hanging). The program does not terminate (or needs considerable more time than anticipated).

To assess the likelihood of such events, fault injection is typically used. The injection can be done in hardware for instance by using an FPGA [260] or by using the processor debugging interface [117]. Alternatively software methods by using an architectural simulator [290, 63] can be used. In contrast to software-level approaches, low-level hardware injection captures the microarchitectural level of the processor, which is abstracted in software level simulation (unless explicitly modeled). The abstraction makes it questionable if software level approaches have sufficient accuracy [286, 57].

ASTEROID detects any of the previously mentioned failure classes. A crash is detected and handled by the operating system service, silent data corruption is detected in the voting process and a timeout mechanism points out a hanging application.

Further trade-offs which are not in scope of this work such as shared-memory handling and voting synchronization software implementation strategies are discussed in [77] and [75].

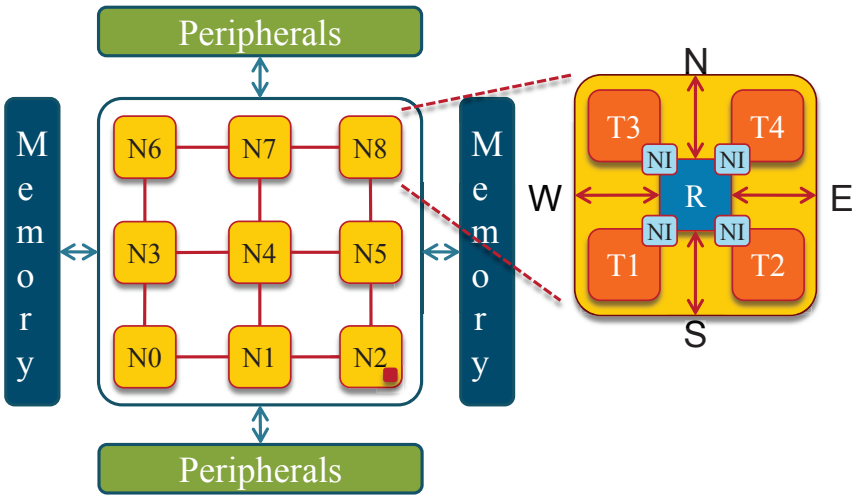


Figure 2.4: A future many-core design: Integrated Dependable Architecture for Many-Cores.

2.2.1 IDAMC Integrated Many-Core

As a representative future many core setup and research vehicle, the IDAMC architecture is used.

The IDAMC architecture is flexible and consists of up to 64 nodes which are interconnected by a mesh NoC topology. The NoC is based on [160] with additional Quality of Service (QoS) which provides isolation guarantees for bandwidth and latency constrained traffic classes [71]. Such traffic is typically found in embedded data streaming applications (e.g. radar, network processor) as well as distributed control (e.g. autopilot).

The advantage from a resiliency standpoint is the inherent redundancy offered by multi- as well as many-core designs. The vast number of cores can be used for high performance as well as reliable redundant execution. This is the central theme of ASTEROID (An Analyzable, Resilient, Embedded Real-Time Operating System Design).

The IDAMC architecture features a mesh-based network on chip architecture which connects up to 64 nodes. Here, each node can contain up to 4 tiles. Each tile encompasses a modified LEON3 multiprocessor [98] with optional peripherals such as memory controller, controller area network, and more.

The actual setup is highly customizable through a synthesis configuration file. This allows to control the number of processors in each tile, type and size of local scratchpad memory and the instantiation of peripherals.

During runtime, the architecture is controlled by a dedicated trusted supervisor tile, which has elevated administrative rights. Only software

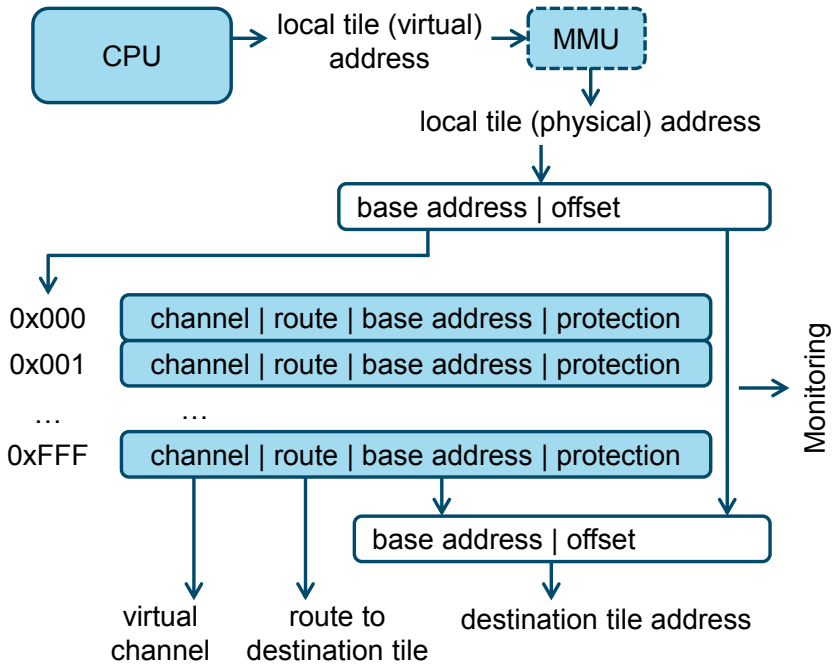


Figure 2.5: Address translation as performed in the network interface.

running on this tile can change the network on chip QoS settings and setup the memory layout of the user tiles.

The packetization of local tile data into network on chip packets is performed by the tile's *network interface*. The IDAMC network on chip used source routing where the sending terminal must know and setup the packet's route. This allows to reduce the complexity of routers, enables flexible routing and reduces contention in the network. To speed up the packetization process, a hardware accelerated memory-mapped I/O scheme, similar to a TLB (translation lookaside buffer) was designed. To physically enforce containment across the entire platform, all accesses from a tile to the network on chip are proxied through and translated by the network interfaces.

The translation process is shown in Figure 2.5. First, a memory IO operation issued by a local CPU is processed by the local MMU (if enabled), then it enters the network interface. The network interface contains a lookup table of configurable length (typically 64). Each line holds the route information to the target tile, the base address used at the target as well as additional protection bits which encode the allowed access type (read, write, execute). Optionally, monitoring devices [196] are connected to the address translation which monitor the access pattern such as datarate and jitter to detect misbehaving software (i.e. babbling idiot).

	#Registers	#LUTs	#BRAMs
IDAMC	33834 (4%)	52317 (11%)	114 (16%)
NoC	11409	15060	12
Tile 0	5016	11381	79
Tile 1/2	4348	9222	10
Tile 3	2247	3919	3
NI 0/1/2	2610	5395	4
NI 3	1744	3000	2
MON 0/1/2	489	1072	1
NI 2nd Generation	1462	1507	2

Table 2.1: Synthesis results of IDAMC with first generation network interface broken down into hierarchical units. Synthesis results for second generation network interface are shown for comparison.

All address-translation tables can be programmed by the supervisor tile during runtime. This has several advantages: First, only a trusted component can reconfigure the logical communication topology. Second, it allows to dynamically reconfigure the platform in case of hard errors such as a broken router or tile.

Table 2.1 shows the synthesis results on a Virtex 6 FPGA. The presented system comprises a minimal setup used for research and development purposes, as it allows fast turn-around times with little synthesis times. The system contains four tiles each connected to a dedicated router. All routers are connected by a 2x2 mesh network. The address translation supports 64 entries. Tile 0, the system controller, contains a LEON 3 processor, 256 kB on-chip RAM as well as 1 kB on-chip ROM which contains bootstrap information. Processing tiles 1 and 2 feature a processor with 1 kB cache (data and instruction) as well as an interrupt controller. Tile 3 connects the system to the external DDR2 memory and contains a memory controller. Hence, to access the main memory, all processors communicate with Tile 3.

The synthesis results of the LEON3 internals are not shown as they are identical to an off-the-shelf LEON system. The target frequency is set to 80 MHz, except the DDR2 interface which runs at 160 Mhz. The network interface of Tile 3 occupies less resources compared to the other tiles because it does not feature a address translation mechanism since no AMBA Master components are instantiated. A detailed discussion of the synthesis results are discussed in [187].

The described system features the first generation network interface used for bring-up only. It includes all functionality but offers poor resource utilization and performance as it was only targeted for prototyping. A

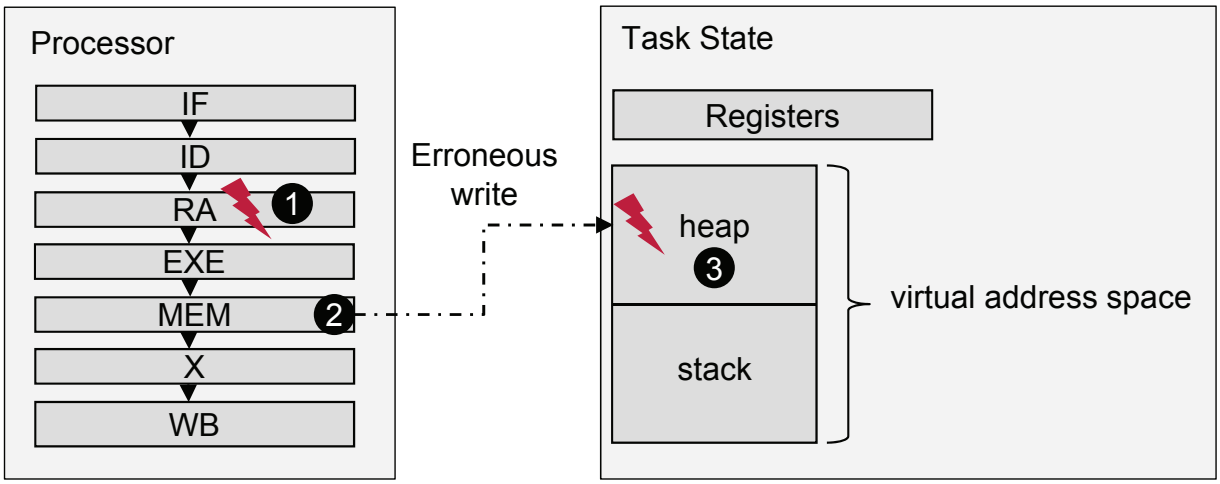


Figure 2.6: Illustrative example of an error in the processor pipeline (1) which causes an erroneous write (2) and leads to an error in the heap state (3).

second generation network interface implements pipelining, and allows efficient data streaming with little packetization overhead compared to the first generation device. The new network interface uses roughly 50% less registers and 27% of logic resources compared to the previous version, offering nearly the same functionality [1].

2.2.2 Hardware-assisted State Comparison

The software architecture of ASTEROID as described previously relies on an efficient voting mechanism to compare the state of the replica processes. In the following paragraphs we revisit how errors propagate internally and discuss the voting process in detail. Figure 2.6 shows how pipeline errors propagate from the processor into the task's state. Here the task state consists of the entire virtual memory space as well as the architecturally visible registers.

For instance an illegal register access causes an erroneous operand fetch ❶. When the content of this register is used later for memory accesses such as a write ❷ the state of the task such as the main memory ❸ is modified illegally. The obvious objective is to identify and signal such alterations. However, there further design goals:

- *Error coverage*, which is the fraction of errors which are detectable by the mechanism, should be as high as possible

[1] DMA unit not included

- *Error latency*, which is the time from error occurrence to error detection, should be as low as possible.
- *Additional overhead* (performance penalty, chip area, code size) should be as low as possible.

To perform the voting, the Romain architecture compares the state on externalization only (e.g. on system-calls and exceptions). Thus, the error coverage of Romain is sufficiently high, because all data is eventually subject to a comparison before it becomes visible. As already discussed, without further consideration of shared-memory communication the execution time overhead of the presented approach is reasonably low. But it also comes with some inherent drawbacks with respect to our requirements: The major issue is that the error latency is not bounded. An error in the task's state as depicted in Figure 2.6 can stay dormant for long time until the erroneous state is externalized. An arbitrary long detection latency can potentially render an error recovery mechanism useless if real-time requirements are involved.

To circumvent this problem hardware assisted fingerprinting is used, which was introduced in [255]: A dedicated fingerprint unit which resides in the pipelines of all cores in the processor hashes all retired instructions. This generates a fingerprint which represents an unique hash for a specific instruction/data sequence. Since the same code is executed on redundant cores we can use the fingerprint as a basis for DMR voting. In the original work from [255], voting between redundant cores is performed when cache lines become visible on the system bus. However, this approach has some inherent drawbacks, especially in the field of real-time systems and with respect to mixed-critical applications. Since the mechanism relies on the cache coherency protocol as a synchronization primitive for comparison, the mechanism implicates a high degree of timing uncertainty (e.g. when comparisons are performed and how often). Also, no differentiation between task contexts is made, thus all instructions end up in one single fingerprint and redundancy cannot easily be performed task-wise.

Thus, we propose to use fingerprinting differently and implemented context-aware fingerprinting, where a fingerprint is generated per context (if required). We extended the LEON 3 processor [98] with a fingerprint unit as shown in Figure 2.7. The unit consists of three building blocks: a instruction counter which counts retired instructions, the data fingerprint which taps the data path of the pipeline and the instruction fingerprint which is fed with the retired instruction word. All of these registers are implemented as ancillary state registers (ASRs) which can be read by software.

The unit works the following way: Both fingerprint registers continuously hash data and instructions. The hash function can be selected at synthesis time. We implemented three variants: CRC-32, single CRC-16

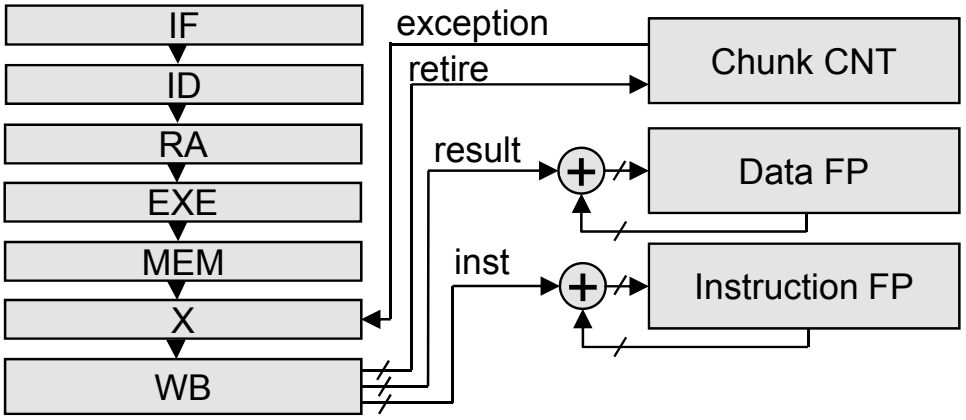


Figure 2.7: Leon 3 pipeline with fingerprinting extensions.

and double CRC-16, which hashes the upper and lower half words independently.

In case of interrupts or traps, the processor stores a copy of the recent fingerprint and the operating system may store the fingerprint in the task control block. In the same way an old fingerprint can be restored on a return-from-interrupt instruction. Thereby, per-task fingerprints can be implemented by the operating system and we are able to handle asynchronous events.

Data and instruction fingerprint reflect a hash over the task state and can be used in the Romain master for voting. However, this approach still exhibits the drawback of an unbounded detection latency, because a task first needs to raise a CPU exception to trigger comparison.

To artificially increase the voting-frequency in a predictable way, *Chunk Checking* is implemented. Chunk checking is a feature which is controlled by the operating system to control the error detection latency for long-running workloads. Per se, the operating system has no method to interrupt two copies at a predictable instant in time (on exactly the same instruction) in order to compare intermediate results. Here, we use the chunk counter which is decremented with each executed instruction and causes a trap if it reaches zero. This enables the operating system to compare intermediate results without using the highly inefficient single-stepping mode.

A third mode of operation is the signature checking mode. In this mode we leverage from the fact that we have an individual instruction fingerprint. By construction it is possible to pre-compute instruction fingerprints for each basic block. This can be done by the compiler or by dynamic recompilation during runtime as part of an operating system service. Such a precomputed fingerprint is restricted to a single basic block which has no data dependency with respect to its control flow. This enables to implement signature checking for basic blocks: A dedicated match-fingerprint

	#Registers (Overhead)	**#LUTs** (Overhead)
CRC32	1503 (22%)	6768 (31%)
Double CRC 16	1508 (22%)	6551 (26%)
Single CRC 16	1496 (21%)	6328 (22%)
Baseline	1230	5170

Table 2.2: Synthesis results of integer unit pipeline with different fingerprinting implementations.

instruction tests the target and the actual fingerprint which may result in a fingerprint-miss trap.

Table 2.2 shows the synthesis result on a Virtex 6 FPGA platform. The Fingerprinting adds up to 31 % overhead on top of the integer unit logic. This is mainly caused by the hashing function. An unpipelined, single cycle CRC consists of 32 cascaded xor stages. To control the fingerprint logic, nine additional 32 bit registers (total of 288 bit) were added. Thus, the hashing algorithm has little influence on the number of required registers. It must be noted, that little optimization was applied to the fingerprinting. The fingerprinting unit resides soley in the writeback stage, but could have been pipelined and distributed among the previous processor pipeline stages.

For ASTEROID, we decided to only use a CRC-based approach. Other work such as [183, 52] also consider Fletcher's Checksum (FC) [95]. FC is computationally more efficient as a checksum is divided into blocks which can be processed in parallel. The downside is that FC is not as robust as CRC with respect to the error detection probability as also highlighted in [52].

2.3 Comparison and Performance Overview

The ASTEROID approach is similar to the dynamically coupled cores (DCC) approach presented in [162]. It uses a similar hardware assisted state comparision as ASTEROID, which is presented later in more detail. However, in DCC, the comparison is controlled by hardware only, a modified cache is used to hold preliminary data. Thus, comparisons are more frequent. Depending on the checking interval, the overhead is between 3% and 20 % for selected microbenchmarks.

Dynamic Dual Modular Redundancy (DDMR) as proposed in [103] uses a configurable ring bus topology to pair cores. Checking is performed using a CRC fingerprint as done in ASTEROID. The slowdown depends on the number of paired cores and the checking interval. For a long interval check

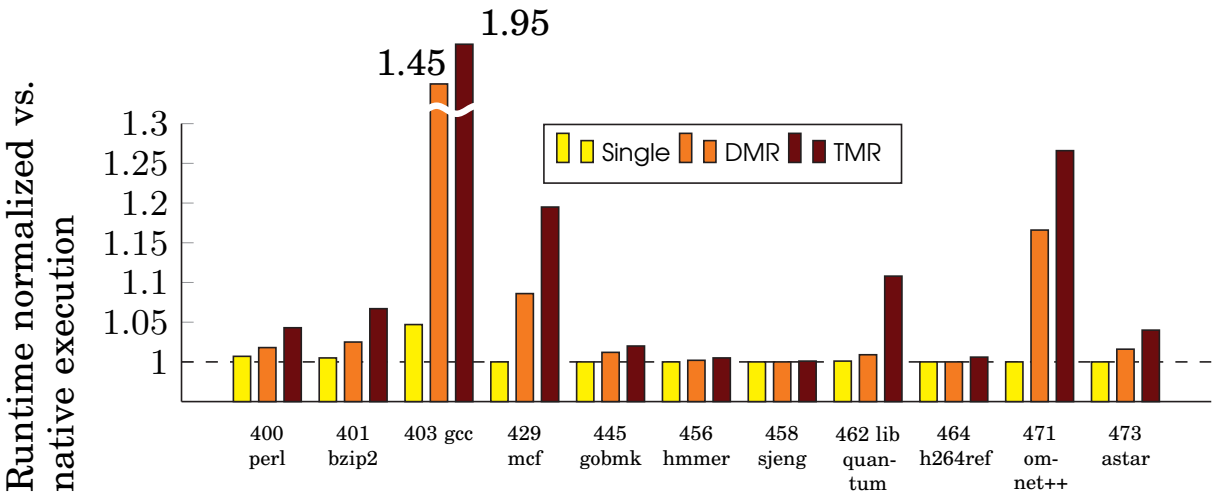


Figure 2.8: Overhead for replicating the SPEC INT 2006 benchmarks with one, two, and three replicas compared to native execution. (Measured by Döebel et al. [78]).

(thousands of instructions) the overhead is between 4% and 30%. This is consistent with the performance figures for DCC.

Döbel et al. evaluated the performance of the ASTEROID Romain Framework in [78, 79, 77]. Here, we will briefly summarize the key findings and conclusions. For the experiments the SPEC INT 2006 benchmarks were used [122]. The benchmark `483.calancbmk` was left out, as it uses deprecated C++ STL features that are not supported by L3Re. SPEC INT 2006 is mainly used as a processor benchmark and thus contains processor bound tasks. This includes a broad field of applications such as compilers, video processing, and compression algorithms.

All applications ran on a two socket board, each containing an Intel Xeon X5650 CPU running at 2.667 GHz. The system features a total of 3 GB RAM running 32-bit executables on a total of 12 processors. All advanced features such as turbo-boost, dynamic frequency scaling were turned off.

Figure 2.8 shows the performance figures as measured by Döebel et al. [79]. It shows the normalized execution time overhead for Simplex, DMR as well as TMR against a non-replicated (normal) execution. For most the benchmarked kernels, the overhead is in the 2% mark (geometric mean for TMR is 2.51%). However, outliers are gcc, mc, libquantum and omnet. This is mostly due to a large number of memory reorganizations (remap, realloc) which is not a typical use-case in embedded devices that have a static memory configuration. Also it was shown that the performance can be further improved by clever replica to core as well as replica to socket mapping.

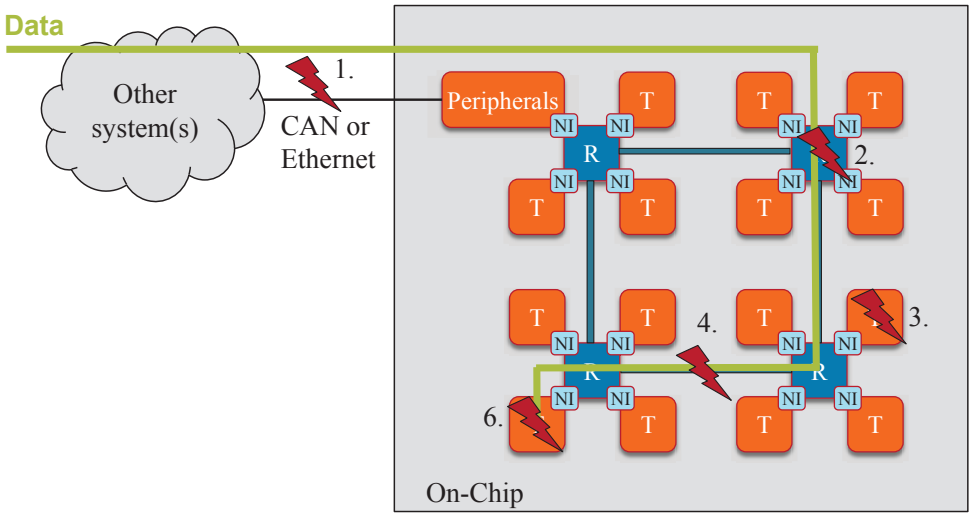


Figure 2.9: Transient errors in the processing and communication infrastructure on-chip as well as off-chip lead to timing uncertainty. Subsystems which are affected by errors are enumerated for reference.

2.4 Summary and Challenges of ASTEROID

The ASTEROID architecture has several advantages over existing approaches:

1. The platform allows safe sharing of critical shared resources.
2. Monitoring capabilities can be used to enforce bounded contention.
3. Fault-tolerance can be easily implemented by redundancy.
4. Fault-containment is supported as errors never propagate uncontrollable.

ASTEROID shows a competitive performance, compared with related approaches that use redundancy for safety-critical applications as shown in [76]. The presented performance values constitute average-case, measurement based values. Obviously, they give a general idea of the performance and applicability but conceal the importance of worst-case design as introduced in the first chapters.

Figure 2.9 shows a bigger picture in which the ASTEROID platform is integrated in a distributed system. Data is fed into the ASTEROID platform through a peripheral interface such as Controller Area Network [233] or Ethernet (e.g. [131]), routed to the target processing tile and eventually consumed.

ASTEROID is capable to detect and correct most transient errors on this path. Functional units which could be affected are highlighted in Figure 2.9.

For instance, CAN messages are retransmitted in case of errors, routers are fault-tolerant (2) and optionally may include link-level or end-to-end error coding techniques [227, 189]. The Romain framework will detect errors and restart or rollback the application in case of errors. Functionally, errors in these subsystems are covered and the ASTEROID platform is likely to properly react potential error scenarios.

However, the performance under such error events is not guaranteed per se. In ASTEROID the timing is vastly influenced by error detection and correction mechanisms and depends on the actual error scenario.

Subsystems which can be affected by errors and interfere with system timing are highlighted in Figure 2.9. Data frames which arrive over CAN or Ethernet can arrive delayed if frames need to be retransmitted ❶. Network on chip packets must eventually be rerouted due to faulty routers ❷ or corrected ❹. Faulty processors may lead to babbling idiot which imposes higher (but bounded) load to the communication infrastructure ❸. And finally timing overhead is added by the error detection and correction facility which enables the redundant execution ❺. This leads to the following questions, which are answered through the course of this thesis:

1. What is the *timing influence* for a given error-scenario?
2. What is the *likelihood* of such errors?
3. How likely is a *system timing failure*?

CHAPTER 3

Timing Verification of Safety-Critical Real-Time System

In the first chapters, we motivated the importance of verification and testing in safety-critical systems and presented the ASTEROID architecture as a prototype. In this chapter, we introduce the mathematical framework required for a formal consideration of the system timing. First, we discuss the model, this includes an architectural model as well as a timing model of the applications. Then we show how individual tasks and resources are analyzed and timing metrics such as response-time and buffer sizes are derived. Finally, we show how these results are composed into system view.

The purpose of a timing verification is to prove the non-functional timing correctness of complex, safety-critical real-time applications such as a distributed control application (e.g. an anti-lock-braking system). It is most important that key properties such as *jitter*, the *latency* between sensing and actuation, *sampling period*, *response time*, and *timing independence* between critical and non-critical parts are guaranteed.

Exhaustive simulation is often not able to reveal corner cases. Due to the tremendous system complexity (i.e. system size and functional interactions) even very long simulation runs are not able to reliably detect critical scenarios. Also assisted simulation, where worst-case patterns are injected does not help, since a combination of component corner-cases does not always lead to a system corner case. Another intrinsic problem of simulation is that very accurate models are required. Often these models (e.g. binary implementation, cycle-level CPU model) are not available during early design times when functional integration must be decided.

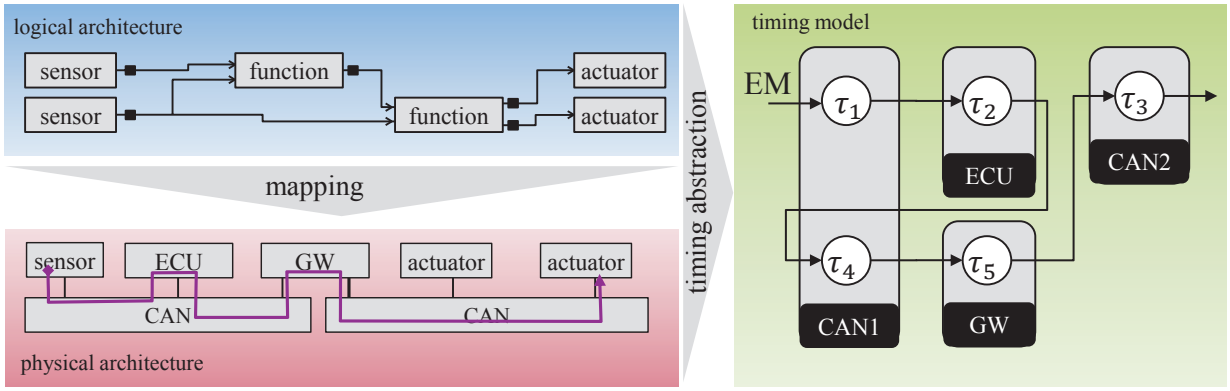


Figure 3.1: From the logical and physical architecture to the timing model.

Here formal verification helps to estimate the timing behavior during early stages and is able to validate timing during integration.

Formal verification is based on abstraction from the physical and logical architecture to a timing model domain. Let us recapture the V-Model, described in Chapter 1. Early during the design process, a functional architecture is specified. Later, when the physical architecture such as the network topology, ECU and gateway devices have been decided on, a mapping from logical functions into the physical domain is crafted. This includes signal to frame mapping, function to task mapping, task to ECU mapping as well as task to processor mapping for multicore ECUs.

The left part of Figure 3.1 shows this mapping process. To precisely model the timing behavior, knowledge from the functional level *and* the physical architecture must be known. In almost all cases, timing properties are derived on a functional behavior level but timing effects are subject to physical effects.

For instance, the designer wants to know whether the sampling period of the anti-lock braking system is always met. A logical view is required to know which actuators, distributed control loops and sensors are involved, the physical mapping to network segments, CAN frames and gateways is required to predict the physical behavior.

Thus, the timing model, as shown in the right part of Figure 3.1, is an abstraction from both domains. The algorithms and model transformation are based on well-known principles from the real-time community [232, 120, 272, 283]. For the sake of consistency we will introduce the model and baseline algorithms which are needed throughout this document.

3.1 Related Work in System-Level Analyses

As discussed, the purpose of timing verification is to prove that a number of applications mapped to a hardware platform met their timing requirements such as deadlines. Also more complex metrics such as output jitter, path latencies (cf. violet path in the lower left in Figure 3.1), processor utilization as well as memory utilization can be derived.

System-Level analysis [120, 282, 272, 168], contrary to component schedulability analysis [297, 61, 172, 39, 29], ensure that communicating tasks which exchange data over long chain of processor and network resources meet their timing goals.

As pointed out in [258], proposed approaches for system-level analysis can be divided into two classes. *Holistic* approaches capture the system analysis as a single, complex problem instance. The major concern with these approaches is the inherent computational complexity involved in solving the holistic problem. The advantage is that the determined timing figures are tight, which means that the derived bounds are close or match the real system's behavior. However, not infrequently a holistic analysis of large systems is intractable caused by an exponential growth of complexity with the number of resources.

Compositional approaches tackle this problem in a divide and conquer fashion. The system analysis is broken down into sub-problems: the analysis of components. A component is typically a bus, processor, or Ethernet switch. The analysis results of these sub-problem are coupled using standardized interfaces. As we will see later, dependencies between subsystems are resolved by iteration. This has several advantages as each component analysis needs only a limited view on the entire system. It is easy to evaluate subsystems in isolation and reuse performance results hierarchically.

There is a large body of system-level analyses approaches available to assist the designer during the design and integration process. Yen and Wolf [292] proposed holistic approaches to compute the delay through a task graph mapped to multiple processing resources. This work was later extended [213, 104]. The holstic framework MAST is freely available [104]. Other groups model complex distributed embedded systems using timed automata [119, 200] and apply model checking [296] do derive feasibility guarantees. This concept was also successfully implemented in the commercial Uppaal tool suite [35].

Compositional Performance Analysis (CPA) [120] adapt existing component schedulability analysis [172, 61, 39, 220, 270, 193]. In a second step, a system-wide analysis is composed by interfacing the behavior by using generic event-model interfaces as introduced in [232, 168, 272]. Often, systems have functional and non-functional cycles, where timing behavior of one component is subject to the behavior of another, and vice versa. Acyclic graphs can be analyzed by performing component analyses in topological

order. Cycles are resolved by iteration until a fixed point is found [259, 141]. In case some component analysis expect a parametrized form of the event model (i.e. jitter and period), conversion functions are available [232]. CPA is successfully commercialized by Symtavision, also a research focused implementation is freely available [69, 70].

Another commonly applied compositional approach is Modular Performance Analysis (MPA) [282] which is based on Real-Time Calculus (RTC) [272] which is built upon Network Calculus [168]. The system model is similar to CPA. A graph represents the application, with nodes being tasks and edges represent the interface between these tasks. The incoming interface resembles a service curve, which is the guaranteed service provided by a resource to the task, an arrival curve, which is in fact a generalization of the previously discussed event-model. On the outgoing interface of the task, the remaining service, which can be used by other tasks, as well as the arrival curve of the outgoing event stream. The remaining service and output event stream of each task is computed using MaxPlus algebra [25].

Comparisons of some of the previously discussed approaches such as SymTA/S [120], MPA [282], and MAST [104] are presented in [210, 261, 154]. The studies show that the analyses vary in terms of runtime and analysis accuracy. It was shown that each tool has its strenghts and weaknesses depending on the actual system's characteristics.

3.2 System Model

In the course of this chapter, we will introduce the modeling formalism required for the rest of this thesis. Similar to [194, 258], we differentiate between two aspects, the *structural model*, which abstracts the software and the underlying hardware architecture. The second aspect is the timing model, which captures the timing properties and interactions between components and tasks.

3.2.1 Structural Model

The structural system model consists of an *application* as well as a *platform* model. An application is a set of communicating *tasks*. In this context, tasks are the smallest entity. The platform model is consisting of a set of *resources* and connections abstracts physical processors, buses, Ethernet switches or network on chip routers. Tasks are mapped to resources according to a static mapping and consume some amount of service, be it processing time or network bandwidth. A scheduler (or arbiter for communication resources) distributes the available service among competing tasks.

Communication between tasks is modeled by a task graph in which edges are referred to as *event streams*. Event streams act as an interface between tasks and encode the activation pattern of a task.

Hence, a system models the physical architecture, topology information as well as functional mapping.

Definition 1 (System)**.**
A system $\mathcal{S}$ consists of a platform, a set of applications and a mapping.

$$\mathcal{S} = \langle \mathcal{P}, \mathcal{A}, \mathcal{M} \rangle \tag{3.1}$$

Definition 2 (Platform)**.**
A platform $\mathcal{P}$ is a directed graph with a set of resources $\mathcal{R}$ as vertices and a set of edges $\mathcal{E}$ indicating the connectivity between the resources.

$$\mathcal{P} = \langle \mathcal{R}, \mathcal{E} \rangle \tag{3.2}$$

$$\mathcal{E} \subseteq \{(r_a, r_b) \mid r_a \neq r_b, r \in \mathcal{R}\} \tag{3.3}$$

Definition 3 (Resource)**.**
A resource r consists of the physical entity which provides service together with a scheduler Ψ which distributes the available service according to a scheduling policy.

A set of associated functions (cf. Figure 3.1) are captured by an application. Special vertices with no incoming edged are referred to as sources and those with no outgoing edges as sinks. For sources, the event streams are specified as boundary conditions.

Definition 4 (Application)**.**
An application $\mathcal{AP}$ is a directed graph consisting of a set of tasks, sinks and sources $\Gamma = \{\tau_1, \tau_2, \dots\}$ and a set of edges $\mathcal{ES}$ representing the event streams which model the task's communication.

$$\mathcal{AP} = \langle \Gamma, \mathcal{ES} \rangle \tag{3.4}$$

A software task consumes service according to its core execution time, whereas a CAN frame occupies the bus for the number of bit times required to transmit the frame under the given protocol.

Definition 5 (Task)**.**
A task τ consumes service provided by a resource.

The mapping of tasks to resources is modeled by the system mapping.

Definition 6 (Mapping)**.**
The mapping $\mathcal{M}$ is a function which assigns each task $\tau \in \Gamma$ in the system $\mathcal{S}$ to a resource $r \in \mathcal{R}$.

$$\mathcal{M} : \Gamma \to \mathcal{R} \tag{3.5}$$

3.2.2 Timing Model

The previously introduced structural model resembles architecture, but it does not capture the timing aspects. In this sense, it must be clarified on a sufficient abstraction level how data is processed, stored and forwarded. In a real system, a task is triggered by a timer, external interrupt or inter-process signal.

In almost all literature on real-time performance analysis, the timing is abstracted from the underlying data. In this scope, we speak of an *event* if "something of interest happens" [242, 258]. An event models the task's activation behavior and abstracts from the actual data being transmitted. Thus throughout this thesis, a task is *triggered by an event*, where in the corresponding physical system, the task obviously is triggered by some incoming data. Hence, in most situations the occurrence of an event can be understood as data been passed from one entity to another. An event stream groups events which semantically belong together such as a collection of events that activate the same task.

The software implementation of a task typically involves multiple, not always distinct activities: Data is loaded from multiple queues, mailboxes or external devices, processed and passed around. In CPA a task is associated with a single input queue of unlimited size and the cyclic execution behavior consists of three distinct phases:

1. Input event is *read* from the queue. If no data is available, the task waits and can be suspended.
2. The event is *processed* and consumes service in the interval of a best-case and worst-case execution time $[C^-, C^+]$.
3. An output event (the result) is *produced* and optionally fed into the input queue of the preceding task.

The best-case and worst-case execution times of software tasks can be obtained by exhaustive simulation which yields a worst-observed execution time. For safety-critical applications a worst-case execution time analysis based on call-graph extraction and static program analysis is carried out. A list and comparison of methods and tools on the worst-case execution time problem is given in [288]. For bus-based communication, the transmission time is usually known as it mostly depends on the protocol structure and the payload size. Some protocols such as CAN use bit-stuffing or scrambling techniques which add additional bits, however a conservative approximation is easily possible [61].

Also communication resources such as CAN buses can be modeled using this task model. Here, input data is consumed and a data frame is constructed, data is serialized and transmitted, and finally data is reconstructed and fed into the receiving task. There are multiple possible

communication semantics such as register communication [89], event triggering and more , but in this work we restrict the communication to event triggered communication (non-destructive writing, destructive reading).

Forks and joins of event streams (i.e. AND, OR joins) are thoroughly discussed in [242, 141, 108]. A special case of fork-join topologies, namely fork-join tasks are studied in Chapter 6, hence, we do not further consider this special case in this chapter.

Now, that we have introduced the idea of events, we can formalize the concept of a trace.

Definition 7 (Event Trace)**.**
An event trace σ is a function

$$\sigma : \mathbb{N}^+ \rightarrow \mathbb{N}^+ \tag{3.6}$$

where $\sigma(n) = t$ indicates the absolute time at which the n-th event occurs associated.

Depending on the system uptime, a trace is potentially unbounded. It follows, that there is an infinite number of possible traces. Depending on the resource and task behavior, each two pairs of input traces could lead to different output traces. Exhaustive consideration of all combinations leads to intractable algorithms.

Event models abstract from the actual trace by only capturing worst and best-case behavior. Any trace which is in the permissible region of the event model is said to satisfy the event model. Vice versa, if a given trace satisfies the event model, any analysis carried out with the event model representation is valid for this trace.

Definition 8 (Event Model)**.**
An event model $\mathcal{EM}$ consists of pair of distance functions

$$\mathcal{EM} = \langle \delta^+(n), \delta^-(n) \rangle \tag{3.7}$$

with

$$\delta^+ : \mathbb{N}^+ \rightarrow \mathbb{R} \tag{3.8}$$

$$\delta^- : \mathbb{N}^+ \rightarrow \mathbb{R} \tag{3.9}$$

which return an upper / lower bound on the time interval between the first and the last event of any sequence of n event arrivals.

For the sake of simplicity, we assume that event-models (and otherwise related functions) are indexed according to their associated task. In this sense, δ_i is the input event model for task τ_i.

Event distance functions are often given as a compact representation [232] such as the parametrized period (P), jitter (J), minimum-distance (d_{min}) model. This model is of particular interest in the automotive domain where most communication is carried out in a periodic fashion.

$$\delta^-(n) = \max\{(n-1)P - J, d_{min}\} \tag{3.10}$$

$$\delta^+(n) = (n-1)P + J \tag{3.11}$$

Alternatively, an event model can be reverse-engineered from a *sufficiently accurate* trace. It must be noted, that this process is only justified in non safety-critical aspects of the design since it is hard to prove sufficient accuracy.

The distance functions have a pseudo-inverse counter part [168, 242], the event arrival functions. The concept is similar to *arrival curves*, as known from real-time or network calculus [272, 168].

Definition 9 (Event Arrival Functions)**.**
The upper / lower event arrival function η^+ / η^- is an upper / lower bound on the number of events in any half-open time interval of size Δt.

$$\eta^+(\Delta t) : \mathbb{R}^+ \to \mathbb{N} \tag{3.12}$$

$$\eta^-(\Delta t) : \mathbb{R}^+ \to \mathbb{N} \tag{3.13}$$

Event arrival functions are sub-additive, whereas event distance functions are super-additive. More precisely, as [194] showed, the shifted function $\delta(n-1)$ is super-additive.

As Schliecker pointed out in [242], event distance functions and event arrival functions can be derived from each other. Formally, it is sufficient to specify either a pair η, or δ. Our event model definition (Def. 8) uses event distance functions because in practical implementations they can be processed much faster due to the discrete domain and can be implemented by integer operations. The following equations show how η is obtained from δ[1].

$$\eta^+(\Delta t) = \begin{cases} 0 & \text{if } \Delta t = 0 \\ \max\limits_{\forall n \in \mathbb{N}^+} \left\{ n \mid \delta^-(n) < \Delta t \right\} & \text{else} \end{cases} \tag{3.14}$$

$$\eta^-(\Delta t) = \min_{\forall n \in \mathbb{N}^+} \left\{ n \mid \delta^+(n+2) > \Delta t \right\} \tag{3.15}$$

[1] We revised Eq. 3.5 - 3.8 as provided by [242]

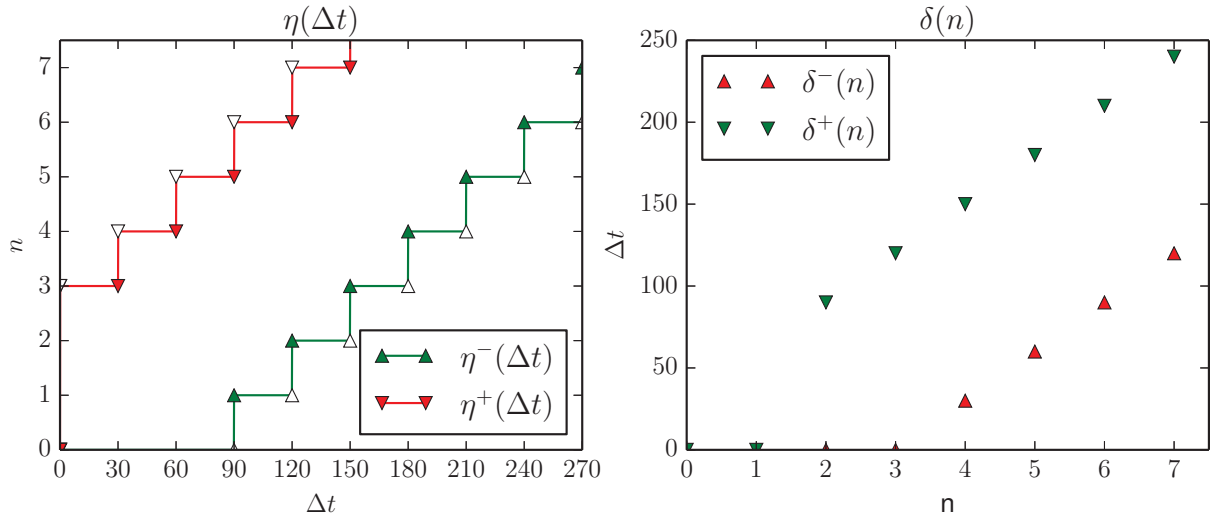


Figure 3.2: Event model for a periodic activation with a period of $P = 30$ and a jitter of $J = 60$.

and δ from η, respectively.

$$\delta^-(n) = \inf_{\Delta t \geq 0, \Delta t \in \mathbb{R}} \left\{ \Delta t \mid \eta^+(\Delta t) \geq n \right\} \tag{3.16}$$

$$\delta^+(n) = \sup_{\Delta t \geq 0, \Delta t \in \mathbb{R}} \left\{ \Delta t \mid \eta^-(\Delta t) < n \right\} \tag{3.17}$$

The relationship between event distance functions and event arrival functions is also shown in Figure 3.2. Both domains represent the same information: A bursty event model with a period $P = 30$ and a jitter $J = 60$.

3.3 Resource Analysis

Resource analysis as used in the CPA framework derives local timing properties such as the task's response time or backlog which typically translates to buffer sizes. Today's analysis approaches emerged from early schedulability analysis such as [178].

Such early work focuses on periodic tasks and computes schedulability based on device utilization for tasks with implicit deadline. Later, algorithms to compute the response-time under more expressive event models (e.g. periodic with jitter) were presented [147, 274, 172]. The approach is to compute the largest time interval for which a resource is busy processing tasks, hence the underlying concept was coined the *busy period* approach. It is based on the *critical instant* assumption in which tasks are assumed to be activated simultaneously.

Most often the critical instant assumption is an over approximation because event streams are correlated. This is because event streams have a common source, correlation is imposed by scheduling or offsets are artificially introduced to improve timing. Analysis support for inter-stream context is presented in [208, 142, 121, 293, 236]. Also most applications have state and thus context between activations. A popular example is a MPEG video decoder in which the I-frames need more processing time than P and B-frames. Such intra-stream context are discussed in [142] and can be analyzed using the busy-period approach [185, 31].

As discussed, the primary goal of the resource analysis is to derive the worst-case / best-case response-time of the task. Throughout the following paragraphs, we will introduce the definitions and concepts of the local resource analysis. The definitions are in-line with [68] and are more generic[2] than the ones provided in [232, 242]. The problem with related response-time formulas is the *stopping condition*. It can be informally phrased as "How many events do we need to consider during analysis to find the worst-case behavior?". Related work in the field of local resource analysis answer this question tailored towards specific scheduling policies such as strict priority preemptive (SPP) and strict priority non-preemptive (SPNP). However, there is no overarching formalism for the stopping condition which applies all scheduling policies (e.g. including FIFO, Round-Robin, and others).

3.3.1 Generalization and Formalism

We only give the main concepts used throughout this thesis, further proofs, remarks, and discussions (especially regarding the difference to [232, 242]) can be found in [68]. For an intuitive understanding we later give a generic formulation of the strict priority preemptive (SPP), strict priority non-preemptive (SPNP) and First In - First Out (FIFO) scheduling policies. These are later extended by adding protocol specific overhead and considering errors.

The timing behavior for each scheduling policy can be described fully by two functions: The multiple-event processing time as well the scheduling horizon. Other properties such as the response-times and the buffer backlog can be derived from these functions.

Definition 10 (Scheduler)**.**
The timing behavior of a scheduler is a set of functions

$$Scheduler = \{H, B^+, B^-\} \tag{3.18}$$

where H is the horizon function and B^+, B^- are the processing functions.

[2] The generalization of the local analysis approach is a joint work together with Jonas Diemer

Definition 11 (Multiple-Event Processing Time)**.**
The maximum and minimum q-event processing time $B^+(q)$ / $B^-(q)$ return lower and upper bounds on the time interval between the arrival of the first event and the completion of the q-th event for any q consecutive events of task τ assuming that all q but the first activation arrive within the scheduling horizon of their predecessors. (see [68])

Informally speaking, the multiple event processing time is the time to process q consecutive events which arrive in the same busy-period. Related work such as [242] assumed "events arrive sufficient early" without further constraints. The notion of the *scheduling horizon* tells us whether two events (or rather associated task executions) influence each other timing-wise. Naturally, if two events are spaced very far apart, there is no influence. That is the execution of the first has no timing impact on the production of the second. However, if two events arrive very closely (i.e. burst), the processing of the second event is delayed by the first event.

Definition 12 (Maximum Multiple-Event Scheduling Horizon)**.**
The maximum q-event scheduling horizon $H(q)$ of any sequence of q events of task τ is a right half-open interval starting with the arrival of the first and ending just prior to the latest time where a hypothetical $q+1$-st activation would receive ϵ service.

In this definition, ϵ service practically means that a $q+1$-th activation would be served an infinitesimal time. We define that an event arriving at the half-open end of a scheduling horizon will get ϵ service. This assumption leads to a bounded scheduling horizon at 100% load and is in practical designs not of interest.

As we will discuss later, for some schedulers also the queuing delay is of interest (i.e. strict-priority non preemptive). The exact definition is given later. However, the interested reader who is familiar with the concept could now be confused between the difference of the scheduling horizon and the queuing delay, hence we give a very simplified reasoning in advance. The queuing delay is the time until at least one *full* processing cycle is available for a task. The scheduling horizon is the time until ϵ service is available, where ϵ can be smaller than a processing cycle.

Practically, the scheduling horizon tells us the scope of influence of q-events to any future events of the same task. Thus, if a $q+1$-th event arrives outside of the scheduling horizon (that is after time $H(q)$ has elapsed measured from the arrival of the first), the execution is unaffected of the previous events.

For backward compatibility to former methods and approaches, we can use the scheduling horizon to compute the busy period.

Definition 13 (Busy Period)**.**
The busy period of a task τ is the right half-open interval starting with the arrival of the first and ending with the scheduling horizon of the q^+ event

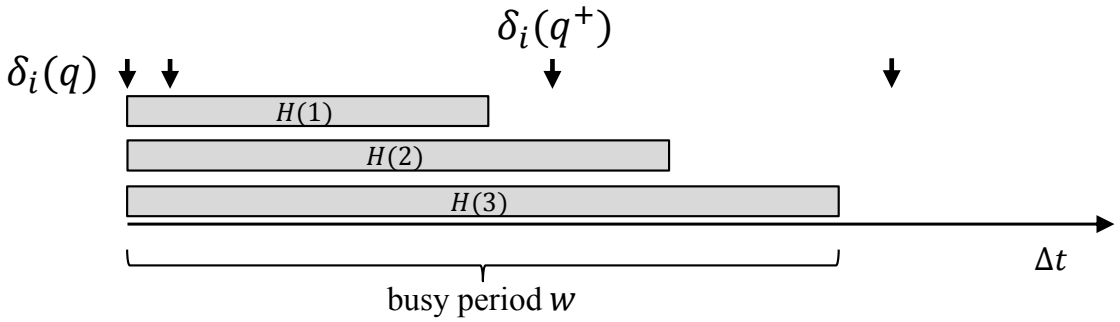


Figure 3.3: The relationship between scheduling horizon $H_i()$, busy-period w and maximum number of events q^+.

such that all but the first of the q^+ events arrive in the scheduling horizon of the preceding event except the $q^+ + 1$-th event.

From this definition we can directly deduce how the busy period is constructed from the scheduling horizon.

$$w = \max_{q \geq 1} \left\{ H(q) | H(q) < \delta^-(q) \right\} \tag{3.19}$$

Also, we can derive the maximum number of events q^+ which reside in the same busy period. This is the maximum number of events which interfere with each other, as there is ϵ idle time between the q^+ and the $q^+ + 1$-th event.

$$q^+ = \eta^+(w) \tag{3.20}$$

This tells us that we can find the worst-case behavior under the first q^+ events in the busy period. We can use this fact, and restrict all further considerations to the first q^+ events.

For the rest of the thesis, we will use either the busy-period definition or the scheduling horizon, depending on which is more intuitive and better fits the considered problem. As sometimes it is more generic to supply the scheduling horizon equation and sometimes it is easier to directly supply the busy-period equation. Generally it is sufficient to derive either the horizon function or the busy-period function.

The illustrative example shown in Figure 3.3 depicts the relationship between the busy-period and the scheduling horizon. The busy-period is the scheduling horizon of the last q^+ event that falls in the previous horizon. In this particular example, the fourth event occurs after the horizon of the third.

Bounding the Timing

Now, all functions are introduced and show how we can find the timing behavior of a task. This includes the response times as well as the timing uncertainty (jitter).

Definition 14 (Response Time)**.**
The response time of a job of task τ is the time from activation of the task until it has been fully processed (and the output event is generated).

Theorem 1. *The worst-case response-time R_i^+ is upper bounded by*

$$R_i^+ = \max_{\forall 0 < q \leq q^+} B_i^+(q) - \delta_i^-(q) \tag{3.21}$$

Proof. Here, $\delta^-(q)$ is by construction a lower bound and $B_i^+(q)$ an upper bound. Thus, the difference is guaranteed to be an upper bound. The maximum number of events which can mutually interfere is bounded by q^+. The following $q+1$-th event will not suffer from interference of the previous q^+ events. If all individual response times for the first q^+ events are maximized, it is guaranteed to find the worst-case response time among them. □

For some scheduling policies such as FIFO, the provided bound is not tight. The scheduling scenario that leads to the construction of the multiple event processing time B^+ contradicts the assumption that events arrive as early as possible according to δ^-. Hence, Theorem 1 is a valid bound but it may be possible that the worst-case response time estimation can further be improved by a scheduling tailored equation. We show this when we discuss the scheduling equations for the FIFO scheduling policy.

Theorem 2. *The best-case response-time R_i^- is lower bounded by the best-case execution time:*

$$R_i^- = B_i^-(1) = C_i^- \tag{3.22}$$

Proof. The proof is straight forward: An event can never be processed faster than the best-case execution time of the associated task. □

For this it is assumed that the best-case execution time is either fixed during analysis or monotonically decreasing for each analysis step. Otherwise, the global fixed point is not guaranteed to be conservative [221].

The uncertainty of the responsiveness of a task is called the jitter.

Definition 15 (Response-Time Jitter)**.**
The response-time jitter is the response-time uncertainty and defined as the difference between the worst-case scenario and the best-case scenario.

$$J_i = R_i^+ - R_i^- \tag{3.23}$$

Bounding the Backlog

Buffer sizes in real implementations are scarce and it must be guaranteed that the queues do not run over, otherwise data is lost. This can happen if events arrive in a large burst and cannot be processed sufficiently fast. Naturally, a lower bound on the backlog is to assume all buffers are empty. However, the lower bound is usually of no further interest. From the scheduling functions, we can compute a worst-case buffer backlog for the queued events.

Theorem 3. *The worst-case backlog is upper bounded by*

$$backlog = \max_{1 \leq q \leq q^+} \{\eta^+(B^+(q)) - q + 1\} \tag{3.24}$$

Proof. As shown before, we only must consider the first q^+ events, as later events perform better or equal. An upper bound on the number of events that arrive during the processing time of the q-th is given by $\eta^+(B^+(q))$, of these events $q-1$ events have already been processed. These can be subtracted which directly leads to eq. 3.24. □

3.3.2 Strict Priority Preemptive (SPP)

We demonstrate how the previous definitions and theorems are applied to a set of independent tasks scheduled by a strict priority preemptive policy. In such a scheduling policy, each task has an associated priority level. At any time, the task in the ready queue with the highest priority is admitted to run on the resource. If during the execution of one task, another task with a higher priority arrives the currently running task is preempted, moved to the ready queue, and the new task is admitted. For the sake of simplicity we neglect context switch overhead [51] as well as blocking due to shared resource access [226], although this can easily be integrated in the following equations. Furthermore, variable execution times is briefly mentioned in [242]. Is can be supported in the model by replacing the execution time terms (e.g. $q \cdot C$) by workload arrival functions which are able to capture the context through multiple incarnations of the task [185]. The following theorem represents the baseline and is extended in later chapters.

Theorem 4. *The maximum Multiple-Event Processing Time for a SPP scheduler (neglecting context switch overhead) is upper bounded by*

$$B^+(q)^{SPP} = q \cdot C_i^+ + \sum_{\forall j \in hp(i)} \left\{ \eta^+ \left(B^+(q)^{SPP} \right) \cdot C_j^+ \right\} \tag{3.25}$$

- *hp(i) is the set of all higher priority tasks mapped to the same resource as task τ_i.*
- *C_i^+, C_j^+ is the worst-case execution time of task τ_i, τ_j.*
- *$\eta_i^+(\Delta t)$ is the maximum number of events of task τ_i in any half-open time interval of length Δt.*

Proof. The proof was given in [242] and [274]. The argumentation is that the q-th event is processed, once the accumulated workload of all higher priority tasks has been processed and all q events of length $q \cdot C_i^+$ are processed. □

Also, as mentioned in related work such as [242, 274, 172, 61], B occurs on both sides of the equation. The ceiling operator introduces discontinuities, thus no simple rearrangement is possible. Due to the monotonicity of the η function, the right side of the equation is monotonic in B, thus a fixed point can be found through iteration [147].

Theorem 5. *For SPP scheduling, the multiple event scheduling horizon for SPP scheduling is given by the corresponding maximum multiple-event processing time.*

$$H(q)^{SPP} = B^+(q)^{SPP} \tag{3.26}$$

Proof. In [172], a proof for the busy period for SPP is presented. As stated in Definition 13 as well as eq. 3.19 the busy period is the scheduling horizon for the last activation (q^+) that falls into a previous scheduling horizon. As shown in [68], the proof given in [172] can be generalized to $q \leq q^+$. □

3.3.3 Strict Priority Non-Preemptive (SPNP)

The Strict Priority Non-Preemptive scheduling analysis is frequently used to analyse fixed priority bus arbitration. If multiple messages are outstanding, the order is determined by the message priority. Once a message is in transmission it is not preemptable and cannot be canceled. Also the automotive operating system OSEK [207] as well as AUTOSAR [12] support a non-preemptive scheduling policy in which task switches are only possible at certain yield points (e.g. task termination). Again, for the following approach we neglect blocking caused by shared resources as well as context switching overhead.

For a SPNP scheduler, the processing time is determined through the queuing delay, which is the time a job of a task is waiting until it receives service. For non-preemptive systems, we know that once the task receives service, there will be no further interference for this event.

Theorem 6. *The maximum Multiple-Event Processing Time for a SPNP scheduler is upper bounded by*

$$B^+(q)^{SPNP} = Q(q) + C_i^+ \tag{3.27}$$

$$Q(q) = (q-1) \cdot C_i^+ + \sum_{\forall j \in hp(i)} \left\{ \eta_j^+(Q(q) + t_{cycle}) \cdot C_j^+ \right\} \tag{3.28}$$

- $Q(q)$ *is the largest time interval from the arrival of the first event until the* q*-th event of* τ_i *receives* ϵ *service.*
- *hp(i) is the set of all higher priority tasks mapped to the same resource as task* τ_i*.*
- C_i^+, C_j^+ *is the worst-case execution time of task* τ_i*,* τ_j*.*
- $\eta_i^+(\Delta t)$ *is the maximum number of events of task* τ_i *in any half-open time interval of length* Δt*.*
- t_{cycle} *is the bittime or cycletime, which needs to be considered due to boundary effects [32].*

Proof. The proof was given in [61]. □

Adding the cycle time t_{cycle} in the argument of the event arrival function is often neglected. This models the discreteness of service. Let's consider a slow bus which runs at a rate of $1/At_{cycle}$. An event arriving ϵ time after a bittime has started cannot get served immediately but has to wait for the start of the next bittime. For processors this effect is usually negligible as events (e.g. interrupts, software calls) are synchronized to the processor clock.

Theorem 7. *The maximum multiple event scheduling horizon for a SPNP scheduler (neglecting context switch overhead) is upper bounded by*

$$H(q)^{SPNP} = q \cdot C_i^+ + \sum_{\forall j \in hp(i)} \left\{ \eta_j^+ \left(H(q)^{SPNP} \right) \cdot C_j^+ \right\} \tag{3.29}$$

- *hp(i) is the set of all higher priority tasks mapped to the same resource as task* τ_i*.*
- C_i^+, C_j^+ *is the worst-case execution time of task* τ_i*,* τ_j*.*
- $\eta_i^+(\Delta t)$ *is the maximum number of events of task* τ_i *in any half-open time interval of length* Δt*.*

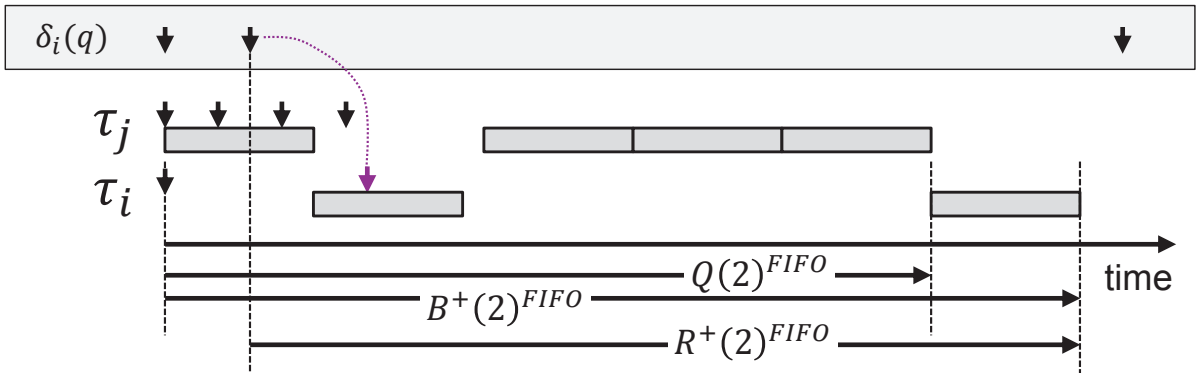


Figure 3.4: Worst-case processing time for FIFO: Interfering jobs from τ_j arrive as quickly as possible, last event (violet) from τ_i arrives just after the last event of τ_j arrives.

Proof. The proof for the busy period was given in [61]. The adaption to multiple events is presented in [68]. The argumentation follows the one for SPP as given in [274], with the difference that the push through interference [61] caused by the non-preemptiveness must be considered. □

3.3.4 First In - First Out (FIFO)

The possibly easiest to implement scheduling and arbitration policy is First In - First Out [253]. A single queue is implemented and jobs are processed in the order they enter the queue. This scheme is often found in switched networks (e.g. Ethernet, NoC) and bus controllers (e.g. some CAN controllers as discussed in [62]) where arbitration is implemented in hardware and resources are scarce.

The analysis of FIFO scheduling is similar to Earliest Deadline First Scheduling as EDF behaves like FIFO if deadlines are set to $D = 0$ [257, 208, 111]. Note that this analogy is restricted to modelling and the deadline does not have the notion of a deadline but rather a scheduling parameter. In this sense the task will not fail if this deadline of zero is exceeded. An analysis targeted towards Ethernet AVB [130] is presented in [74]. It applies to mixed fixed-priority policy with multiple FIFO queues but does not sufficiently capture non-preemptiveness.

The FIFO analysis is an example for which the previously provided theorems provide conservative results but can further be improved to capture the peculiarities of the scheduling policy. Hence, we first present the set of equations and then explain the problem.

Theorem 8. *The maximum Multiple-Event Processing Time for a FIFO scheduler is upper bounded by*

$$B^+(q)^{FIFO} = Q(q) + C_i^+ \tag{3.30}$$

$$Q(q) = (q-1) \cdot C_i^+ + \sum_{\forall j \in \textit{fifo(i)}} \left\{ \eta_j^+(Q(q) + t_{cycle}) \cdot C_j^+ \right\} \tag{3.31}$$

- $Q(q)$ *is the largest time interval from the arrival of the first event until the* q*-th event of* τ_i *receives* ϵ *service.*
- *fifo(i) is the set of all tasks mapped to the same queue as task* τ_i*.*
- C_i^+, C_j^+ *is the worst-case execution time of task* τ_i*,* τ_j*.*
- $\eta_i^+(\Delta t)$ *is the maximum number of events of task* τ_i *in any half-open time interval of length* Δt*.*
- t_{cycle} *is the bittime or cycletime, which needs to be considered due to boundary effects [32].*

Proof. Under worst-case assumptions, the q-th event arrives ϵ time after all previous $q-1$ jobs finished and all interfering jobs arrived. If the interfering jobs arrive as quickly as possible, and the q-th event of task τ_i as late as possible, the interference is maximized. □

The construction of the worst-case multiple event processing time is also shown in Figure 3.4. The upper part of the figure in the grey box shows the earliest arrival times of events which belong to task τ_i. The Gantt diagram below, shows the scenario that leads to the worst-case processing time. Here the q-th event (violet) arrives just after all interferer events arrived. Note that it is not conservative to assume that events of task τ_i arrive as quick as possible (i.e. what is shown in the grey box).

Theorem 9. *The maximum multiple event scheduling horizon for a FIFO scheduler (neglecting context switch overhead) is upper bounded by*

$$H(q)^{FIFO} = q \cdot C_i^+ + \sum_{\forall j \in \textit{fifo(i)}} \left\{ \eta_j^+ \left(H(q)^{FIFO} \right) \cdot C_j^+ \right\} \tag{3.32}$$

- *fifo(i) is the set of all tasks mapped to the same queue as task* τ_i*.*
- C_i^+, C_j^+ *is the worst-case execution time of task* τ_i*,* τ_j*.*
- $\eta_i^+(\Delta t)$ *is the maximum number of events of task* τ_i *in any half-open time interval of length* Δt*.*

Proof. A worst-case is constructed if all events from all τ_i as well as all q events of τ_i arrive as quickly as possible, generating the highest possible workload. Then, the scheduling horizon is determined by the time, the queue busy. After the queue is busy it can provide at least ϵ service. The latest time after which the queue has processed all events and thus is empty again, is after all workload associated with these events is processed. The workload is the sum of the execution times which arrived during the processing according to eq. 3.32. When the queue is empty a potential $q+1$-th event would get ϵ service. □

Now, the response time is computed according to eq. 1 by subtracting the earliest arrival time from the latest processing time. This is also shown in Figure 3.4. However, in FIFO scheduling, the scenario that leads to the worst-case processing time is (in some cases) mutual exclusive with the assumption that events of task τ_i arrive as quickly as possible. Thus, there is no consistent scenario that actually leads to the depicted response time of $R_i^+(2)$. For the computation of the response time, we assume an earlier arrival time of what we assumed for the construction of the processing time. Again, it must be noted that the result is a conservative safe bound, but we want to improve the response-time bound as good as possible.

Obviously, in FIFO scheduling the interference depends on the arrival time of the q-th event. The earlier it arrives, the less interference.

Corollary 1. *The response-time of the q-th event of task τ_i, assuming it arrives in the scheduling horizon of the $(q-1)$-th event is upper bounded by*

$$R_i^+(q,a) = Q_i(a) - a + C_i^+ \tag{3.33}$$

- *a is the arrival time of the q-th event of task τ_i.*
- *C_i^+ is the worst-case execution time of task τ_i.*
- *$Q_i(a)$ is the queueing delay for the q-th event, assuming it arrives at time a.*

Proof. Eq. 3.33 directly follows the definition of the response time (Definition 14), it is time from the arrival (a) until the event has been processed. The latest time, the event is processed is the queueing delay $Q_i(a)$ plus the execution time C_i^+. □

A graphical interpretation of eq. 3.33 is shown in Figure 3.5. The queuing delay for some fixed q is shown over a range of values a. The function $Q(q,a)$ is stepped with respect to a, this is because the interference arrives in chunks accoring to the η function. For a given value of a the response-time is the distance from the queuing delay curve to the linear function $a - C^+$. The queuing delay can be obtained according to the following theorem.

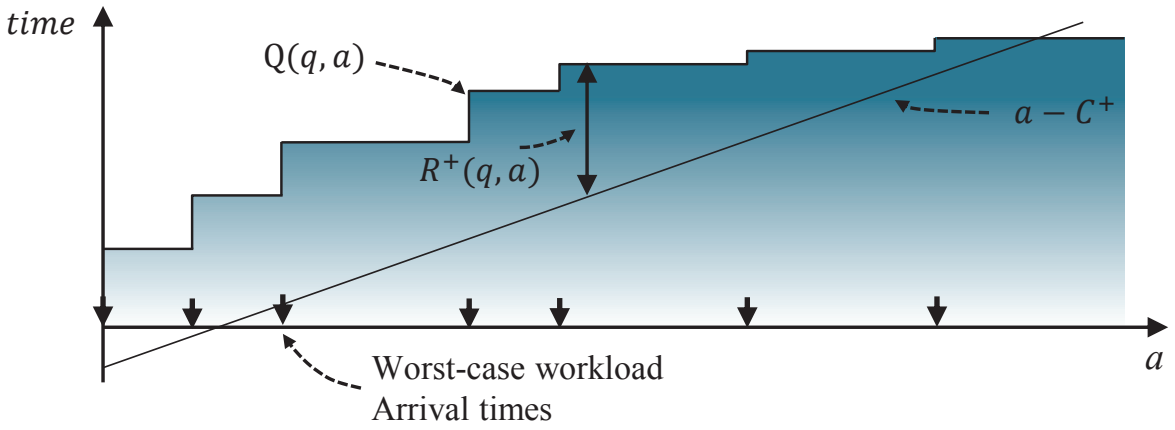


Figure 3.5: Graphical interpretation of eq. 3.33. The response time $wcresponse[i]q, a$ is the distance between the curves $Q(q, a)$ and the linear function $a - C_i^+$.

Theorem 10. *The maximum queuing delay for the q-th event released at time a and assuming the release is in the scheduling horizon of the preceding event is bounded by*

$$Q_i(q, a) = (q - 1) \cdot C_i^+ + \sum_{\forall j \in fifo(i)} \left\{ a + t_{cycle} \right) \cdot C_j^+ \} \tag{3.34}$$

- *$Q_i(q, a)$ is the largest time interval from the arrival of the first event until the q-th event of τ_i receives ϵ service.*
- *fifo(i) is the set of all tasks mapped to the same queue as task τ_i.*
- *C_i^+, C_j^+ is the worst-case execution time of task τ_i, τ_j.*
- *$\eta_i^+(\Delta t)$ is the maximum number of events of task τ_i in any half-open time interval of length Δt.*
- *t_{cycle} is the bittime or cycletime.*

Proof. The proof follows eq. 3.31 with the difference that only interference which arrives prior to time a must be considered. Events which arrive later, are queued behind the event of interest and do not interfere. If events arrive simultaneously at time a, we assume that the q-th event arrives t_{cycle} after the others, hence the worst-case order is assumed and all events are added to the interferer set. □

The question remains which value of a leads to the worst-case response-time. A conservative approach is to evaluate all candidates in the interval

bounded by the earliest possible arrival and the latest possible arrival determined by the scheduling horizon $a \in [\delta_i^-(q), H_i(q))$. This can be a challenging task and given $a \in \mathbb{R}$, the number of candidates is uncountable. As already mentioned, the queuing delay is a step function in a, with steps at the earliest event arrival according to $\delta_j^-(n)$. Without further proof, we can tell that the maximum of the difference of a step function and a linear function is obtained by only considering the steps.

Theorem 11. *The worst-case response time $R_i^+(q)$ is found by evaluating eq. 3.33 at the steps of $Q_i(q,a)$.*

$$R_i^+(q) = \max_{\forall a \in \mathcal{A}} \{R_i^+(q,a)\} \tag{3.35}$$

$$\mathcal{A} = \bigcup_{\forall j \in \mathit{fifo}(i)} \left\{\delta_j^-(n) \mid \delta_i^-(q) \leq \delta_j^-(n) < H_i(q)\right\} \tag{3.36}$$

Proof. After a step occurs in $Q(q,a)$, the function stays constant, whereas $a - C^+$ is linearly increasing in a. Thus, the difference between the function $Q(q,a)$ and $a - C_i^+$ is strictly monotonically decreasing to the next step. Thus, we can conclude that it is sufficient to evaluate the steps to find the maximum response-time. An illustrative example is shown in Figure 3.5. □

Similarly, other scheduling policies or protocols show improvement potential, where a direct response time equation, similar to 3.33 gives tighter results, than the approximation according to eq. 1. As already mentioned, earliest deadline first scheduling, which is not discussed in this thesis, belongs to this class of schedulers. Often, for a first timing assessment the reader is advised to start with a simple approximation and refine when really necessary.

3.4 System Analysis

We presented the local resource analysis step which considers resources and tasks in isolation. In complex system, tasks are cascaded, exchange data and have complex non-functional interactions. The goal of the system analysis is to analyse complex application graphs with large task and event chains. As discussed in [120], there are functional as well as non-functional dependencies among tasks. Functional dependencies directly arise from the communication relations, whereas non-functional dependencies are introduced by scheduling artifacts. A higher priority task for instance will delay a low priority task on the same resource. Naturally, this affects the entire communication chain of the low priority task. Also event models

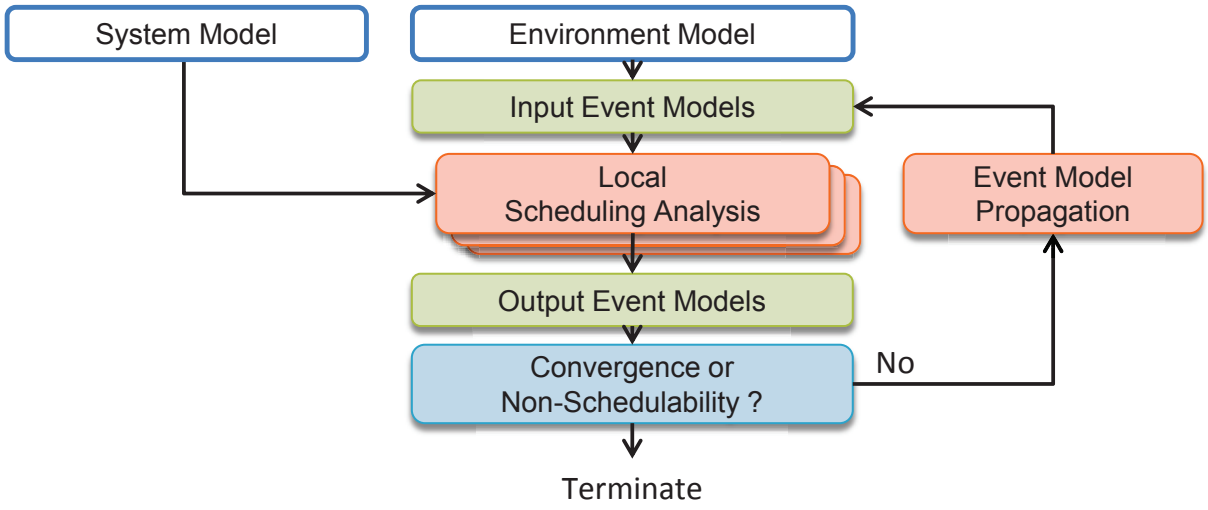


Figure 3.6: Compositional analysis flow with the major steps *local scheduling analysis* and *event model propagation*.

along the task chains are not known a-priori. Also arbitrary complex cyclic dependencies prevent a simple feed-forward analysis where tasks are analyzed in isolation. To solve the problem of mutual dependencies, the system analysis can be formulated as a fixed point problem [210, 232, 242, 258]. This concept is also used in related approaches such as real time calculus (RTC) [282, 272].

The analysis flow, as shown in Figure 3.6, consists of two interleaved steps: the *local analysis* (explained in the previous section) and the *propagation* of event models. The environment model specifies the boundary conditions for each stream under which the system is operating. This is usually given as external event models which describe the characteristics of external sources (i.e. video cameras, controllers). Other, yet unknown, event models are initialized with optimistic guesses that are iteratively updated during the analysis. The event models are used for the local resource analysis during which the local behavior is considered in isolation. Among other results, the local analysis yields the response-time jitter J_{resp}. As shown in [120, 232], given the input event model as δ_{in}^{-}, we can obtain the output event model δ_{out}^{-} using the following equation:

$$\delta_{out}^{-}(q) = \max\{(q-1)\cdot C^{-}, \delta_{in}^{-}(q) - J_{resp}\} \tag{3.37}$$

This output event model is then used as the input event model for the following task of the chain. More sophisticated approaches to compute a tighter bound for the output event model is presented in [241] and [222]. These approaches leverage the fact that out of q events, not all events are delayed by the worst-case response time.

The iteration is stopped and a system fixed point is found, if the worst-case response times and thus the all event models remain stable.

Often, the latency along a path is of interest. This is the case in automotive systems where signals are transfered across multiple ECUs, buses and gateways. A simple approach to compute the latency is to add the worst-case response times along a path. Often, the scenarios that lead to the worst-case situation on one resource are mutual exclusive with a scenario that leads to the worst-case response time on another resource. This can be exploited to improve the path latency bound. A recursive path latency algorithm which performs better than a simple response-time summation is presented in [242].

Often data is fragmented over multiple frames [246, 64, 137]. A typical example for this is Ethernet or NoC communication, where large data streams are first packetized and then sent in flits and phits. To compute the latency of the data packet we must compute the latency of multiple events which are associated with the smallest data entity (e.g. phits or Ethernet frames). Here, we can use the following equation (cf. [73]):

$$L(q)_i = \delta_i^-(q) + \sum_{\forall j \in \mathit{Path}(i)} R_j^+ \tag{3.38}$$

Here, $\mathit{Path}(i)$ is the set of all tasks, which belong to a path i. The idea is to predict the injection time of q events and assume the last event observes the worst-case path latency. By causality, all previously sent events must have arrived by then.

3.5 Summary

In this chapter, we summarized the state of the art in timing verification of complex embedded systems. We have shown how larger systems are decomposed into a functional as well as architecture model. These models are the starting point for a scalable system analysis concept through which response-time, path latency and buffer backlog values can be computed by nested fixed-point iteration.

The building blocks of CPA were harmonized by providing a unified theory which is valid through a wide range of scheduling policies, where approaches presented in literature required a formalism tailored towards a specific scheduling problem. Here, we showed that the busy-period known from literature is a special case of the presented scheduling horizon. We proofed the conservativeness of these unified bounds and showed scenarios where a hand-tailored set of equations (i.e. FIFO) can yield tighter results.

CHAPTER 4

Multi-Master and Point-to-Point Communication

This chapter addresses the effects of bit errors, packet errors and error handling in buses supporting multi-master as well as master-slave communication. The approaches can be applied to on-chip buses such as AMBA [7] or large field-buses such as the Controller Area Network [233] and its successor CAN-FD [113]. In scope of the CPA system analysis approach, the presented methodologies are resource analyses as they operate on a local level. In scope of the ASTEROID architecture, the presented methodology is applied to on-chip communication (AMBA) as well as off-chip communication (CAN) to interface ASTEROID with the environment.

In the first section, the error models and their properties are presented. The second and third section focus on the two approaches to obtain a probabilistic response-time bound considering random error effects. Finally, the approaches are applied to an automotive use-case to obtain the performance bounds and judge the reliability of communication considering typical environments.

The chapter is partially based on the work published in [21, 15, 16].

4.1 Channel Model

Communication in digital as well as in analog systems take place between a sending station and one (or multiple) receiving station(s). Data is transmitted over a channel such as a wire, optical or over the air. Errors caused by electrical surroundings superimpose the actual data stream. There are external sources for errors such as high power radar stations, but also in-

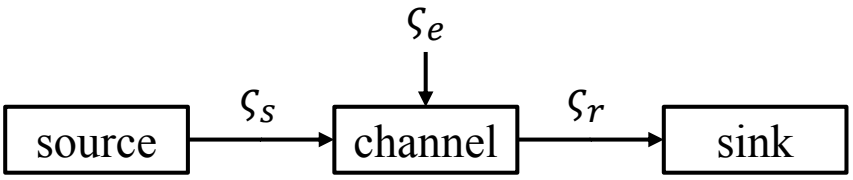


Figure 4.1: Channel model: source transmits a data stream ς_s, the channel superimposes an error stream ς_e, and a altered data stream ς_s is received at the sink.

ternal sources such as the inductive switching characteristics of an electric motor. Figure 4.1 shows the channel model. The source (i.e. sending node) sends a stream which is superimposed by errors and then received at the sink (i.e. receiving nodes).

If errors occur during the transmission, one (or multiple) receivers sense the error as an altered symbol (i.e. bitflip). The receiver eventually recognizes the problem depending on the coding properties and either corrects the data stream on the fly or signals an error condition on a return path. For the correction on the fly error-correction codes are used, whereas error-detection codes are used for the latter approach. However, the actual mechanism and its resilience is protocol dependent and discussed throughout the analysis sections.

Now, we formalize the channel model.

Definition 16 (Data Stream)**.**
A data stream of length $n \in \mathbb{N}^+$ (the stream length), over field $A = \mathbb{F}^2$ is defined as $\varsigma \in A^n$.

For our considerations, we restricted the definition to a binary alphabet $\mathbb{F}^2 = \{0, 1\}$, however the following concepts are valid for finite fields of length m with mod-m addition and multiplication. Since information on typical field busses is modulated represented as bits (e.g. NRZ coding), such modeling is natural[1].

Definition 17 (Syndrome)**.**
Given a transmitted data stream ς_s and a received data stream ς_r, the error syndrome is defined as

$$\varsigma_e = \varsigma_s \oplus \varsigma_r \tag{4.1}$$

with $\oplus$ being the element-wise mod-m addition.

[1]Gigabit Ethernet uses 5-PAM with $m = 8$

In binary transmission $\oplus$ is a xor, thus the syndrome is retrieved as difference (xor) between the transmitted and received data.

Practically, the syndrome represents the error pattern which occurred during the transmission.

Definition 18 (Error Event)**.**
An error event is the state in time $t \in \mathbb{N}^+$ at which the syndrome is non-zero $\varsigma_s(t) \neq a_0$.

Practically, an error event is a discrete time at which an error alters the transmitted data. Hence, for binary systems, a syndrome is simply a list of error events.

4.2 Error Models

A vast number of error models have been presented in literature. Error models can be subdivided into *descriptive* [1] as well *generative* [148] error models. A generative model is used to generate and simulate the error process, whereas a descriptive model is typically used to assess the statistics (e.g. bit error rate, variance) of a sampled process. Throughout the rest of the section, we first introduce key descriptive parameters and then discuss selected generative models. Although we do not use these models for simulation purposes, but rather for a formal stochastic consideration.

In signal processing, we must differentiate on which level an error model is applied. Parametrized models for the *analog domain* (e.g. voltage level) have been presented. Here, additive white gaussian noise (AWGN) is one of the most simple models [202], in which the amplitude follows a normal distribution. Such a model can be used to describe the effects of thermal noise in (e.g. in resistors) and simple wireless channels. More sophisticated error models exist to reflect more complicated effects such as fading in multipath channels, scattering, interference and other wireless effects. For instance Rice fading and Rayleigh fading models [209] can be applied to capture and simulate the analog effects by using MATLAB simulation.

For digital processes, such as bit-level or packet-level transmission, generative models such as Markov-chains or Hidden Markov Models (HMM) are used [284]. Of special interest due to their simplicity are the Gilbert model [100], the Gilbert-Elliot model [85] as well as the Fritchman model [254].

The task of the error models, which are introduced in the following sections, is to model the (to be expected) error syndrome with justifiable accuracy.

Throughout the rest of this section, we introduce two commonly used bit error models. These models can also be interpreted as packet (or frame)

error models. We choose two simple error models that can be used as generators for simulations but also to derive additional statistical information such as the predicted number of error events in a given time frame.

4.2.1 Descriptive Parameters for Lossy Channels

We will now introduce some key parameters which are typically used to quantify the error environment. It must be noted that the following metrics are not directly considered as an error model, as they are of descriptive nature.

Definition 19 (Bit Error Rate)**.**
The bit error rate (BER) λ of a channel is the ratio of altered symbols of the received data stream ς_r to the total number of transmitted symbols n:

$$\lambda = \frac{d(\varsigma_s, \varsigma_r)}{n} \tag{4.2}$$

with $d(\varsigma_a, \varsigma_b)$ being the hamming distance between both data streams:

$$d(\varsigma_a, \varsigma_b) = |\{i \mid 1 \leq i \leq n, \varsigma_a(i) \neq |\varsigma_a(i)\}| \tag{4.3}$$

The hamming distance is the number of bits which are different in both data streams.

The previous definition of the bit error rate is based on the standpoint of an external clairvoyant observer, who only sees the transmitted *and* received streams. If the syndrome is known (or was derived), a more straight forward computation of the BER is possible by counting the number of non-zero symbols in the syndrome.

$$\lambda = e/n \tag{4.4}$$

with

$$e = |\{i \mid 1 \leq i \leq n, \varsigma_s(i) \neq a_0\}| \tag{4.5}$$

where a_0 is the additive identity of the field $\mathbb{F}^m$, (i.e. for $\mathbb{F}^2$, $a_0 = 0$).

Another key property is the correlation of the error events. Obviously, it makes a difference if errors are identically distributed over the entire transmission time or if they are clustered and errors occur in bursts. There is no consistent definition of a burst. Figure 4.2 shows an example of a transmission which is disturbed by errors. There are four clusters of error events. It is unclear if the two error clusters in the middle of the figure belong to a single burst or are in fact two individual bursts.

We use the latter interpretation, because this allows to identify bursts uniquely which are always separated by a at least one correct symbol.

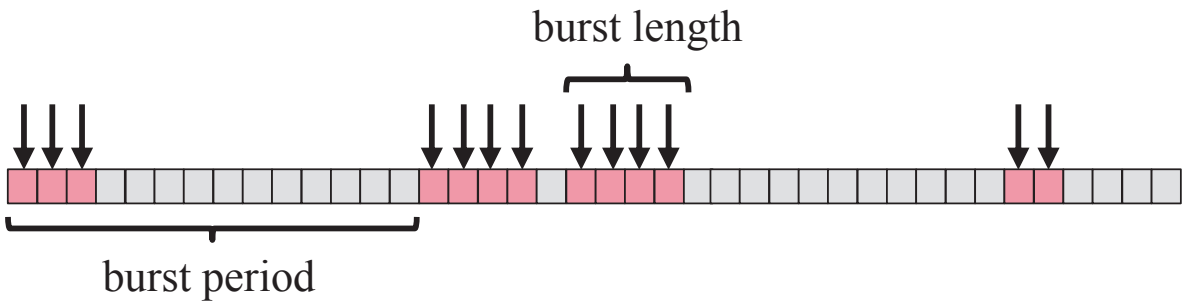


Figure 4.2: Burst errors and key properties such as burst length and burst period.

Definition 20 (Burst)**.**
A burst of length b is a syndrome ς_s of length b with non-zero elements:

$$\varsigma_s(i) \neq a_0 \forall 1 \leq i \leq b \tag{4.6}$$

Any syndrome ς_s can be decomposed into a sequence of alternations of a non-zero syndrom (a burst error, nz) and a zero-syndrome (no errors, z).

$$\varsigma_s = \ldots, \varsigma_1^{nz}, \varsigma_1^{z}, \varsigma_2^{nz}, \varsigma_2^{z}, \ldots \tag{4.7}$$

The previous definition is a pragmatic model, which fits the needs of this research. There are also other burst definitions. For instance, in [82] the distance between the first and the last erroneous bit determines the length of the burst.

Definition 21 (Burst Length)**.**
For a syndrome ς_s with m bursts, the average burst length $\bar{b}$ is defined as

$$\bar{b} = \frac{1}{m} \sum_{i=1}^{m} n_i^{nz} \tag{4.8}$$

where n^{nz} is the number of symbols in ς_i^{nz}

Definition 22 (Burst Period)**.**
For a syndrome ς_s with m bursts, the average burst Period $\bar{b}$ is defined as

$$\bar{b} = \frac{1}{m} \sum_{i=1}^{m} \{n_i^{nz} + n_i^{z}\} \tag{4.9}$$

where n^{nz} is the number of symbols in ς_i^{nz}

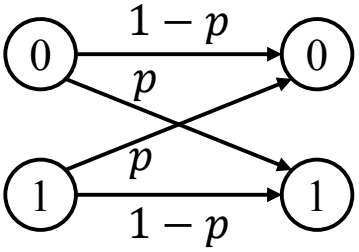


Figure 4.3: Binary symmetric channel: p denotes the probability of a flipped bit.

4.2.2 Binary Symmetric Channel

As we do not know the actual pattern of the syndrome a priori, we can approximate it using a stochastic process. The random variable in this process is the error over time. This is a common modelling approach [231].

Let X_t be the value of a random variable at discrete time $t \in \mathbb{N}$, then the series of random variables $X_0, X_1, \ldots$ is called a process. We use the following definition as given in [38]:

Definition 23 (Stochastic Process)**.**
A stochastic process is defined by the collection of random variables $\{X_t : t \in I\} \in S$ *on a probability space with a sample space* Ω*, a set of events* $\mathcal{F}$ *with the assigned probabilities* P*.* S *is called the state space, and* I *the index set.*

The series ς_s can be grasped as a stochastic process with $\varsigma_s(t) = X_t$, and $\Omega = \{1, 0\}$. Then S is the set of all possible error scenarios over the mission time.

The simplest error model for digital transmission over a binary channel is the binary symmetric channel (BSC) [175] or sometimes called single bit error model. In this model, error events are uncorrelated and each bit is considered in isolation. Figure 4.3 shows the conceptual model. The left side shows what is transmitted and the right what is received. The transmitted information is distorted with a probability $p \in [0, 1]$, and the receiver receives the correct information with a probability $1 - p$. We assume that the error process is stationary, hence p does not change over time (or changes so slowly that it is reasonable to assume stationarity).

The channel capacity of a BSC, which is the relative amount of transmitted information per time can be computed as follows:

$$c = 1 + p\log(p) + (1 - p)\log(1 - p) \tag{4.10}$$

Assuming a CAN bus with a data rate of 500 kbit/s and $p = 1e - 6$, the capacity is 0.999985 which results in an "usable" datarate of $499,992$ bits/s.

In case each error is independent and identically distributed (i.i.d.) and the probability of a flipped bit is p for all bits, we can model the channel behavior using a Bernoulli process[150].

Definition 24 (Bernoulli Process)**.**
A Bernoulli process is a stochastic process with $\Omega = \{0, 1\}$ *and* $I = \mathbb{N}$ *where each random variable* X_t *obeys*

$$\begin{aligned} P[X_t = 1] &= p \\ P[X_t = 0] &= 1 - p \end{aligned} \tag{4.11}$$

Assuming we transmit n symbols over a BSC, we want to compute the probability that a given error sequence ς_s occurs. The probability can be computed by multiplication, because the random variables are independent:

$$P[X = \varsigma_s] = \prod_{i=0}^{n} P[X_i = \varsigma_s(i)] \tag{4.12}$$

Often, it is necessary to know the probability to observe a given number of errors $n_e \in \mathbb{N}$ during a total transmission of length n.

Theorem 12. *The number of errors in a transmission of length* n *in a BSC follows a Bernoulli distribution:*

$$P[n_e, n] = \binom{n_e}{n} p^{n_e} \cdot (1 - p)^{n - n_e} \tag{4.13}$$

Proof. Proof is straight forward (i.e. [266]). □

Theorem 13. *The distance between error events is a geometric random variable* N_c *with the following distribution:*

$$P[N_c = x] = p^x (1 - p) \tag{4.14}$$

Proof. Proof is straight forward (i.e. [266]). □

When we consider large data frames or a long stream of data, the computation of $P[n_e, n]$ for $n_e \ll n$ can become a tedious task because it is complex to compute the binomial $\binom{n_e}{n}$.

Theorem 14. *The probability distribution for a number of errors* n_e *in a transmission of length* n *in a BSC can be approximated by a Poisson distribution:*

$$P[n_e, n] \approx \frac{(np)^{n_e}}{n_e!} e^{-np} \tag{4.15}$$

Proof. Reasoning is given in [266]. □

As a good rule of thumb, this approximation is valid under the following conditions [198]:

> "The sample size n should be equal to or larger than 20 and the probability of a single success, p, should be smaller than or equal to 0.05. If $n \geq 100$, the approximation is excellent if np is also ≤ 10."

This is the case for almost all further considerations, as typical data frame sizes are larger than 20 bits and the error probability p is typically in the order (or below) of $p = 10^{-6}$.

Practically, the presented approximation transforms the discrete process into a continuous time domain of infinitesimal symbol time (or bit time for binary channels) where the time distance between exactly N_c errors is exponentially distributed:

$$P[N_c = x] = (np)e^{-npx} \tag{4.16}$$

It must be highlighted that the geometric distribution as well as the (continuous) exponential distribution are memoryless. That is, the time between errors equals the time to error. Practically, this means if an observer begins the observation at time t_0, eq. 4.14 and 4.16 can be used to determine the distribution of when the next error is observed.

Obtaining Model Parameters

The binary symmetric channel model has only one parameter, p. In practical application p determines the bit error rate. Hence, if measurements (data traces and syndromes) are available, we can use eq. 4.4 and directly set $p = \lambda$. Often, only the packet error rate is available and a bit error rate must be derived.

Definition 25 (Packet Error Rate)**.**
The packet error rate λ_p is the relative number of corrupted packets p_c to the transmitted packets n_p.

$$\lambda_p = p_c/n_p \tag{4.17}$$

Hardware controllers, such as CAN and Ethernet controllers are only capable to count the packet loss and corruptions. There are no further means available to detect and identify single corrupted bits. For packets of length n we find that on average a packet is corrupted with probability p_p by applying eq. 4.13.

$$\begin{aligned} p_p &= 1 - P[n_e = 0, n] \\ p_p &= 1 - (1 - \lambda)^n \end{aligned} \tag{4.18}$$

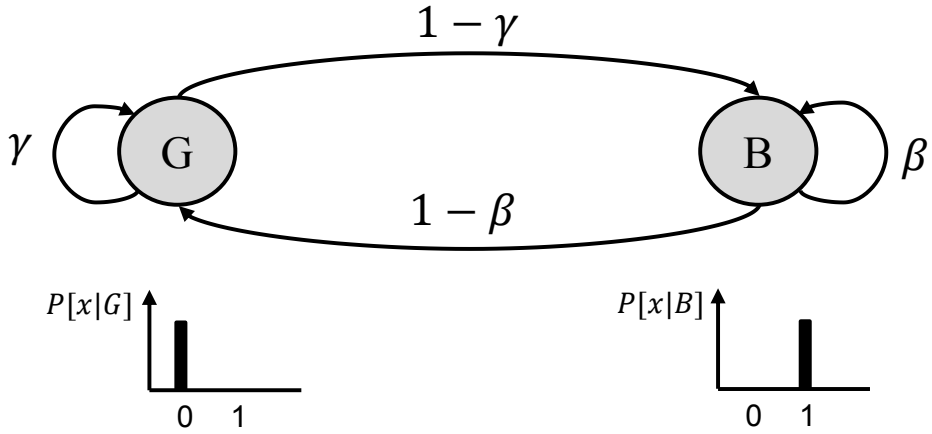


Figure 4.4: Two-state Gilbert loss model.

by rearranging the equation for λ, we get

$$\lambda = \sqrt[n]{1 - p_p} \tag{4.19}$$

4.2.3 Two State Gilbert Loss Model

The previously introduced binary symmetric channel model is not capable to reflect burst errors which can occur due to fading in wireless channels but also in wired communication caused by repetitive emission of electro magnetic interference (EMI) bursts as seen in electric vehicles. In such environments it is more likely that an error occurs if an error has been seen in the recent past. This can lead to scenarios where an error spans multiple symbols or data frames and certain error patterns are more likely than others. Note that in this section model the occurrence of (burst-) error events, the effect on packets is part of an analysis which is discussed later in this thesis.

In this work, we use the two state Gilbert loss model [100], which is a special case of a Hidden Markov Model (HMM) [86] and is an extension of a discrete time Markov chain.

The two-state Gilbert loss model is depicted in Figure 4.4. The state *G* (good) represents the reception of correct bits whereas the state *B* (bad) represents a bit-flip. The conditional transition probabilities are given as $P[G|G] = \gamma$ as well as $P[B|B] = \beta$, respectively. The probability of observing the system in the good/bad state $g[t+1]$ / $b[t+1]$ at time $t+1$ purely depends on the state previous to the observation at time t. In the context of Markov processes, this property is called memorylessness.

$$g[t+1] = \gamma g[t] + (1-\beta)b[t] \tag{4.20}$$

$$b[t+1] = (1-\gamma)g[t] + \beta b[t] \tag{4.21}$$

Similarly, the steady state probabilities of finding the system in the good or bad state at some given time are given as

$$g_{ss} = \frac{1-\beta}{2-\gamma-\beta} \tag{4.22}$$

$$b_{ss} = \frac{1-\gamma}{2-\gamma-\beta} \tag{4.23}$$

From this, we can conclude that the average bit error rate λ for a Gilbert loss model is given by $\lambda = b_{ss}$. Often, we are interested in particular error patterns and the probability $P(n_e, n)$ of observing n_e errors in n transmitted bits.

Theorem 15. *The number of erroneous bits n_e given a total transmission of length n and ending in the* G/B *state follows the distribution given by $P[n_e, n, G]$ / $P[n_e, n, B]$.*

$$\begin{aligned} P[n_e, n, G] =& g[0]\gamma^{n-2n_e}(1-\beta)(1-\gamma) \\ & \cdot \sum_{i=0}^{n_e-1} \binom{n_e-1}{i} \binom{n-n_e}{i+1} (\beta\gamma)^{n_e-1-i}[(1-\beta)(1-\gamma)]^i \\ & + b[0]\gamma^{n-2n_e-1}(1-\gamma) \\ & \cdot \sum_{i=0}^{n_e} \binom{n_e}{i} \binom{n-n_e-1}{i} (\beta\gamma)^{n_e-i}[(1-\beta)(1-\gamma)]^i \end{aligned} \tag{4.24}$$

$$\begin{aligned} P[n_e, n, B] =& g[0]\gamma^{n-2n_e+1}(1-\gamma) \\ & \cdot \sum_{i=0}^{n_e-1} \binom{n_e-1}{i} \binom{n-n_e}{i} (\beta\gamma)^{n_e-1-i}[(1-\beta)(1-\gamma)]^i \\ & + b[0]\gamma^{n-2n_e}(1-\beta)(1-\gamma) \\ & \cdot \sum_{i=0}^{n_e} \binom{n_e}{i+1} \binom{n-n_e-1}{i} (\beta\gamma)^{n_e-i-1}[(1-\beta)(1-\gamma)]^i \end{aligned} \tag{4.25}$$

where $g[0]$ / $b[0]$ denote the initial conditions that the channel is in the good / bad state.

Proof. The closed form as given above as well as a recursive formulation was provided in [295]. □

The probability of n_e errors out of n transmitted bits, irrespective of the final state of the Markov chain is the sum of the previously provided equations:

$$P[n_e, n] = P[n_e, n, B] + P[n_e, n, G] \tag{4.26}$$

Analogous to eq. 4.13, this can be used to obtain the likelihood of observing a given number of errors, regardless of the actual pattern.

Obtaining Model Parameters

As in most models, the parameters are not directly observable and must be retrieved from measurements. In [100], a convenient way is described to derive γ and β by computing $P[1]$ and $P[1|1]$ from a measured error syndrome. For further considerations, the estimation provided in [100] is slightly simplified[2]. This leads to the following estimations:

$$\gamma = P[1] \tag{4.27}$$

$$\beta = 1 - \frac{P[1](1 - P[1|1])}{1 - P[1]} \tag{4.28}$$

Alternatively, if no measured syndrome is available but other key properties such as average burst length L and bit error rate λ can be estimated, we can use these values to obtain γ and β:

$$\gamma = \frac{\lambda(2 - \beta) - 1}{\lambda - 1} \tag{4.29}$$

$$\beta = 1 - \frac{1}{L} \tag{4.30}$$

The previous estimations are derived from the associated burst length distribution $b[n|B]$:

$$b[n|B] = \beta^n \tag{4.31}$$

The mean burst length is obtained as the limit for $n \to \infty$ of eq. 4.31 which forms a geometric series and directly leads to eq. 4.30. In a second step, the steady state probability eq. 4.23 can be solved for γ which directly leads to eq. 4.30.

4.3 Probabilistic Response-Time Analysis under Errors

In the previous section, we discussed the channel model, single bit errors as well as a simple burst error model. In bus communication and switched

[2]The error rate in the B state is set to 1.

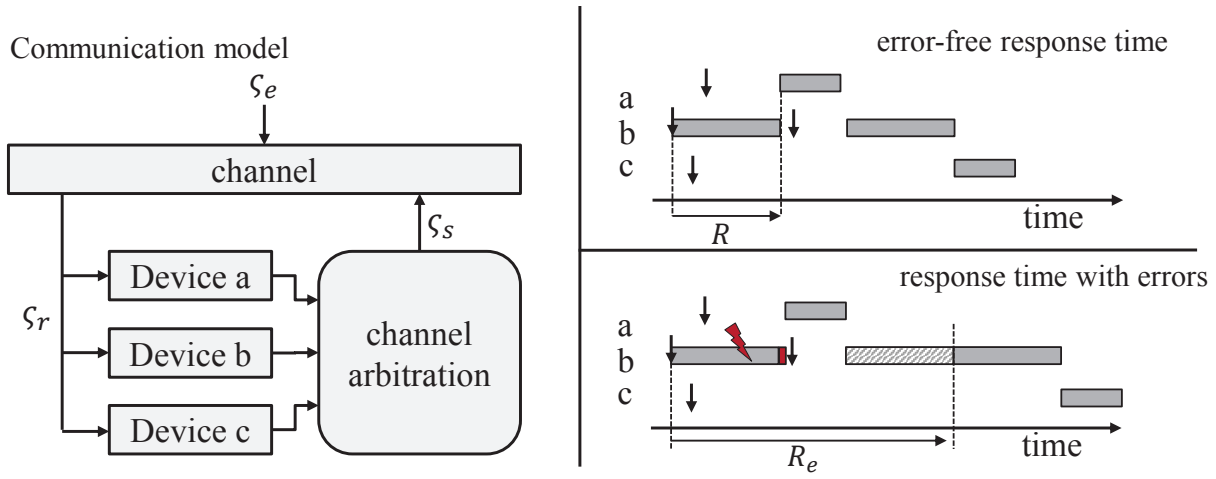


Figure 4.5

networks, data is typically sent in a sender-receiver (e.g. LIN [177], Ethernet) or publisher-subscriber (e.g. CAN [233]) fashion. In case multiple bus masters try to talk simultaneously, an arbitration process decides on the order. The communication model, consisting of multiple devices an arbitration process and the channel model is depicted in Figure 4.5. Data is transmitted as a data frame which, next to the actual payload, contains control information. This typically includes a generic identifier, source- or destination information, as well as a field for an error detection or correction code. Depending on the protocol and physical layer, the data frame may be scrambled by adding stuffing bits or using additional encoding (e.g. 8b/10b). Scrambling techniques are used to maintain DC-balance and provide sufficient change in the signal to allow clock recovery. As motivated in Chapter 3, a frame is modelled as a task τ with an execution time according to its transmission time. Throughout this chapter, we reference a frame belonging to stream i by the associated CPA task τ_i.

Definition 26 (Frame)**.**
A frame is the smallest entity of data, associated with a unique stream i, payload and has an associated worst-case / best-case transmission time of C_i^+ / C_i^- observed on the physical layer.

Let us focus on the dynamic behavior for a fixed-priority arbitration scheme, which is commonly used in real-time communication (e.g. CAN, AMBA, Ethernet 802.11q). The right part of Figure 4.5 shows the timing behavior of frame-based communication in two scenarios. The transmission of three frames transmitted by device a, b, and c in the error-free case (top) and the error case (lower part) is shown. Also the response time R of one exemplary frame is indicated. Obviously, as the frame transmitted by device b is corrupted (lower part) it must be recovered. In the depicted case

and throughout the chapter, we assume a retransmission scheme is used (retransmission is shown as a hatched grey frame) and error signaling is neglected. This increases the response time R_e by the retransmission as well as by an additional higher priority frame.

Naturally, the response time under errors depends on the number of errors which have occurred. Unfortunately, the number of error events cannot be bounded conservatively due to its stochastic nature. It is extremely unlikely, yet possible, that a very large number of errors affect the communication which leads to an arbitrary large response time. Hence, a worst-case bound under errors is not feasible and the response time must be grasped as a stochastic process.

To quantify the effect of errors, we need to quantify the error associated overhead. In our communication model, we account the error overhead by considering a signaling overhead as well the actual recovery.

Definition 27 (Error Signaling Overhead)**.**
The error signaling overhead O^s is an upper bound on the time required to cancel the current transmission and notify all terminals. This includes the time to bring all devices into a consistent state.

Depending on the protocol, the signaling overhead can be composed of the transmission of an error frame or the deliberate transmission of invalid frame.

Definition 28 (Error Recovery Overhead)**.**
The error recovery overhead O_i^r for a frame of stream i is an upper bound on the time to recover the error after all devices are in a consistent state.

For retransmission schemes, the recovery overhead is determined by the frame transmission C_i, whereas in systems which use forward error correction, O_i^r it is determined by the FEC runtime.

The goal of the following sections is two-fold. First we model the worst-case impact of errors on the response-time and second, we predict the likelihood of observing an increased response-time caused by errors. With reference to the example given in Figure 4.5, we want to obtain worst-case bounds for R and R_e and know the probability that we observe those response times. The illustrative example shows only one error, however an arbitrary number of error scenarios can occur and must be considered in a formal context.

4.3.1 Related Work

A large number of related literature has discussed the effects of errors (e.g. error signaling and retransmission overhead). One of the first considerations of errors on the response time was presented in [216]. The busy

period approach was extended to include error overhead which is modeled as higher priority load with a known minimum inter-arrival time T_F. A reliability value was derived in [49] by calculating the probability that the distance between two error events is never smaller than T_F over a given mission time.

In [44] the approach was picked up and a tree-based method is presented, where different error scenarios are evaluated iteratively. In a second step, these scenarios are translated to probabilities and a *worst-case deadline failure probability* is calculated. The approach was extended in [45], and the tree-based was superseded by a simpler, more accurate approach. However, both methods [45, 44] allow only deadlines smaller than the periods, which is a limit for practical use since bursty CAN traffic is not supported. Such bursts can occur at a Gateways which connect highspeed and lowspeed communication media such as Ethernet and CAN. Hence, it is especially important to support queued messages.

Later work [190, 110] also accounted for burst errors where a generalized Poisson process is used. The authors are then able to compute the worst-case deadline failure probability. However, the complexity of their burst-model lead to computational problems such as numerical accuracy of the presented approach.

In [248], an approach is presented to bound the reliability of periodic, synchronized messages tightly. Therefore, a reliability metric $\mathcal{R}(t)$ is defined which denotes the probability that after time t the packets are transmitted without any deadline misses. Reliability is calculated based on the hyperperiod, which is the time when the event pattern of a periodic message set repeats itself. Hence, the complexity of the algorithm depends on the number of events in the hyperperiod. This algorithm is suitable for automotive message sets in which periods are typically multiples of 10ms and deadlines are given implicitly (i.e. period as deadline). However, if messages are not synchronized, or the relative phasing is unknown or can change, the approach is not applicable.

The problem of obtaining the response time under the probabilistic effect of errors is very similar to effects known from variable execution time research where execution times are modeled by probability mass functions (pmf). In [67] an algorithm was presented to derive stochastic response times for individual events of a given trace (e.g. periodic). Based on this methodology, an average response-time distribution can be derived [66]. As we will show in the following sections, we can adopt the underlying idea.

4.3.2 Busy-Period Fixed-Priority Arbitration

The following analysis is based on [45, 61], generalized to queued frames and arbitrary event models as introduced in the CPA chapter. The goal is

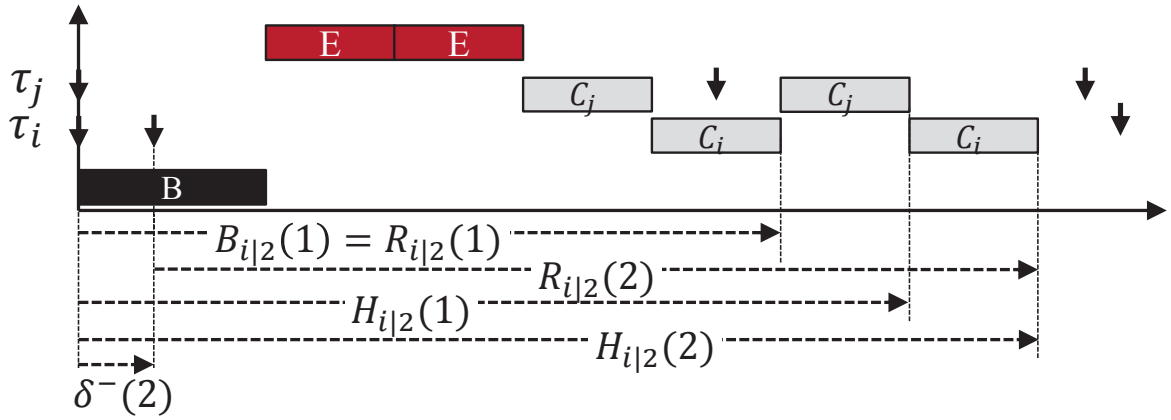


Figure 4.6: Illustrative example for the response time computation under $k = 2$ errors for a fixed priority non-preemptive scheduler.

to derive the response time under the assumption that the transmission is affected by k error events.

Definition 29 (K-Error Response Time)**.**
The K-Error response time $R^+_{K|i}$ is an upper bound on the response time, assuming k errors arrived in the scheduling horizon of the affected frame.

The illustrative example from Figure 4.6 shows the response time for $k = 0$ as well as $k = 1$. Now we are presenting a method to obtain the response time under error conditions for an arbitrary value of k systematically. First, we introduce the necessary definitions which apply for the error-case.

As discussed before, the error penalty from which frame τ_i suffers in case of one error event is composed of the protocol overhead of error signaling O^s plus the actual retransmission. Here, we make a worse-case approximation by assuming the largest frame of equal or higher priority is affected.

Theorem 16. *The worst-case overhead for k errors under SPNP scheduling is upper bounded by*

$$E_{i|k} = k \cdot (O^s + O^r_i) \tag{4.32}$$

- *k is the number of errors*
- *O^s is the signaling and detection overhead*
- *O^r_i is the recovery overhead, bounded by the largest execution time of all tasks of higher or equal priority ($hep(i)$) than task τ_i.*

$$O^r_i = \max_{\forall j \in hep(\tau_i)} C_j \tag{4.33}$$

Proof. As stated in [45], under worst-case conditions each error hits a different packet. In the worst-case the largest packet is hit k times. □

Now we can extend the response-time analysis for non-preemptive, fixed priority scheduling as given in Chapter 3 and incorporate the error overhead. As for the error-free case, which was presented in Chapter 3, we must consider all higher priority frames, a lower priority blocker which arrived shortly before as well as the additional cost caused by the k error overhead.

The presented response-time equations can easily be adapted by assuming that the overhead for k errors is added to all the relevant terms. An illustrative example is shown as a Gantt chart in Figure 4.6. The scheduling horizon, processing time and response time are indicated. The worst-case abstraction assumes that *all* k errors occur prior to all frame transmissions. Obviously, this model is not a consistent schedule which could be observed in reality: Retransmissions appear before the actual frames are transmitted. Now we establish the formalism which leads to the results shown in the example.

Definition 30 (k-Error Scheduling Horizon)**.**
The k-error, multiple event scheduling horizon $H_{i|k}(q)$ of any sequence of q events of frame τ_i under SPNP scheduling is the right half-open scheduling horizon under the assumption k error events occur in this interval.

Theorem 17. *Given k errors occur during the transmission of frame τ_i with a corresponding error overhead of $E_{i|k}$, the k-error, q-event scheduling horizon $H_{i|k}(q)$ is upper bounded by*

$$H_{i|k}(q) = q \cdot C_i + E_{i|k} + B_i + \sum_{j \in hep(\tau_i)} C_j \cdot \eta^+(H_{i|k}(q)) \tag{4.34}$$

with

- *k is the number of errors*
- *C_i is the worst-case execution time of task τ_i*
- *$E_{i|k}$ is the error overhead for k errors*
- *B_i is the execution time of the lower priority blocker*
- *$hep(\tau_i)$ is the set of higher or equal priority tasks than τ_i*
- *$\eta^+(\Delta t)$ gives an upper bound on the number of events in any time interval Δt.*

Proof. The proof is analogous to the error-free case. In addition to the workload from q activations of task τ_i, lower priority blocking B_i and the higher priority workload released during the time interval, an additional error overhead for k errors must be added $E_{i|k}$. □

Definition 31 (k-Error Multiple Event Queuing Time)**.**
The maximum k-error multiple event queuing time $Q_{i|k}(q)$ of any sequence of q events of frame τ_i is the right half-open multiple event queuing time under the assumption k error events occur in this interval.

Theorem 18. *Given k errors occur during the transmission of frame τ_i with a corresponding error overhead of $E_{i|k}$, the k-error, q-event scheduling horizon $H_{i|k}(q)$ is upper bounded by*

$$Q_{i|k}(q) = (q-1) \cdot C_i + E_{i|K} + B_i + \sum_{j \in hep(\tau_i)} C_j \cdot \eta^+(Q_{i|k}(q) + t_{cycle}) \quad (4.35)$$

with

- *k is the number of errors*
- *C_i is the worst-case execution time of task τ_i*
- *$E_{i|k}$ is the error overhead for k errors*
- *B_i is the execution time of the lower priority blocker*
- *$hep(\tau_i)$ is the set of higher or equal priority tasks than τ_i*
- *$\eta^+(\Delta t)$ gives an upper bound on the number of events in any time interval Δt.*
- *t_{cycle} is the cycle time (bittime) of the bus (processor) which accounts for boundary conditions.*

Proof. The q-th event gets ϵ service, when all workload consisting of the lower priority blocker, error overhead caused by k errors, all previously released $q-1$ events and the higher priority events have been processed, (cf. Theorem 17). □

The cycle time t_{cycle} accounts for an event which arrives shortly after the start of a bittime and has to wait for the next bit slot. The queuing time is then used to compute the processing time.

Definition 32 (k-Error Multiple Event Processing Time)**.**
The maximum k-error multiple event processing time $B^+_{i|k}(q)$ of any sequence of q events of frame τ_i is the right half-open multiple event queuing time under the assumption that k error events occur prior to the start of the transmission of th q-th frame.

Theorem 19. *The maximum k-error multiple event processing time is upper bounded by*

$$B^+_{i|k}(q) = Q_{i|k}(q) + C_i \tag{4.36}$$

Proof. The proof follows the one of Theorem 6. Additional error overhead of $E_{i|k}$ is considered in the queuing delay term. □

From this we can compute the k-error response time using standard CPA methodology.

Theorem 20. *The k-error response time is upper bounded by*

$$R^+_{i|k} = \max_{\forall 0<q\leq q^+} B^+_{i|k}(q) - \delta^-_i(q) \tag{4.37}$$

Proof. The proof is analogous to the one of Theorem 1. For each k-error scenario, it is necessary to evaluate which event leads to the worst-case response time. □

4.3.3 Busy-Period First-In First-Out Arbitration

In First-In-First-Out arbitration, all events are fed into a single queue and are processed in the order of their arrival. The error recovery protocol in FIFO bus arbitration is not standardized and is implementation dependent. We assume that the sending terminal is immediate (or sufficiently early) notified in case of transmission errors. This is typically the case in broadcast-like bus architectures as well as point to point links (e.g. NoC). Often standardized protocols do not dictate the scheduling and error handling explicitly and leave this up the implementation. The AXI [174] embedded bus standard for instance provides user-defined signals (`TUSER`) which can be used to implement parity checking and provide recovery information.

When the sending terminal recognizes a latent error situation, we consider only a *Greedy retransmission* scheme. A greedy retransmission scheme retries until the frame is received by all terminals. Such a scheme is for instance used for input queued switches where arbitration happens at the ingress port and re-arbitration is not easily possible.

Naturally, there are other possible recovery resolution strategies. For instance a packet can be dropped when a recovery is deemed impossible, for example after either a number of unsuccessful trials or a timeout interrupt. Such protocols would then discard the data and eventually inform higher level software or assume that a higher level protocol takes care. In this section, we only consider protocols which are guaranteed to make progress

(data is never dropped) given unlimited time is available. Later in Chapter 5 we lift this assumption and consider higher level protocols which cope with dropped data.

The general approach is similar to the previously presented fixed priority analysis. In fact, we give a set of equations which bounds the previously introduced greedy scheme. Under worst-case assumptions, we can assume that the task with the largest worst-case execution time is affected by an error and the entire frame is retransmitted over and over k times. Thus the worst-case overhead can be computed similarly to the fixed priority case as presented in Lemma 16, with the difference that all frames mapped to the same queue have to be taken into consideration.

Corollary 2. *The worst-case overhead for k errors under FIFO scheduling is upper bounded by*

$$E_k = k \cdot (O^s + O^r) \tag{4.38}$$

- *k is the number of errors*
- *O^s is the signaling and detection overhead*
- *O^r is the recovery overhead, bounded by the largest worst-case execution time of all tasks that share the queue with task τ_i.*

$$O^r = \max_{\forall j \in \mathit{fifo}(\tau_i)} C_j^+ \tag{4.39}$$

Theorem 21. *Given k errors occur during the transmission of frame τ_i under FIFO arbitration, the k-error, q-event scheduling horizon $H_{i|k}(q)$ is upper bounded by*

$$H_{i|k}(q) = q \cdot C_i + E_k + \sum_{j \in \mathit{fifo}(\tau_i)} C_j \cdot \eta_j^+ \left(H_{i|k}(q)\right) \tag{4.40}$$

- *k is the number of errors*
- *E_k is the signaling and detection overhead*
- *$\mathit{fifo}(\tau_i)$ is the set of all tasks mapped to the same fifo as τ_i.*
- *$\eta^+(\Delta t)$ is an upper bound on the number of events in the time interval Δt.*

Proof. The proof is analogous to the one of Lemma 3.32. Additionally to the workload of all released frames, we need to add the total worst-case workload induced by k errors which is bounded by E_k. As the additional workload for k errors for both schemes (greedy and requeue) is the same, the horizon also is. □

Corollary 3. *The response-time of the q-th event of task τ_i under k errors, assuming it arrives in the scheduling horizon of the $(q-1)$-th event is upper bounded by*

$$R^+_{i|k}(q,a) = Q_{i|k}(a) - a + C^+_i \tag{4.41}$$

- *k is the number of errors*
- *a is the arrival time of the q-th event of task τ_i.*
- *C^+_i is the worst-case execution time of task τ_i.*
- *$Q_{i|k}(a)$ is the queueing delay for the q-th event, assuming it arrives at time a and k errors arrived prior to a.*

Proof. Proof follows the one of eq. 3.33 assuming an increased queuing delay, caused by workload of k errors. □

4.3.4 Probability Computation

The remaining question is to combine the response-time under errors with the error model and compute the likelihood that a given threshold is exceeded. The exceedance function as a metric has been used in related work in the field of response times under errors such as in [44, 45] and is also adapted as the key metric in this thesis.

Definition 33 (Worst-case Response-Time Exceedance Function)**.** *The worst-case response-time exceedance function is an upper bound on the probability $P[R_{i,j} > t]$, that the response-time of some job of task τ_i exceeds a given threshold t.*

$$X^+_i(t) \geq \max_{\forall j} P[R_{i,j} > t] \tag{4.42}$$

Hence, for any observed frame, the probability of exceeding the response-time threshold is smaller than $X^+_i(t)$. The deadline failure probability which is for instance used in [190] and [110], is a special case for $X^+_i(D_i)$, where D_i is the deadline associated with task τ_i. Also the more sophisticated reliability metric $\mathcal{R}(t)$ can be derived from the exceedance function. The reliability function $\mathcal{R}$ gives the probability that the function is still operational (no missed deadlines) after time t.

Theorem 22. *The reliability $\mathcal{R}(t)$ of a task τ_i is lower bounded by*

$$\mathcal{R}(t) = \left(1 - X^+_i(D_i)\right)^{\eta^+_i(t)} \tag{4.43}$$

- *t is the uptime*
- *D_i is the relative deadline associated with task τ_i*

- $\eta_i^+(\Delta t)$ *is an upper bound on the number of events in the time interval* Δt.

Proof. By construction the exact reliability function is computed as the product of the probabilities that a deadline is met under the condition the previous deadlines were met:

$$\mathcal{R}_{\text{exact}}(t) = P[\tau_{i,0}] \cdot P[\tau_{i,1}|\tau_{i,0}] \cdot \cdots P[\tau_{i,n}|\tau_{i,n-1} \wedge \tau_{i,n-2} \wedge \cdots \wedge \tau_{i,0}] \quad (4.44)$$

here n is the last event which arrives prior the time t. By definition the exceedance function is an upper bound on any event j, regardless of the context. Hence, the converse probability is a lower bound on any event and we can conclude that

$$P[\tau_{i,n}|\tau_{i,n-1} \wedge \tau_{i,n-2} \wedge \cdots \wedge \tau_{i,0}] \geq \left(1 - X_i^+(D_i)\right)^n \quad (4.45)$$

It follows that

$$\mathcal{R}_{\text{exact}}(t) \leq \left(1 - X_i^+(D_i)\right)^n \quad (4.46)$$

Also by definition, $\eta_i^+(t)$ is an upper bound for n. Since η is an upper bound and $1 - X_i^+(t)$ is a lower bound, the power is a lower bound. □

Note that eq. 4.45 also holds for burst errors but is very pessimistic in such a case. The reason is that $X_i^+(D_i)$ is a conservative probability that some event will miss its deadline. This value is derived by assuming that the burst will affect the frames or computation in the worst-possible way. The extrapolation made in eq. 4.45 assumes that the worst possible burst arrival is repeated frame after frame (even if this is very unlikely according to the used error model).

The remaining question is how to obtain the exceedance function under the presented error models. Here we follow the approach presented in [45, 21] and generalize it to arbitrary schedulers for which the k-response time as well as the k-error scheduling horizon is available.

Single Bit Error Model

A naive approach would be to apply the probability equations directly as presented in Section 4.2. For a single bit error model, we can use the Poisson process (eq. 4.15) and feed the scheduling horizon directly into the Poisson equation:

$$P[k, \Delta t] = \frac{(\Delta t \cdot \lambda)^k}{k!} e^{-\Delta t \cdot \lambda} \quad (4.47)$$

Unfortunately, this is not directly possible as it was shown in [44, 45]. The consequence of an error event on the response time depends on the exact

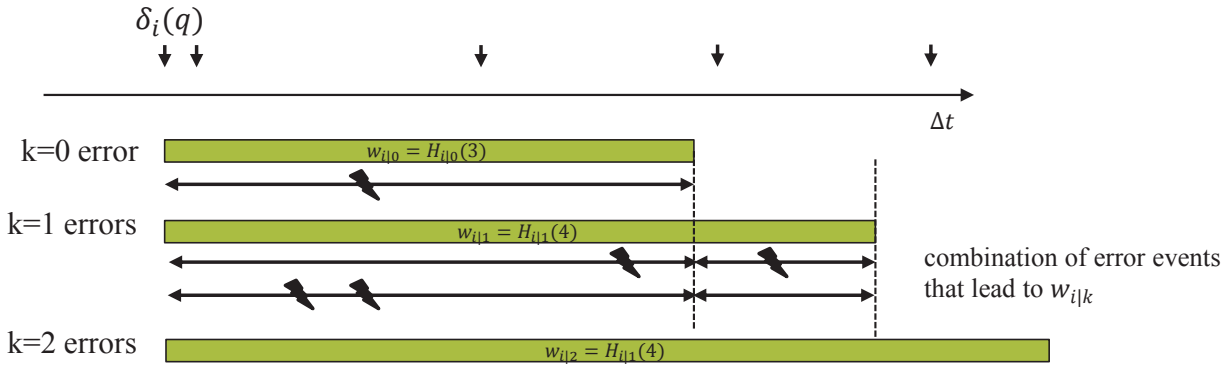


Figure 4.7: Scheduling Horizons $H_{i|k}(q)$ and busy-period $w_{i|k}$ for values for $k = 0, 1, 2$ and the associated busy period shown in green. Black arrows in-between show one potential error arrival scenario.

arrival time in the scheduling horizon. In the formal scheduling analysis which was presented earlier, we assumed error events arrive as early as possible. However in reality they can occur at any time. The problem is that if errors are spaced too far apart, they will have no effect. Similar to regular events, they must fall into the scheduling horizon of the previous error event in order to have a timing impact. This is shown in Figure 4.7.

In order to observe a response time $R^+_{i|k=1}$, the error must occur in the associated scheduling horizon $H_{i|k}(q^+)$. It is a necessary condition but not sufficient. This is shown in Figure 4.7. Errors must arrive in a specific pattern. In order to observe a busy-period of length $w_{i|k=1}$, an error event must arrive sometime prior to the end of the zero-error busy period $w_{i|k=0}$. Otherwise, the error will not affect the timing of events which arrived previously. In order to observe a two-error busy period $w_{i|k=2}$, there are two cases (also indicated in Figure 4.7):

1. Two error events arrive in zero-error window.
2. One error event arrives in the zero-error windows, which leads to a one-error window and a second error appears in the interval $[w_{i|k=0}, w_{i|k=1})$.

The number of combinations that lead to a three-error busy period is even larger. Broster et al. [45] have shown that the number of combinations forms a Catalan Series and thus an explicit enumeration is not feasible.

Theorem 23. *An upper bound for the probability $P[\leq w_{i|k}]$ to observe a $w_{i|k}$ busy period or smaller and thus a $R^+(i|k)$ response-time is given by*

$$P[\leq w_{i|k}] = P[k, w_{i|k}] - \sum_{l=0}^{k-1} P[\leq w_{i|l}] \cdot P[k-l, w_{i|k} - w_{i|l}] \tag{4.48}$$

Proof. Given in [45]. The argumentation is that k errors in the time window $w_{i|k}$ can occur in several ways. Either the errors fall in a "lucky" pattern and we actually observe a busy period $w_{i|k}$ with probability $P[\leq w_{i|k}]$ or we observe a smaller busy period of some length $w_{i|l}$ with probability $P[\leq w_{i|l}]$ and the remaining $k - l$ error events fall in the excess window of length $w_{i|k} - w_{i|l}$. □

In the worst-case it follows that, if we have seen a busy-period of length $w_{i|k}$, it is conservative to assume that the events in this busy period are delayed and a response-time $R^+(i|k)$ is observed.

$$X_i^+(t) \leq 1 - \sum_{\forall k | R_{i|k} < t} P[w_{i|k}] \tag{4.49}$$

Burst Error Model

In this section, we extend the previously presented single bit probability analysis to burst errors assuming a Gilbert loss model as introduced in Section 4.2.3. In order to keep the problem traceable, we make some conservative approximations. Typically an error-burst most likely hits bits which belong to one frame. Analogous to the single-bit error case, we assume that each error event hits a dedicated data frame. This obviously is a major problem, if the average burst length on a bit-level is large. If the burst length is in the order of a frame size, then at most two frames are affected.

This overestimation can be reduced if, instead of a bit-error process as described in Section 4.1 a packet-error process is used. A method to obtain a packet error process is described in [109]. But packet error statistics can also be obtained from channel simulation [143]. Special care must be taken, since it has been shown [146] that the packet error process for bursty channels cannot be modeled by a time homogeneous Markov process. Alternatively, gap error models can be used [90], here the consecutive number of error-free bits surrounded by error events is modeled.

We adapt the key approach from eq. 4.48 and apply eq. 4.24 and eq. 4.25. First, we can make the observation that the last bit in any busy-period must be always correct.

Lemma 1. *Under a Gilbert loss error model, a k-error busy-period of $w_{i|k}$ ends in the GOOD state.*

Proof. The proof is by contradiction. Assuming a busy-period of length $w_{i|k}$ which ends in the bad state. This implies that the last bit must have been corrupted. If the last bit was corrupted, the entire frame must be recovered and leads to a retransmission which further increases the busy-period. This violates the initial assumption that the busy-period is of length $w_{i|k}$. □

Let us use the following notation to reference the probability of observing k errors in a window w ending in state S and starting with a probability of g_0 in the GOOD state and $b_0 = 1 - g_0$ in the BAD state, respectively.

$$P[k, w, S, g_0] \tag{4.50}$$

The actual value can be computed by feeding the parameters in eq. 4.24 and eq. 4.25.

Theorem 24. *The probability $P[\leq w_{i|k}]$ to observe a busy-period of length $w_{i|k}$ or smaller under the Gilbert loss error model is upper bounded by*

$$P[\leq w_{i|k}] = P[k, w_{i|k}, G, g_{ss}] - \sum_{l=0}^{k-1} P[\leq w_{i|l}] \cdot P[k-l, w_{i|k} - w_{i|l}, G, 1] \tag{4.51}$$

- *k is the number of errors*
- *$w_{i|k}$ is an upper bound of the busy-period assuming k error events*
- *$P[\leq w_{i|k}]$ is the probability to observe a busy window of $w_{i|k}$ or smaller*
- *$P[k, w, S, g_0]$ is the probability to observe k errors in a time interval w ending in state S.*
- *g_{ss} is the steady state probability to observe the Gilbert model in the GOOD state.*

Proof. The proof is by construction. It is reasonable to assume that the model is in the steady state at the beginning of a busy-period. k errors in a time window $w_{i|k}$ can be seen in two ways:

1. In a busy-period that ends in the BAD state.
2. In a busy-period that ends in the GOOD state.

According to Lemma 1, the probability of the first case is zero. Thus only the latter case must be considered. These k errors in a time window $w_{i|k}$ under the assumption that we start in the steady state and the last bit is healthy (GOOD) can lead to the following mutually exclusive scenario:

1. a busy-period of length $w_{i|k}$,
2. a busy-period of length $w_{i|k-1}$ assuming one error falls in the window $[w_{i|k} - w_{i|k-1})$, starting in the GOOD state,
3. a busy-period of length $w_{i|k-2}$ assuming the two errors falls in the window $[w_{i|k} - w_{i|k-2})$, starting in the GOOD state,
   ...

4. a busy-period of length $w_{i|0}$ assuming the k errors falls in the window $[w_{i|k} - w_{i|0})$, starting in the GOOD state.

It is not possible to observe a busy-period of length $w_{i|k-j}$ assuming the j errors falls in the window $[w_{i|k} - w_{i|k-j})$, starting in the BAD state because each busy period must end in the GOOD state (Lemma 1). We can put the previous construction into the following form:

$$P[k, w_{i|k}, G, g_{ss}] = P[\leq w_{i|k}] + \sum_{l=0}^{k-1} P[\leq w_{i|l}] \cdot P[k-l, w_{i|k} - w_{i|l}, G, 1] \quad (4.52)$$

Simple rearrangement leads to eq. 4.51. □

Once the busy-period probabilities are derived, the exceedance function is computed according to eq. 4.49

4.4 Convolution Analysis for Fixed-Priority Arbitration

By construction, the previously presented analysis induces additional pessimism because recovery overhead is conservatively approximated to be the worst case recovery overhead among all higher priority tasks. In this section we present an alternative approach which reduces this pessimism and trades accuracy with analysis runtime.

4.4.1 Related Work

The problem of obtaining the response time under the probabilistic effect of errors is very similar to effects known from variable execution time research. There, execution times are modeled by probability mass functions (pmf). In [67] an algorithm was presented to derive stochastic response times for individual events of a given trace (e.g. periodic). Based on this methodology, an *average* response-time distribution can be derived [66]. As we will show in the following sections, we can adopt the underlying idea to cover the behavior under errors. However, the drawback with this approach is that it relies on the memorylessness of the underlying process, thus it only works for the single bit error model and not for the Gilbert loss model. Note that this approach significantly differs from the work in [66] where the average-case distributions of all events is derived. However, we apply a similar methodology.

The following stochastic error analysis consists of two steps. First, the task and error model are transformed into an equivalent variable execution time problem by using a probability mass function (pmf) instead

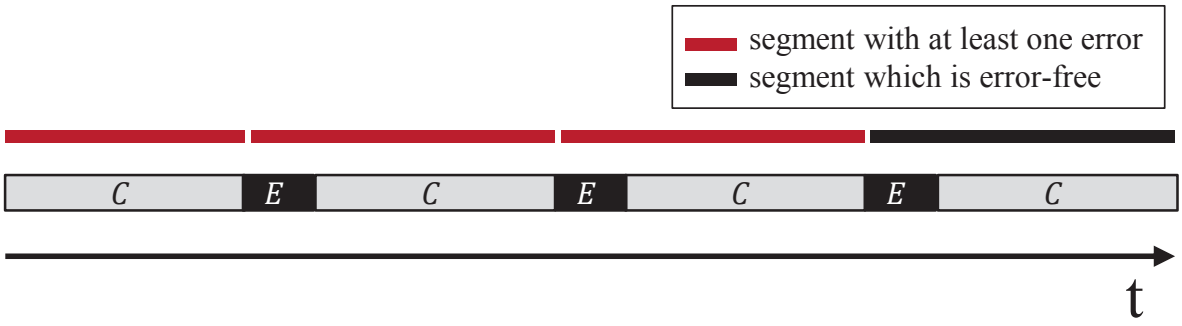


Figure 4.8: An example scenario that leads to $k = 3$ error recoveries. The initial transmission of length C must be hit by at least one error (red bar), the following two recovery attempts consisting of error signaling and retransmission ($C + E$) must be hit by at least one error and the final recovery must remain intact.

of a single worst case execution time. In a second step, similar to [67], we iteratively apply *splitting*, *convolution* and *merging* of the pmfs to calculate conservative stochastic response-time bounds.

From Errors to Variable Execution Times

Let us consider a single instance of a τ_i in isolation. We model the variable execution time overhead caused by error events by using execution time probability mass functions (pmfs). This approach assesses the overhead in a fine grained fashion, rather than using the worst-case error overhead among all higher priority tasks as done previously. This is possible because the Poisson error model is memoryless, hence it allows us to examine the execution time behavior of tasks in isolation. For the following steps it is necessary to distinguish between error-events (i.e. the bit flip) and the manifestation (i.e. recovery operation consisting of error signaling and re-transmission). According to eq. 4.15, and a constant bit error rate λ the probability to see no errors in a time-interval Δt is given by

$$P^{ok}[\Delta t] = e^{-\lambda \Delta t} \tag{4.53}$$

and the probability to see at least one error in the window is determined by the converse probability:

$$P^{err}[\Delta t] = 1 - e^{-\lambda \Delta t} \tag{4.54}$$

Theorem 25. *For an instance of task τ_i, a lower bound on the probability $P_r[k]$ for exactly $k \geq 0$ recovery operations is given by:*

$$P_r[k] = \begin{cases} P^{ok}[C] & \text{if } k = 0 \\ P^{err}[C] \cdot \left(P^{err}[C+E]\right)^{k-1} \cdot P^{ok}[C+E] & \text{if } k > 0 \end{cases} \tag{4.55}$$

- *k is the number of errors.*
- *C is the worst-case execution time.*
- *E is the total worst-case overhead for one error (i.e. $E_{i|k=1}$ cf. Theorem 16).*
- *$P^{ok}[\Delta t]$ / $P^{err}[\Delta t]$ is the probability to observe no error / at least one error in the time Δt.*

Proof. The case for $k = 0$ is trivial and directly obtained by applying eq. 4.53. In order to see k recoveries, $k-1$ previous attempts must have failed. The first initial transmission does not include additional error signaling overhead, whereas the other $k-1$ recoveries include additional error signaling of length E. The last final recovery and retransmission must be intact, otherwise another recovery will take place. □

An illustrative example for $k = 3$ is shown in Figure 4.8. We can obtain a lower bound on the probability that we observe less or equal than k recoveries by summation

$$P_r[\leq k] = \sum_{i=0}^{k} P_r[i] \tag{4.56}$$

In that sense, a lower bound on $P_r[k]$ is conservative because this implies that the likelihood $(1 - P_r[k])$ for more than k recoveries is higher, which cause even more recovery overhead.

Now we need to bound the time which is spent by a job of task τ_i for handling k recoveries. By using the worst-case execution time C and the worst-case error signaling overhead E, we can bound the execution time for error situations with exactly k recoveries by $C + k(C + E)$. This is also shown in Figure 4.8. By combining the overhead for k recoveries with the probability for the occurrence, we retrieve an execution-time probability mass function (pmf).

Definition 34 (Worst-Case Execution Time pmf)**.**
The worst-case execution time pmf $e(t)$ is the probability that the overall execution time of an instance of a task including errors and recoveries is of length t.

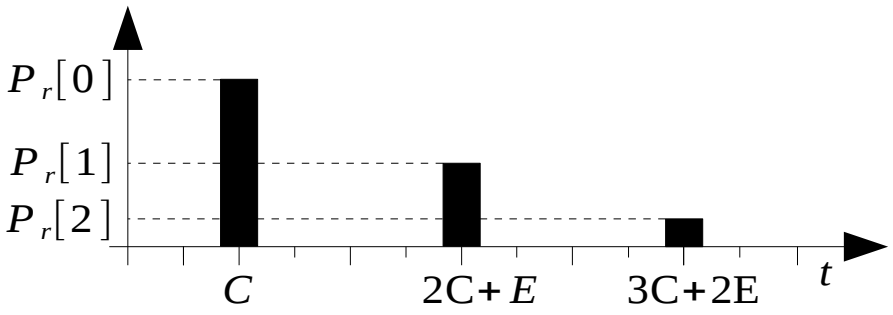


Figure 4.9: Worst-Case execution time pmf $e(t)$.

Theorem 26. *A conservative upper bound for the worst-case execution time pmf $e(t)$ including error-signaling and recovery operations is given by:*

$$e(t) = \begin{cases} P_r[k] & \textit{if } t = C + k(C+E), k \geq 0 \\ 0 & \textit{otherwise} \end{cases} \tag{4.57}$$

Proof. The proof is straightforward. An upper bound for the execution time with exactly k recoveries is given by $C + k(C + E)$ (cf. Figure 4.8). The probability that a k-recovery situation will occur is bounded by $P_r[k]$ (Equation 4.55). □

Based on the lower-bound notion observed for $P_r[k]$, we can conclude that the cumulative distribution function $\sum_{x=0}^{t} e(x)$ is a lower bound that the observed execution time (including error handling and recovery) of a given job is less or equal than t. An example for the worst-case execution time pmf is shown in Figure 4.9. Here, the probability for the task executing for no longer than time C is given by $e(C)$ which corresponds to the error-free case, i.e. the case where no recoveries are observed.

The previous consideration assumes that all k retransmissions occur one after another and are not interleaved with instances of other tasks. We can lift this implicit assumption and show that under a memoryless Poisson single bit error model, an arbitrary interleaving is allowed.

Theorem 27. *The probability that no error event occurs during the execution of τ_i is independent of the number of preemptions and interleaving with other tasks.*

Proof. Assuming the task τ_i was preempted $n \in \mathbb{N}$ times and sliced in arbitrary intervals of length Δt_j such that

$$\sum_{j=1}^{n} \Delta t_j = \tilde{C}_i \tag{4.58}$$

then the probability that no error event occurs while τ_i is executing is calculated through Equation: 4.15

$$\prod_{j=0}^{n} e^{-\lambda \Delta t_j} = e^{-\lambda \sum \Delta t_j} = e^{-\lambda \tilde{C}_i} \tag{4.59}$$

which is independent of n. □

4.4.2 Stochastic Busy Window

As discussed in Chapter 3, in the non-stochastic case it was shown that the *critical instant* assumption - all events arrive as early as possible - leads to the worst-case busy-period for fixed-priority scheduling [172]. The worst-case response time can then be found among all events in the worst-case busy-period.

Since the busy-period fixed-point equation as presented in previous sections does not support the concept of probabilistic execution times, we need to transform the equation in a stochastic domain. As for the non-stochastic busy-period approach, the critical instant assumption remains valid also for the case where the execution time is a random variable. The workload is maximized if all events arrive as early as possible, independent of their execution time. The remaining question is, what is the likelihood that a worst-case busy period of length t will occur under errors.

Definition 35 (Level-i Busy-Period pmf)**.**
The level-i busy-period pmf $\Omega_i(t)$ is the probability that the critical instant initiates a level-i busy-period of length t. That is, time t after the critical instant all messages of higher and equal priority than task τ_i released prior to time t have been transmitted successfully.

In the following steps we explain, how to determine the worst-case level-i busy-period pmf.

Obviously, the function approaches zero for $t \to \infty$. For practical applications however, it is sufficient to know $\Omega_i(t)$ only for a limited co-domain. Thus, the analysis can be stopped if a sufficient accuracy is reached (i.e the probability that the busy period exceeds t falls below a predefined threshold). In the following steps, we use convolution and shift operations through which it is possible to systematically evaluate candidates.

The idea can be sketched informally as follows: We iterate over the events released after the start of the critical instant in the order of their earliest release, pick up their execution time and “add” them up. After a sufficient number of events has been considered, we know the stochastic busy window. A sufficient amount has been considered, if the probability of new events falling in the busy window is sufficiently small.

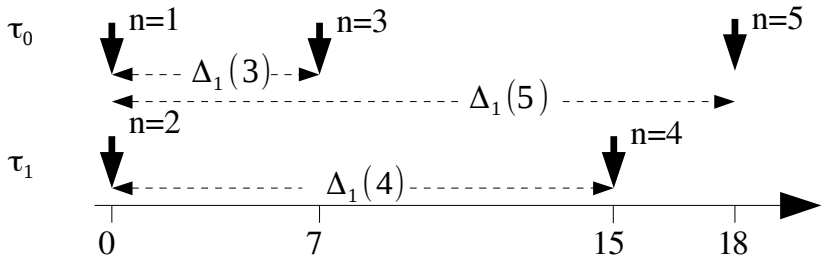


Figure 4.10: Critical-instant scenario of task τ_0 and τ_1. All events are numbered according to their release time (i.e. $n = 5$ is the 5-the event released at time $\Delta_1(5)$.)

Starting with the critical instant, we can define a strict total ordering on all events. That is, $\Delta_i(n)$ denotes the earliest release time of the n-th job among all jobs of higher and equal priority than τ_i. In case release times are equal, events are ordered according to their priority and their event ordering (events of the same task cannot pass each other). Figure 4.10 shows an example. τ_0 has a high priority and a periodic with jitter event model $P = 11$, $J = 4$. Task τ_1 has a low priority and a periodic event model $P = 15$. Arrows denote the earliest release times of the associated event, n gives the index with respect to to its release time and priority.

$$\Delta_i(n) = \inf_{\Delta t \geq 0} \{\Delta t \mid \sum_{\forall j \in hp(i)} \eta_j^+(\Delta t) \geq n\} \tag{4.60}$$

Accordingly, $e_n(t)$ is the execution time pmf of the n-th event associated with $\Delta_i(n)$.

Assuming two messages are released at the same instant at time $t_0 = 0$. We are now interested in the stochastic execution-time pmf consisting of both transmissions including error overhead.

Lemma 2. *The execution-time pmf including error and recovery overhead for two messages which are released simultaneously and have execution time pmf's of e_0 and e_1 is obtained by convolution as follows:*

$$e_{sum}(t) = (e_0 * e_1)(t) \tag{4.61}$$

where

$$(e_0 * e_1)(t) = \sum_{\tau=-\infty}^{\infty} e_0(t-\tau)e_1(\tau) \tag{4.62}$$

and

- e_0, e_1 *are the execution time pmfs of event 0 and 1.*

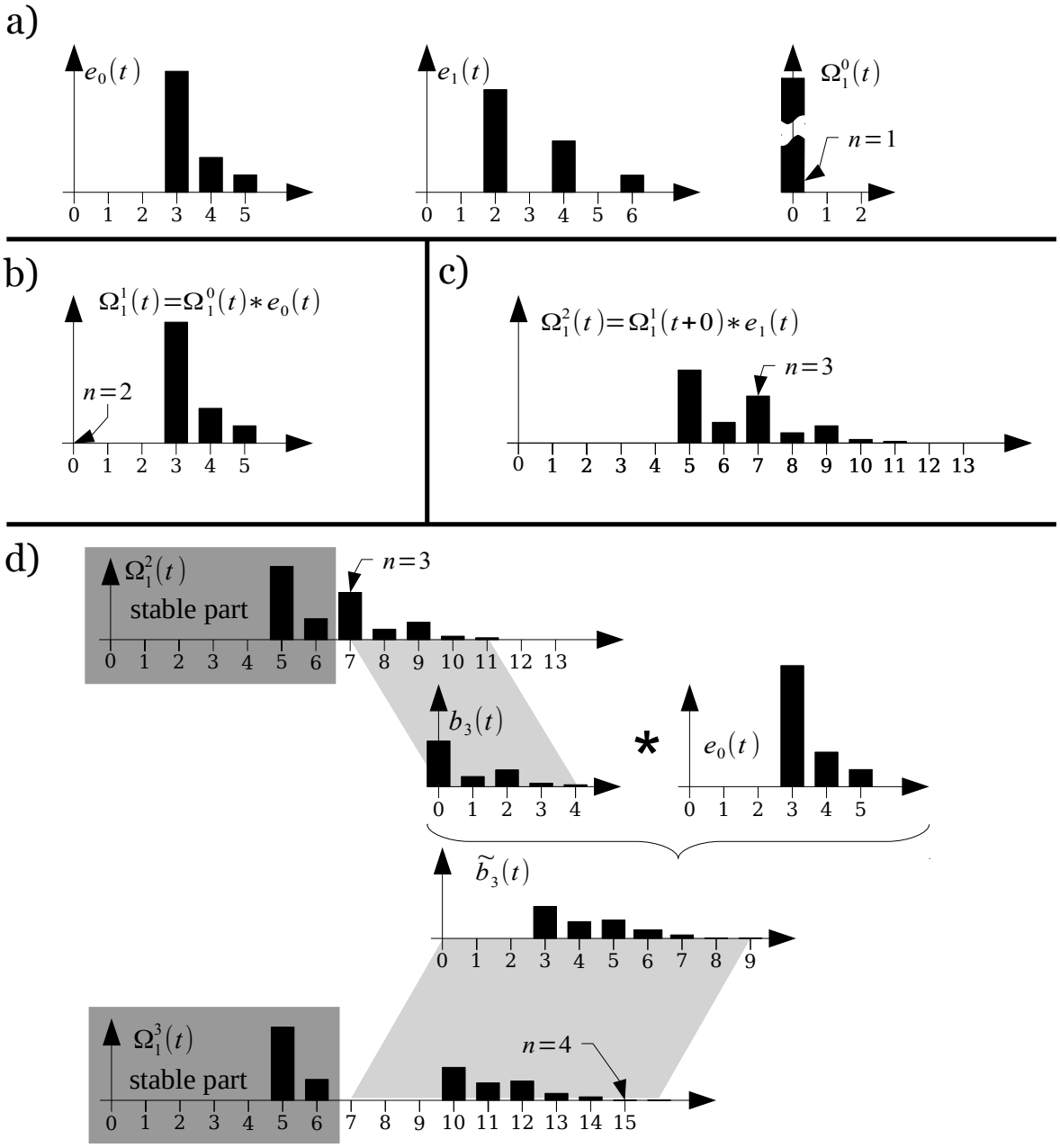


Figure 4.11: Illustrative example for constructing the busy-period pmf Ω_1.

- e_{sum} *is the execution time pmf of the combined workload.*

Proof. As already motivated, under the single bit Poisson error model, the execution time variables C_1, C_2 are independently distributed. According to [106] (Definition 7.1 for discrete variables, and Theorem 7.1 for continuous variables), the pmf of the sum of two independently distributed variables is obtained by convolution of the pmf's of the individual variables. □

Before we formalize the approach, we sketch the algorithm by using the illustrative critical instant Gantt-chart from Figure 4.10 and derive the stochastic busy period for the task τ_1. It must be noted that there is no low priority blocker, since there are only two tasks in the system. Individual steps of the process are shown in Figure 4.11.

Figure 4.11 a) shows the initial input for the algorithm, the execution time pmfs of task τ_0 and τ_1 as well as an initial busy period of length zero which is used as an optimistic starting point. The execution time pmfs $e_0(t)$ and $e_1(t)$, are computed by using eq. 4.57. The busy-period pmf $\Omega_1(t)$ is retrieved incrementally by iterating over all events of the tasks in order of their releases and "adding" their execution times.

Like for the non-stochastic analysis, we start with an initial busy period of length zero. In the stochastic domain, a busy period of length zero $\Omega_1^0(t)$ is represented by a pmf with a peak at $t = 0$ as shown in Figure 4.11 a) on the right. In the following steps, we consider all 5 events in the order of their occurence:

$n = 1$ The first event $n = 1$ arrives at $t = 0$ earliest which will increase the busy-period pmf by "adding" its execution time pmf to $\Omega_1^0(t)$. This event arrives at time $\Delta_1(1) = 0$ and is associated with task τ_0 (cf. Figure 4.10). The pmf of the sum of the busy-period plus execution time $e_0(t)$ of the new event can be calculated by using eq. 4.61.

The resulting busy-period pmf $\Omega_1^1(t)$ is shown in Figure 4.11 b).

$n = 2$ The second event arrives at $t = 0$ earliest. We can see in Figure 4.10 that event $n = 2$ is associated with task τ_1, and adds the execution-time pmf of task τ_1 to the busy-period pmf, thus we apply eq. 4.61 a second time to get $\Omega_1^2(t)$. The busy-period pmf now includes the first two events.

$n = 3$ The third event at $\Delta_1(3) = 7$ resembles the most interesting case. It depends on the previously seen workload if the event falls in the busy period:

a) If the busy period is smaller than $\Delta_1(3)$ (i.e. < 7), the third event does not contribute to the busy period at all, since it arrives after the window ended.

b) If the busy period is larger than $\Delta_1(3)$ (i.e. ≥ 7), the third event will further increase the length of the period.

The first case implies that due to causality, event 3 cannot alter the probability that a busy window is smaller than $\Delta_1(3)$. Thus, we call the interval $[0, \Delta_1(3)]$ of $\Omega_1^2(t)$ *stable* (cf. bottom left in Figure 4.11).

In the other cases where the busy period is larger than $\Delta_1(3)$, the execution time of event 3 must be added to the busy period. This increases the length of the tail of the busy period pmf. Practically, this is achieved by applying eq. 4.61 only to the tail of $\Omega_1^2(t)$ as shown in the bottom part of Figure 4.11 d). Thus, only busy periods which are larger than $\Delta_1(3)$ are increased by the execution time of $n = 3$. The bottom of Figure 4.11 d) shows how the unstable part of the busy-period is convolved with the execution time pmf of $n = 3$ and forms the new tail. After the new tail is merged with the stable part, we have computed $\Omega_1^3(t)$.

$n \geq 4$ The probability that the 4-th event at time $t = 15$ falls into a busy window is sufficiently small, so we stop the iterative process.

To formalize the process, we assume we knew $\Omega_i^n(t)$ which includes the first n events. We will show now how to derive $\Omega_i^{n+1}(t)$ which includes the first $n+1$ events. The $n+1$-th event will obviously add additional workload to $\Omega_i^n(t)$, but in what way does it alter the busy-period? Therefore, we first introduce the notion of backlog.

Definition 36 (Relative Backlog pmf)**.**
We define the relative backlog pmf $b_i^n(t)$ as the probability that the accumulated workload of events of higher or the same priority than τ_i released before event n is processed at time t.

Theorem 28. *The relative backlog pmf of event $n+1$ can be calculated using the stochastic busy-period, since it already includes all events released prior to $n+1$. The relative backlog pmf $b^{n+1}(t)$ of event $n+1$ is given by the tail of $\Omega_i^n(t)$.*

$$b_i^{n+1}(t) = \begin{cases} \Omega_i^n(t + \Delta_i(n+1)) & \text{if } t \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{4.63}$$

Proof. The proof is given in [66] for variable execution time tasks. □

Note that the relative backlog pmf does not necessarily sum up to one. This is because the probability that a new event $n+1$ falls in a previously busy period can be smaller than one (i.e. $n+1$ only falls in the busy window if no or few recoveries were observed).

This operation is called *splitting*. It divides the busy window pmf in two parts, a *stable* and an *unstable* part. Due to causality, the new event $n+1$ cannot alter the busy window pmf in the stable interval $[0, \Delta_i(n+1)]$. Vice-versa only the unstable part of the busy window pmf is altered by event $n+1$.

In the next step, the relative backlog of the busy-period pmf and the workload originating from the new event $n+1$ have to be added.

The resulting backlog which includes the old backlog plus additional execution time $e_{n+1}(t)$ is called $\tilde{b}_i(n+1)(t)$. It is calculated by applying eq. 4.61:

$$\tilde{b}_i^{n+1}(t) = \left(b_i^{n+1} * e_{n+1}\right)(t) \tag{4.64}$$

The busy-period function Ω_i^{n+1} which includes the first $n+1$ events is generated by *merging* the stable busy-period interval and the new busy-period tail:

$$\Omega_i^{n+1}(t) = \begin{cases} \Omega_i^n(t) & \text{if } t < \Delta_i(n+1) \\ \tilde{b}_i^{n+1}(t) & \text{otherwise} \end{cases} \tag{4.65}$$

The procedure for the next event $n+2$ consists of the same three steps: *split* (eq. 4.63), *convolution* (eq. 4.64) and *merge* (eq. 4.65). The algorithm terminates if the probability that a new event falls in the busy window is below a threshold. This threshold is then a bound of the probability that a larger busy window is seen and can be used as a confidence value for the analysis.

In the previous steps we have elaborated how to derive $\Omega_i^{n+1}(t)$ from $\Omega_i^n(t)$. For the starting value of the iterative process, we assume that the busy-period size is the length of the lower priority blocker as described in [61]. This allows us to start the algorithm with the following initial busy period:

$$\Omega_i^0(t) = \begin{cases} 1 & \text{if } t = B_i \\ 0 & \text{otherwise} \end{cases} \tag{4.66}$$

with

$$B_i = \max_{\forall j \in lp(i)} C_j \tag{4.67}$$

Algorithmic Formalization

Algorithm 1 gives a pseudo code implementation of the stochastic busy-window computation. As described, the individual steps are composed of splitting which separates the unstable from the stable part of the busy-window pmf, convolution, which integrates the arrived workload and the backlog, as well as merging which integrates the new backlog in the busy window pmf.

It is assumed that $\Omega_i(t)$ is implemented as a vector and the operation $[t:]$ slices the vector at the t-th index and returns the right slice. Analogous, $[:t]$ slices the vector at the t-th index and returns the left slice. Similarly, $[f, c]$ concatenate vectors f and c.

Algorithm 1 Discrete Computation of the stochastic busy period

function BUSY WINDOW(τ_i, Δ, w_i^0, set of pmfs $e(t)$)
  $n \Leftarrow 0$
  $w \Leftarrow w_i^0$
  while $1 - \sum_{t=0}^{\Delta_i(n)-1} w(t) > \epsilon$ **do**
    $t \Leftarrow \Delta_i(n)$
    u = $w[t:]$ ▷ split, unstable part
    s = $w[:t-1]$ ▷ split, stable part
    $c = u * e_n$ ▷ convolution
    $w \Leftarrow [s, c]$ ▷ merge
    $n \Leftarrow n + 1$
  end while
  return w
end function

4.4.3 Stochastic Queuing Delay and Response Time

Similar to the error-free strict-priority non-preemptive case (cf. Chapter 3), the response time can be obtained by checking all events in the busy-period and find the one with the largest response-time. For the stochastic case this means we need to evaluate the response-time distribution for the first q^+ events and find a worst-case among them. Later we will elaborate on the number of events to consider q^+.

Analogous to the non-stochastic, error-free case, a task τ_i can start executing once all previously released events of τ_i and other higher priority tasks have executed. Once τ_i starts executing, it cannot be interrupted. Additionally, in the error-case we must also wait until all higher and same priority execution attempts and their recoveries are finished.

Similarly to the busy-period we introduce a probabilistic version of the queuing delay.

Definition 37 (Queuing Delay pmf)**.**
The queuing delay pmf $\Omega_{q,i}(t)$ is the probability that the q-th event is queued for time t and fully transmits without errors right after.

Analogous, we introduce the stochastic version of the processing time.

Definition 38 (Processing Time pmf)**.**
The queuing delay pmf $\mathcal{B}_{q,i}(t)$ is the probability that the q-th event is fully transmitted at time t including eventual errors and recovery.

as well as a the stochastic response time:

Definition 39 (Response Time pmf)**.**
The response time pmf $\mathcal{R}_{q,i}(t)$ is the probability that the response time under the critical instant assumptions of the q-th event is time t including eventual errors and recovery.

The definition implies that all erroneous transmission and recovery attempts of the q-th event itself have finished before time t, so that the response time pmf of the q-th event can be calculated as the queuing delay pmf shifted by the execution time.

The computation of the queuing delay pmf associated with the q-th event is very similar to the computation of the busy period with minor modifications: Again, we iterate over the events in order of their release ($\Delta_i(n)$). However, for the computation of $\Omega_{q,i}(t)$ only the first q events of τ_i must be included as well as all events of the tasks of higher priority.

Naturally, the stochastic queuing delay includes execution and recovery time spend for the first $q-1$ transmissions of task under consideration τ_i. Only for the q-th event itself the last successfull transmission *must not* be included. However potential retransmissions of the q-th event *must* be included. Hence, we shift the execution time pmf by C_i. This effectively subtracts C_i from the total execution time.

$$e_n^q(t) = e(t + C_i) \tag{4.68}$$

Note that all other tasks have an execution time pmf according to eq. 4.57.

To determine the actual queuing delay pmf we use the same algorithm as for the busy-period pmf. That is, we apply eq. 4.63, eq. 4.64 and eq. 4.65 to $\Omega_{q,i}(t)$ rather than $\Omega_i(t)$. The starting condition obtained by eq. 4.66 can also be used for $\Omega_{q,i}^0$. For the sake of completeness we provide the slightly modified equations. Split, to compute the relative backlog pmf (this resembles eq. 4.63):

$$b_i^{n+1}(t) = \begin{cases} \Omega_{q,i}^n(t + \Delta_i(j+1)) & \text{if } t \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{4.69}$$

Convolution, to include the execution time of event $n+1$. Note that this step is slightly different from eq. 4.64: If the q-th event is encountered, the modified execution time 4.68 is used.

$$\tilde{b}_{q,i}^{n+1}(t) = \begin{cases} \left(b_i^{n+1} * e_{n+1}^q\right)(t) & \text{if } j+1 = \tau_{i,q} \\ \left(b_i^{n+1} * e_{n+1}\right)(t) & \text{otherwise} \end{cases} \tag{4.70}$$

Merging of the new backlog in the queuing delay pmf (this resembles eq. 4.65:

$$\Omega_{q,i}^{n+1}(t) = \begin{cases} \Omega_{q,i}^n(t) & t < \Delta_i(n+1) \\ \tilde{b}_{q,i}(t) & \text{otherwise} \end{cases} \tag{4.71}$$

Based on the queuing delay $\Omega_{q,i}$, we can compute the probabilistic processing time of the q-th event by adding (right-shifting) the execution time of the q-th event of τ_i:

$$\mathcal{B}_{q,i}(t) = \Omega_{q,i}(t - C_i) \tag{4.72}$$

Accordingly, the response time pmf is calculated by subtracting (left-shifting) the earliest arrival time.

$$\mathcal{R}_{q,i}(t) = \mathcal{B}_{q,i}(t + \delta_i^-(q)) \tag{4.73}$$

At that point we need to evaluate $\mathcal{R}_{q,i}(t)$ for some q, to determine the worst-case. Theoretically, an infinite number of events must be considered, since the derived stochastic busy window is infinitely long. In practice, the evaluation can be stopped when the probability that the q-th event falls in the busy-period is smaller than a given threshold.

$$q^+ = \max_{q>0, q\in\mathbb{N}} \left\{ q \mid \left(1 - \sum_{t=0}^{\delta_i^-(q)} \Omega_i(t) \right) < \epsilon \right\} \tag{4.74}$$

In practical cases it is sufficient to set ϵ to the machine epsilon (i.e. 2^{-52} on 64-bit floating point machines). It is virtually impossible that a larger busy-period is observed in real systems.

The exceedance function can be computed by the converse probability of the response time pmfs.

Theorem 29. *The probability that the q-th event exceeds some response-time t is given as:*

$$X_{q,i}(t) = 1 - \sum_{x=0}^{t} R_{q,i}(x) \tag{4.75}$$

Proof. The $R_{q,i}(t)$ is a conservative bound, that a response time less or equal to t is seen. Hence, $\sum_{x=0}^{t} R_{q,i}(x)$ is a conservative bound that a response time less or equal than t is observed. The converse probability, thus is a conservative upper bound that the response time t is exceeded. □

Thus, for any event q which arrives in the busy-period, we obtain a unique exceedance function. These functions can be merged to get a conservative view on any event.

Theorem 30. *An upper bound to the probability that any event in any possible busy window exceeds a response time t due to interference and errors is:*

$$X_i^+(t) = \max_{1\leq q\leq q^+} X_{q,i}(t) \tag{4.76}$$

Proof. Individual $X_{q,i}(t)$ are conservative by construction. Let's assume two given functions $X_a(t)$, $X_b(t)$. For a given t_0 a larger value exceedance value implies a higher probability of exceeding the threshold t_0. This obviously holds for any t_0. Thus, a piecewise maximum of both functions $X_a(t)$, $X_b(t)$ yields a conservative exceedance function which approximates $X_a(t)$ as well as $X_b(t)$ conservatively. This argumentation can be transfered to q individual functions which yields aforesaid equation. □

4.5 Experiments

In the following sections we show the application of the previously presented approaches to off-chip as well as on-chip communication. Additionally, we want to evaluate the performance of systems under different (burst) error rates and compare the tightness of the results. A typical representative for off-chip automotive communication is Controller Area Network, CAN [233] and its successor CAN-FD [113]. The standard in on-chip communication and IP integration is the Advanced Microcontroller Bus Architecture (AMBA).

4.5.1 Controller Area Network

The Controller Area Network (CAN) [3] is one of the most prominent buses used in various fields (e.g. automotive, industrial, aerospace) today. Despite its age, it has been introduced in the 80's, it is still in use today due to its cost advantage, versatility and robustness against errors. Due to its simplistic nature - CAN is a priority driven serial bus - it is often used for real-time system where the worst-case timing delays of transmissions must be predictable.

The CAN protocol is a multi-master, differential, serial bus. On the physical layer, data is Non-Return-to-Zero (NRZ) encoded. The CAN transceiver output consists of an open-collector or "wired and" circuit, thus the wire can be driven to two states: dominant (0) or recessive (1). All connected transceivers may drive the bus at the same time and the state is determined by the logical AND function of all driver inputs. Thus a CAN bus subscriber can "overwrite" the bus-line by sending a dominant bit.

Data is transmitted in frame entities which are non-preemptable, where each frame consists of the following blocks: A start of frame marker, an 11-bit frame identifier, control field, up to eight data bytes, a 15-bit CRC followed by an acknowledgement field and an end-of-frame marker.

An exact protocol description can be found in the official specification [3]. In 1991, CAN 2.0 introduced extended-frames which allow a 29-bit frame identifier. The arbitration scheme uses carrier sense multiple access/bitwise arbitration (CSMA/BA) and is based on the fact that dominant bits "win"

the access to the physical medium. Thus, the smaller the CAN bus identifier, the higher the priority of the frame.

The actual length of one frame for s payload data bytes is not necessarily fixed due to *bit stuffing*, where the controller inserts certain bits in order to avoid long sequences of 1's and 0's. In [61] it was shown, that the maximum length in bit times t_{bit} for one base frame of size s is

$$C = (55 + 10s)\ t_{bit} \tag{4.77}$$

and for extended frames as:

$$C = (80 + 10s)\ t_{bit} \tag{4.78}$$

All receiving nodes constantly check for protocol consistency during the transmission of frames. If framing rules are violated (e.g. missing stuffing-bits, missing acknowledgement), or the CRC field does not match the payload, an error can be signaled by all bus subscribers. The CAN standard defines two error frames for this purpose: the active error frame and the passive error frame.

When an error frame is transmitted, other nodes drop the frame and a retransmission of the broken frame is triggered. The worst-case overhead for an error frame can be given as

$$O^s = 31\ t_{bit} \tag{4.79}$$

Then, after an error frame has been transmitted, the re-transmission has to complete in a new arbitration phase.

To evaluate the approach, we use a modified version of the 17-messages SAE benchmark as presented in [273]. The benchmark includes sporadic messages, as well as periodic messages. Sporadic messages are modeled by assuming a minimum interarrival time, also we assume that the CAN bus is part of a larger distributed, automotive network and the data which ought to be transmitted has been processed on different ECUs which results in an increased released jitter (e.g. due to scheduling on upstream ECUs). Thus, the used message set covers a broader spectrum and may be more applicable to today's automotive networks. This is, for some frames we relaxed the deadline and increased the jitter to 1ms. Besides from that, the benchmark was used as it is.

Single Bit Errors

We carried out the following experiment using a 125 kbit/s CAN bus and a single bit error rate of 10^{-7}, which was measured by [91] in an aggressive environment. We show the accuracy of the following three approaches:

1. Busy-Period approach (cd. Section 4.3.2 and Section 4.3.4).

Table 4.1: SAE CAN Benchmark

Prio	DLC	C [1] (bits)	E [2] (bits)	T (ms)	D (ms)	R (ms)
17	1	62	13	1000	5	1.416
16	2	72	13	5	5	2.016
15	1	62	13	5	5	2.536
14	2	72	13	5	5	3.136
13	1	62	13	5	5	3.656
12	2	72	13	5	5	4.256
11	6	112	13	10	10	5.016
10	1	62	13	10	10	8.376
9	2	72	13	10	10	8.976
8	2	72	13	10	10	9.576
7	1	62	13	100	100	10.096
6	4	92	13	100	100	19.096
5	1	62	13	100	100	19.616
4	1	62	13	100	100	20.136
3	3	82	13	1000	1000	28.976
2	1	62	13	1000	1000	29.496
1	1	62	13	1000	1000	29.52

[1] for standard ID, without intermission space
[2] active error frame minus end of frame

2. Convolution-based approach (cf. Section 4.4).

3. Monte Carlo Error Simulation of the Critical Instant.

In the following, we coined the busy-period approach "Broster's approach" [44, 45], which we extended to support multiple schedulers as well as queued events. For the Monte Carlo analysis, we simulated the critical instant 10^6 times under the aforementioned single bit error-model and recorded the worst-case response times. The exceedance function was computed via the empirical CDF.

Figure 4.12 shows the results for Frames F1 and F9. Note that the Monte Carlo results are only statistically significant down to 10^{-5} due to the sample size. Unsurprisingly, the convolution-based method (c) matches the Monte Carlo results (m) in which the critical instant under errors is simulated. This is because not additional pessimism is introduced during the computation and the model captures all effects without over approximation. Also unsurprisingly, the convolution-based (c) approach is significantly tighter than Broster's.

The results tell us that Frame F1, virtually never exceeds its deadline of 1000 ms. Even a response time of 140 ms only occurs with a probability of 10^{-15}. However, for Frame F9, the situation is slightly different. Given a deadline of $D = 10$ ms, a deadline miss can be seen in one out of hundred cases. Whether this is sufficient depends on the actual application. How-

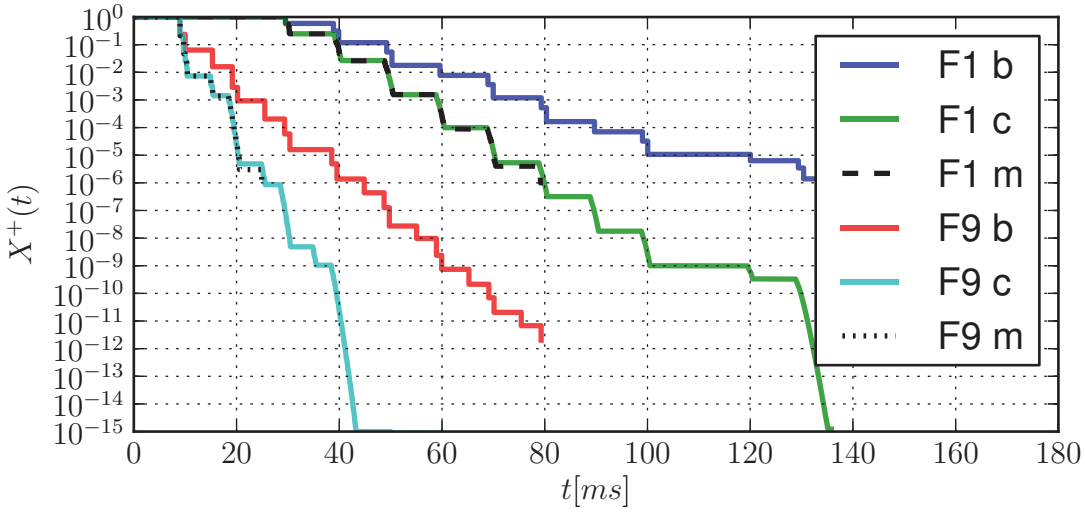


Figure 4.12: Exceedance functions according to modified Broster (b), convolution based approach (c) and Monte Carlo simulation (m) with 10^6 samples. $\lambda = 0.00024$ per bit $\cong$ 30 per sec @ 125kbit/s.

ever, we can also tell that a response time of 20 ms is never exceeded for one in a million events. Thus, in typical automotive controller applications these figures are most likely sufficient. Otherwise, better EMI shielding must be used to decrease the error rate or the critical messages must be transmitted at a higher priority level.

4.5.2 On-Chip Interconnect Arbitration

In the following experiments, we evaluate the effect of errors in on-chip communication. Therefore, we consider a typical scenario in which a number of processing units (PU) are connected to a common, shared memory. This memory can be external or internal RAM. Each processing unit is an ARM processor clocked at 1.1 Ghz with a split L1 cache. D$ as well as I$ are assumed to be 32kB. For the experiments, we assume a 32-bit interconnect datapath and 64 byte L1 cache lines. Hence, an interconnect transaction consists of a single cycle address phase and 16 cycles data (write or read)

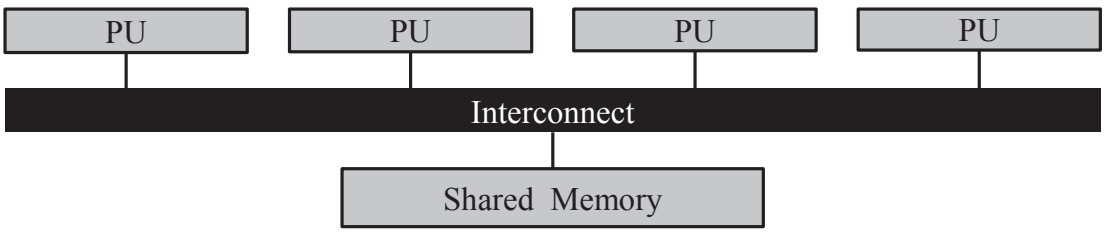


Figure 4.13: MPSoC architecture for interconnect experiments. It is implicitly assumed that the processing units (PU) include a cache.

burst. Memory latency is assumed to be constant and four system clock cycles.

Processing units execute application kernels. For our experiments we used workload from the EEMBC [211] benchmark suite. In particular, we used seven automotive kernels such as angle to time conversion, fast fourier transform, bit manipulation, CAN remote data request, iir filter and others. For this experiment, we used EEMBC application kernels as listed in Table 4.2. A full specification of the kernels can be found in [211].

As an input for our timing analysis, we need the access pattern to the interconnect. Following the approach of [84], we obtained the access pattern by tracing each PU without external interference, dedicated access to the memory and cold caches. It is straight forward to obtain the minimum distance function $\delta^-(n)$ from an event trace, as explained for instance in [194, 168]. Therefore, we used the data kindly provided by [84] for our analysis. This yields a good approximation on a worst-case event model.

The access pattern of almost all kernels consists of alternating phases of memory bound and CPU bound phases. However, the length of these phases depends on the benchmark. Large memory bound phases lead to relatively large busy periods compared to off-chip networks (such as CAN) where only a few packets are buffered. This is because we conservatively assume that software is executing as quick as possible, where in fact the memory path will cause backpressure.

Figure 4.14 depicts the event models obtained from simulation traces using the timed ARM model of a GEM5 simulator. It can be seen that the I/O bound phase for task bitmnp01 takes approximately 25,000 cycles. After this time window little to no requests are performed. However, benchmark aifft01 consumes more data (1200 accesses) until computation starts at 40,000 cycles. The initial slope of all event models is roughly one event in 30 cycles or 17 bit per cycles (each access is 64 bytes). This is

Kernel	R^+ [us]	Memory Burst Size [accesses]	Priority
a2time01	122.46	839	7
aiifft01	95.36	1419	6
bitmnp01	79.10	806	5
canrdr01	60.72	936	4
idctrn01	39.73	1067	3
iirflt01	14.49	893	2
ttsprk01	0.35	224	1

Table 4.2: EEMBC kernels mapped to the processing units. Worst-case respone time (error-free), memory bust size as well as priority level are indicated.

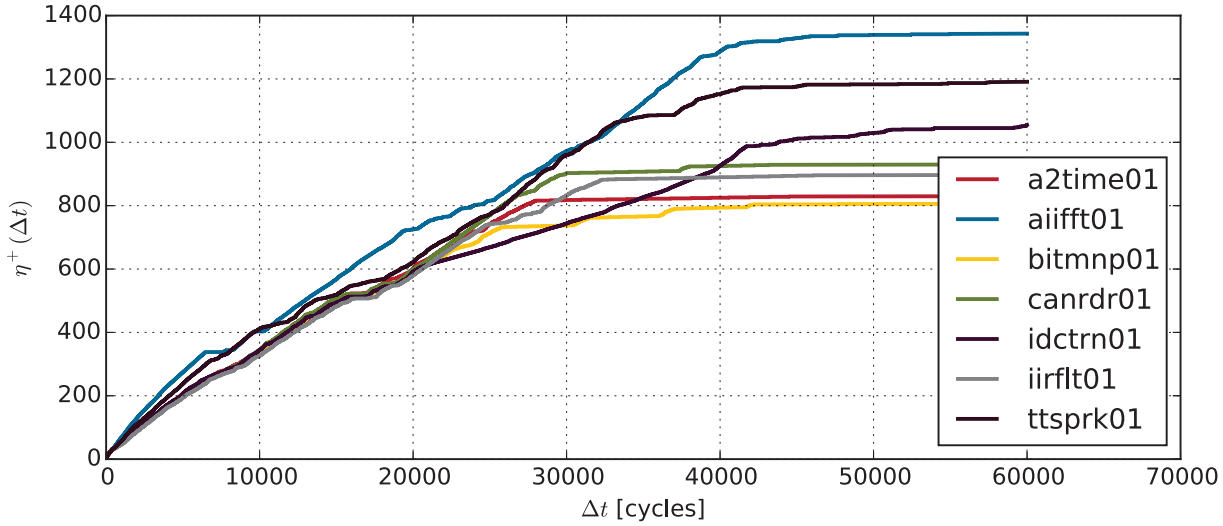


Figure 4.14: Event model obtained by simulation [84].

the maximum throughput of the simulated system under best-case (no-contention) conditions.

We mapped the applications to individual PUs. This causes contention in the interconnect. We assumed a fixed priority access arbitration scheme in which each PU has an assigned priority level. We analyzed the memory access latency of each application in the system (cf. Figure 4.13). Each transaction is composed of a single cycle address phase, a 16 cycle data burst and 4 cycles memory latency. The (error-free) analysis revealed that the busy-period of the interconnect is extremely large (152,481 cycles). During this time all kernels fetch their workingset data and instructions. The number of accesses during this time is also indicated in Table 4.2 (Memory Burst Size). The response times seem large at first glance, but are only in the order of a few microseconds (at 1.1 Ghz) which is negligible considering the execution times are in the order of a few milliseconds.

Single Bit Error

There is no reliable number on the transient error rate of today's 20nm silicon processes. Thus, we first assume a single bit error distribution with an soft error rate (SER) of $\lambda = 10^{-6}$ errors per cycle. This translates to an average of 1000 errors per second on a 1 Ghz system. Later we also cover burst errors.

We assume that an erroneous transaction is detected by parity checks. The controller schedules a re-arbitration phase as well as a single cycle error response phase. Then the transaction is restarted and eventually completes. This leads to a total error overhead which only causes 23

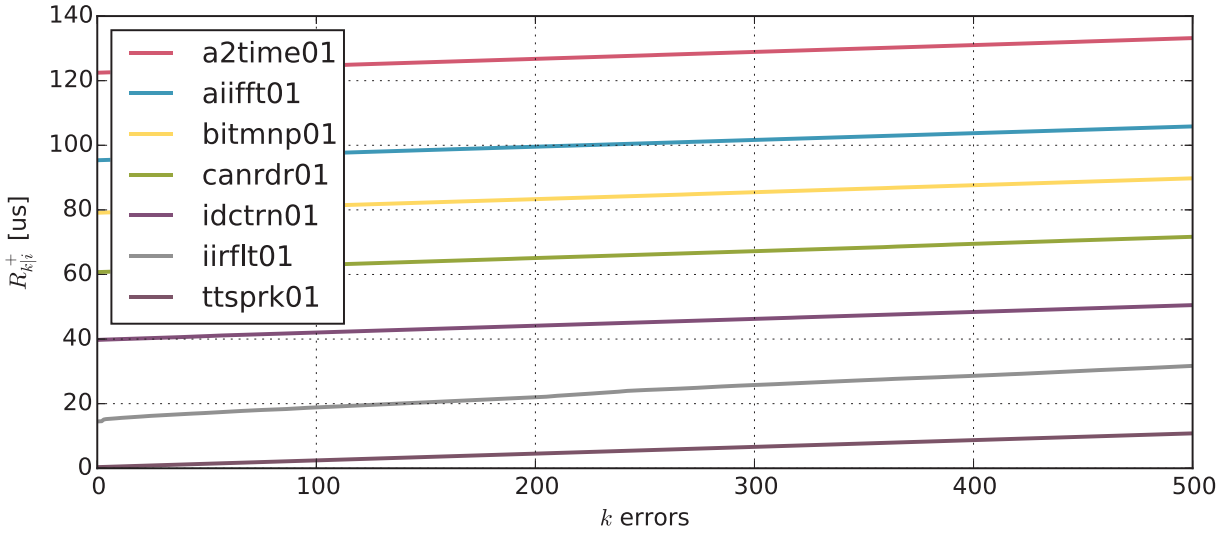


Figure 4.15: Response time $R^+_{i|k}$ over a given number of errors $k$in the busy-period .

cycles overhead (including retransmission) per error Figure 4.15 shows the response time $R^+_{i|k}$ as a function of the number of errors in the busy-period. The response time naturally increases with the number of errors. However, the relative increase is small and typically only a few percent of the response time. This is because data arrives in huge bursts consisting of many hundreds to thousands of transactions. The overhead by errors is little compared to the overall number of transactions during the busy period.

The exceedance function is shown in Figure 4.16. It can be seen that the function for all applications decays rapidly. There is a considerable difference, if we compare the results with the experiments carried out for CAN (cf. Figure 4.12). For CAN the decay is by far not as rapid as for the on-chip interconnect.

For instance bitmnp01 exhibits an error-free response time of 79 us. We can see that the memory latency will be in no cases larger than this since $X^+(79us) = 1$ since the exceedance function sharply drops to values below 10^{-10}. Virtually no response time larger than that will ever be observed.

Generally, we can conclude that the impact of errors naturally is higher in communication systems with larger transactions and slower links (such as CAN). For on-chip communication which occurs in large bursts that leads to long busy-periods, additional errors only add very little overhead compared to the already existing workload. This leads to the conclusion that on-chip single bit errors in today's processors do not cause significant timing problems if error detection and correction works correctly. However,

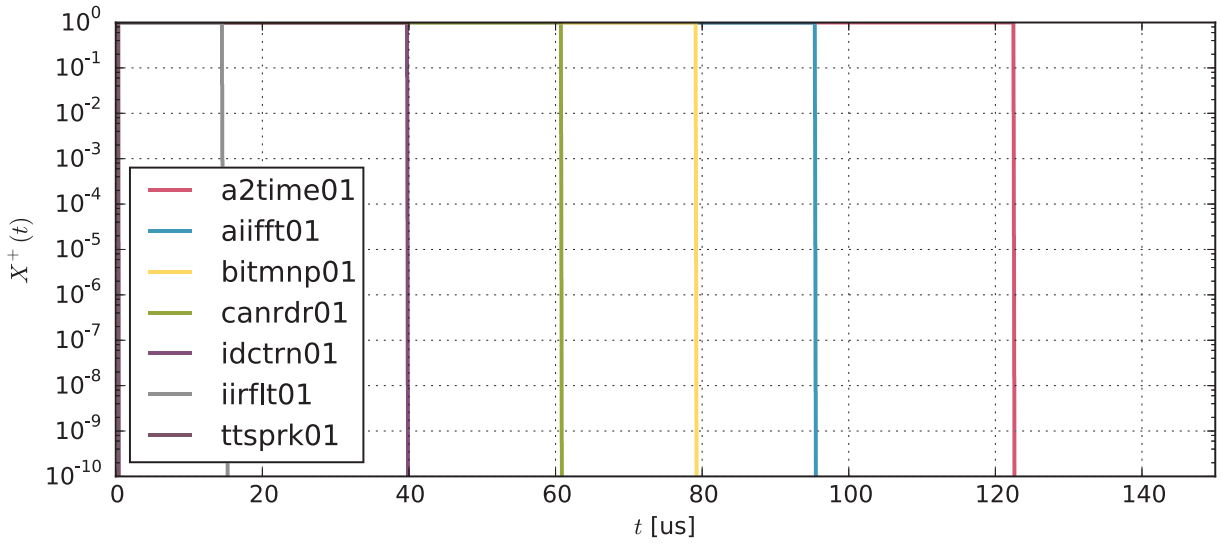


Figure 4.16: Exceedance funtion for the example taskset. Single bit errors $\lambda = 10^{-6}$ per cycle.

for on-chip communication with larger transaction length such as NoC communication, this is not necessarily true[3].

Burst Error

Now, we assume a burst error process is superimposed to the communication. A realistic reason for such errors is transient voltage drops caused by high transient currents. Again, exact parameter values cannot be obtained, thus we assumed an average burst length of 25 cycles and an average bit error rate of $\lambda = 10^{-6}$. Hence, the long term error rate matches the one from the single bit error experiment. Since the k-error response times are not a function of the underlying bit-error process, the k-error response times match. In this sense, Figure 4.15 is also valid for the burst experiment.

The resulting exceedance function is shown in Figure 4.17. For the experiment, we computed probabilities considering up to 300 error events. Beyond this number, the binomial coefficient used in the probability computation are intractable and further approximation is needed. Generally, it can be seen that the timing impact for burst errors is much larger: the exceedance functions still decay fast compared to the CAN experiment but not as fast as for single bit errors. The results for bitmnp01 tell us that a response-time larger than 85 us can be observed with a probability of approximately 10^{-8}.

[3]Under the assumption that the NoC is additionally hardened so that a transient error never leads to permanent error.

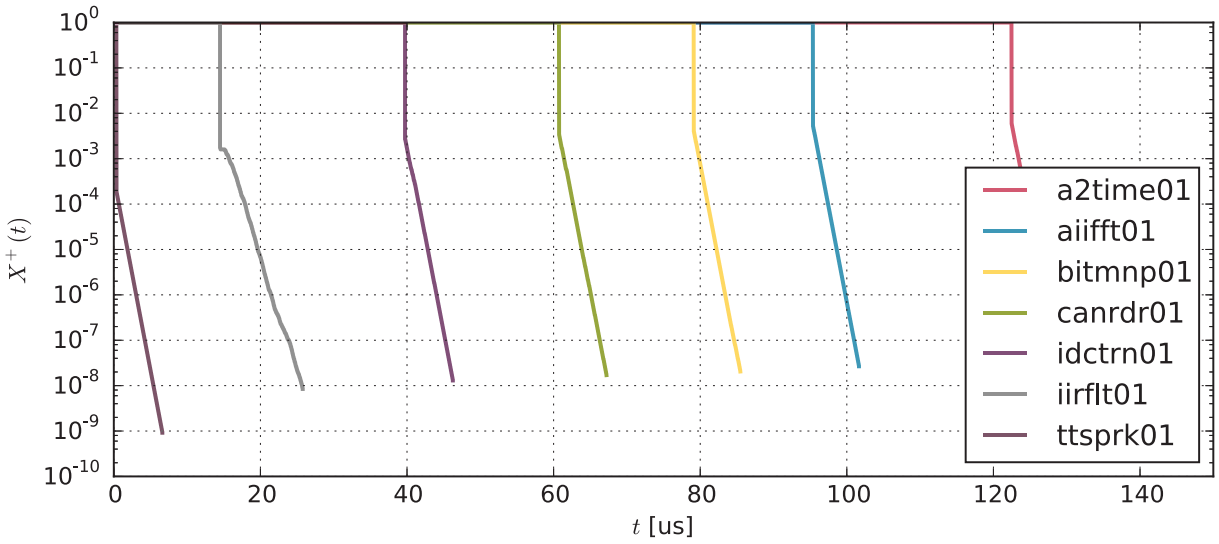


Figure 4.17: Exceedance funtion for the example taskset. Burst error, average burst length of 25 cycles and average long term bit error rate of $\lambda = 10^{-6}$.

One reason for this is that the arrival of a burst will have a larger timing impact because of the increased overhead (more transactions are affected). On the other hand, we must also note that the analysis over approximates this overhead by considering each bit-error as a transaction error. This most likely does not occur in reality. As discussed, compared to the single bit error experiment, burst errors have a higher probability to cause deadline violations. Generally, for all of the analyzed applications the likelihood of exceeding the error-free worst-case response time by more than 20 us is so small that such events will never be seen in a running system. Under worst-case error-conditions (20 us worst-case memory latency overhead), Eq. 3.38 tells us that the overall execution time of the associated application is only increased by 20us. This is reasonable and allows us the add this margin to a processor scheduling analysis.

4.6 Summary

In this chapter, we discussed communication aspects of the ASTEROID system, covering multi-master busses such as Controller Area Network as well on-chip interconnects (e.g. AMBA). We formalized the underlying communication channel and presented two commonly used error models: binary symmetric channel for single bit errors as well as the two state Gilbert loss model which captures burst errors.

The timing impact of errors on individual messages is modelled and conservatively approximated. We have presented two approaches to capture the stochastic nature of errors in a worst-case performance context. First, we used the busy-period approach and extended related work [45] to reflect queued activations and bursts. Second, we presented a more accurate convolution-based approach that can be used for single-bit errors.

The experiments show that the timing impact of errors in off-chip communication is higher compared to on-chip communication. This is because off-chip transactions (i.e. CAN frames) are larger and the transmission speed is slower compared to quick, parallel on-chip communication.

CHAPTER 5

Switched Networks

In the previous chapter, we focused on errors in traditional busses or point-to-point communication. In this chapter, we consider switched networks such as Network on Chip or Ethernet-based systems. Contrary to the previous chapter, we focus on the performance analysis of switched networks under errors only without deriving reliability bounds.

Depending on the internal switch architecture and the transmission protocol, a correct packet delivery cannot be guaranteed under all conditions. There are two reasons for lost packets, both are depicted in the illustrative example shown in Figure 5.1. The first cause are bit errors on the wire generated by interference such as crosstalk, noise, or an imprecise sampling point synchronization (cf. Chapter 4). In most on-chip and off-chip protocols (e.g. Ethernet and QPI [299]), packets are protected by checksums for instance by a CRC, so it is most likely that bit errors are detected at the switch or receiver.

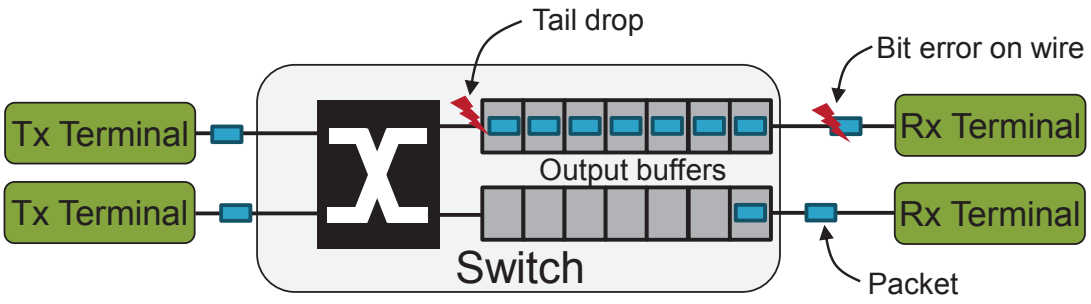


Figure 5.1: Four terminals connected to a switch. Packets can be dropped caused by buffer overrun (tail drop) or due to bit errors on the wire.

The second cause of dropped packets is inherent to non-blocking switches. Obviously, if multiple packets arrive at different input ports and are targeted towards the same output port, congestion is unavoidable. Packets are temporarily stored in queues (cf. Figure 5.1) and scheduled for transmission. However, if packets arrive too fast the queues will run over and packets will eventually be dropped.

Thus, for data transfers which require a reliable in-order delivery, a higher level transport protocol such as Automatic Repeat ReQuest (ARQ) [176, 264] is necessary. In scope of the ASTEROID platform (cf Chapter. 2) this affects the timing of on-chip NoC communication as well as off-chip, high speed communication.

In this chapter, we first discuss the related work in the field of error control protocols as well as performance evaluation. In particular we revisit end-to-end error control protocols such as ARQ and Go-Back-N which take care of an integer, in-order data transmission in case of dropped packets. We then present a formal worst-case latency prediction which is used in ASTEROID to maintain safe timing bounds in on-chip as well as off-chip communication. We specifically evaluate the off-chip characteristics of Go-Back-N and Stop and Wait in an automotive Ethernet use-case. The work presented in this chapter is based on [22].

5.1 Related Work

There are various flavours of ARQ such as Stop-and-wait ARQ as well as Go-Back-N and Selective Repeat (and further variations thereof) [264]. For instance, GigE Vision, an Ethernet-based standard for high-performance industrial cameras, implements a variant of Selective Repeat on top of UDP. All of these protocols share a similar concept: Successful (or unsuccessful) delivery is signaled back (ACK/NACK) on a return path by the receiving terminal, so the sending terminal knows when to retransmit certain packets.

Worst-case, error-free analysis of switched networks without error-control protocols are broadly available. This included AFDX [33], standard Ethernet [237, 270], Ethernet AVB [74, 72]. Also standard busy-period-based approaches for network on chip communication are available [252, 245, 73]. However, none of these articles discusses the impact of high-level error control protocols.

A survey of existing end-to-end error-control protocols was presented in [53]. Further consideration of errors in NoC, associated errors as well as error-control are summarized in [189]. Approaches that we presented in Chapter 4 to assess the timing under error effects [44, 45, 248] do not capture end-to-end error protocols and cannot be applied to these networks.

Although an ARQ-based mechanism is used, these approaches are only capable to model broadcasting busses or single point-to-point links.

There is a major difference between a bus-based communication and a switched network: In bus-based systems, a packet corruption is detected by all terminals simultaneously and the retransmission is immediately scheduled. In switched networks, an unsuccessful delivery is noticed only after an acknowledgement timeout or negative acknowledgement (depending on the actual protocol). Hence, the methods used for busses as presented in the previous chapter cannot be applied to switched networks such as Ethernet or NoCs.

There is a large body of stochastic considerations of ARQ-based schemes that can be applied to wired and wireless networks [294, 46, 278, 126]. These results were mostly obtained by simulation or queuing theory (e.g. [167]). The obtained data is typically expressed as probability distributions on the packet loss, queue length as well as latency. However, probabilistic (average-case) approaches are not suitable to the problem in scope of this work since they cannot give hard worst-case bounds. For instance, these approaches are not capable to deliver hard latency guarantees not even under the error-free case.

5.2 Error Control Protocols

As previously motivated, dropped or corrupted packets are a common scenario in most networks. During the following sections, we assume that all packets are equipped with a checksum of sufficient error-detection capabilities. Furthermore, we focus on the network and assume that the sending and receiving terminals are never faulty themselves. That is, terminals are always capable to detect checksum mismatches, do not crash, and do not lose packets due to buffer overflows. As already motivated, switches are allowed to drop packets due to buffer overrun or bit errors. We furthermore assume that switches never stop operating for an unlimited time (permanent error). In the following sections we outline the operation of two popular error control protocols the simple *Stop and Wait ARQ* as well as the more complex *Go-Back-N ARQ*.

5.2.1 Stop and Wait ARQ

The simplest approach to provide a transparent error-free communication link is the Stop and Wait ARQ protocol [264]. An exemplary sequence chart of Stop and Wait ARQ is shown in Figure 5.2. The operation is straight forward. The sender sends the first data packet ❶ and upon arrival, the receiver acknowledges the data ❷. Once the acknowledgment is received by the sending terminal, the next packet can be send. Sometimes a data

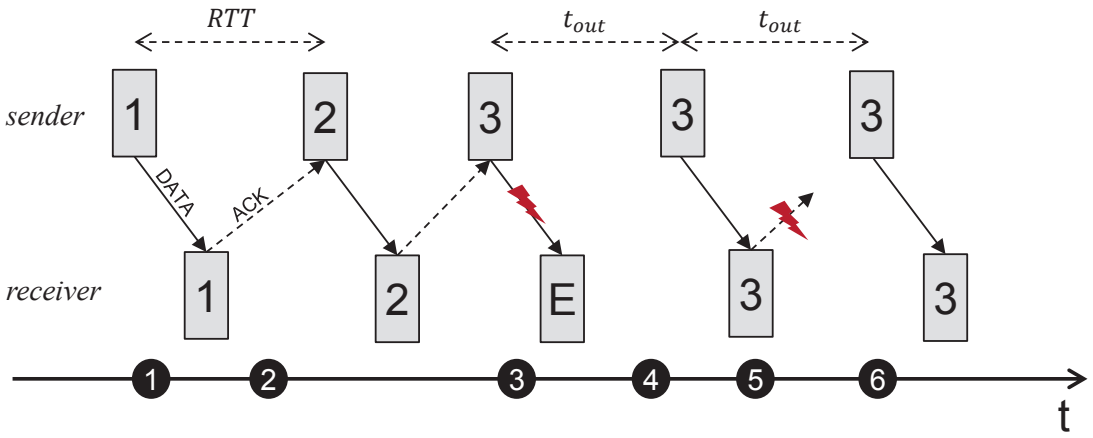


Figure 5.2: Illustrative example for Stop and Wait ARQ. The sending terminal waits for a positive acknowledgement before sending the next packet. Errors are detected after a timeout.

packet is corrupted or entirely dropped ❸ and the receiving terminal does not send the ACK. After a timeout, the sender notices the error and retransmits the data packet ❹. Eventually, an ACK packet is dropped ❺, so the receiving terminal receives an integer data packet but the sending station nevertheless retransmits the packet after the timer expired. In this case, it is possible that the receiver ❻ has received a duplicate packet. To eliminate such duplicates, a one-bit sequence number is typically used [264]. Note, that throughout the following sections we assume no negative acknowledgment (NACK) is used.

Definition 40 (Round Trip Time)**.**
The best-case / worst-case round trip time RTT^- / RTT^+ is the lower / upper bound of the time interval from the start of the transmission of a data packet at the sending terminal until a positive acknowledgment has been received by the sending terminal.

Obviously, the achievable throughput for Stop and Wait ARQ is constrained by the round-trip time RTT_i^+ of the network. Under worst-case conditions, the sustainable datarate can never be higher than C_i/RTT_i^+. Otherwise, more and more packets will arrive at the ARQ buffer which eventually overflows. Also the latency, especially for bursty traffic, is weak as packets are held back until acknowledged.

5.2.2 Go-Back-N

Go-Back-N is an extension of the simple Stop and Wait ARQ, which enables faster transmission by admitting a limited number of packets, while still

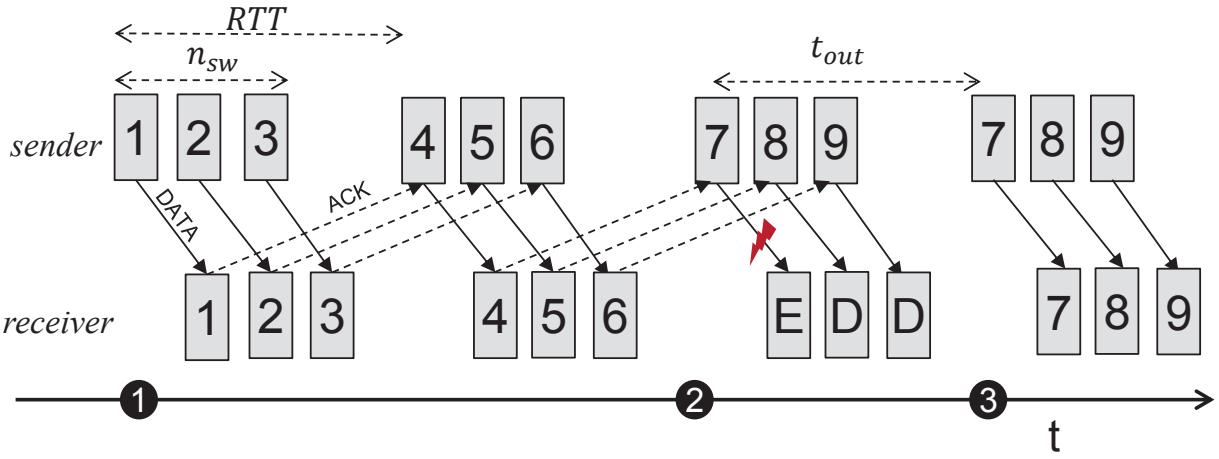


Figure 5.3: Illustrative example for Go-Back-N ARQ. Sending terminal sends out n_{sw} packets, then waits for a positive acknowledgement before sending the next packet. Hence, there can be n_{sw} packets in-flight. In case of a dropped data packet, a retransmission of at most n_{sw} packets is scheduled after a timeout t_{out}.

waiting for outstanding acknowledgements. This approach is widely used for instance in the High-Level Data Link Control (HDLC) as well as X.25 an ITU-T standard protocol for packet switched wide-area network communication. Figure 5.3 shows an illustrative example of the Go-Back-N operation. Contrary to Stop and Wait ARQ, Go-Back-N immediately sends out a number of packets ❶. The maximum number of packets which are allowed in the network pipeline is called *send window* (n_{sw}). For normal operation (highest throughput), the send window should be in the order of the bandwidth-delay product of the network topology. The bandwidth-delay product is the typical capacity of simultaneous in-flight messages (pipeline depth) of the network.

At time ❷ in Figure 5.3, the ACK of the first four packets were received and the next packets are transmitted. Packet 7 arrives corrupted (E) at the receiving node and is not acknowledged. As for Stop and Wait, no negative acknowledge (NACK) mechanism is used. All packets which arrive later and are out of sequence (packet 8 and 9) are discarded (D). Thus, for simple Go-Back-N, the receiver has a receive window of 1, as it accepts only the next valid packet. After the sending terminal runs into the timeout at time ❸ it retransmits all packets that remain unacknowledged.

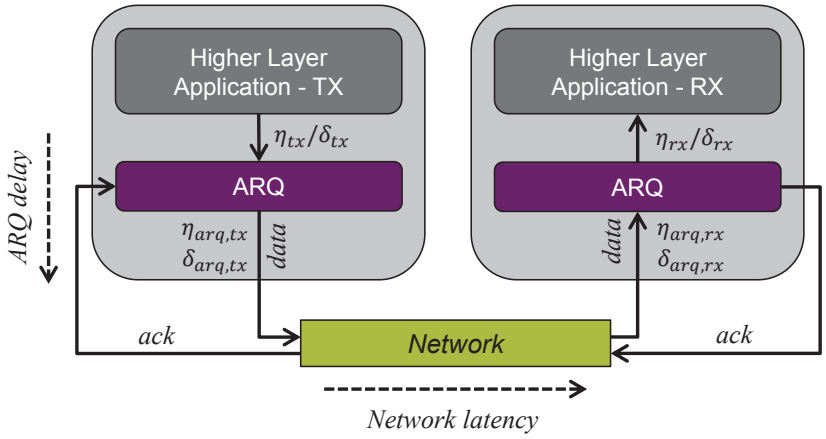


Figure 5.4: Timing model of an ARQ-based protocol.

5.3 Performance of Stop and Wait ARQ

In the following sections, we derive the worst-case latency of a packet as well as a sequence of packets for the error-free as well as for the error case (pathological case) under the Stop and Wait protocol.

5.3.1 ARQ Timing Model

For the sake of simplicity, we assume that an application is associated with a single stream ($stream(i)$), thus there is a unique sender, receiver relationship between nodes. In a communication system which uses no error control protocol, packets are send immediately by the terminals once data is ready to be transmitted. Accordingly, packets are immediately scheduled for transmission once they are assembled by the sending application. As we have seen in the previous section, this is not the case for Stop and Wait ARQ, where packets are held back by the sending terminal until a positive acknowledgement has been received (or a timeout occurred). In this section, we model this effect and extend CPA accordingly.

Figure 5.4 shows the ARQ timing model which we use throughout the next steps. The figure depicts the following aspects, which are relevant for timing considerations:

1. Sending application (Higher Layer Application TX).
2. ARQ protocol stack (ARQ) of the sending terminal.
3. The network topology is simplified as a black box (Network).
4. ARQ protocol stack (ARQ) of the receiving terminal.
5. Receiving application (Higher Layer Application RX).

Any data exchange between these units is abstracted by event models. The TX application tries to send data according to the injection event model η_{tx}. The data packet is fed into the ARQ module which eventually delays a packet. The ARQ module then feeds data into the network. The event stream at this edge is bounded by the ARQ event model $\eta_{arq,tx}$. After the packets have passed the network, a stream which is characterized by $\eta_{arq,rx}$ arrives at the ARQ unit of the receiving terminal and is passed to the application η_{rx}. Intuitively, is seems natural that $\eta_{arq,tx} = \eta_{tx}$, however this is not the case. Unacknowledged data could be held back by the ARQ unit and queue up. Then, if the network allows this, acknowledgements arrive as quickly as possible and a large output burst can be seen (best-case after worst-case).

5.3.2 Latency in the Error-Free Case

For each packet which is sent out by the application, the overall system latency is composed of two parts: The ARQ delay induced by data link layer (ARQ) as well as the network latency, which is the worst-case time a packet travels through the topology until it is received by the receiving terminal. Unluckily, both the ARQ delay as well as the network delay are mutually dependent. That is, the ARQ delay is a function of the network delay, since packets have been held back until an ACK is received. Vice versa, the network latency depends on the ARQ behavior because the more packets are injected the more interference occurs.

This dependency is resolved by the CPA fixed-point iteration loop and the following analysis is interleaved with the CPA analysis. Similarly to the analysis of shared resources in CPA as presented in [240, 242], we add an analysis step to the outer CPA fixed-point iteration. This *ARQ analysis* step is also depicted in Figure 5.5.

First, we compute the worst-case network latency. Then, we derive the worst-case ARQ latency and update the ARQ event models accordingly. These steps are repeated until convergence. Once the fixed-point is found, the overall latency can be computed by summation over the response times. We assume, that once the packet is received, it is immediately processed and handed over to the application with no additional delay. Modelling protocol stack latencies and jitter is easily possible but omitted for the sake of clarity.

For the following steps we assume that once $\eta_{arq,tx}$ is available, we use approaches from related work (cf. Section 5.1) to compute the worst-case latency and worst-case round-trip time through the topology.

So the first step is to derive $\eta_{arq,tx}/\delta_{arq,tx}$ which can be used for the analysis of the switched network. The ARQ event model is a function of the application's event model, specified by δ_{tx} and the worst-case / best-case ARQ delay R^+_{arq}/R^-_{arq}. For the following analysis steps, we assume that the

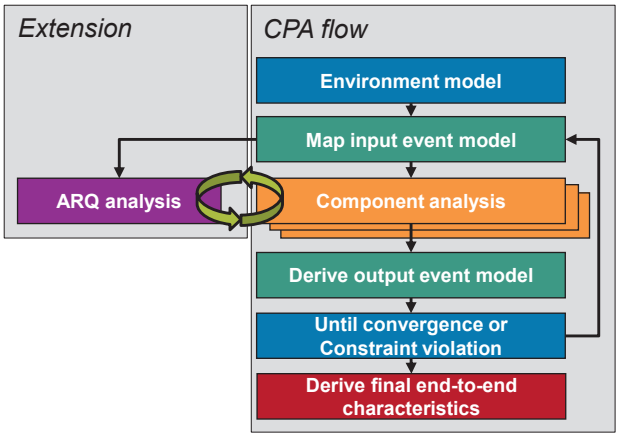


Figure 5.5: Compositional Performance Analysis Flow (right) and ARQ extension (left).

round-trip time through the network is significantly smaller than the time t_{out}. This obviously must be the case in all properly configured networks (a packet is never assumed to be lost, although it is still in the network pipeline). As described in Chapter 3, δ can be converted into η, which is then used for the response-time analysis. Thus, it is sufficient to only derive δ.

Theorem 31 (ARQ Event Model)**.** *The effective ARQ event model $\delta^-_{arq,tx}$ which is emitted by the data link layer, if no packets are ever dropped or corrupted is bounded by*

$$\delta^-_{arq,tx}(q) = \max\left\{\delta^-_{tx}(q) - R^+_{arq} + R^-_{arq}, B^-_{arq}(q-1)\right\} \tag{5.1}$$

where

- *δ^-_{tx} denotes the minimum distance function of packets send out by the TX application.*
- *R^-_{arq}/R^+_{arq} are the best-case/worst-case response times of the ARQ.*
- *$B^-_{arq}(q)$ is the smallest time interval in which any q packets can ever pass the ARQ unit.*

Proof. For a proof two cases need to be considered. Both are conservative bounds as we will show later. Accordingly, both cases can be computed independently and the better case is used (hence the max in the equation).

For the first case, we can model the ARQ unit as a processing element which delays a packet some time in the interval $[R^+_{arq}, R^-_{arq}]$. For this we can use Eq. 3.37 to derive the output event model for jitter propagation. A proof for this is given in [243] and is a generalized form of Eq. 3.37.

On the other hand, packets can never be forwarded faster than they get acknowledged. Here, $B^-_{arq}(q-1)$ is used as a valid lower bound on the distance of any q packets at the output of the ARQ unit. Assuming one packet is emitted at some time, then the following $q-1$ packets are emitted after time $B^-_{arq}(q-1)$ earliest. Thus, $B^-_{arq}(q-1)$ is a lower bound on the distance between any q events at the output. □

Throughout the chapter, we assume that the best-case round-trip time RTT^- is computed by adding the best-case transmission times of the data packet and the corresponding ACK packet along the (forward and backward) path through the network according to the worst-case latency equations given in Chapter 3 (cf. eq. 3.38). The component analyses are carried out and the worst-case response times along the stream ($stream(i)$) are summed up to derive the network latency:

$$L_i = \sum_{\forall j \in stream(i)} R^+_j \tag{5.2}$$

Similarly, the worst-case round-trip time RTT^+_i can be computed by also considering the return path of the acks (*ackstream(i)*).

$$RTT^+_i = L_i + \sum_{\forall j \in ackstream(i)} R_j \tag{5.3}$$

5.3.3 Stop and Wait Response Time

We are going to show how to derive the worst-case ARQ delay (R^+_{arq}), the largest time interval which a packet is resting in the ARQ buffer. Similar to Eq. 3.19 which is used to derive the busy-period for fixed priority arbitration, we can establish a busy-period equation for the ARQ protocol. We model the ARQ behavior by a single task which executes for time RTT^+_i. This resembles the behavior that a transmission of a packet starts earliest after the previous ACK arrives after time RTT^+_i.

Theorem 32 (Stop And Wait Busy Period)**.** *The largest time interval $w_{arq,i}$ in which packets arrive at the terminal and need to be queued at the ARQ buffer while the terminal is waiting for ACKs of previous packets is given by*

$$w_{arq,i} = \eta_{tx,i}(w_{arq,i}) \cdot RTT^+_i \tag{5.4}$$

where

- *$\eta_{tx,i}(\Delta t)$ is an upper bound on the number of packets send by the application.*

- *and RTT_i^+ is the round trip time through the network.*

Proof. Similar to the argument as used in [61], the worst-case is seen, if the queue is empty and packets arrive as fast as possible. It is straight forward to show that after time $w_{arq,i}$ the ARQ protocol immediately transmits the next packet because all previous packets have been sent and the corresponding ACKs have been received. □

The next step is to compute the time is takes the ARQ unit to process q packets under worst-case conditions:

Theorem 33 (Worst-Case Multiple Packet Forwarding Time)**.** *The worst-Case multiple packet forwarding time is the largest time interval to forward a sequence of q packets under an error-free, Stop and Wait ARQ. It starts with the arrival of the first and ends with the transmission of the q-th packet and all but the first packet arrive before the preceding one is acknowledged. It is computed by*

$$B_{arq,i}^+(q) = (q-1) \cdot RTT_i^+ \tag{5.5}$$

where

- *q is the number of packets*
- *RTT_i^+ is the worst-case round-trip time of a packet through the network.*

Proof. The first packet out of the sequence of q packets can be transmitted right away. Once the ACK of the already in-flight packet is received after time RTT^+, the next packet is transmitted. Hence, by induction the q-th packet is transmitted at time $(q-1) \cdot RTT^+$. □

Theorem 34 (Best-Case Multiple Packet Forwarding Time)**.** *The minimum time interval to forward a sequence of q packets under Stop and Wait ARQ is lower bounded by*

$$B_{arq,i}^-(q) = (q-1) \cdot RTT_i^- \tag{5.6}$$

where

- *q is the number of packets*
- *RTT_i^+ is the worst-case round-trip time of a packet through the network.*

Proof. The proof is straight forward: The first packet can pass immediately, the following packets must wait the best-case round-trip time until they are dispatched. □

As for fixed-priority scheduling, the worst-case response time can be found among all packets which arrive during the busy-period.

Theorem 35 (ARQ Worst-Case Response Time)**.** *The worst-case response-time of the ARQ protocol is upper bounded by*

$$R^+_{arq,i} = \max_{1 \leq q \leq \eta^+_{tx,i}(w_{arq,i})} \left\{ B^+_{arq,i}(q) - \delta^-_i(q) \right\} \tag{5.7}$$

Proof. See proof of Theorem 1. □

Analogously, the best-case response time R^-_{arq} can be assumed to be zero as packets are forwarded immediately under optimal conditions. In most situations ARQ is used to transmit a large burst of data (i.e. a large dataset such as a video frame), which is composed of multiple packets. It is worthwhile to know the latency of a number of q packets that belong to one data entity (e.g. the video or LIDAR frame).

Theorem 36 (System Latency)**.** *The overall system latency for q data packets which are to be transfered assuming an error-free Stop and Wait ARQ channel is given by:*

$$\mathbb{L}_i(q) = \delta^-_i(q) + L_i + R^+_{arq,i} \tag{5.8}$$

Proof. This is a generalization of the path latency presented in Eq. 3.38.

It takes time $\delta^-(q)$ to inject q packets into the system. In the worst-case scenario, the last (q-th) packet experiences the worst-case ARQ delay ($R_{arq,i}$) as well as the worst-case network latency (L_i). Due to causality (FIFO semantics, packets cannot pass each other), we know that all previous packets must have been received, if the last packet arrives at the receiver. Thus, Eq. 5.8 is a valid upper bound for the overall system latency of q packets. □

5.3.4 Timing Under Errors

In the previous section, we assumed that all packets were delivered intact. This naturally includes acknowledgements as well as data packets. We lift this assumption and evaluate the pathological cases in which either data or ACK packets are dropped or corrupted. Throughout this section, we assess the timing under a given number of dropped or corrupted packets and evaluate the latency under this scenario.

Generally, valid packets can be dropped (i.e. in case of buffer overflow) or corrupted (in case of bit errors). A store-and-forward switch most likely detects the bit error and drops the packet. A virtual cut-through switch forwards the corrupted packet and the receiving terminal notices the error. Through the rest of the paper we speak of “dropped packets” which

includes packets dropped caused by buffer overflows as well as bit errors. Figure 5.2 illustrates how a timeout mechanism detects missing ACKs and correspondingly triggers retransmission of data.

Throughout the following paragraphs, we extend the previously introduced equations to derive the latency bounds under the assumption of k-dropped packets. This value then serves as a sensitivity figure for the consequence of dropped packets.

Let us construct the worst-case timing scenario under the assumption that a single packet is dropped. The drop can occur at any hop (switch/terminal), however, in the worst-case scenario a packet is lost at the last hop right at the receiving terminal. Thus, it has imposed the maximum interference on the network but never arrives healthy at the destination. Obviously, data as well as ACK packets can be subject to this problem.

It makes a difference which kind of packet is lost. A lost ARQ packet is handled differently than a lost data packet. The interested reader is advised to consult [264]. Timing-wise, it is worse to drop an ACK packet, right before it reaches the destination tx-terminal. Thus, data as well as ACK packet fully congest the network but the sender still runs into a timeout and has to retransmit.

The sending terminal notices the problem after a time t_{out} and resends the last data packet (cf. Figure 5.2). Hence, similar to the busy period as introduced in the previous section, we can derive the k-error busy period. This is the busy period under the assumption that k packets have been dropped during transmission. We extend the previously established equations by adding the worst-case blocking caused by dropped ACK packets.

Theorem 37 (k-Error Stop and Wait Busy Period)**.** *The largest time interval $w_{arq,i}(k)$ in which packets arrive at the terminal and need to be queued at the ARQ buffer while the terminal is waiting for ACKs of previous packets under the assumption that k packets were lost is upper bounded by*

$$w_{arq,i}(k) = k(t_{out} + RTT_i^+) + \eta_{tx,i}(w_{arq,i}(k)) \cdot RTT_i^+ \tag{5.9}$$

where

- *k is the number of errors*
- *t_{out} is the ARQ timeout and*
- *RTT_i^+ is the worst-case round trip time.*

Proof. The proof is analogous to the one of Theorem 32 under the assumption that there is an additional worst-case blocking time. □

Similarly, we extend the multiple packet forwarding time to include the overhead caused by errors.

Theorem 38 (k-Error Multiple Event Forwarding Time)**.** *The k-error multiple event forwarding time is upper bounded by*

$$B^+_{arq,i}(q,k) = k(t_{out} + RTT^+_i) + (q-1) \cdot RTT^+_i \tag{5.10}$$

where

- *k is the number of dropped packets*
- *t_{out} is the ARQ timeout and*
- *RTT^+_i is the worst-case round trip time.*

Proof. A packet is admitted to the network if the previous $q-1$ ACKs where received assuming k packets where dropped. This leads to a worst-case overhead of $t_{out} + RTT^+_i$ for each retransmission of where there are k. □

Finally, we are able to compute the worst-case ARQ latency, assuming k dropped packets, by substituting the multiple event forwarding time in Eq. 5.7 by the k-error multiple event forwarding time. This leads us to the following ARQ response-time bound:

$$R^+_{arq,i}(k) = \max_{1 \le q \le \eta^+_{tx,i}(w_{arq,i}(k))} \left\{ B^+_{arq,i}(q,k) - \delta^-_{tx,i}(q) \right\} \tag{5.11}$$

The k-error system latency can be computed by using Eq. 5.8.

5.4 Performance of Go-Back-N

5.4.1 Latency in the Error-Free Case

In this section, we derive the worst-case response time of the Go-Back-N ARQ scheme assuming that no packets are lost. The steps are similar to the Stop and Wait ARQ. That is, we compute the worst-case busy period to evaluate the largest time interval in which the Go-Back-N unit is busy waiting for outstanding ACKs. Then, we compute the largest multiple packet forwarding time for each packet in the busy period which leads to the worst-case response time.

The peculiarity of Go-Back-N is that at any time there can be up to n_{sw} packets in-flight. Thus, if no packets are queued, the first n_{sw} packets are transmitted immediately, whereas the $(n_{sw}+1)$-th packet must wait until the first packet is acknowledged and so on. We assume that the send window is known and does not change over time. In fact, we can color each arriving packet according to its waiting partners. In Figure 5.3 there are $n_{sw} = 3$ groups: we can group $\{1, 4, 7, \dots\}$, $\{2, 5, 8, \dots\}$, and $\{3, 6, 9, \dots\}$.

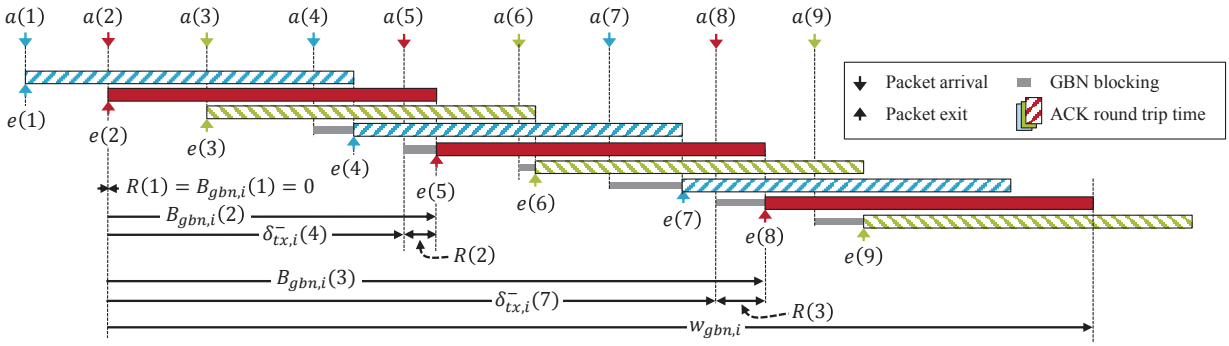


Figure 5.6: Go-Back-N Gantt chart. Packets and the ACK round trip are colored according to their position in the send window. $a(q)/e(q)$ indicate arrival and exit time of the q-th packet.

These groups are scheduled independently from each other given sufficient number of packets to be transmitted are available.

The concept of groups is also illustrated in Figure 5.6 for the first 8 packets of the stream (under no errors). Here $a(q)/e(q)$ denote the arrival/exit times of individual packets into and out of the ARQ unit, respectively. Packets are colored according to their position in the send window. Blue packets wait for blue ACKs, red packets wait for red ACKs and so on. Note that packet coloring repeats in a cyclic way. Obviously, the red packet 5 which arrives at time $a(5)$ only has to wait for acknowledgements of red packets which have arrived earlier. Someone could argue that a very late arrival time $a(1)$ could delay the exit time $e(5)$, (i.e., if $a(1) > a(2)$) but this violates causality. Vice versa, we can conclude that if packets arrive as quickly as possible, the interference for later packets is always maximized. Hence, the critical instant assumption (packets arrive as quick as possible) leads to a worst-case scenario for Go-Back-N.

To derive the worst-case response time for a Go-Back-N scheduler we consider the groups independently from each other. As we will see later, it is a safe approximation to assume that all groups exhibit the same timing under worst-case conditions, hence, it is sufficient to derive the worst-case timing for one group (i.e. red in Figure 5.6) and deduce the timing behavior for all groups (hatched blue and green).

Theorem 39 (q-Packet Group Forwarding Time)**.** *The q-packet group forwarding time $B_{gbn,i}(q)$ of a $stream(i)$ is given by the time it takes to forward a sequence of packets which contains precisely q packets of a given group. It starts with the arrival of the first and ends with the transmission of the q-th packet of that group, and all but the first packets arrive before the preceding is acknowledged. The maximum q-packet group forwarding time $B^{+}_{gbn,i}(q)$*

for Go-Back-N under no errors is independent of the actual group and upper bounded by

$$B^+_{gbn,i}(q) = (q-1) \cdot RTT^+_i \tag{5.12}$$

- q *is the number of (potentially non consecutive) packets which belong to the* same *group*
- *and* RTT^+_i *is the round trip time.*

Proof. The last (q-th) packet of that group can be sent out latest when the ACKs for all previous $q-1$ packets of that group are received. Since each acknowledgment takes at most time RTT^+, we obtain Eq. 5.12. Also as the right hand side of Eq. 5.12 is independent of the actual group (and thus position in the send window), so we can conclude that also $B^+_{gbn,i}$ is. □

Note, that the semantics of the functions B_{gbn} and B_{arq} are actually identical as Stop and Wait is a special case of Go-Back-N for $n_{sw} = 1$. Thus, a Stop and Wait stream consists of only a single group.

Theorem 40 (Group Busy Period)**.** *Given any group* G *of stream* i*, the group busy period is the maximum time interval* $w_{gbn,i}$ *in which the ARQ protocol has outstanding ACKs of group* G*. The maximum group busy period is upper bounded by*

$$w_{gbn,i} = \left\lceil \frac{\eta^+_{tx,i}(w_{gbn,i})}{n_{sw}} \right\rceil \cdot RTT^+_i \tag{5.13}$$

- n_{sw} *is the number of outstanding unacknowledged packets*
- RTT^+_i *is the round trip time*
- *and* $\eta^+_{tx,i}(\Delta t)$ *is an upper bound on the number of packets send by the application.*

Proof. During any time window Δt, there can be at most $\lceil \frac{\eta^+_{tx,i}(\Delta t)}{n_{sw}} \rceil$ packets of group G. Thus, the longest waiting time is bounded by the worst-case round-trip time RTT^+ times the number of packets of group G. □

The group busy period starts with the arrival of a packet of group G and ends with the reception of the acknowledgement of the same group. During the group busy period, also packets of other groups can arrive and may have outstanding ACKs at the end of $w_{gbn,i}$. Again, for the special case $n_{sw} = 1$, the group busy period for Go-Back-N matches the busy period of the Stop and Wait protocol.

Theorem 41 (Worst-Case Go-Back-N Response Time)**.** *The maximum response time $R^+_{gbn,i}$ of a Go-Back-N terminal of stream i is obtained by*

$$R^+_{gbn,i} = \max_{1 \leq q \leq \tilde{\eta}^+_{tx,i}(w_{gbn,i})} \{R(q)\} \tag{5.14}$$

with

$$R(q) = B^+_{gbn,i}(q) - \delta^-_{tx,i}\left((q-1) \cdot n_{sw} + 1\right) \tag{5.15}$$

and

$$\tilde{\eta}^+_{tx,i}(\Delta t) = \left\lceil \frac{\eta^+_{tx,i}(\Delta t)}{n_{sw}} \right\rceil \tag{5.16}$$

where

- $R(q)$ *is the response time of the q-th packet of a sequence of q belonging to the same group*
- $B^+_{gbn,i}(q)$ *is the time to forward q packets of the same group*
- $\eta^+_{tx,i}(\Delta t)$ *is an upper bound of the number of packets, regardless of the group*
- $\tilde{\eta}^+_{tx,i}(\Delta t)$ *is an upper bound of the number of packets which belong to the same group*
- n_{sw} *is maximum number of unacknowledged packets*
- $\delta^-_{tx,i}(q)$ *is the minimum distance between any q packets, regardless of the group.*

Proof. The response time is by definition the time interval from the arrival of the packet at the ARQ unit until it is forwarded to the network. Similar to the busy-period in fixed-priority scheduling, there are at most $\eta^+_{tx,i}(w_{gbn,i})$ packets which can interfere with each other (cf. Eq. 3.20). However, as we know, only packets of the same group can interfere. Out of $\eta^+_{tx,i}(w)$ packets there are at most $\tilde{\eta}^+_{tx,i}(w)$ packets of the same group. All of these packets must be checked for their response time.

The response time $R(q)$ of the q-th group packet is maximized, if the forwarding time is maximized and the arrival time minimized. The latest forwarding time of the q-th packet is by definition obtained by the maximum multiple packet forwarding time $B^+_{gbn,i}(q)$.

Now, we need to obtain the minimum distance between any q packets of the same group. Since the group pattern is repeated in a cyclic fashion (cf. Figure 5.6), we know that if we start with some group, the next packet of

the same group is seen after time $\delta^-_{tx,i}(n_{ws}+1)$ earliest, the second after time $\delta^-_{tx,i}(2n_{ws}+1)$. If we expand and generalize the sequence we get the total number of packets which must be observed in any sequence which contains q packets of the same group, as follows:

$$n = (q-1) \cdot n_{sw} + 1 \tag{5.17}$$

The minimum time window in which n packets can be observed is by definition $\delta^-_{tx,i}(n)$. Thus, $R^+_{gbn,i}$ is maximized. □

Similarly, we can derive the best-case behavior for any sequence of q packets, regardless of the group.

Theorem 42 (Best-Case Go-Back-N Multiple Packet Forwarding Time)**.** *The best-case Go-Back-N multiple packet forwarding time $B^-_{gbn,i}(q)$ for a stream i for any seqeunce of q packets is lower bounded by*

$$B^-_{gbn,i}(q) = \left(\left\lceil \frac{q}{n_{sw}} \right\rceil - 1 \right) \cdot RTT^-_i \tag{5.18}$$

where

- RTT^-_i is the best-case round trip time
- and n_{sw} is maximum number of unacknowledged packets.

Proof. The proof is analogous to the one of Theorem 40. □

5.4.2 Timing under Errors

As for Stop and Wait ARQ, we now consider the pathological case for Go-Back-N ARQ. Since Go-Back-N is more complicated, there are more cases to consider. At any time there are at most n_{sw} unacknowledged packets in-flight, so any of those packets can be corrupted. In case the first packet in the send window is affected (cf. packet 7 in Figure 5.3), up to n_{sw} must be retransmitted. However, if the last packet is affected and the send window did not advance because no additional data is available at the sending terminal, only one packet must be retransmitted. The case of a lost ACK is more subtle and we have to consider two cases[1].

1. An ACK is lost and the associated data packet was the most recent packet from the sender: In that case, the missing ACK is noticed after t_{out} time and a retransmission of the recent packet is initiated.

[1] These cases are not shown in Figure 5.3

2. Secondly, an ACK is lost but the transmitting terminal has sent further packets which were correctly acknowledged. In this case the sending terminal can implicitly assume the outstanding ACK because subsequent packets are confirmed which would remain unacknowledged otherwise.

For the following analysis, we approximate the real protocol behavior the following way: We assume that all data packets arrive at the receiver but the associated ACK is dropped on the last hop. As explained previously, subsequent ACKs implicitly acknowledged *all* previously sent data. We assume that this does not happen and the receiving terminal still waits for the missing ACK. This obviously is an overestimation but captures all possible effects at once: Data and ACK packets impose the highest load on the network while still waiting for the timeout.

Theorem 43 (K-Error Group Busy Period)**.** *Given any group G, the k-error group busy period is the maximum time interval $w_{gbn,i}(k)$ in which the ARQ protocol has outstanding ACKs of group G assuming exactly k dropped packets of group G during that time. The maximum k-error group busy period is upper bounded by*

$$w_{gbn,i}(k) = k(t_{out} + RTT_i^+) + \left\lceil \frac{\eta_{tx,i}^+(w_{gbn,i}(k))}{n_{sw}} \right\rceil \cdot RTT_i^+ \tag{5.19}$$

Proof. During any time window Δt, there can be at most $\lceil \frac{\eta_{tx,i}^+(\Delta t)}{n_{sw}} \rceil$ packets of group G. Thus, the longest waiting time is bounded by the worst-case round-trip time RTT^+ times the number of packets of group G plus an additional waiting time for each dropped packet. Under worst-case conditions a dropped packet leads to a timeout and an retransmission, thus the waiting time per error is upper bounded by $t_{out} + RTT^+$. □

Theorem 44 (k-Error, q-Packet Group Forwarding Time)**.** *The k-error, q-packet group forwarding time $B_{gbn,i}(q,k)$ of a $stream(i)$ is given by the time it takes to forward a sequence of packets which contains precisely q packets of a given group under the assumptions of k corruptions in that group. It starts with the arrival of the first and ends with the transmission of the q-th valid packet of that group, and all but the first packets arrive before the preceding is acknowledged. The maximum multiple error packet forwarding time $B_{gbn,i}^+(q,k)$ for Go-Back-N is independent of the actual group and bounded by*

$$B_{gbn,i}^+(q,k) = k(RTT_i^+ + t_{out}) + (q-1) \cdot RTT_i^+ \tag{5.20}$$

Proof. The proof is analogous to the one of Theorem 39. Additionally, under worst-case conditions, each error leads to a timeout t_{out} as well as a retransmission which is acknowledged after time RTT_i^+. □

Correspondingly, we can derive the response time under the assumption that k packets were dropped.

Theorem 45 (Worst-Case k-Error Go-Back-N Response Time)**.** *The maximum k-error response time $R^+_{gbn,i}(k)$ of a Go-Back-N terminal is obtained by*

$$R^+_{gbn,i}(k) = \max_{1 \leq q \leq \tilde{\eta}^+_{tx,i}(w_{gbn,i}(k))} \{R(q,k)\} \tag{5.21}$$

with

$$R(q,k) = B^+_{gbn,i}(q,k) - \delta^-_{tx,i}((q-1) \cdot n_{sw} + 1) \tag{5.22}$$

and

$$\tilde{\eta}^+_{tx,i}(\Delta t) = \left\lceil \frac{\eta^+_{tx,i}(\Delta t)}{n_{sw}} \right\rceil \tag{5.23}$$

Proof. The proof is analogous to the error-free counterpart, but instead $B^+_{gbn,i}(q,k)$ and $w_{gbn,i}(k)$ are used which are conservative by construction under the k-error assumption. □

5.5 Experiments

In our experiments we evaluate only Go-Back-N as Stop and Wait is a special case for $n_{sw} = 1$. Throughout the experiments, we apply the presented algorithms to off-chip switched Ethernet networks. However, as already mentioned, the established formalism is also valid for on-chip switched communication.

We assume a standard store and forward Ethernet switch as depicted in Figure 5.7. Ethernet frames arrive at the ingress port, and are parsed and forwarded to the correct egress port. Frames are queued in a first-in, first-out fashion until they are scheduled for transmission. Switches can be cascaded in any arbitrary topology. Sophisticated switches are capable to detect and break cycles (e.g. by using Spanning Tree Protocol STP), however, we restrict our scope to simple acyclic topologies, which are common in the embedded domain. For the analysis, we assume a constant wire delay of $33ns$ which translates to about $10m$ wire length (propagation at the speed of light).

5.5.1 Daisy Chain

At first, we consider a daisy chain topology consisting of five switches as depicted in Figure 5.8. A sending terminal (TX) sends a datastream using Go-Back-N to a receiving terminal (RX). Each frame carries 1024 bytes payload which contains the actual data as well as the Go-Back-N protocol

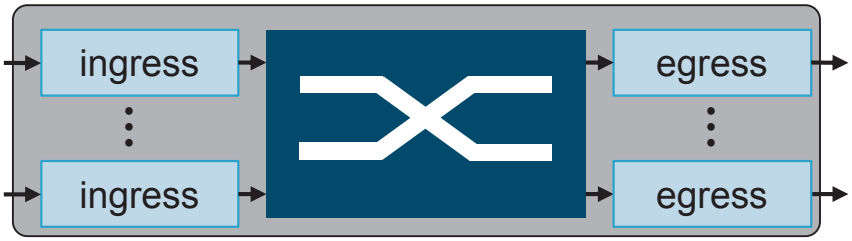


Figure 5.7: Ethernet frames are received on the ingress port and forwarded to the appropriate egress port, where they are queued until transmitted.

information. We assume a bursty datastream (e.g. radar data) which can generate a 20 kb burst (20 frames) with an average rate of 1 Mb/s. ACK packets are sent from the RX terminal back to the TX terminal using a minimum sized Ethernet frame which has 32 bytes payload. Piggypacking, where ACKs are merged with payload data on the return path, is not considered. Hence, a physical ACK frame is sent for each acknowledgement. Additional terminals connected to each switch periodically ($P = 0.5$ ms) send frames with 1024 bytes payload to the rx terminal.

To evaluate the influence of the send window on the latency, we swept over an interval $n_{sw} \in [1, 25]$. The results are depicted in Figure 5.9. For each experiment, we depicted the worst-case end-to-end latency for transmitting $20kb$ of data. The latency tends to drop with an increased send window; this is expected as the limited send window is the major cause for blocking. Surprisingly, the latency is not monotonically decreasing in n_{sw}. This is not very obvious, as an increased send window suggest less ARQ blocking, after all, all terms (i.e. R_{arq}) are monotonically decreasing in n_{sw}. However, an increased n_{sw} leads to increased network load because more packets are admitted in shorter time. This increased transient load leads to a higher RTT^{+} which negatively affects the system latency. We can conclude that for small n_{sw} the ARQ protocol has a self-regulating traffic

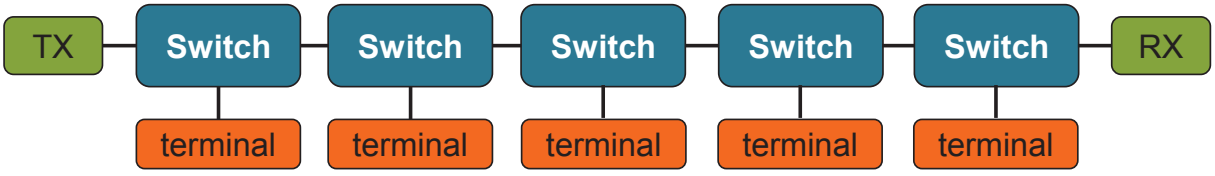


Figure 5.8: Daisy-chain topology. A sending terminal (TX) communicated with the receiving terminal (RX) using the Go-Back-N protocol. Further network congestion is generated by additional terminals.

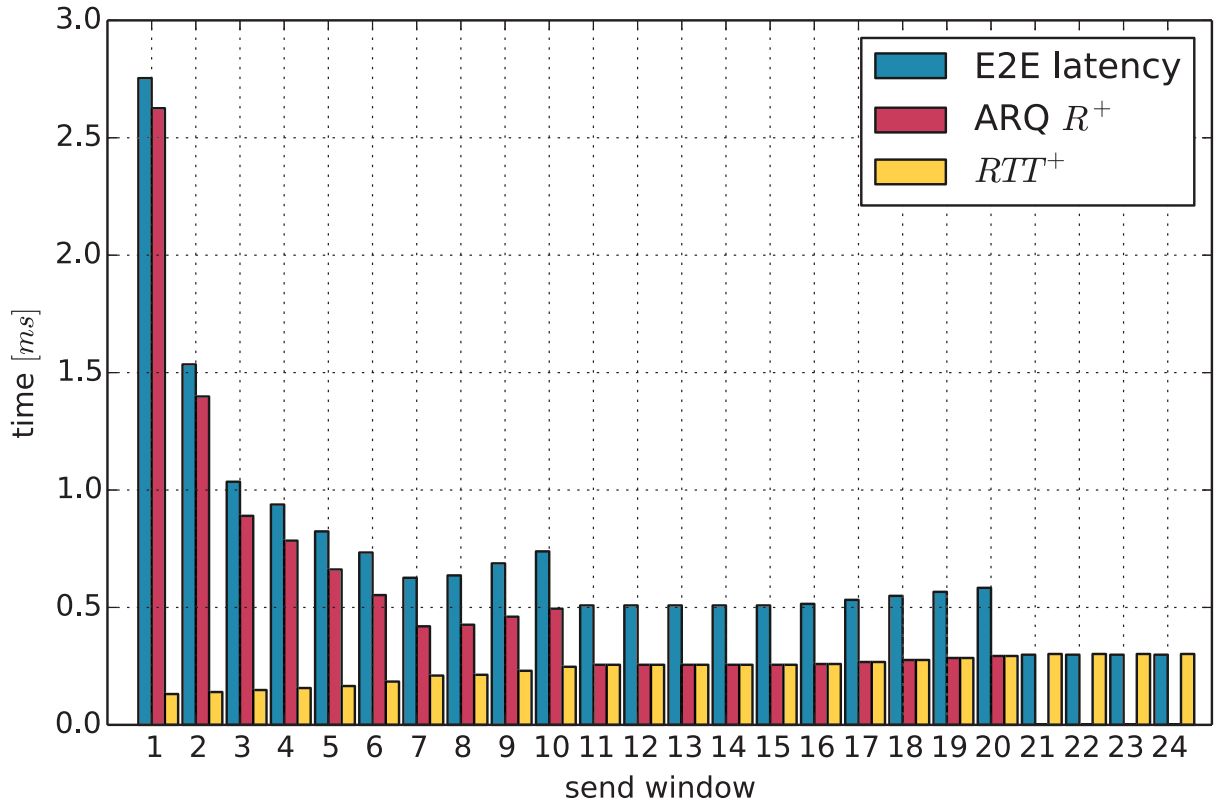


Figure 5.9: Formal worst-case latency.

shaping characteristic limited by the network load (higher round-trip times lead to less congestion).

For completeness, we compare the formal worst-case analysis with a simulation of the topology. We used a discrete event simulator to trace the behavior under the previously introduced setup. We directly injected large bursts (20 kb) into the ARQ unit and ran each experiment for 100 s of simulated time. The results are depicted in Figure 5.10. Note that the latency decreases monotonically with n_{sw} contrary to the formal analysis. It seems that the non-monotonicity, which is seen in formal analysis, is caused by the worst-case approximations. However, it must be noted that the simulation shows the *observed* worst-case and the actual worst-case is somewhere between the formal analysis results and the simulation.

The overestimation of the formal analysis compared to the simulation depends on the exact send window size but is in the order of a factor 3. This is partly due to the complex feedback of overestimations. Worst-case assumptions are considered in isolation for each hop which makes the line topology unfavorable. This is a trade-off between accuracy and analysis complexity. These local worst-case scenarios are partially mutual exclusive and, thus, the overestimation, as for any CPA analysis, increases with

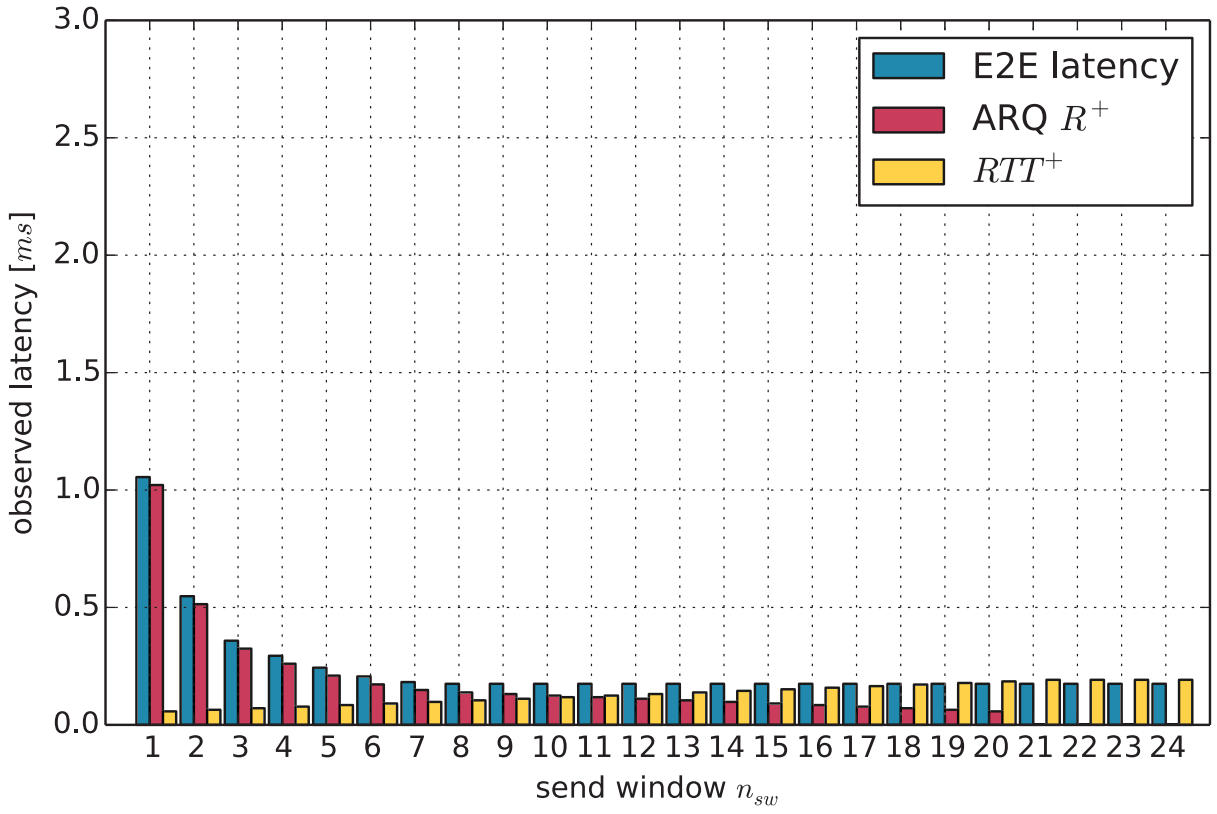


Figure 5.10: Simulated latency (worst-observed).

the number of hops. Still the absolute latency figures, obtained by the formal analysis, of less than 3 ms are promising. These values are still suitable for hard real-time data communication with very tight end-to-end constraints. This way, it gives solid, reliable worst-case guarantees which simulation results cannot provide. Also it must be noted that the runtime of the simulation (172 min) is 4 orders of magnitude larger than the formal analysis runtime (4s) on an Intel Core i7 processor.

5.5.2 Two Switches Automotive Setup

In this section, we apply the approach to a more realistic automotive Ethernet setup. As a realistic automotive use-case, we use a topology and traffic characteristics from a real in-car setup as presented in [173].

The Ethernet network consists of 14 end nodes which exchange data with different timing constraints. Two switches provide the interconnect for all devices as shown in Figure 5.11. A processing unit (HeadUnit) operates as a data sink for control, camera, and bulk streams. Two side cameras, as well as a rear camera transmit driver assistance video streams to the HeadUnit. The unit computes a bird's eys view from these streams used

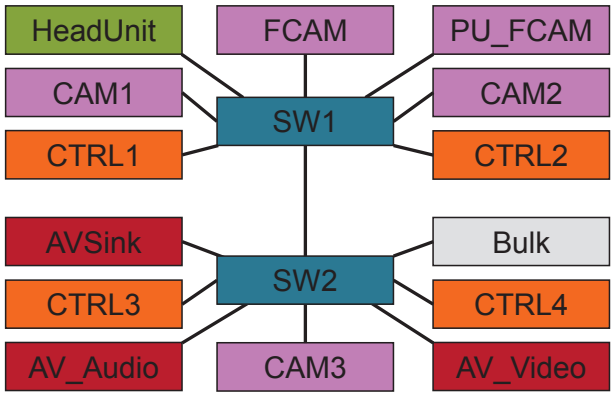


Figure 5.11: Automotive topology with two swiches.

as an input for further applications (i.e. pedestrian detection). The fourth camera (FCAM) captures a front video which is send to a dedicated front view processing unit (PU_FCAM).

The rear seats of the car are equipped with an AV entertainment system. Here, AVSink represents the rear seat entertainment system (RSE) which operates as the sink for AV_Audio and AV_Video streams. BulkTraffic resembles best-effort data (e.g. 4G Internet communication) which is send to the HeadUnit. The authors of [173] argue that all CAN and almost all Flexray control data can be packaged in 20 bytes UDP frames. They traced an existing system and found that inter-frame time of CAN and Flexray messages are between 10 and 100 ms (including event-driven as well as cyclic frames). As we need a conservative, worst-case approximation for the traffic classes, we assume an inter-frame time of 10 ms for all control frames.

It is assumed that driver assistance videos data is MPEG2-TS encoded with a bitrate of 25 Mbit/s. Each camera transmits an UDP frame each 0.25

Node Name	UDP/TCP Payload [byte]	Ethernet Payload [byte]	Period [ms]	Net Bandwith	Gross Bandwith
Ctrl1,...,Ctrl4	20	48	10	16 kbit/s	38.4 kbit/s
CAM1,...,CAM3	786	814	0.25	25.1 Mbit/s	26 Mbit/s
FCAM	786	814	0.25	25.1 Mbit/s	26 Mbit/s
Audio	1472	1500	8.4	1.4 Mbit/s	1.43 Mbit/s
Video	1472	1500	1	11.8 Mbit/s	12 Mbit/s
Bulk Traffic	1400	1440	1	11.2 Mbit/s	11.52 Mbit/s

Table 5.1: Traffic characteristics as used for the automotive setup.

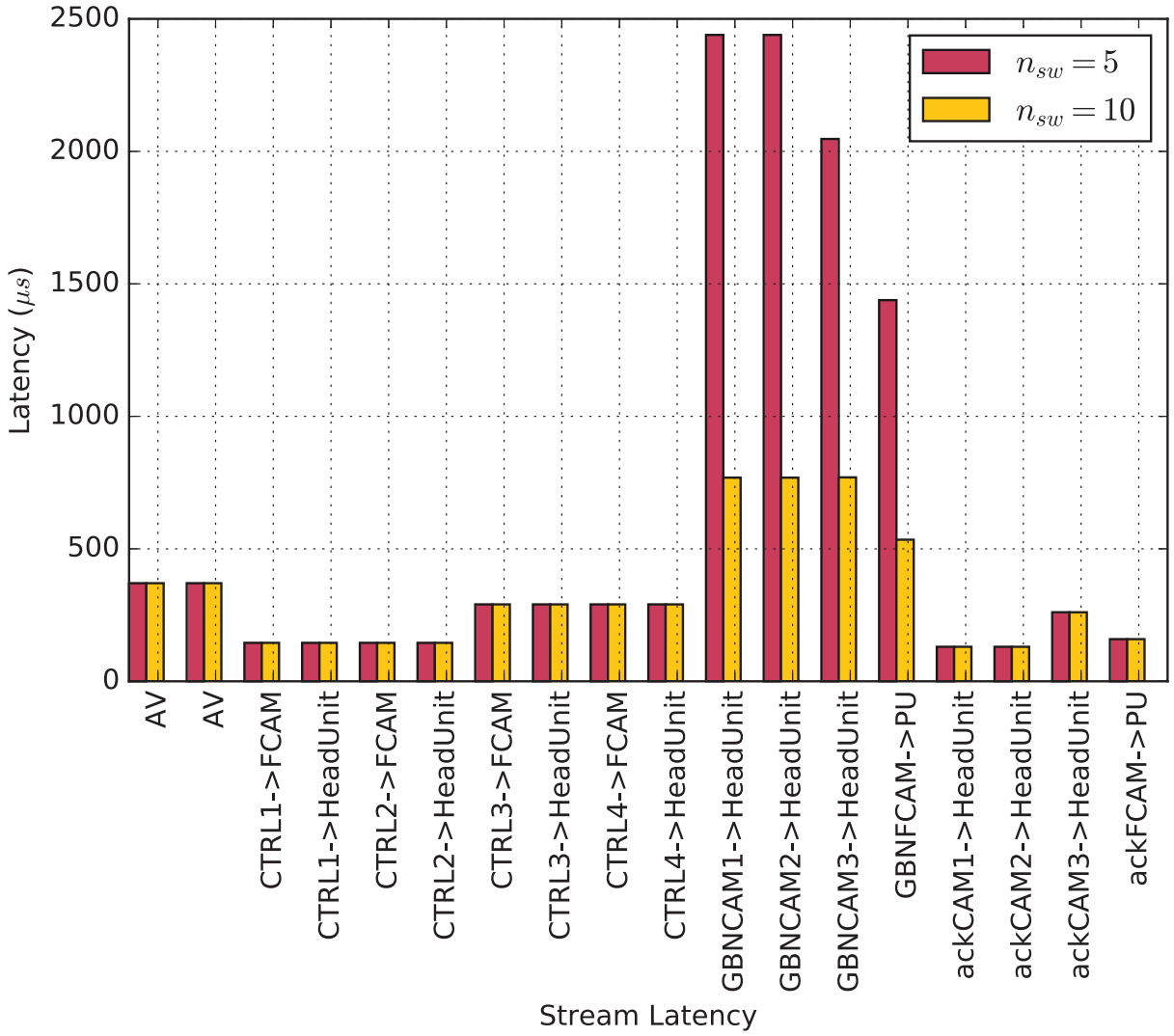


Figure 5.12: Latency for all streams in the automotive setup. *GBN* indicate the forward stream (i.e. CAM to Headunit), where *ack* indicates the latency on the acknowledgment path.

ms, with bursts of 5 packets. All camera streams use Go-Back-N to protect the data against corruptions. Control data such as sensor data is assumed to be updated and send so frequently that an additional error detection and correction is not necessary. Similarly, the RSE consists of DVD video data and 44.1 kHz stereo uncompressed audio (audio CD). In [173], the BulkTraffic consists of 15 TCP connections to the HeadUnit with up to 11.2 Mbit/s. As our model cannot capture the complex TCP flow-control protocol accurately, we model the TCP streams by independent frames to capture the imposed workload. For a detailed description of the traffic characteristics,

the interested reader is advised to consult [173]. An overview of the traffic description is shown in Table 5.1.

The authors of [173] did assume non-prioritzed Ethernet, we assign reasonable priorities for video and control data, which are in-line with the overall automotive application. Latency critical traffic such as control data (CTRL1,2,3) is mapped to the highest priority. Thus, it is guaranteed, that critical data is forwarded as soon as possible. All video and audio streams (all CAMs, FCAM, AV_Video and AV_Audio) were mapped to medium priority. BulkTraffic is mapped to the lowest priority, that way isolation is guaranteed.

Figure 5.12 shows the latencies of all streams (except best-effort Bulk). Latencies for Go-Back-N traffic is broken down in forward path (GBN) and backward path (ack) as indicated. We analyzed the topology using two different parameters for the send window $n_{sw} = 5$ and 10. For send-window sizes of smaller than 4, the analysis was not capable to find a fixed point and deemed the system unschedulable. Interestingly, the results are still promising with latencies of below 3 ms in all cases. Generally, a worst-case latency in the order of 10 ms is considered as low in the automotive domain and error control protocols seem very well applicable to hard real-time traffic.

5.6 Summary

In this chapter we have introduced high level error control protocols and how they detect and correct packet drops. The chapter focused on Stop and Wait as well as Go-Back-N in particular, as they are the most simple and well understood concepts. First, we dissected the protocols with respect to their timing behavior and constructed the worst-case behavior. Then we formally considered the error-free as well as the error case and derived the event models, response times and system latencies for aforesaid protocols. It seems possible to extend the presented approaches to other schemes such as Selective Repeat or Hybrid schemes.

Contrary to the considerations in previous chapters, we did not derive reliability figures using error models. In this work we focused on a conservative consideration of ARQ protocols - also under errors. Merging this work with the previously introduced error models remains to be addressed in future research. A challenge for this is that multiple error models are involved: one per link and one per device[2]. This “error-parallelism” needs to be captures in a reliability analysis.

By using our methodology, contrary to common belief, we could show that end-to-end error control protocols are generally adequate for hard

[2] In case the switches are also considered to be faulty.

real-time systems such as safety-critical high data-rate video applications, although the application of ARQ protocols leads to increased end-to-end latencies and higher worst-case transient load. Applied to the ASTEROID platform, this allows us to conservatively predict the latency of switched networks under errors be it on-chip or off-chip.

CHAPTER 6

Multiprocessor on Chip

To maintain a high degree of fault-tolerance, the ASTEROID platform executes a task redundantly on multiple cores until a common synchronization point (voting) is reached. The architectural novelty compared to simple lock-stepping is that only a subset of all tasks (only the safety-critical tasks) are replicated. This gives more flexibly and helps to reduce cost (cf. Chapter 2).

The redundancy protocol Romain is implemented by the research group of TU Dresden in the operating system Fiasco.OC [276]. The Romain master waits until all redundant copies have synchronized (or timed out), compares the intermediate result and, in case of no errors, continues the execution until the next voting point is reached [17]. In case of errors, either the correct state is copied over the erroneous (roll forward) or a recent checkpoint is restored.

Naturally, this has some interesting timing effects which are studied in this chapter: Since a task is simultaneously executed on multiple cores and is also subject to voting and checkpointing, previously established timing and reliability analysis approaches cannot be directly applied to predict the timing due to synchronization effects.

This chapter is divided into three sections: First, we establish a timing model of replicated execution. Then we evaluate the worst-case performance of fault-tolerant tasks running on ASTEROID. Finally, we establish an approach to accurately predict the mean time to failure (deadline miss) under the ASTEROID scheme. The approach and results presented in this chapter are based on [19, 20]. Geoffrey Nelissen[1] kindly pointed out two problems with the approach presented in [19] which leads to optimism

[1] grrpn@isep.ipp.pt

in the response time analysis. We address these shortcomings and provide a simplified, revised approach which resolves these problems. All experiments in this chapter were performed using the revised approach.

6.1 Error Detection and Recovery Model

The MPSoC executes an application consisting of replicated tasks as well as unprotected tasks. This reflects the requirement that some functions are of higher criticality than others. In ASTEROID, processors communicate over a local interconnect which is capable to detect and correct errors on its own. Here the methods and approaches of Chapter 4 can be used.

Tasks are managed by an operating system which uses a fixed-priority scheduler and are statically mapped to individual cores. In most cases this is a valid assumption, since partitioned scheduling schemes are widely used (e.g. AUTOSAR [10]) and well understood [47].

Additional communication overhead such as NoC overhead, cache coherency traffic or shared resource accesses is not explicitly modelled. We assume that any overhead is accounted in the execution times or application graph as described below.

6.1.1 Fault-Tolerant Tasks

Our system consists of non-fault tolerant (non replicated) task as well as fault-tolerant (replicated) tasks. A non-fault tolerant task τ behaves like an independent task as discussed in Chapter 3. Once activated, it occupies the processor for some time between its best-case and worst-case execution time, furthermore it is mapped to a single core and has a unique priority.

Fault-tolerant (ft) tasks are extensions of regular tasks which are replicated among several cores to increase reliability.

Definition 41 (Fault-Tolerant Task)**.**
A fault tolerant task is an independent task with an annotated best-case / worst-case execution time C^+/C^+, which is executed redundantly (in space or time) and subject to voting.

We assume that in regular, predefined intervals, the application triggers the creation of a checkpoint. The checkpoint includes all recently changed memory regions of the task's address space, as well as the current register contents. When a checkpoint is established, the processor writes the relevant memory content to a fault-tolerant memory region. This can be protected main memory (e.g. by using an ECC) as well as dedicated checkpoint memory, e.g. as used in the SafetyNet [256] approach. Furthermore, we assume that the creation of a checkpoint is an atomic transaction and

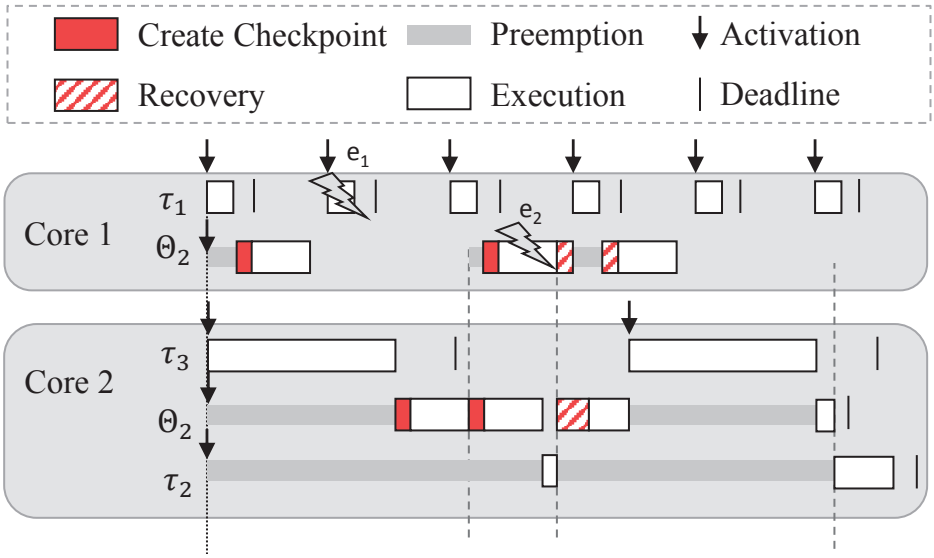


Figure 6.1: Gantt diagram showing non-replicated tasks τ_1, τ_3, τ_4 as well as replicated task Θ_4.

the hardware circuitry (e.g. DMA controller) takes care, that the checkpointing process itself is fault tolerant. That means, if checkpoint creation itself is affected by errors it will simply restart until it has completed the process. Once an error in the program state is detected, the most recent checkpoint is restored, and task execution is resumed from this point. Generally, the imposed assumptions are compatible with most of the checkpoint and rollback approaches which are summarized in [265].

For the error detection, we assume a voting process as introduced in Chapter 2 which is based on a fast fingerprinting hardware support. An example of the voting and checkpointing sequence is shown in Figure 6.1. Here fault-tolerant task Θ_2 consists of two checkpoints. First, a checkpoint is created (red box), then computations are performed (white box). The state comparisons are performed at intermediate points in computation (dashed vertical lines). An error hits a non fault tolerant task τ_1 at time e_1. This directly leads to a failure associated to the function of that task. An error event e_2 hitting a replicated task Θ_2 leads to a state corruption which is detected at the next state comparison (second dashed line). A recovery process is initiated and the execution is resumed from the last checkpoint. In this scenario all deadlines (vertical bars) are met.

To model the overhead of checkpointing and recovery, a fault-tolerant task is associated with further parameters. Checkpointing and redundancy imposes additional execution time overhead described by parameters n, t_{cov} and t_{rov}. For a ft-task, the execution time C is divided into n checkpoint stages of arbitrary length $t_{cp,i}$ so that $C = \sum t_{cp,i}$. At the beginning of each execution interval (stage), a checkpoint is established causing a *creation*

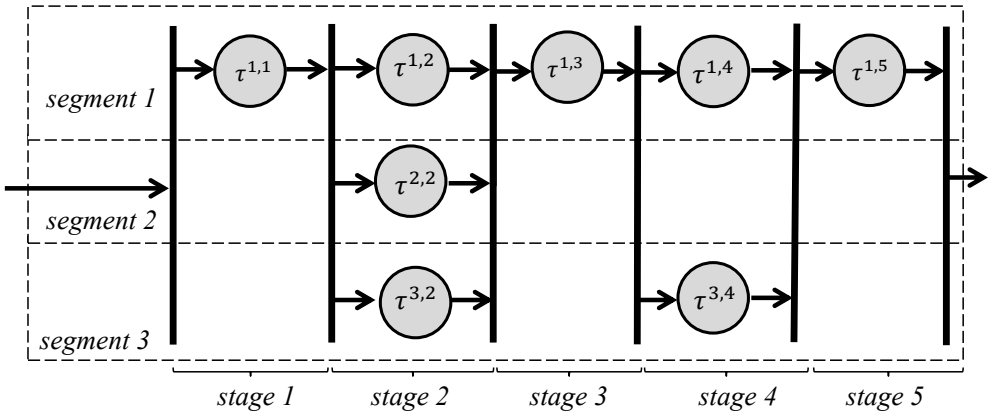


Figure 6.2: A generic fork-join model as used in OpenMP or Romain replication framework [75] as used in ASTEROID. A fork-join task can be subdivided into vertical stages and horizontal segments.

overhead of t_{cov}. At the end of each stage, fingerprints of all redundant execution streams are compared. In case of an error, all replicas re-execute the recent stage with an additional *recovery overhead* of t_{rov}.

We assume that the time required for detection and recovery for interconnect errors is negligible short. Previously presented results for a priority-based arbitration scheme confirm such assumptions. Also, to keep the presented analysis clear, we assume that there is no additional communication overhead which needs to be considered, instead all overhead is attributed implicitly to the tasks execution time. The operating system and the processor hardware (MMU) take care that an erroneous task does not interfere with other tasks in the system. In particular, this implies that a faulty task cannot write to memory regions not assigned to this task. However, timing interference is possible when an erroneous task keeps demanding processing time.

6.1.2 Fork-Join Task Model

The rest of this chapter abstracts from the fault-tolerant task and maps the problem into the domain of DAG task graphs [42]. Any fault-tolerant task using replication can be modeled by a directed acyclic graph as shown in Figure 6.2. A fork-join task Θ is an extension of an independent task and consists of multiple stages with further data dependency.

Definition 42 (Fork-Join Task)**.**
A fork-join task Θ is a directed acyclic graph with a set of independent tasks as nodes and edges between these tasks which represent the precedence relations.

In any fork-join graph, we can identify *segments* and *stages* as annotated in Figure 6.2. The stages reflect the checkpoint intervals whereas the segments model the redundancy in space.

In that sense, a segment can only be started once all segments released in the previous stage have finished their execution. Each segment is modeled by an independent task, thus it has a worst-case execution time, priority and unique mapping. Nested forks in which only some segments have common synchronization points are not modelled (and not supported).

The execution semantics is similar to the ones of independent tasks: We assume an infinite large fifo queue in front of the fork-join task. Once an event arrives at this queue, the first stage of the fork-join task is released and the sub-tasks are spawned. Once all segments of the first stage have fully executed, the following stage is triggered. The event is processed if the last stage has fully executed (and all segments have finished). If further events arrive at the fork-join task during that time, they are queued at the fifo. Thus, at any time the fork-join task is only processing one event. We can conclude that the considered fork-join DAG is not pipelined, opposed to the generic task graph introduced in the SymTA/S approach [120]. For the sake of simplicity, we do not consider shared resources such as semaphors (e.g. Multi-processor Priority Ceiling Protocol [225]). However, as our analysis follows the compositional CPA approach, related approaches in the field of shared resources can easily be integrated into the presented formalism.

Now, it is obvious that any fault tolerant task can be represented by an associated fork-join task. The segments represent the independent executing replicas, whereas the "join" semantics models the voting mechanism. A DMR-based replication is modeled using two segments, a TMR approach features three segments. The number of stages depends on the number of checkpoints n.

To reference individual nodes in the fork-join DAG, we introduce the following shorthand notations. A fork-join task Θ is represented by a set of regular tasks which we call subtasks.

Definition 43 (Fork-Join Subtask)**.**
A subtask $\tau^{\sigma,s}$ of a fork-join task Θ is the σ-th segment in the s-th stage, which has no further data dependency.

Subtasks are shown in Figure 6.2 as circles. Similarly, a fork-join task is parametrized by a set of execution times and priorities $C^{\sigma,s}, p^{\sigma,s}$ one for each subtask $\tau^{\sigma,s}$. The execution time depends on the number of checkpoints n and the total execution time C of the associated fault tolerant task. Hence, for n checkpoints, there are n stages.

Corollary 4. *Assuming, a task is split into n checkpoints of the equal length (C/n). Under DMR/TMR, there are two/three segments and we get the following subtask execution times.*

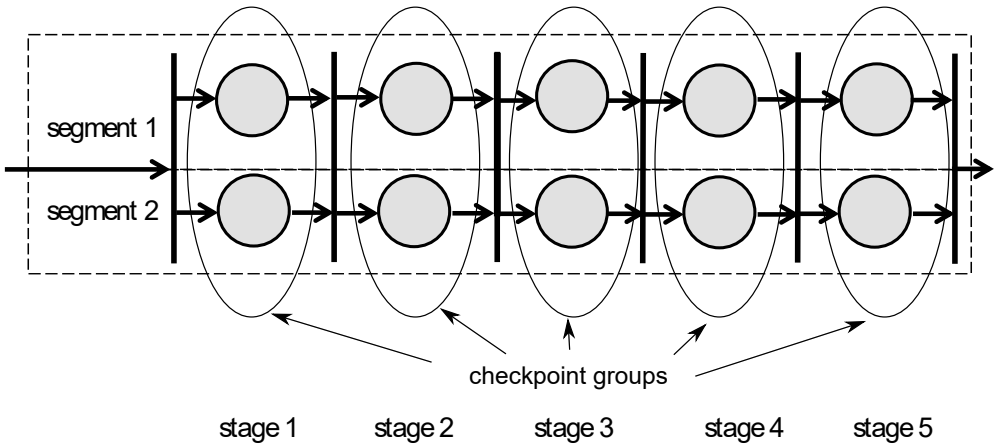


Figure 6.3: Fork-join model for a fault-tolerant task. Checkpoint groups are indicated.

$$\forall s \leq n, \sigma \leq 2 \ : \ C^{\sigma,s} = \frac{C}{n} + t_{cov} \tag{6.1}$$

Proof. By construction, each subtask executes C/n-th of the workload plus the additional checkpoint creation time. □

For a fault tolerant task Θ_i, we can group all segments in a given stage s in a set.

Definition 44 (Checkpoint Group)**.**
A checkpoint group is the set of fork-join subtasks which belong to the s-th stage of Θ_i. $CG_{i,j,s}$ is the j-th instance of a checkpoint group associated with the s-th stage of Θ_i.

Thus,a checkpoint group contains all segments which (in an error-free environment) computes the same result and leads to a comparison. An example of a fork-join model which reflects a fault tolerant task is given in Figure 6.3. Also the checkpoint groups are indicated.

Without loss of generality, we assume that all subtasks in one segment are mapped to the same core: This means that tasks in the same row in Figure 6.2 are mapped to one core. This is the case for redundant execution, as the segments in the fork-join task graph represent the replicas. Thus, the segment (σ) encodes implicitly the mapping for fork-join tasks, and we can say $\tau^{\sigma,s}$ is mapped to core σ. In that sense σ can be a segment as well as a core.

6.1.3 Failure Modes and Error Handling

An error can affect a task in a number of ways, eventually causing a task to stop providing service. Unprotected tasks are assumed to fail on the first error, which is a conservative assumption based on the possible failure modes of the processor. For tasks which are replicated, error events can occur during the following execution phases:

1. checkpoint creation
2. regular execution
3. recovery process

For the first case we assume that an error during checkpoint creation is detected and corrected on-the-fly by Romain. For the second case the error is detected after all segments have finished. It might happen that due to errors a task gets stuck in a loop and will not yield processing-time properly. Romain implements a budgeting (timeout) mechanism which enforces maximal execution times to detect programs which got stuck. A similar kind of budgeting is also used in PharOS [55]. Then the recovery process is triggered with the result that the recent checkpoint group is re-executed. The third scenario is treated like the second, assuming that an error during the recovery process does not corrupt the recent checkpoint. By these means it is possible to detect errors at the end of each task to avoid error propagation in the system (domino effect). Errors do not have an effect on tasks when a core is idling.

Figure 6.4 shows an example Gantt chart as a possible execution trace of a system with two cores and tasks τ_1 - τ_4 as well as Θ_2. Here priorities are assigned according to the occurrence in the chart, i.e. τ_1 has a higher priority than Θ_2 and so on. In this example Θ_2 is a fault-tolerant DMR task with two equally distributed checkpoints ($n = 2$). Checkpoint groups are indicated by $CG_{2,1,1}, \ldots, CG_{2,2,2}$. Important events have been numbered in the order of their occurrence: ❶ All tasks activate simultaneously. Both replicas of Θ_2 start creating a checkpoint. ❷ An error event e_1 affects task τ_1, causing this task to fail. Further activations of τ_1 may still dissipate processor time but have no further value for the service delivered by τ_1. ❸ The intermediate results of all segments in $CG_{2,1,1}$ are available and match. Then, a checkpoint is established. ❹ An error event e_2 hits Θ_2. A segment in $CG_{2,1,2}$ is affected. ❺ After the execution of segments in $CG_{2,1,2}$, the voting mechanism detects a mismatch and a recovery is initiated and a reexecution follows. This is modelled by an additional fork-join stage indicated by $CG'_{2,1,2}$. ❻ The voting after the reexecution $CG'_{2,1,2}$ agrees, no further error is detected. First activation of Θ_2 successfully

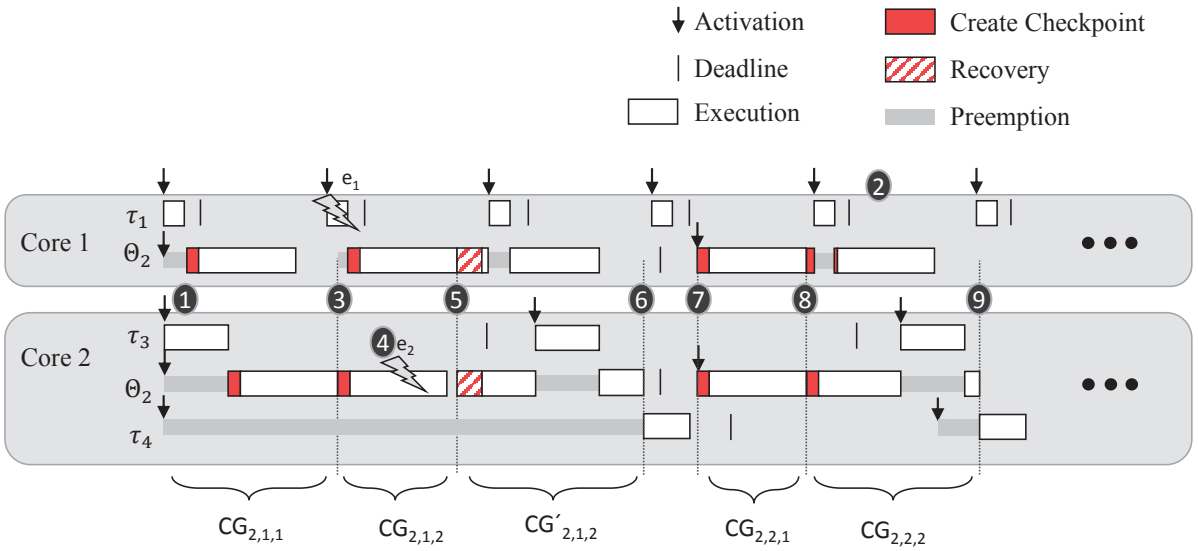


Figure 6.4: Illustrative example: Task Θ_2 is mapped redundantly to both cores and split in $n = 2$ checkpoints.

completed and deadline met. ❼ A second activation of Θ_2 arrives, checkpoint is established. ❽ Intermediate result in $CG_{2,2,1}$ is available and the voting data matches, a checkpoint is established. ❾ Intermediate result in $CG_{2,2,2}$ is available and match. The second activation of Θ_2 successfully completed and deadline met.

In this example, we can see that the effect of an error (recovery and reexecution) can be modelled by adding an additional stage. For a single error, it is a conservative assumption that the stage with the largest execution time is reexecuted. Thus, a fault-tolerant task representing this behavior has $n+1$ stages, n from the original fork-join task graph plus an additional stage which models one reexecution. This idea can be generalized to k errors:

Corollary 5. *A fault-tolerant task which is affected by k errors during its execution is conservatively approximated by a fork-join task graph with a total of $n + k$ stages.*

Proof. A fault-tolerant task in an error-free environment is modeled by n stages, where n is the number of comparisons (votings). Under worst-case situations, an error hits the stage with the largest execution time, which leads to an additional comparison (voting), recovery and reexecution. Again, in the worst-case scenario, this reexecution is affected by another error leading to another recovery/reexecution.

In the worst case, each of the k errors hits the worst-case segment. This leads to a total of $n + k$ stages, with n being the original stages and the

addional k stages are repetitions of the stage with the largest execution time. □

Corollary 6. *If all checkpoints are equally distributed, the execution time for a subtask which models error recovery and execution time is given by:*

$$\forall n > s \geq n + k, \sigma \leq 2 \ : \ C^{\sigma,s} = \frac{C}{n} + t_{rov} \tag{6.2}$$

Proof. Each recovery subtask consists of the actual recovery time t_{rov} plus the reexecution time C/n. □

6.2 Performance of Fork-Join Tasks

This section discusses the timing implications of redundant multicore execution in which the workload associated with the application is distributed on multiple isolated cores and executed in parallel. In particular, we derive a worst-case bound on the response time for fork-join tasks which are used to model replicated execution.

The presented methodology is not only restricted to redundant execution of reliable applications, but can be also applied to predict the timing of high performance, hard real-time applications which use a state of the art, parallel computing paradigm.

6.2.1 Related Work

Task-parallel programming models facilitate splitting application logic into sequential and parallel parts. Toolkits, such as OpenMP [205] and Intel's Thread Building Blocks [133] support the programmer by automating most of the parallelization and synchronization work. Depending on the scheme, this leads to an unpredictable system, or makes a real-time analysis challenging.

Parallel task models that support such semantics have been presented for instance in [163, 218, 34]. Early work in distributed systems in which events are synchronized was presented in [108]. The fork-join tasks considered in this work are a special case of directed acyclic graph (DAG) task models for instance [96, 30, 42].

Holenderski et al. [124] addressed multi-resource scheduling in which parallel tasks can access local and global resources which can be preemptible as well as non-preemptible. In that scope a generalized shared resource protocol (Parallel-SRP) was presented. Baruah et al. presented a generalized parallel task model [30] which also supports fork-join tasks. They analyze the schedulability under EDF and concludes that EDF has a speedup bound of 2. Fork-join task models in particular were considered

in [239], the presented model is compatible with our work. The authors decomposed fork-join task constraints into a set of sequential deadline constraints under implicit deadline assumption. Based on this, schedulability bounds for global EDF scheduling were given. Lakshmanan et al. introduced a task stretch transformation in [163] where parallel workload is converted into sequential workload which effectively avoids fork-join structures where possible. The work was extended in [87, 218] (segment stretch transformation). The resource augmentation bound for SST and TST was shown to be 3.42. This implies that any taskset, feasible on m unit speed processors is also schedulable by SST / TST on m processors that are 3.42 faster.

There exist a large variety of scheduling and mapping techniques to handle fork-join tasks such as [192]. Here deadlines were assigned to a fork-join task and an EDF-like scheduler was used to schedule all subtasks in a fork-join task independently. In [94] a heuristic algorithm (Fisher-Baruah-Baker First-Fit-Dreceasing) was given to map tasks to cores under a partitioned multiprocessor scheme. Fork-join decomposition and priority assignment were put together in [92] in scope of RT-OpenMP.

Most of related work in the field of parallel task graphs only considers global EDF or variations thereof under implicit deadline assumptions (i.e. [58, 164, 30, 239, 92]). In this work we show how to predict the response-time of such applications under a fixed-priority scheduling scheme conservatively.

This work is based on [19] which discussed a response-time analysis methodology for fork-join tasks. However Geoffrey Nelissen pointed out two shortcomings that lead to optimism.

The first problem is related to blocking. A fork-join task behaves in a way quite similar to two independent tasks mapped to an MPSoC which block on a single shared resource. For shared resource, the challenge, model and analysis is discussed in [191]. The same blocking effect can delay the execution of the fork-join tasks on one processor and artificially causes an increased input jitter.

The second problem is that the work in [19] assumes a greedy approach in which interference is assumed as early as possible. This does not always lead to the worst-case as we will show in a later example. The work presented in this section still uses the concepts presented in [19] and revises the approach.

6.2.2 Response-Time of Independent Tasks Under the Presence of Fork-Join Tasks

Through the course of this section, we strive to obtain the fork-join response time.

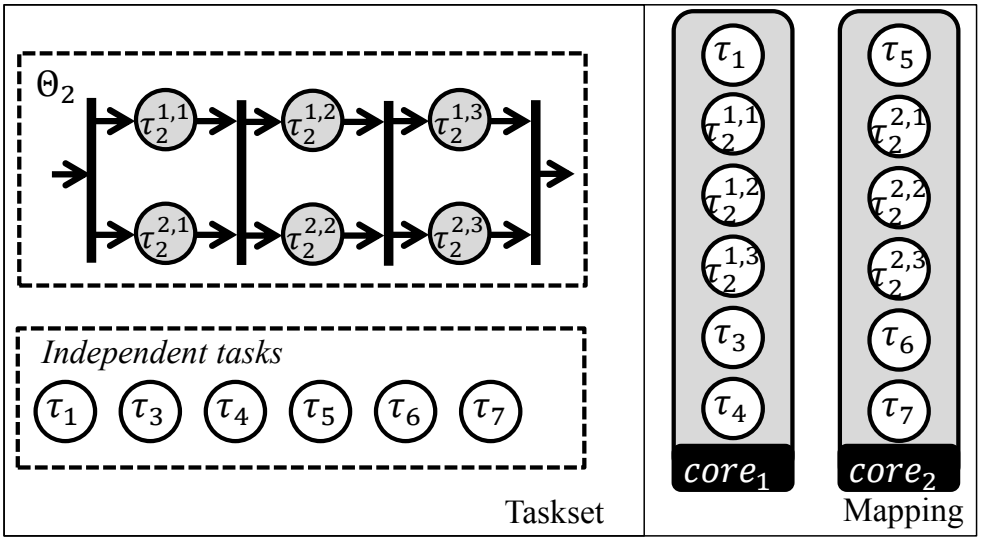


Figure 6.5: Example taskset and mapping as used for illustrative purposes. Two cores, one fork-join task Θ_2 and some individual tasks.

Definition 45 (Fork-Join Response Time)**.**
The response time R^+ of an event of fork-join task Θ is the time interval defined by the time when the event arrives at a fork-join task until it exits the last stage.

The worst-case (fork-join) response time R^+ is an upper bound to any response time which can be observed. The goal of the following analysis steps is twofold: First, we determine the response time of independent tasks under the interference of fork-join tasks. Then we show how to obtain the worst-case response time bounds for fork-join tasks.

We now derive the worst-case response time for any independent task τ_i which is not part of a fork-join task. Contrary to systems which solely consist of independent tasks, an independent task τ_i can be preempted not only by independent tasks but also by fork-join tasks Θ_j. The following analysis includes these timing effects. To illustrate the effects caused by a fork-join task on an individual task we use the example taskset as shown in Figure 6.5. In the example we have six independent tasks as well as one fork-join task mapped to two cores (Core 1, Core 2). The fork-join task consists of three stages and two segments. We now demonstrate how to obtain the response time of task τ_4 running on Core 1. We choose this task as an example, since it has a lower priority than all subtasks of Θ_2 as well as τ_1.

According to Theorem 1, to compute the worst-case response-time it is sufficient to bound the multiple event processing time as well as the scheduling horizon (or busy-period).

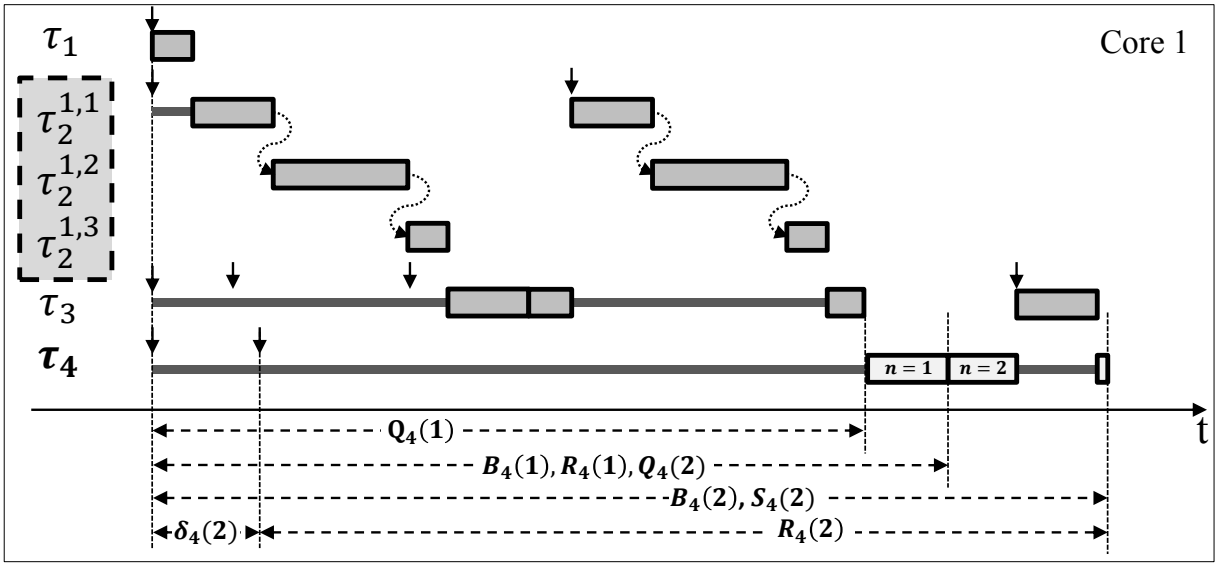


Figure 6.6: Worst-case schedule for independent task τ_4 running on core 1. Queuing delays as well as multiple-event busy times are indicated for the first two events.

Corollary 7. *The multiple event processing time $B_i^+(q)$ of an independent task τ_i under partitioned, fixed-priority preemptive scheduling is given by*

$$B_i^+(q) = q \cdot C_i + I_{i,IND}(B_i^+(q)) + I_{i,FJ}(B_i^+(q)) \tag{6.3}$$

where

- *C_i is the worst-case execution time of task τ_i,*
- *$I_{i,IND}(\Delta t)$ is an upper bound for workload caused by higher priority independent tasks in any time window of length Δt,*
- *$I_{i,FJ}(\Delta t)$ denotes the interference caused by fork-join (sub-) tasks of higher priority mapped to the same core.*

Proof. To execute q instances of an independent task τ_i, the workload of the task itself $q \cdot C_i$ must be executed as well as all higher priority workload released during the processing time. The higher priority load consist of other independent tasks as well as fork-join tasks. □

Similarly, we retrieve the queuing delay for the q-th event as follows.

Corollary 8. *The multiple event queuing delay $Q_i(q)$ of an independent task τ_i under partitioned, fixed-priority preemptive scheduling is obtained by*

$$Q_i(q) = (q-1) \cdot C_i + I_{i,IND}(Q_i(q)) + I_{i,FJ}(Q_i(q)) \tag{6.4}$$

where

- *C_i is the worst-case execution time of task τ_i,*
- *$I_{i,IND}(\Delta t)$ is an upper bound for workload caused by higher priority independent tasks in any time window of length Δt,*
- *$I_{i,FJ}(\Delta t)$ denotes the interference caused by fork-join (sub-) tasks of higher priority mapped to the same core.*

Proof. Analogous to the one of Corollary 7. □

Corollary 9. *The scheduling horizon for an independent task τ_i under partitioned, fixed-priority preemptive scheduling is bounded by the multiple event processing time $B_i^+(q)$.*

$$H_i(q) = B_i^+(q) \tag{6.5}$$

Proof. This follows from the SPP scheduling concepts discussed in Chapter 3. For the proof, we use the queuing delay. By definition the queuing delay is the time at which the q-th event gets at least a full processor cycle service. The scheduling horizon of q events is by definition the time after which the $q+1$ event gets ϵ service. Substituting q by $q+1$ in the queuing delay equation Equation 6.4 yields an upper bound for the scheduling horizon. This substitution leads directly to Equation 6.3, the multiple event processing time. □

The higher priority interference of independent tasks can be classically computed [172] by considering all tasks that are of higher or equal priority (denoted by hp_{ind}).

$$I_{i,IND(\Delta t)} = \sum_{\forall \tau_j \in hp_{ind}(i)} \eta_j(\Delta t) \cdot C_j \tag{6.6}$$

To compute the interference caused by fork-join tasks, it is necessary to derive the event model at the input of subtasks.

Definition 46 (Fork-Join Sub Event Model)**.**
A sub event model ($\eta_i^{\sigma,s}$ or $\delta_i^{\sigma,s}$) of a fork-join task Θ_i describes the worst-case event arrival at the corresponding subtask $\tau_i^{\sigma,s}$.

Theorem 46. *The number of events that arrive in some time interval of length Δt at the input of a subtask $\tau_i^{\sigma,s}$ can be conservatively approximated by the input event model η_i of the corresponding fork-join task Θ_i*

$$\eta_i^{\sigma,s}(\Delta t) \leq \eta_i(\Delta t + R_i^+) \tag{6.7}$$

Proof. Under worst-case assumptions, the fork-join task is blocked on some other core[2]. Thus, events queue up at the FIFO. In the worst-case, the fork-join task is blocked for R_i^+, its response-time. Thus, at the start of the busy-period, we must account for additional events which arrive during time R_i^+. This is analogous to the response-time computation in multiprocessors under the impact of shared resources. Given a multicore system and two tasks (τ_i, τ_j) mapped to different cores which access the same shared resource. Under worst-case assumptions task τ_i acquires the shared resource and blocks, thus delaying, the execution of the other task τ_j. The task starts executing latest when τ_i releases the shared resource after its response time. The effect is discussed and a formal analysis is presented in [240, 191].

Furthermore it is conservative to assume that all stages are activated simultaneously, hence it is conservative to use the event model of the first stage for all stages. □

This is intuitively shown in Figure 6.6, for each event of Θ_2 each subtask is activated once in a cascading fashion. Note that the effect of inter-core blocking is not explicitly depicted in the figure. The event arrivals which are shown are already shifted by R_i^+. The interested reader should consult [240].

From the previous reasoning we can conclude that the interference by higher priority fork-join tasks is given by

$$I_{i,FJ}(\Delta t) = \sum_{\forall \tau_j^{\sigma,s} \in hp_{fj}(i)} \eta_j(\Delta t + R_i^+) \cdot C_j^{\sigma,s} \tag{6.8}$$

Here, hp_{fj} is the set of all higher priority fork-join subtasks which are mapped to the same core as the task under analysis, τ_i.

6.2.3 Response-Time of Fork-Join Tasks

Similar to the previously presented analysis of independent tasks, we also use the busy-window approach to derive the response time for fork-join tasks. However, previously introduced formulas cannot be applied directly to fork-join constructs. Mainly, because the behavior of the fork-join task depends on a complex interaction between multiple cores. That is, some segments in one stage finish earlier than others, inducing a "waiting time" in which one or more cores are potentially idle, waiting to finish a segment on other cores. Before we go into detail, the following analysis does not support event-pipelining in which a fork-join task is processing

[2]This effect was not sufficiently considered in [19].

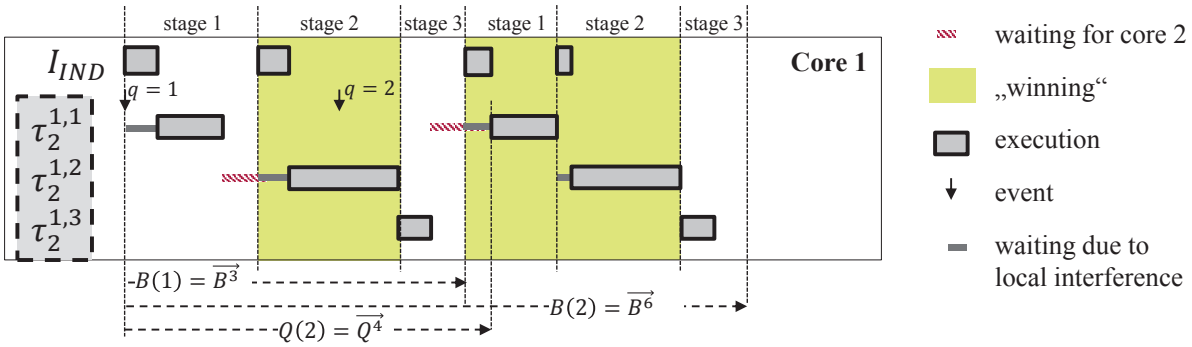


Figure 6.7: Effect of multiple events (i.e. q=2). Each event cascades through all three stages. Effects of waiting induced by core 2 is indicated as a red bar (but core 2 is not explicitly shown). Higher priority interference is denoted as I_{IND}. Queuing delay as well as the multiple event busy times are shown for the first two events. Note that $B(1)$ as well as $B(2)$ align with the end of the stage, since the completion times are delayed by core 2 (red bar).

multiple events in a pipeline fashion and multiple stages are executing simultaneously. We restrict ourselves to the redundancy case in which the fork-join task chain must have fully been executed until the following event is admitted.

Before we go through the analysis, we highlight the relation between events and stages. An example is shown in Figure 6.7. The Gantt diagram shows only the behavior on core 1 and core 2 is not shown. I_{IND} denotes the higher priority interference, abstracting from all other tasks mapped to the core. Hatched (red) bars denote where core 1 has to wait for core 2 to finish the previous stage, although the activity on core 2 is not explicitly shown in this figure.

The first event $q = 1$ as indicated by the arrow, arrives right at the start of the busy period. Obviously, this event ripples through all three stages. The second event ($q = 2$) cascades through all three stages, thus the first two events together execute a chain of six stages in total. We can conclude, that the behavior of two events is equivalent to the behavior of one event which triggers a fork-join tasks consisting of six stages. Without loss of generality, we model the behavior of multiple events by repeating the sequence of stages assuming the initial graph consists of s_{max} stages. To model the execution of q successive events, an equivalent task graph with $s_{max} \cdot q$ stages is used. For further notation, we extend the subtask definition to stages beyond s_{max} the following fashion:

$$\tau^{\sigma,s} \equiv \tau^{\sigma,(s \bmod s_{max})} \tag{6.9}$$

Here, the $\equiv$ operator refers to all task parameters such as priority and execution time.

Now, similar to the multiple event busy time for independent tasks, we can define a stage-completion time for fork-join tasks.

Definition 47 (Stage-Completion Time)**.**
The stage-completion time $\overrightarrow{B}^s$ of fork-join task Θ is the largest time interval from the start of the busy-period until all segments in the s-th stage have executed fully.

Similarly, we can define the window from the start of a stage to the end of that stage by using the completion times.

Definition 48 (Stage-Completion Window)**.**
The stage-completion window $\overleftrightarrow{B}^s$ is defined as the time interval from the start of the s-th stage until all segments of the s-th stage are fully executed.

The stage-completion time can be used to formulate the multiple event processing time by evaluating $q \cdot s_{max}$ stages.

$$B(q) = \overrightarrow{B}^{q \cdot s_{max}} \tag{6.10}$$

In every stage, each processor adds to the total window. However, the total interference naturally depends on the interferer set of the processors and it is not immediately clear which processor delays the stage of the fork-join task most. Hence all cases must be considered and can be potential candidates.

Definition 49 (Stage-Completion Window Candidate)**.**
The stage-completion window candidate $\overleftrightarrow{B}_i^{\sigma,s}$ is the time interval from the start of the s-th stage of fork-join task Θ_i until the stage is completed on core σ.

At this point we can also formalize the concept of maximizing a stage. A core maximizes a stage if the stage-completion window candidate is the largest among all others.

Theorem 47. *The stage-completion time can be computed from the completion time of the previous stage plus the largest stage-completion window candidate of the s-th stage, by computing a candidate for each core and choosing the largest.*

$$\overrightarrow{B}_i^s = \overrightarrow{B}_i^{s-1} + \max_{\forall \sigma} \left(\overleftrightarrow{B}_i^{\sigma,s} \right) \tag{6.11}$$

Proof. The proof is by induction. The first stage-completion time is conservative by construction as it is analogous to single core scheduling theory. For the subsequent stages a conservative stage-completion window is added which is *independent* of the previous stages. Thus, all stages are maximized in isolation and then added together. □

Theorem 48. *The stage-completion window candidate* $\overleftrightarrow{B}_i^{\sigma,s}$ *of fork-join task* Θ_i *can be computed using the following recurrence relation:*

$$\overleftrightarrow{B}_i^{\sigma,s} = C_i^{\sigma,s} + I_{i,IND}^{\sigma,s}(\overleftrightarrow{B}_i^{\sigma,s}) + I_{i,FJ}^{\sigma,s}(\overleftrightarrow{B}_i^{\sigma,s}) \tag{6.12}$$

Where $I_{i,IND}^{\sigma,s}$ *is an upper bound on the interference caused by higher priority interference of independent tasks and* $I_{i,FJ}^{\sigma,s}$ *bounds the higher priority interference of other fork-join tasks*[3].

$$I_{i,IND}^{\sigma,s}(\Delta t) = \sum_{\forall \tau_j \in hp_{ind}(i)} \eta_j^+(\Delta t) \cdot C_j \tag{6.13}$$

$$I_{i,FJ}^{\sigma,s}(\Delta t) = \sum_{\forall \tau_j^{\sigma,s} \in hp_{fj}(i)} \eta_j^{\sigma,s}(\Delta t + R_j^+) \cdot C_j^{\sigma,s} \tag{6.14}$$

Proof. Naturally, the subtask of execution time $C^{\sigma,s}$ must be executed plus all higher priority interference released during $\overleftrightarrow{B}_i^{\sigma,s}$. The interference of higher priority tasks (fork-join and independent) is maximized as the number of events in the s-th stage is maximized by the stage interference event model. For fork-join tasks, we further must consider the inter-core blocking as already presented in the previous section. According to [172], it is assumed that each stage starts with a critical instant regardless of what happend in previous stages. □

Using the established formulas we can compute the multiple event processing time. From the multiple event processing time we can derive the response time as shown in Chapter 3. However, we have not discussed the scheduling horizon which is also required in order to decide how many events need to be considered under worst-case conditions. For this, we use the queuing delay as a conservative approximation for the scheduling horizon:

Theorem 49. *The worst-case queuing delay* $Q_i(q+1)$ *is a safe upper bound for the worst-case scheduling horizon* $H_i(q)$*:*

$$H_i(q) \leq Q_i(q+1) \tag{6.15}$$

[3]Here is the difference between the work presented in [19] which used an optimized Stage Interference Event Model which causes non-conservatism.

Proof. The worst-case scheduling horizon is by definition the largest time interval until a $q+1$-th event gets at least ϵ service. The worst-case scheduling horizon is by definition the largest time interval until a q-th event gets at least a full processor cycle t_{cycle}. Therefore a $q+1$-th event also gets at least ϵ service after the queuing delay. □

Definition 50 (Stage Queuing Time)**.**
The stage queueing time $\overrightarrow{Q}_i^s$ is the largest time interval from the start of the busy window until all segments of fork-join task Θ_i get at least a full processor cycle service in the s-th stage.

Definition 51 (Stage Queuing Window Candidate)**.**
The stage queuing window candidate $\overleftrightarrow{Q}_i^{\sigma,s}$ is the largest time interval in which subtask $\tau_i^{\sigma,s}$ is blocked in the s-th stage by higher priority task interference until it gets a full processor cycle service.

To evaluate the queuing delay of the q-th event, we must check the largest stage queueing time of the first segment of that event. Analogous to the completion formulas we can derive the stage queuing formulas:

$$Q_i(q) = \overrightarrow{Q}_i^{s_{max}\cdot(q-1)+1} \tag{6.16}$$

$$\overrightarrow{Q}_i^s = \overrightarrow{B}_i^{s-1} + \max_{\forall\sigma}\left(\overleftrightarrow{Q}_i^{\sigma,s}\right) \tag{6.17}$$

$$\overleftrightarrow{Q}_i^{\sigma,s} = I_{i,IND}^{\sigma,s}(t_{cycle} + \overleftrightarrow{Q}_i^{\sigma,s}) + I_{i,FJ}^{\sigma,s}(t_{cycle} + \overleftrightarrow{Q}_i^{\sigma,s}) \tag{6.18}$$

6.2.4 Worst-Case Timing Evaluation of Replication

Here we evaluated the *Romain* framework with respect to worst-case timing. Therefore, we map the Romain replication to a fork/join execution model: One replica acting as the master spawns the other replicas. These replicas execute user code concurrently. Once they reach a point where they externalize state, concurrent execution is interrupted and one of the replicas executes master code in sequential execution mode. After handling the event in master mode, control is returned to concurrent replica execution. In a DMR setup, there are two fork-join segments, whereas a TMR setup yields three parallel segments. The number of stages depends on the number of comparisons (voting) and is application dependent. Applications which perform many I/O operations have more stages than performance bound applications.

We obtained the execution characteristics from a set of benchmarks from the MiBench benchmark suite [107]. These benchmarks were executed using the *Romain* framework. To obtain reasonable timing models, we executed the benchmarks with one, two, and three replicas respectively

on a state-of-the-art Intel Core i7 processor with 2.6 GHz [4]. We measured their execution times and specifically observed the time spent executing sequentially (state comparison, system call handling etc.), as well as the time spent executing concurrently (executing user code) in order to derive the execution times $C^{\sigma,s}$ as used in the analysis. Thus, first the number of stages is counted and the execution time spent in each stage is traced. The number of parallel segments is fixed to 2/3 due to the dual/triple modular redundancy use case.

In total we ran each benchmark 150 times to get a sufficiently large sample size. Then we removed the first samples which show major transient anomalies caused by memory layout organization. These where mostly page faults because the benchmarks did not lock pages, nor mark them as sticky in advance. These page faults only appear once at the beginning of the execution trace and vanish in the steady state as the memory is fully mapped. Also embedded processor without a MMU will not show such a behavior since the memory layout is fixed at design time. Since the benchmarks are executed on a live system, we see interference from interrupt handlers (i.e. timer) from time to time which tamper with the execution-time measurements. To filter these effects, we choose the execution times as the 0.9-quantile over the used samples per stage, assuming that rare outliers are caused by IRQ handlers.

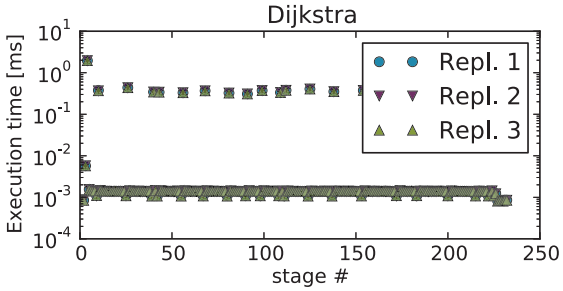
We found that in most cases state comparison is in the order of a few μ seconds or less and compared to the computation-heavy segments of the benchmarks it is negligibly small. Additionally, the system calls themselves may block due to hard drive access and other hardware interaction which is not an inherent part of the benchmark unless the operating system and hardware performance shall be evaluated which is not the case. Thus, for the following experiments, we deliberately excluded time spent in system calls and only focus on usercode which is the intrinsic part of the benchmark.

Execution Time Measurements

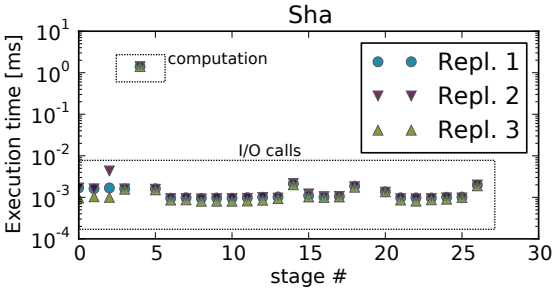
An overview of the benchmark data such as the total execution time and the number of stages per benchmark can be found in Table 6.1. We see, that the number of stages differs drastically from benchmark to benchmark ranging from 8 to 358 stages. This is mostly due to different I/O patterns of the benchmarks.

In most cases I/O is caused by printfs used to print (intermediate) results. Figure 6.8b and 6.8a show representative execution time behavior and other benchmarks show a very similar pattern. The diagram shows the stage

[4] Romain is only available for x86 architectures



(a) Execution times of segments in Dijkstra benchmark. I/O phases and computation bound phases alternate.



(b) Execution times of segments in SHA benchmark. Most stages prepare I/O (e.g. printf).

Figure 6.8: Execution characteristics

number on one axis and the execution time on the other. As we traced replicas independently the execution times is shown per replica.

Worst-Case Evaluation of Romain Configurations

As a replica resembles a segment in a fork-join task, we are able to model the Romain Framework as well as the benchmarks using the previously introduced fork-join task model. For the following experiment we use a dual core with the same clock frequency as the architecture used for the benchmarking (2.6 GHz).

In these experiments we evaluate the worst-case response time of the Bitcount and Rijndael (AES) benchmarks in a single core and dual core setup. Therefore, we mapped 20 independent tasks as well as one fork-join application on both cores and varied the utilization in order see the effects on the response time. For the task parameters of the 20 independent tasks we use UUniFast algorithm [39].

We compare the performance of the parallel setup (dual core) with a purely sequential execution (single core). Note that we do not explicitly

Table 6.1: Number of stages per Benchmark

Benchmark	Stages	Total C [ms]
Security/Rijndael	14	4.9
Security/SHA	27	1.53
Automotive/Bitcount	8	271.2
Automotive/QSort	358	18.35
Networking/Dijkstra	233	9.05

model core to core communication, memory effects. For this experiment, we assume that a task has the same worst-case execution time on one core or the other. In the sequential setup all interference tasks and all fork-join segments are mapped to just one core. So the single core executes the entire workload which was distributed in the dual-core experiment. In reliability terms, the sequential mapping resembles a *redundancy in time* as we still execute a fault tolerant task twice.

Figure 6.9a and 6.9b show the results. The axes show the load on the processors and the relative improvement of using a dual core, parallel approach over a sequential one. Naturally, a single core setup is only schedulable if the sum of the loads is less than 1. Thus, no improvement indication can be given for setups where $Load_1 + Load_2 \geq 1$.

The naive assumption would be that a parallel mapping would outperform a sequential mapping. Interestingly there is no clear indication of this. Under worst-case assumptions, the Bitcount benchmark performs clearly better if mapped to two cores, whereas the Rijndael benchmark exhibits a worst-case response time twice as large as compared to a sequential setup[5].

The reason for this is the number of stages and the execution times in these stages. The Rijndael benchmark has 14 stages. Of these stages are 13 IO bound stages of only a couple of thousand processor cycles and one computation driven very long stage. In contrast to this, Bitcount has 8 stages. Three of these stages are IO bound the rest are long computation bound stages. We can conclude that a parallel setup leads to a real-time improvement if the fork-join task has an equally distributed stage execution time.

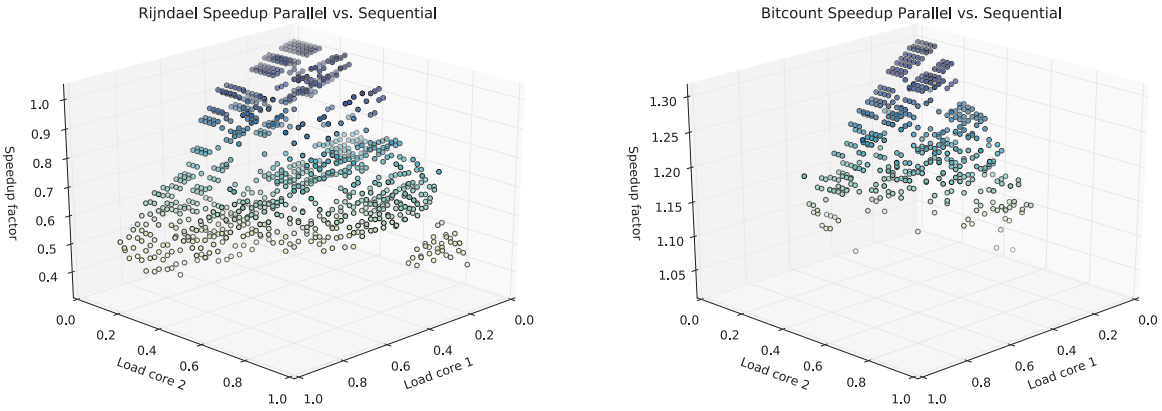
6.3 Reliability Prediction of Replication

In the previous section we discussed the model and response time analysis of replicated applications. Intuitively, redundant execution should increase reliability. However, to evaluate the final safety-intergrity level we need to obtain the likelihood of a task meeting its deadline.

In this section, we analyze the likelihood that a task misses a deadline due to transient load caused by recovery of erroneous tasks. This leads us to a task-wise reliability analysis which allows to validate criticality constraints (safety integrity level [135]). The goal is to derive the reliability as tight as possible to reduce over-provisioning to a minimum.

The handling of permanent errors is not considered in this section and considered as an orthogonal problem which is tackled with other

[5]These are worst-case considerations under the presented approximation, simulations might indicate other reasoning but cannot easily capture the worst-case scenario



(a) Speedup for rijndael benchmark. (b) Speedup for bitcount benchmark.

Figure 6.9: Worst-case time improvement (speedup) single core vs. dual core mapping.

approaches as explained in Chapter 2. Permanent errors are covered by e.g. [102, 144, 214].

The detection scheme in ASTEROID is assumed to be perfect, thus, all errors are detected and the detection mechanism itself is sufficiently reliable. Here, "sufficient means that the mean time to failure has to be at least an order magnitude higher than other sources of error. The final MTTF of the system (or function) is then sufficiently approximated by the MTTF of the application with the lowest MTTF. First, this is a reasonable assumption, also done in related work [212, 298]. The error detection coverage scheme used in ASTEROID is measured in [79] and found to be 100 % and the recovery rate for TMR majority voting is almost 100 %.

6.3.1 Related Work

Related work in the area of fault-tolerant system analysis focus either on logical or on temporal correctness of systems or components. Whenever formal techniques are introduced to compute the reliability with respect to timing constraints, the probability of logical failures is neglected. Similarly considering fault tolerance mechanisms with respect to logical correctness does not take timing effects into account. In this section we will focus on the research of analyzing timing failures.

Baruah et al. describe a method in [28] to account for mixed criticality in real-time systems. They assume that task characteristics such as the worst-case execution time (WCET) are annotated with a certain confidence (criticality level). The response time analysis for a particular task is then carried out in consideration of the interference based on the criticality level.

However, fault-tolerance mechanisms and reliability in particular are not considered.

Izosimov et al. [140, 212] suggest another analysis methodology for fault-tolerant real-time systems. In this work re-execution and replication are treated as design alternatives. They assume static execution order scheduling within a MPSoC. Based on this assumption they derived the maximum number of tolerable errors that can be corrected in time. Furthermore, they optimized this number by combining re-execution and replication corresponding to a heuristic optimization algorithm. The drawback is the assumptions of static execution order scheduling which limits the applicability in real-life systems.

In [48, 49] Burns et al. presented an extension of the worst-case response time analysis to incorporate re-execution. This method gives upper reliability bounds extrapolated from the critical instant, which is the worst-case activation scenario where all tasks are released simultaneously. Also in [216] Punnekkat et al. extended this work to derive an optimal checkpointing strategy.

Timing prediction for erroneous bus communication as presented in Chapter 4 are partially applicable to rollback and checkpointing. For instance [44] can be used to model and analyze the exeuction time overhead for error detection, rollback recovery and re-execution. However, redundant execution as employed in ASTEROID is not supported.

We extend the work presented in [248], initially targeted towards bus communication assuming fixed-priority, non-preemptive arbitration. The authors present a framework to compute the reliabiltiy (mean time to deadline miss). To bound the reliability as tight as possible a job-wise analysis approach is introduced, where each job in the hyperperiod, which is the time in periodic systems after which the activation pattern starts to repeat itself, is analyzed. In a second step the results are composed to a task reliability function. We extended this idea to support redundant execution on multi-core SoCs.

6.3.2 Error Model and Metrics

The occurrence of errors on a core is modelled using Poisson processes with a given constant error-rate λ_i per core. A single error rate can be derived from the errors rate of memory and core components such as ALU and FPU. The Poisson model is widely used in literature [127, 251, 101, 298] and generally accepted.

Error events are independent which is a realistic assumption for transient errors induced by radiation. Most likely the Poisson model is not suitable for thermally induced errors as these errors are not homogeneous unless the error rate is conservatively approximated for all environmental conditions such as frequencies, voltages and workloads. Specifying

per-core error rates accounts for architectures in which some cores may be inherently more reliable than others. Error rates can be reduced by low-level hardware mechanisms (e.g. [9]) or by using more reliable but low-performance process parameters.

For the Poisson model, the following equations give the probability for correct execution of the i-th processor during the time interval Δt and the converse probability that at least one error occurred.

$$P(\textit{no error in time } \Delta t) = e^{-\lambda_i \Delta t} \tag{6.19}$$

$$P(\textit{errors in time } \Delta t) = 1 - e^{-\lambda_i \Delta t} \tag{6.20}$$

We have used the Poisson model before to model independent errors in communication (cf. Chapter 4). The first equation is deduced from eq. 4.15 by setting n_e to zero, $n = \Delta t$ and $p = \lambda_i$. The second equation is the converse probability of the first.

Throughout the rest of the chapter we consider the deadline failure probability of each job of a task individually. In this scope, we need to recall that the j-th job of a task τ_i is denotes as $\tau_{i,j}$.

In the error-free case there are exactly n checkpoint groups, one for each stage of the task. As already mentioned, in case of errors during the execution, the total number of checkpoint groups for the for task Θ_i is increased by the number of affected checkpoint groups (cf. Corollary 5). Naturally, the number of erroneous checkpoint groups is not known a priori, hence we do not know the actual phenotype of the fork-join model describing a job $\Theta_{i,j}$. But the effects of a given number of errors on a fault tolerant task can be modeled by using the appropriate precedence model.

For this analysis we restrict our task model to periodic tasks with deadlines smaller than their periods. Hence, tasks are periodically activated every T time units, execute for at most time C and must have finished execution before their deadline $D \leq T$.

Since we focus on hard real-time systems, we are particularly interested if tasks adhere to their deadline constraint under error scenarios. The response time $R_{i,j}$ of a job is the difference between its finishing time and its release time, which is the time when the job becomes ready and is computed by the equations presented in the previous sections.

Now, the following question remains: “How reliable is a replicated fault tolerant task?” We will use the common notion of reliability $\mathcal{R}$ as a metric Practically $\mathcal{R}(t)$ denotes the probability that a task is still operating without a deadline failure in the interval $[0, t]$.

$$\mathcal{R}(t) = P(\text{no failure in interval } [0, t]) \tag{6.21}$$

The same metric is used in [248] to evaluate the reliability of periodic data frames transmitted over a bus.

The goal of the formal analysis is to derive $\mathcal{R}_i(t)$ for each task from which we can easily calculate other common metrics such as the MTTF:

$$MTTF_i = \int_0^\infty \mathcal{R}_i(t)\ dt \tag{6.22}$$

6.3.3 Formal Reliability Analysis

In this section we present an algorithm which allows to compute $\mathcal{R}_i(t)$ accurately for arbitrary values of t. The general approach is similar to [248] extended to checkpointing and rollback on MPSoCs. For this formal reliability analysis we consider tasks individually. Thus, a reliability per task is retrieved. This is done by assessing the success of each job individually and combining these values into a per-task reliability. By this, a more precise result can be obtained because the interference characteristic of each job is account. This trades accuracy and flexibility as the presented approach only works for synchronized systems.

In a periodic taskset, the activation pattern appears repetitively after the hyperperiod of length $lcm(T_1, \ldots, T_n)$. For the rest of this analysis approach, we assume that the response time of each job $\tau_{i,j}$ as well as $\Theta_{i,j}$ can be computed using the previously established approaches. It must be highlighted that we only presented an approach to compute the worst-case response time for all jobs, but the equations can easily be used to compute the response time of individual events (thus jobs). Now we introduce some important definitions. Let us assume that we want to compute the reliability of a task τ_i. This task can be a fault tolerant task as well as a non fault-tolerant task.

Definition 52 (Success)**.**
The fact that the j-th activation of task τ_i is logically correct (computes the correct value) and meets its deadline is referred to as $S_{i,j}$.

Definition 53 (Success Probability)**.**
The success probability $P[S_{i,j}]$ is the probability that job $\tau_{i,j}$ succeeds.

It is important to note that it is of no relevance for the response time of a ft-task which erroneous segment job in a checkpoint group caused additional delay *exactly*. The ultimate effect of errors in $CG_{i,j,k}$ is always the same, regardless of the actual faulty subtask. This can be seen in Figure 6.4, if e_2 would have hit the corresponding job on core 2, the outcome would have been the same.

For the analysis it is important to know which jobs interfere with another job. The set of tasks and thus the set of jobs can be obtained by the *timing dependency graph*[6] which indicates the functional and non-

[6]Not to be confused with the application graph.

functional dependencies between the tasks in the example from Figure 6.4. The nodes of the timing dependency graph correspond to the tasks in the system and the directed edges represent functional and non-functional dependencies between tasks. Functional dependencies are given through the task graph and non-functional dependencies arise from the local scheduling on a processor. A particular job $\tau_{i,j}$ may be delayed by jobs of higher priority than $\tau_{i,j}$ which are released prior to $\tau_{i,j}$ running on the same resource as $\tau_{i,j}$ (non-functional dependency). Due to precedence constraints (functional dependency), a job $\tau_{i,j}$ can also be delayed by jobs running on another core. We are not interested in the subtasks which may delay a given job, but instead in the corresponding checkpoint groups which may delay a job.

Definition 54 (Interference Set)**.**
The interference set $\xi_{i,j}$ is the set of all checkpoint groups CG which potentially delay job $\tau_{i,j}$.

Note that if $\tau_{i,j}$ itself is a ft-task its own $CG_{i,j,k}$ are always in $\xi_{i,j}$. Errors in non-ft tasks have no timing effect on other tasks and therefore do not need to be considered in the interference set. The cardinality of $\xi_{i,j}$ is arbitrarily large. Practically we limit the set to a given number of checkpoint groups which have been released time d before the release of $\tau_{i,j}$, thus for large d we gain arbitrarily good estimates of $\xi_{i,j}$. By this, we exploit that jobs which are released sufficient time before $\tau_{i,j}$ do not interfere with $\tau_{i,j}$.

Choosing a too small value for d may lead to underestimation, leading to non-conservative results due to additional interference which is not covered throughout analysis. By analyzing for increasing values of d the result converges towards the actual reliability.

Based on the given definitions, we can express the reliability of a task τ_i as the probability that all jobs $\tau_{i,1} \dots \tau_{i,j}$ have succeeded their execution that have been released in the interval $[0, t]$:

$$\mathcal{R}_i(t) = P[S_{i,1} \wedge \ldots \wedge S_{i,j}] \tag{6.23}$$

Feasible Error Scenarios

The algorithm we propose consists of two independent steps that are carried out for each task: In the first step all *feasible scenarios* for each job of the task τ_i throughout the hyperperiod are enumerated. A feasible scenario is a specific error constellation in which no deadline miss (failure) for the task τ_i occurs. In a second step, we transform these scenarios into probabilities through which it is possible to derive the characteristic reliability function $\mathcal{R}_i(t)$.

Coming back to the example in Figure 6.4, we focus on the scenario enumeration for job $\tau_{4,1}$. The response time of job $\tau_{4,1}$ depends on the

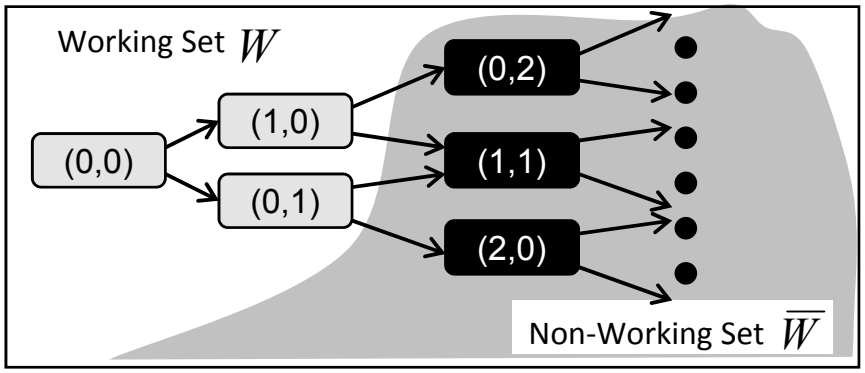


Figure 6.10: Working-set for job $\tau_{4,1}$

reexecution pattern of checkpoint groups which are in the interference set of $\tau_{4,1}$ which is $\xi_{4,1} = \{CG_{2,1,1}, CG_{2,1,2}\}$. For the particular situation depicted, $CG_{2,1,2}$ handles one error (re-execution denoted as $CG'_{2,1,2}$) without causing $\tau_{4,1}$ to miss its deadline. Further analysis would reveal that a scenario where $CG_{2,1,1}$ is exposed to one error would also lead to a feasible schedule, whereas an error in both $CG_{2,1,1}$ and $CG_{2,1,2}$ causes $\tau_{4,1}$ to miss its deadline. To formalize this concept we introduce the *error scenarios*.

Definition 55 (Error Scenario)**.**
An error scenario $s^{i,j,k} : \xi_{i,j} \rightarrow \mathbb{N}_0$ is a function which specifies the number of reexecutions including recovery for each checkpoint group $CG \in \xi_{i,j}$.

This function can conveniently be expressed as a tuple, where the s-th component in the tuple denotes the number of errors for the s-th checkpoint group in the (ordered) interference set $\xi_{i,j}$. The example in Figure 6.4 shows the error scenario $s^{4,1,1}$ for which the interference set is $\xi_{4,1} = \{CG_{2,1,1}, CG_{2,1,2}\}$. Now, we can simply specify the scenario as $s^{4,1,1} = \{CG_{2,1,1} = 0,\ CG_{2,1,2} = 1\}$ or more conveniently $s^{4,1,1} = (0, 1)$.

As discussed, it is possible to construct an equivalent DAG model for each error scenario. This model can then be fed into a response-time analysis to verify deadline constraints. Thus, each scenario can either lead to a feasible schedule or to an infeasible schedule, depending on the response time analysis of the error scenario model. We can summarize that $\tau_{4,1}$ meets the deadline for the scenarios $\{(0, 0), (1, 0), (0, 1)\}$. Figure 6.10 shows exactly these feasible scenarios which we call working set.

Definition 56 (Working Set)**.**
The working set $\mathcal{W}_{i,j}$ of a job $\tau_{i,j}$ is the set of error scenarios $s^{i,j,k}$ for which job $\tau_{i,j}$ is guaranteed to meet its deadline constraint.

Originating from the error-free case as the root node, we can construct an error graph as shown in Figure 6.10. In this graph, edges are error

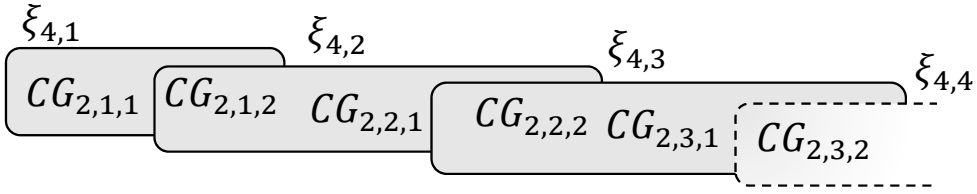


Figure 6.11: Illustrative example of some interference sets for task τ_4 which are mutually overlapping, causing stochastic dependence for scenarios. Note that the interference sets are truncated to three members to keep the problem tractable.

events whereas nodes are error scenarios. The graph can be built by using depth first search: For each node we perform a response time analysis and evaluate the real-time constraint $R^+ < D$. Each node is then colored with respect to the schedulability of the error scenario which it presents. In case the node is schedulable, we recursively evaluate all successor nodes in the same way until we find all working scenarios. All nodes which are marked *feasible* form the working set $\mathcal{W}_{i,j}$.

Practically, the graph can be arbitrarily large but it is sufficient to enumerate only a sub-graph. By doing so, we obtain a subset of the real working set which is conservative because all other nodes are considered as non-working. A pessimistic approach which only considers the worst-case activation would only yield to one particular $\mathcal{W}_i$ which resembles the situation with the least number of working scenarios.

The success probability of job $\tau_{i,j}$ can be expressed as the probability that one scenario of all working scenarios in $\mathcal{W}_{i,j}$ will actually occur:

$$P[S_{i,j}] = P[s^{i,j,1} \vee \ldots \vee s^{i,j,k}],\ s^{i,j,1}, \ldots, s^{i,j,k} \in \mathcal{W}_{i,j} \tag{6.24}$$

Probability Computation

Once we have obtained all working sets for all jobs in the hyperperiod we can derive success probabilities. It is not sufficient to calculate scenario occurrence probabilities based on individual working sets alone. The scenarios from successive jobs of the same task are not mutually independent. The reason for this is that consecutive interference sets include common checkpoint groups: $\xi_{i,j} \cap \xi_{i,j+1} \neq \emptyset$. This is also depicted in Figure 6.11. Note that the interference sets are artificially truncated for the sake of simplicity.

This problem becomes obvious in the following simple example: We assume that the working sets for $\tau_{4,1}$ and $\tau_{4,2}$ have been determined independently in a previous step. Now, since $\xi_{4,1}$ and $\xi_{4,2}$ overlap, job $\tau_{4,2}$ will only see manifestations of $CG_{2,1,2}$ that led to a feasible schedule for job

$\tau_{4,1}$, namely zero or one repetitions of $CG_{2,1,2}$ (cf. Figure 6.10). To express these mutual dependencies, we introduce conditional success probabilities.

Definition 57 (Conditional Success Probability)**.**
The conditional success probability is the probability that job $\tau_{i,j}$ meets its deadline given that previous jobs of τ_i have already met their deadlines.

$$P[S_{i,j}|S_{i,j-1} \wedge \ldots \wedge S_{i,1}] \tag{6.25}$$

By using conditional probabilities, it is possible to express the reliability function (cf. Equation 6.23) by simple multiplication:

$$\begin{aligned} \mathcal{R}_i(t) = \quad & P[S_{i,1}] \cdot P[S_{i,2}|S_{i,1}] \cdot \ldots \\ & \cdot P[S_{i,j}|S_{i,j-1} \wedge \ldots \wedge S_{i,1}] \end{aligned} \tag{6.26}$$

The remaining challenge is to derive the conditional success probabilities used in equation 6.26. We can be sure that job $\tau_{i,j}$ will meet its deadline if the actual error scenario is in $\mathcal{W}_{i,j}$. Beyond that, it is also necessary that previously activated jobs of task τ_i have successfully terminated. Because all scenarios in $\mathcal{W}_{i,j}$ are mutually exclusive, equation 6.25 can be written as a sum of conditional scenario probabilities:

$$P[S_{i,j}|S_{i,j-1} \wedge \ldots \wedge S_{i,1}] = \sum_{s \in \mathcal{W}_{i,j}} P\left[s|S_{j-1} \wedge \ldots \wedge S_1\right] \tag{6.27}$$

Remember, we are only interested in the time until the first failure. We process jobs in the order of their activation. Hence, when we look at an arbitrary success $S_{i,j}$ this always includes the fact that all jobs released before $\tau_{i,j}$ have succeeded. From the perspective of $S_{i,j}$, the history of working scenarios forms a complete probability space. Also, when we process job $\tau_{i,j}$ to calculate $P[S_{i,j}|S_{i,j-1}, \ldots, S_{i,1}]$, we know the conditional probability of the predecessor job $P[S_{i,j-1}|S_{i,j-2}, \ldots, S_{i,1}]$ and the probabilities of all scenarios in the working set of $\tau_{i,j-1}$, because they have already been evaluated. But as already mentioned in Section 6.3.3, sometimes scenarios of successive working sets are mutually exclusive. To formalize this we introduce scenario consistency.

Definition 58 (Consistency)**.**
Two error scenarios $s_a = s^{i,j,a}$ and $s_b = s^{i,j-1,b}$ are said to be consistent if there is no $CG \in \xi_{i,j} \cap \xi_{i,j-1}$ so that $s_a(CG) \neq s_b(CG)$, where $s(CG)$ denotes the number of errors in checkpoint group CG in scenario s.

Practically, this means that consistent error scenarios may contain the same checkpoint group CG in the interference set and if so, the number of repetitions $s_a(CG)$ and $s_b(CG)$ must be the same, thus s_a and s_b are

not contradictory. If two scenarios s_a and s_b are consistent, we can reduce one scenario s_a and remove those checkpoint groups which are already contained in s_b:

Definition 59 (Reduced Scenario)**.**
Given two consistent scenarios s_a and s_b, the reduced scenario $\tilde{s}_a$ is defined as $\tilde{s}_a : \{\xi_a \setminus \xi_b\} \rightarrow \mathbb{N}_0$, where $\tilde{s}_a(CG) = s_a(CG) = s_b(CG)\ \forall\ CG \in \{\xi_a \setminus \xi_b\}$

For the example from Figure 6.11, a reduced scenario $\tilde{s}^{4,2,l}$ from $\mathcal{W}_{4,2}$ would not contain checkpoint group $CG_{2,1,2}$ since this is specified by the predecessor scenario. By application of consistency and reduced scenarios we can calculate the conditional scenario probability as follows:

$$P[s_{i,j,a}|s_{i,j-1,b}] = \begin{cases} P[\tilde{s}_{i,j,a}] & \textit{if } s_{i,j,a}, s_{i,j-1,b} \textit{ consistent} \\ 0 & \textit{otherwise} \end{cases} \tag{6.28}$$

Then we can put all building blocks together to obtain the final conditional success probability, based on the the scenarios from the previous working set $\mathcal{W}_{i,j}$ which has already been processed in the previous step.

$$\begin{aligned} & P[s_{i,j,k}|S_{i,j-1} \wedge \ldots \wedge S_{i,1}] \\ = & \sum_{s \in \mathcal{W}_{i,j-1}} P[s_{i,j,k}|s] \cdot \tilde{P}[s|S_{i,j-2} \wedge \ldots \wedge S_{i,1}] \end{aligned} \tag{6.29}$$

Where $\tilde{P}$ is the normalized probability of the scenario s. The probabilites are normalized because all scenarios in $\mathcal{W}_{i,j-1}$ form a total probability space:

$$\sum_{s \in \mathcal{W}_{i,j-1}} \tilde{P}[s|S_{i,j-2} \wedge \ldots \wedge S_{i,1}] = 1 \tag{6.30}$$

The absolute probability for the entire scenario $P[s]$ is a simple multiplication, since the checkpoint groups in a scenario are independent.

$$P[s] = \prod_{CG \in \xi_s} P[R = s(CG)] \tag{6.31}$$

Equation 6.31 only considers timing errors and is only valid for ft-tasks. Errors which affect and corrupt non ft-tasks are currently not considered. In order to integrate the fact that non ft-tasks do not only fail in case of a deadline violation, but also when they are hit by an error event, it is necessary to extend the equation by a term (cf. Equation 6.20) which reflects that the job under analysis will not be hit by an error itself.

$$P_{\textit{non-ft}}[s] = (1 - e^{-\lambda_i C_i}) \cdot \prod_{CG \in \xi_s} P[R = s(CG)] \tag{6.32}$$

Finally, it is trivial to calculate the probability for certain re-execution patterns by applying Equation 6.19 and 6.20 with appropriate values of Δt.

The probability for *exactly* $s(CG_{i,j,l}) = R$ repetitions of a checkpoint group $CG_{i,j,l}$ for a given scenario s can be calculated as follows, where t_e is the execution time of the segments in $CG_{i,j,l}$. For the sake of simplicity we assume the error rates for all cores are the same $\lambda = \lambda_p \forall p \in \mathcal{P}$. Here β denotes the degree of redundancy, i.e. $\beta = 2$ implies a DMR setup with two replica executions in a checkpoint group.

$$P[R = 0] = \left(1 - e^{-\lambda(t_{cov}+t_e)}\right)^{\beta} \tag{6.33}$$

$$P[R = 1] = (1 - P[R = 0]) \cdot \left(1 - e^{-\lambda(t_{rov}+t_e)}\right)^{\beta} \tag{6.34}$$

$$P[R = r, r > 0] = (1 - P[R = 0]) \cdot \left(1 - \left(1 - e^{-\lambda(t_{rov}+t_e)}\right)^{\beta}\right)^{r-1} \cdot \left(1 - e^{-\lambda(t_{rov}+t_e)}\right)^{\beta} \tag{6.35}$$

For ft-tasks the system tries to recover from errors until one successful re-execution took place. Thus, for the probability calculation it is known that all erroneous re-executions are always followed by one correct re-execution.

By inserting the conditional success probabilities in Equation 6.26, it is possible to obtain $\mathcal{R}_i(t)$. Assuming we have job success probabilities for all jobs in the interval $[0, t_{hyper}]$, with $t_{hyper} = lcm[T_1, \ldots, T_n]$, we can compute the reliability $\mathcal{R}(t_{hyper})$ through Equation 6.26. From this we can compute the reliability for a given number of A hyperperiods:

$$\mathcal{R}(A \cdot t_{hyper}) = (\mathcal{R}(t_{hyper}))^A \tag{6.36}$$

6.3.4 Experiments

We show the practical applicability and accuracy and analysis speed of the presented approach by comparing our approach with a Monte-Carlo reference simulation.

For the evaluation of the presented formal analysis we use Monte-Carlo simulation as a reference. We assume that tasks are perfectly synchronized, that means all tasks activate simultaneously at time $t = 0$ and there is no drift of the activation pattern. Under these assumptions, it is possible to calculate the exact response times for each scenario because there are no uncertainties in scheduling.

Table 6.2: Mapping (M1) of task-set used throughout evaluation.

Task	$prio_1$	$prio_2$	n	t_{cov}	t_{rov}	C	P	D
Θ_0 (ft)	3	2	2	10	40	60	300	300
τ_1	4	-	-	-	-	50	250	250
τ_2	2	-	-	-	-	10	100	100
τ_3	-	1	-	-	-	50	300	300
Θ_4 (ft)	1	3	2	10	40	40	600	500

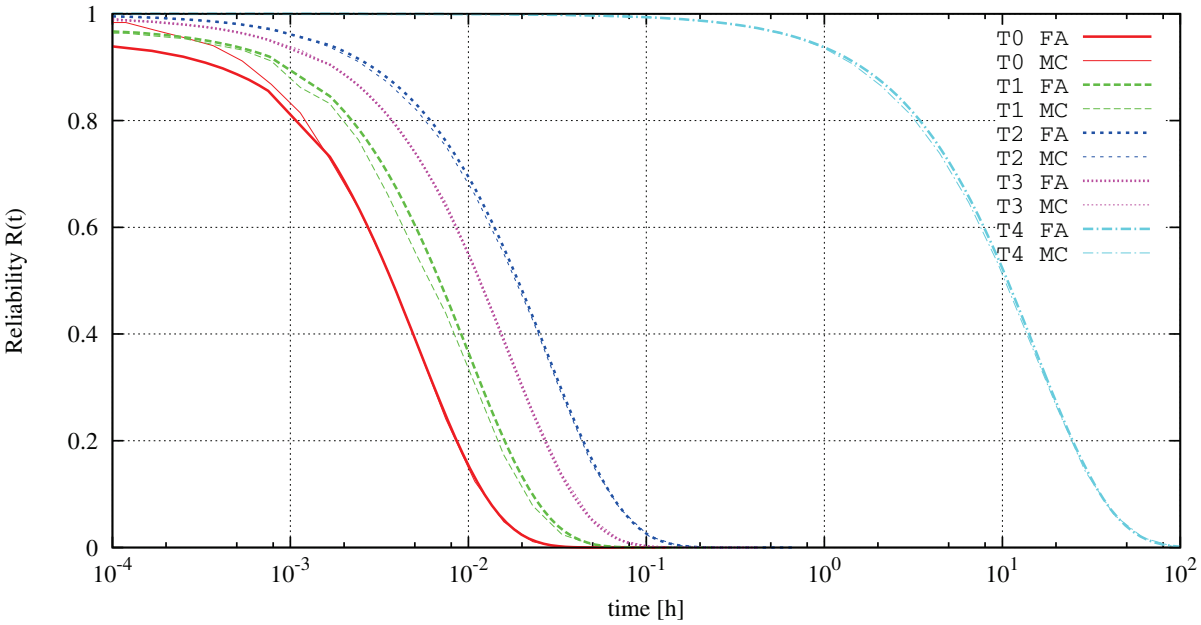


Figure 6.12: Comparison of formal analysis (FA) accuracy with Monte-Carlo (MC) reference simulation. $\lambda_{1,2} = 1/sec$, 10.000 samples.

For the following experiments we use the taskset as shown in Table 6.2. We assigned Θ_0 and Θ_4 to be fault-tolerant tasks, that means both tasks execute redundantly on core 1 and core 2 and create two checkpoints with some additional overhead.

The priority assignment and mapping is chosen in a way that the following effects can be observed:

- functional dependencies due to precedence constraints of checkpoint groups
- non-functional dependencies caused by scheduling of higher priority tasks
- priority inversion caused by inter-core blocking

We can observe that the reliability of Θ_4 is much better than the one of Θ_0. This is due to the computation time and deadline parameters. Task Θ_4 has the chance to accommodate a lot more re-executions until a deadline is missed compared to Θ_0.

By our approach it is now possible to efficiently use one multi-core processor and replicate only critical tasks where conventional approachs would replicate the entire system.

For the Monte-Carlo-Simulation, we have implemented a simple static priority preemptive scheduler that will schedule a given taskset in the same way, as an operating system scheduler would do. However, we do not schedule real tasks but only abstract tokens which represent tasks. For each core in the system, we instantiate one scheduler which is attached to an event-generator. The event-generator will produce error events with an exponentially distributed inter-event time with an average of $1/\lambda_i$.

The response time of each job is monitored, and failure events caused either by logical incorrectness of regular tasks or timing violations of all tasks are recorded. The reliability function $\mathcal{R}_i(t)$ is then the inverse of the cumulative distribution function of the failure events. By recording a sufficient large number of samples it is possible to approximate the exact reliability with respect to our system and task model.

The comparison of Monte-Carlo approach (MC) and our presented formal analysis (FA) is shown in Figure 6.12. The graph shows the reliability function $\mathcal{R}(t)$ versus time, which gives the probability that a task is still functioning after time t. The simulation has been carried out for unrealistic error rates to produce a reasonable number of MC simulation runs in acceptable time. For realistic error rates, Monte-Carlo simulation would not be operational due to excessive run times. The reason is that the time per Monte-Carlo run grows with reduced error rates. Unfortunately, error effects are coupled due to their influence on timing. Coupling makes advanced MC techniques such as importance sampling complicated or requires approximations, such as adapting event frequencies, which changes MC results. We use MC simulation only to demonstrate that our approach has very little pessimism. Already for these high error rate, more than 2 hrs computation time were needed. Since the accuracy of our approach is generally independent of the error rate, we may conclude that the accuracy observed in the experiments also holds for realistic error rates.

The reliability analysis was carried out with a search depth value of $d = 12$, higher values increase analysis time, because more scenarios have to be considered and the accuracy improvement is not noticeable. As mentioned in Section 6.3.3, it is not possible to list *all* feasible scenarios in the working set for a practical implementation. Thus, the working set is artificially truncated, then potentially working scenarios are considered as non-working which is a pessimistic assumption leading to a conservative result. For this experiment, the working set was truncated when the

occurrence probability of a scenario was below a cut-off probability of $1e-12$. This produces equivalent results compared to the Monte-Carlo approach. There is a pessimistic derivation of the formal analysis for very small times $t < t_{hyper}$. The reason for this is that the Monte-Carlo simulator will start executing tasks at time $t = 0$ and jobs in the first hyperperiod have no interference from a previous hyperperiod which could cause deadline violations. Note that the Monte-Carlo result and the formal analysis result converge quickly (see Θ_0 in Figure 6.12).

However, with 17 sec run time, our analysis framework was significantly faster than the processing of 10.000 MC runs which took about 2 hrs. For our approach the run time only depends on the accuracy that shall be achieved and the number of task activations in a hyperperiod.

6.4 Summary

In this chapter we presented a model for fault tolerant tasks which uses replication and rollback (or rollforward) recovery. Therefore, we mapped the fault tolerant task to a timing-wise equivalent fork-join task graph. A fork-join task graph reflects the redundancy and compare mechanism through a directed acyclic graph model. Furthermore, we showed how errors and recovery operations are modeled by adding additional stages to the fork-join task graph.

In the second section, we presented a worst-case response time analysis approach which computes the timing of heterogeneous tasksets composed of independent tasks as well as fork-join tasks. By applying the approach to the Romain framework, we were able to show that parallel workloads may behave counterintuitive: In some cases the worst-case response time is drastically larger compared to a sequentialized execution. This can be explained by the fact that a fork-join task experiences the worst-case interference of all cores in a combined fashion. This effect is not due to a conservative overestimation but is observable in real-world as long as the correctness of the used event models is guaranteed. We could show that a parallelization does not decrease the worst-case response time in all cases and is connected to subtle design decisions such as the number of stages and prioritization. Fixed-priority scheduling does not seem optimal in case of redundant workload. In this scope, it would be interesting to see how other scheduling policies such as synchronized round-robin behave.

In the last section, we presented a formal approach through which we can analyze reliability constraints for a mixed-critical taskset of which some tasks are protected by redundant execution including checkpointing and rollback and others are unprotected. Contrary to other approaches, we consider a representative hyperperiod and not the worst-case condition because it allows tighter reliability guarantees. The comparison of

our algorithm with a reference Monte-Carlo simulation shows very good accuracy with the benefit of significantly shorter analysis time for realistic parameters compared to simulation.

CHAPTER 7

Conclusion

Embedded system's industry calls for even cheaper, high performance platforms. Traditional fault-tolerance concepts which tackle the reliability challenge at the hardware level are deemed too expensive and inflexible to host a variety of applications with opposing constraints.

In the first part of this thesis, we presented a flexible cross-layer resiliency approach *ASTEROID*. We summarized this hardware/software platform which enables reliable execution by using hardware assisted replication. Our approach provides a solution tailored towards the needs in mixed-critical applications: Isolation between applications of different criticality, fault tolerance for critical application and competitive performance for best-effort services.

Depending on these constraints, low-level hardware, operating system and the Romain service go hand in hand to provide just the level of service which is required without massive over provisioning. ASTEROID achieves this by using dynamic approaches which come with a certain degree of runtime uncertainty, as errors on all levels (processor, on-chip interconnect and off-chip communication) influence end-to-end system timing considerably. The key contribution however is not the platform itself, but the *design methodology* required to out rule such timing uncertainties and craft a system which can be certified without much hassle.

In this sense we followed a compositional approach and *decomposed* the platform into building blocks. For each of these blocks we provided modelling and analysis approaches to capture the impact of errors under different error models. A system analysis in style of the SymTA/S approach can compose a consistent system picture from individual resource models. Each resource model is extended by an error model. Thus, we are able to provide conservative approximations of error-scenarios by using a formal

analysis which is several orders of magnitude faster compared to error injection methods.

For *on-chip and off-chip communication* existing error models such as a Single Bit Error Model as well as Gilbert Loss model for bursts were combined with performance models to obtain probabilistic predictions which are valid bounds in all system states and can be used for safety certification, where guaranteed performance is a key requirement. We considered point-to-point communication with error control as often found in cross-bars but also multi-master busses such as AMBA as well as distributed off-chip networks such as Controller Area Network. We found that the effects of errors in non-switched, on-chip communication (i.e. busses such as AMBA) play a very little role as the overhead for each error is very little compared to the number of overall transactions. However, in environments with large error bursts, the additional latency induced by errors naturally increases. Our approaches allow a fast and accurate approximation of these effects under a parametrizable error model.

In larger networks the communication is typically protected using *end-to-end error control protocols* such as Stop-And-Wait or Go-Back-N. These protocols are extremely hard to predict as the performance depends on contention in the network which affects the actual latency of the data packets, latency of the (worst-case) round-trip time and number of dropped packets but also on the error control protocol state machine in the end nodes. This thesis provides a modelling approach of end-to-end automatic repeat request effects and incorporated this model in a system analysis context. A comparison with simulation showed that our approach gives a good approximation of the expected performance of the network topology and can be used to optimize latency and throughput.

The operating service Romain, implemented by TU Dresden, provides a selective redundant execution framework. In this work we modeled redundant execution by using a fork-join task graph. Furthermore, we provided a response-time analysis for arbitrary (but non-nested) fork-join tasks under fixed-priority scheduling. The presented model is capable to reflect the behavior of replicated execution and error-recovery using rollback (or rollforward). This allowed us to embed the performance analysis into a reliability analysis to predict the mean-time-to-failure for synchronized parallel tasks.

There are multiple open questions which give a direction for future research in the domain of timing-performance under errors. First, a system analysis can only capture a certain error constellation. Thus, a maximum tolerable error limit per resource must be defined in advance, otherwise a system fixed-point cannot be found. The available error-models for a processor analysis are not sufficiently expressive, here the error effect of individual components such as MMU, ALU, cache must be considered closer to get a more precise understanding of failure-rates and the associated

failure modes in these components. Also the performance prediction of redundant execution leaves room for two optimizations: We have drawn the conclusion that partitioned, fixed-priority scheduling is not a good scheduling policy for this problem. Research of cooperative scheduling solutions is the suggested next step. Also the analysis itself can be drastically improved by considering best-case behavior. This can lead to a significant improvement in performance prediction.

APPENDIX A

Publications

This appendix lists all publications by the author. The list is divided in thesis related and thesis unrelated publications.

A.1 Related to the Thesis

A.1.1 Reviewed

Philip Axer, Daniel Thiele, and Rolf Ernst. Formal timing analysis of automatic repeat request for switched real-time networks. In *In Proc. of SIES*, Pisa, Italy, June 2014.

In this paper we present a formal approach to predict system latencies of ARQ-based Ethernet systems. Chapter 5 is based on this work.

Philip Axer, Moritz Neukirchner, Sophie Quinton, Rolf Ernst, Björn Döbel, and Hermann Härtig. Response-time analysis of parallel fork-join workloads with real-time constraints. In *Proc. of Euromicro Conference on Real-Time Systems (ECRTS)*, jul 2013.

This paper presents a response-time analysis for fork-join task graphs. Chapter 6 is partially based on this work.

Philip Axer and Rolf Ernst. Stochastic response-time guarantee for non-preemptive, fixed-priority scheduling under errors. In *In Proc. of Design Automation Conference (DAC)*, jun 2013

A convolution-based approach is presented to derive the response-time exceedance function. Chapter 4 is partially based on this work. In particular Section 6.2.

Philip Axer, Rolf Ernst, Heiko Falk, Alain Girault, Daniel Grund, Nan Guan, Bengt Jonsson, Peter Marwedel, Jan Reineke, Christine Rochange, Maurice Sebastian, Reinhard von Hanxleden, Reinhard Wilhelm, and Wang Yi. Building timing predictable embedded systems. *ACM Transactions on Embedded Computing Systems*, 2013. Accepted.

This article presents challenges and sketches solutions for building time-predictable real-time systems.

Jonas Diemer, Philip Axer, and Rolf Ernst. Compositional performance analysis in python with pycpa. In *Proc. of WATERS*. jul 2012.

This paper presents pyCPA, a python implementation of the compositional performance analysis approach. All experiments in this thesis are done with pyCPA.

Philip Axer, Maurice Sebastian, and Rolf Ernst. Probabilistic response time bound for can messages with arbitrary deadlines. In *Proc. of DATE*, 2012.

This paper generalizes the approach presented in [45] to arbitrary activation pattern. Chapter 4 is partially based on this work.

Philip Axer, Rolf Ernst, Björn Döbel, and Hermann Härtig. Designing an analyzable and resilient embedded operating system. In *Proc. on Software-Based Methods for Robust Embedded Systems*, Germany, 2012.

This paper presents the ASTEROID platform as described in Chapter 2. This includes the Romain approach as well as fingerprinting.

Philip Axer, Maurice Sebastian, and Rolf Ernst. Reliability analysis for mpsocs with mixed-critical, hard real-time constraints. In *Proc. Intl. Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Taiwan, oct 2011.

This publication presents a reliably analysis for redundant tasks running on a MPSoC platform. Chapter 6, and in particular Section 6.3 is based on this work.

Philip Axer, Jonas Diemer, Mircea Negrean, Maurice Sebastian, Simon Schliecker, and Rolf Ernst. Mastering mpsocs for mixed-critical applications. *IPSJ Transactions on System LSI Design Methodology*, 4:91–116, aug 2011.

This publication guides the reader through the challenges of designing hard-real time MPSoCs. Applying formal timing analysis to different aspects of the system design such as shared resources (NoC and memory) and redundant execution are addressed.

A.1.2 Unreviewed

Philip Axer and Rolf Ernst. Timing of can under the influence of random error events. In *Real-Time Systems: the past, the present, and the future*. CreateSpace Independent Publishing Platform, 2013.

This publication summarizes the error analysis presented in [45] and generalizes it to arbitrary, non-parametrized event models.

A.2 Unrelated to the Thesis

Daniel Thiele, Philip Axer, and Rolf Ernst. Improving formal timing analysis of switched ethernet by exploiting fifo scheduling. In *Design Automation Conference (DAC)*, San Francisco, CA, USA, June 2015.

This paper shows how to improve response-time bounds by considering FIFO inherent inter-stream context.

Daniel Thiele, Johannes Schlatow, Philip Axer, and Rolf Ernst. Formal timing analysis of can-to-ethernet gateway strategies in automotive networks. *Real-Time Systems*, 2015.

This article shows how larger automotive communication systems consisting of gatways, switches and end nodes can be analyzed.

Daniel Thiele, Philip Axer, Rolf Ernst, and Jan R. Seyler. Improving formal timing analysis of switched ethernet by exploiting traffic stream correlations. In *Proc. of CODES+ISSS*, New Delhi, India, October 2014.

This paper shows how to improve response-time bounds by considering non-preemptiveness in Ethernet scheduling.

Philip Axer, Daniel Thiele, Rolf Ernst, and Jonas Diemer. Exploiting shaper context to improve performance bounds of ethernet avb networks. In *Proc. of DAC*, San Francisco, USA, June 2014.

This paper presents an approach to exploit the AVB shaper characteristics to improve performance predications.

Adam Kostrzewa, Sebastian Tobuschat, Philip Axer, and Rolf Ernst. Supervised sharing of virtual channels in networks-on-chip. In *In Proc. of SIES*, Pisa, Italy, June 2014.

This paper presents a resource broker approach to supervise resource allocation in NoC communication.

Philip Axer, Daniel Thiele, Rolf Ernst, Jonas Diemer, Simon Schliecker, and Kai Richter. Requirements on real-time-capable automotive ethernet architectures. 2014.

Similar to [268], this article presents problems and derived requirements for automotive Ethernet networks.

Moritz Neukirchner, Philip Axer, Tobias Michaels, and Rolf Ernst. Monitoring of workload arrival functions for mixed-criticality systems. In *RTSS*, 2013.

In this paper a workload monitoring approach is presented. This allows to monitor and thus isolate individual criticality levels from each other.

Daniel Thiele, Philip Axer, Rolf Ernst, Jonas Diemer, and Kai Richter. Cooperating on real-time capable ethernet architecture in vehicles. In *Proc. of Internationaler Kongress Elektronik im Fahrzeug*, oct 2013.

This article presents problems and derived requirements for automotive Ethernet networks. The paper shows where common solutions are required and how implementers and users can compete using standardized protocols.

Daniel Thiele, Philip Axer, Rolf Ernst, Jonas Diemer, and Kai Richter. Cooperating on real-time capable ethernet architecture in vehicles. In *Proc. of Internationaler Kongress Elektronik im Fahrzeug*, oct 2013.

This paper presents a response-time analysis for Weighted Round Robin Scheduling which is used in Ethernet switches.

Moritz Neukirchner, Sophie Quinton, Tobias Michaels, Philip Axer, and Rolf Ernst. Sensitivity analysis for arbitrary activation patterns in real-time systems. In *Proc. of Design Automation and Test in Europe (DATE)*, mar 2013.

This paper shows an approach to derive a sensitivity bound on activation patterns in the form of arrival functions opposed parametrized functions.

Boris Motruk, Jonas Diemer, Philip Axer, Rainer Buchty, and Mladen Berekovic. Safe virtual interrupts leveraging distributed shared resources and core-to-core communication on many-core platforms. In *In Proc. of PRDC*, 2013.

This work presents a safe, architectural approach to distribute interrupts in a large tiled many-core processor. The mechanism takes care that criticality restrictions are obeyed and interrupts are only forwarded to trusted parties.

Moritz Neukirchner, Tobias Michaels, Philip Axer, Sophie Quinton, and Rolf Ernst. Monitoring arbitrary activation patterns in real-time systems. In *RTSS*, 2012.

This work introduces an algorithm to monitor arbitrary activation patterns that are modelled through minimum distance functions.

Maurice Sebastian, Philip Axer, and Rolf Ernst. Utilizing hidden markov models for formal reliability analysis of real-time communication systems with errors. In *Proceeding of the 17th IEEE Pacific Rim International Symposium on Dependable Computing*, dec 2011.

This paper describes an approach to analyze burst errors on a CAN bus. For this a hidden Markov model is used to model the burst behavior.

Maurice Sebastian, Philip Axer, Rolf Ernst, Nico Feiertag, and Marek Jersak. Efficient reliability and safety analysis for mixed-criticality embedded systems. *SAE System Level Architecture Design Tools and Methods*, April 2011

This article highlights the reliably analysis for a safety-critical network design. It applies a CAN and MPSoC analysis to a typical automotive problem.

Bibliography

[1] *Channel error profiles for DECT*, volume 141. IET, 1994.

[2] Das v-modell. online, 1997.

[3] ISO 11898-1:2003 - Road vehicles – Controller area network (CAN) – Part 1: Data link layer and physical signalling, 2003.

[4] Anant Agarwal. The tile processor: A 64-core multicore for embedded processing. In *Proceedings of HPEC Workshop*, 2007.

[5] Richard Anthony, Achim Rettberg, De-Jiu Chen, Isabell Jahnich, Gerrit de Boer, and Cecilia Ekelin. Towards a dynamically reconfigurable automotive control system architecture. In Achim Rettberg, Mauro Cesar Zanella, Rainer Dömer, Andreas Gerstlauer, and Franz-Josef Rammig, editors, *IESS*, volume 231 of *IFIP Advances in Information and Communication Technology*, pages 71–84. Springer, 2007.

[6] ARINC. Arinc 653-3 - avionics application software standard interface, 2010.

[7] ARM Limited. Amba specification (rev. 2). May 1999.

[8] Motor Industry Research Association. *MISRA-C: Guidelines for the Use of the C Language in Critical Systems*. 2004.

[9] T. Austin, D. Blaauw, T. Mudge, and K. Flautner. Making typical silicon matter with razor. *IEEE Computer*, 37(3):57–65, 2004.

[10] AUTOSAR GbR. Specification of Multi-Core OS Architecture v1.0.0. http://www.autosar.org/, November 2009.

[11] AUTOSAR GbR. AUTOSAR: Technical Safety Concept Status Report, 2010.

[12] AUTOSAR GbR. Release 4.1. online, 2014. www.autosar.org.

[13] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on*, 1(1):11 – 33, 2004.

[14] Philip Axer, Jonas Diemer, Mircea Negrean, Maurice Sebastian, Simon Schliecker, and Rolf Ernst. Mastering mpsocs for mixed-critical applications. *IPSJ Transactions on System LSI Design Methodology*, 4:91–116, aug 2011.

[15] Philip Axer and Rolf Ernst. Stochastic response-time guarantee for non-preemptive, fixed-priority scheduling under errors. In *In Proc. of Design Automation Conference (DAC)*, jun 2013.

[16] Philip Axer and Rolf Ernst. Timing of can under the influence of random error events. In *Real-Time Systems: the past, the present, and the future*. CreateSpace Independent Publishing Platform, 2013.

[17] Philip Axer, Rolf Ernst, Björn Döbel, and Hermann Härtig. Designing an analyzable and resilient embedded operating system. In *Proc. on Software-Based Methods for Robust Embedded Systems*, Germany, 2012.

[18] Philip Axer, Rolf Ernst, Heiko Falk, Alain Girault, Daniel Grund, Nan Guan, Bengt Jonsson, Peter Marwedel, Jan Reineke, Christine Rochange, Maurice Sebastian, Reinhard von Hanxleden, Reinhard Wilhelm, and Wang Yi. Building timing predictable embedded systems. *ACM Transactions on Embedded Computing Systems*, 2013. Accepted.

[19] Philip Axer, Moritz Neukirchner, Sophie Quinton, Rolf Ernst, Björn Döbel, and Hermann Härtig. Response-time analysis of parallel fork-join workloads with real-time constraints. In *Proc. of Euromicro Conference on Real-Time Systems (ECRTS)*, jul 2013.

[20] Philip Axer, Maurice Sebastian, and Rolf Ernst. Reliability analysis for mpsocs with mixed-critical, hard real-time constraints. In *Proc. Intl. Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Taiwan, oct 2011.

[21] Philip Axer, Maurice Sebastian, and Rolf Ernst. Probabilistic response time bound for can messages with arbitrary deadlines. In *Proc. of DATE*, 2012.

[22] Philip Axer, Daniel Thiele, and Rolf Ernst. Formal timing analysis of automatic repeat request for switched real-time networks. In *In Proc. of SIES*, Pisa, Italy, June 2014.

[23] Philip Axer, Daniel Thiele, Rolf Ernst, and Jonas Diemer. Exploiting shaper context to improve performance bounds of ethernet avb networks. In *Proc. of DAC*, San Francisco, USA, June 2014.

[24] Philip Axer, Daniel Thiele, Rolf Ernst, Jonas Diemer, Simon Schliecker, and Kai Richter. Requirements on real-time-capable automotive ethernet architectures. 2014.

[25] François Baccelli, Guy Cohen, Geert Jan Olsder, and Jean-Pierre Quadrat. *Synchronization and linearity*, volume 3. Wiley New York, 1992.

[26] Radhakisan Baheti and Helen Gill. Cyber-physical systems. *The Impact of Control Technology*, pages 161–166, 2011.

[27] Theodore P. Baker. Comparison of empirical success rates of global vs. partitioned fixed-priority and edf scheduling for hard real time. Technical report, 2005.

[28] S. Baruah, Haohan Li, and L. Stougie. Towards the design of certifiable mixed-criticality systems. In *Proc. of Real-Time and Embedded Technology and Applications Symp.*, pages 13–22. IEEE, 2010.

[29] Sanjoy Baruah and Theodore Baker. Schedulability analysis of global edf. *Real-Time Systems*, 38(3):223–235, 2008.

[30] Sanjoy Baruah, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, Leen Stougie, and Andreas Wiese. A generalized parallel task model for recurrent real-time processes. pages 63–72, 2012.

[31] Sanjoy K Baruah, Deji Chen, and Aloysius Mok. Static-priority scheduling of multiframe tasks. In *Real-Time Systems, 1999. Proceedings of the 11th Euromicro Conference on*, pages 38–45. IEEE, 1999.

[32] Iain John Bate. *Scheduling and timing analysis for safety critical real-time systems*. Citeseer, 1999.

[33] H. Bauer, J. Scharbarg, and C. Fraboul. Worst-case end-to-end delay analysis of an avionics afdx network. In *Proc. of DATE*, pages 1220–1224, 2010.

[34] Olivier Beaumont, Vincent Boudet, Yves Robert, et al. A realistic model and an efficient heuristic for scheduling with heterogeneous processors. 2001.

[35] Johan Bengtsson, Kim Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. *UPPAAL—a tool suite for automatic verification of real-time systems*. Springer, 1996.

[36] L. Benini. Designing reliable systems with unreliable devices challenges and opportunities. In *Proc. IEEE Int. Electron Devices Meeting*, pages 509–511, 2007.

[37] G. Bernat and A. Burns. New results on fixed priority aperiodic servers. In *Real-Time Systems Symposium, 1999. Proceedings. The 20th IEEE*, pages 68–78, 1999.

[38] Rabi N Bhattacharya and Edward C Waymire. *Stochastic processes with applications*, volume 61. Siam, 2009.

[39] Enrico Bini and Giorgio Buttazzo. Measuring the performance of schedulability tests. *Real-Time Syst.*, 30:129–154, 2005.

[40] C. Bolchini, A. Miele, M. Rebaudengo, F. Salice, D. Sciuto, L. Sterpone, and M. Violante. Software and hardware techniques for seu detection in ip processors. *J. Electron. Test.*, 24(1-3):35–44, June 2008.

[41] Cristiana Bolchini and Antonio Miele. Reliability-driven system-level synthesis for mixed-critical embedded systems. *IEEE Transactions on Computers*, 62, 2013.

[42] V. Bonifaci, A. Marchetti-Spaccamela, S. Stiller, and A. Wiese. Feasibility analysis in the sporadic dag task model. In *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*, pages 225–233, July 2013.

[43] S. Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *IEEE Micro*, 25(6):10–16, 2005.

[44] I. Broster, A. Burns, and G. Rodríguez-Navas. Probabilistic analysis of CAN with faults. In *Proc. of Real-Time Systems Symposium*, pages 269–278. IEEE, 2002.

[45] I. Broster, A. Burns, and G. Rodriguez-Navas. Comparing real-time communication under electromagnetic interference. In *Proc. 16th ECRTS*, pages 45–52, 2004.

[46] H. Bruneel and M. Moeneclaey. On the throughput performance of some continuous arq strategies with repeated transmissions. volume 34, pages 244–249, 1986.

[47] Almut Burchard, Jörg Liebeherr, Yingfeng Oh, and Sang H. Son. New strategies for assigning realtime tasks to multiprocessor systems. *IEEE TRANSACTIONS ON COMPUTERS*, 44(12):1429–1442, 1995.

[48] A. Burns, R. Davis, and S. Punnekkat. Feasibility analysis of fault-tolerant real-time task sets. In *Proc. of Euromicro Workshop Real-Time Systems*, pages 29–33, 1996.

[49] A. Burns, S. Punnekkat, L. Strigini, and D. R. Wright. Probabilistic scheduling guarantees for fault-tolerant real-time systems. In *Proc. of Dependable Computing for Critical Applications*, pages 361–378, 1999.

[50] Alan Burns and Rob Davis. Mixed criticality systems: A review. Technical report, 2013.

[51] Alan Burns and Andy J. Wellings. Engineering a hard real-time system: From theory to practice. *Software: Practice and Experience*, 25(7):705–726, 1995.

[52] J. Caplan, M.I Mera, P. Milder, and B.H. Meyer. Trade-offs in execution signature compression for reliable processor systems. In *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, pages 1–6, March 2014.

[53] Georg Carle and Ernst W Biersack. Survey of error recovery techniques for ip-based audio-visual multicast applications. *Network, IEEE*, 11(6):24–36, 1997.

[54] Gonzalo Carvajal, Miguel Figueroa, Robert Trausmuth, and Sebastian Fischmeister. Atacama: An open fpga-based platform for mixed-criticality communication in multi-segmented ethernet networks. In *Proceedings of the 2013*

IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM '13, pages 121–128, Washington, DC, USA, 2013. IEEE Computer Society.

[55] D. Chabrol, C. Aussagues, and V. David. A spatial and temporal partitioning approach for dependable automotive systems. In *Proc. of Emerging Technologies & Factory Automation*, pages 1–8, 2009.

[56] Liming Chen and A. Avizienis. N-version programminc: A fault-tolerance approach to rellablllty of software operatlon. In *Fault-Tolerant Computing, 1995, Highlights from Twenty-Five Years., Twenty-Fifth International Symposium on*, pages 113–, Jun 1995.

[57] Hyungmin Cho, Shahrzad Mirkhani, Chen-Yong Cher, Jacob A Abraham, and Subhasish Mitra. Quantitative evaluation of soft error injection techniques for robust system design. In *Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE*, pages 1–10. IEEE, 2013.

[58] Sébastien Collette, Liliana Cucu, and Joël Goossens. Integrating job parallelism in real-time scheduling theory. *Information Processing Letters*, 106(5):180–187, 2008.

[59] FlexRay Consortium et al. Flexray communications system protocol specification version 2.1, 2005.

[60] William A Crossley. System of systems: An introduction of purdue university schools of engineering's signature area. In *Proceedings of the Engineering Systems Symposium*, 2004.

[61] R.I. Davis, A. Burns, R.J. Bril, and J.J. Lukkien. Controller area network (can) schedulability analysis: Refuted, revisited and revised. *Real-Time Systems*, 35(3):239–272, 2007.

[62] Robert I Davis, Steffen Kollmann, Victor Pollex, and Frank Slomka. Controller area network (can) schedulability analysis with fifo queues. In *Real-Time Systems (ECRTS), 2011 23rd Euromicro Conference on*, pages 45–56. IEEE, 2011.

[63] F. de Aguiar Geissler, F. Lima Kastensmidt, and J.E. Pereira Souza. Soft error injection methodology based on qemu software platform. In *Test Workshop - LATW, 2014 15th Latin American*, pages 1–5, March 2014.

[64] Giovanni De Micheli and Luca Benini. *Networks on chips: technology and tools*. Academic Press, 2006.

[65] Daniel A DeLaurentis, Oleg V Sindiy, and William Stein. Developing sustainable space exploration via a system-of-systems approach. *The American Institute of Aeronautics and Astronautics, San Jose*, 2006.

[66] J. L. Diaz, D. F. Garcia, Kanghee Kim, Chang-Gun Lee, L. Lo Bello, J. M. Lopez, Sang Lyul Min, and O. Mirabella. Stochastic analysis of periodic real-time systems. In *Proc. 23rd IEEE Real-Time Systems Symp. RTSS 2002*, pages 289–300, 2002.

[67] J. L. Diaz, J. M. Lopez, and D. F. Garcia. Probabilistic analysis of the response time in a real time system. In *Proc. of the 1st CARTS Workshop on Advanced Real-Time Technologies,*, 2002.

[68] Jonas Diemer. Informal discussions on predictable complex networks, 2013, 2014. The results of these discussions will partially be published in Jonas Diemer's Phd thesis.

[69] Jonas Diemer and Philip Axer. pyCPA - a pragmatic Python implementation of Compositional Performance Analysis. `http://code.google.com/p/pycpa`.

[70] Jonas Diemer, Philip Axer, and Rolf Ernst. Compositional performance analysis in python with pycpa. In *Proc. of WATERS*. jul 2012.

[71] Jonas Diemer and Rolf Ernst. Back Suction: Service Guarantees for Latency-Sensitive On-Chip Networks. In *The 4th ACM/IEEE International Symposium on Networks-on-Chip*, 2010.

[72] Jonas Diemer, Jonas Rox, and Rolf Ernst. Modeling of ethernet avb networks for worst-case timing analysis. In *Proc. of MATHMOD*, Vienna, Austria, 2 2012.

[73] Jonas Diemer, Jonas Rox, Mircea Negrean, Steffen Stein, and Rolf Ernst. Real-Time Communication Analysis for Networks with Two-Stage Arbitration. In *EMSOFT'11*, October 2011.

[74] Jonas Diemer, Daniel Thiele, and Rolf Ernst. Formal worst-case timing analysis of ethernet topologies with strict-priority and avb switching. In *Proc. of SIES*, 6 2012. Invited Paper.

[75] B. Döbel, H. Härtig, and M. Engel. Operating system support for redundant multithreading. In *Proc. of EMSOFT*, 2012.

[76] Björn Döbel. *Operating System Support for Redundant Multithreading*. Dissertation, TU Dresden, 2014.

[77] Björn Döbel and Hermann Härtig. Who watches the watchmen? protecting operating system reliability mechanisms. In *8th Workshop on Hot Topics in System Dependability (HotDep'12)*, 2012.

[78] Björn Döbel and Hermann Härtig. Where have all the cycles gone? - investigating runtime overheads of osassisted replication. In *GI-Jahrestagung*, pages 2534–2547, 2013.

[79] Björn Döbel, Robert Muschner, and Hermann Härtig. Resource-aware replication on heterogeneous multicores: Challenges and opportunities. In *Workshop on Resource Awareness and Adaptivity in Multi-Core Computing*, RACING'14, 2014.

[80] Mark Dowson. The ariane 5 software failure. *ACM SIGSOFT Software Engineering Notes*, 22(2):84, 1997.

[81] Elena Dubrova. *Fault-Tolerant Design*. Springer, 2013.

[82] J. Dunlop and D.G. Smith. *Telecommunications Engineering, 3rd Edition*. Taylor & Francis, 1994.

[83] C. Ebert and C. Jones. Embedded software: Facts, figures, and future. *Computer*, 42(4):42–52, April 2009.

[84] Leonardo Ecco, Sebastian Tobuschat, Selma Saidi, and Rolf Ernst. A mixed critical memory controller using bank privatization and fixed priority scheduling. In *Proc. of the 20th IEEE International Conference on Real-Time Computing Systems and Applications (RTCSA)*, Chongqing, China, August 2014.

[85] EO Elliott. Estimates of error rates for codes on burst-noise channels. *Bell system technical journal*, 42(5):1977–1997, 1963.

[86] Robert J Elliott, Lakhdar Aggoun, and John B Moore. *Hidden Markov Models*. Springer, 1994.

[87] Frédéric Fauberteau, Serge Midonnet, and Manar Qamhieh. Partitioned scheduling of parallel real-time tasks on multiprocessor systems. *ACM SIGBED Review*, 8(3):28–31, 2011.

[88] Mohamed Fayad and Marshall P. Cline. Aspects of software adaptability. *Commun. ACM*, 39(10):58–59, October 1996.

[89] N. Feiertag, K. Richter, J. Nordlander, and J. Jonsson. A compositional framework for end-to-end path delay calculation of automotive systems under different path semantics. *Work. on Compositional Theory and Technology for Real-Time Embedded Systems CRTS, Barcelona (E)*, 2008.

[90] M.V.S. Fernandes, E.L. Pinto, and M. Grivet. A novel structured markovian model for burst-error channels. In *Wireless and Mobile Communications (ICWMC), 2010 6th International Conference on*, pages 11–15, Sept 2010.

[91] J. Ferreira, A. Oliveira, P. Fonseca, and J. Fonseca. An experiment to assess bit error rate in can. *Proc. of RTN*, 2004.

[92] David Ferry, Jing Li, Mahesh Mahadevan, Kunal Agrawal, Christopher Gill, and Chenyang Lu. A real-time scheduling service for parallel tasks. *In Proc. of RTAS*, 2012.

[93] Christoph Ficek, Nico Feiertag, Kai Richter, and M Jersak. Applying the autosar timing protection to build safe and efficient iso 26262 mixed-criticality systems. Embedded Real-Time Soft-ware Congress (ERTS^2). Toulouse, France, 2012.

[94] N. Fisher, S. Baruah, and T.P. Baker. The partitioned scheduling of sporadic tasks according to static-priorities. In *Proc. on ECRTS*, pages 10 pp. –127, 0-0 2006.

[95] J. Fletcher. An arithmetic checksum for serial transmissions. *Communications, IEEE Transactions on*, 30(1):247–252, January 1982.

[96] José Fonseca, Vincent Nélis, Gurulingesh Raravi, and Luis Miguel Pinho. A multi-dag model for real-time parallel applications with conditional execution. Technical report, 2015.

[97] Freescale. P2040 qoriq communications processor product brief, November 2011.

[98] Jiri Gaisler. A portable and fault-tolerant microprocessor based on the sparc v8 architecture. In *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, pages 409–415. IEEE, 2002.

[99] G. Giannopoulou, N. Stoimenov, P. Huang, and L. Thiele. Scheduling of mixed-criticality applications on resource-sharing multicore systems. In *Embedded Software (EMSOFT), 2013 Proceedings of the International Conference on*, pages 1–15, Sept 2013.

[100] Edgar N Gilbert. Capacity of a burst-noise channel. *Bell system technical journal*, 39(5):1253–1265, 1960.

[101] Alain Girault and Hamoudi Kalla. A novel bicriteria scheduling heuristics providing a guaranteed global system failure rate. *Dependable and Secure Computing, IEEE Transactions on*, 6(4):241–254, 2009.

[102] M. Glass, M. Lukasiewycz, F. Reimann, C. Haubelt, and J. Teich. Symbolic reliability analysis and optimization of ECU networks. In *Proc. of Design, Automation and Test in Europe*, pages 158–163, 2008.

[103] A Golander, Shlomo Weiss, and R. Ronen. Ddmr: Dynamic and scalable dual modular redundancy with short validation intervals. *Computer Architecture Letters*, 7(2):65–68, July 2008.

[104] M. Gonzalez Harbour, J.J. Gutierrez Garcia, J.C. Palencia Gutierrez, and J.M. Drake Moyano. MAST: Modeling and analysis suite for real time applications. In *Real-Time Systems, 13th Euromicro Conference on, 2001.*, pages 125 –134, 2001.

[105] S. Goossens, J. Kuijsten, B. Akesson, and K. Goossens. A reconfigurable real-time sdram controller for mixed time-criticality systems. In *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2013 International Conference on*, pages 1–10, Sept 2013.

[106] Charles Miller Grinstead and James Laurie Snell. *Introduction to probability*. American Mathematical Soc., 1998.

[107] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proc. of WWC*, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

[108] J.J. Gutierrez Garcia, J.C.P. Gutierrez, and M. Gonzalez Harbour. Schedulability analysis of distributed hard real-time systems with multiple-event synchronization. In *Real-Time Systems, 2000. Euromicro RTS 2000. 12th Euromicro Conference on*, pages 15–24, 2000.

[109] Bo Han and Seungjoon Lee. Efficient packet error rate estimation in wireless networks. In *Testbeds and Research Infrastructure for the Development of Networks and Communities, 2007. TridentCom 2007. 3rd International Conference on*, pages 1–9. IEEE, 2007.

[110] Hans A Hansson, Thomas Nolte, Christer Norstrom, and Sasikumar Punnekkat. Integrating reliability and timing analysis of can-based systems. *Industrial Electronics, IEEE Transactions on*, 49(6):1240–1250, 2002.

[111] M.G. Harbour and J.C. Palencia. Response time analysis for tasks scheduled under edf within fixed priorities. In *Real-Time Systems Symposium, 2003. RTSS 2003. 24th IEEE*, pages 200–209, Dec 2003.

[112] Hermann Härtig and Michael Roitzsch. Ten years of research on l4-based real-time systems. In *Proceedings of the 8th Real-Time Linux Workshop*, 2006.

[113] Florian Hartwich. Can with flexible data-rate. In *Proc. of ICC*, Hambach Castle, Germany, 2012.

[114] Peter Hazucha and Christer Svensson. Impact of cmos technology scaling on the atmospheric neutron soft error rate. *Nuclear Science, IEEE Transactions on*, 47(6):2586–2594, 2000.

[115] Reinhold Heckmann and Christian Ferdinand. Worst-case execution time prediction by static program analysis. In *In 18th International Parallel and Distributed Processing Symposium. IEEE Computer Society*, 2004.

[116] Harald Heinecke, Klaus-Peter Schnelle, Helmut Fennel, Jürgen Bortolazzi, Lennart Lundh, Jean Leflour, Jean-Luc Maté, Kenji Nishikawa, and Thomas Scharnhorst. Automotive open system architecture-an industry-wide initiative to manage the complexity of emerging automotive e/e-architectures. *Convergence*, pages 325–332, 2004.

[117] Andreas Heinig, Ingo Korb, Florian Schmoll, Peter Marwedel, and Michael Engel. Fast and low-cost instruction-aware fault injection. In *GI-Jahrestagung*, pages 2548–2561, 2013.

[118] Günther Heling, Jochen Rein, and Patrick Markl. Silentbsw - silent autosar basic software for safety-related ecus. Technical report, vector, 2012.

[119] Martijn Hendriks and Marcel Verhoef. Timed automata based analysis of embedded system architectures. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 8–pp. IEEE, 2006.

[120] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System Level Performance Analysis - The SymTA/S Approach. *IEE Proc. Computers and Digital Techniques*, 152(2):148–166, March 2005.

[121] Rafik Henia and Rolf Ernst. Improved offset-analysis using multiple timing-references. In *Proceedings of the conference on Design, automation and test in Europe: Proceedings*, pages 450–455. European Design and Automation Association, 2006.

[122] John L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, September 2006.

[123] Martin Hillenbrand. *Funktionale Sicherheit nach ISO 26262 in der Konzeptphase der Entwicklung von Elektrik/Elektronik Architekturen von Fahrzeugen*, volume 4. KIT Scientific Publishing, 2011.

[124] M. Holenderski, R.J. Bril, and J.J. Lukkien. Parallel-task scheduling on multiple resources. In *Proc. of ECRTS*, pages 233–244. IEEE, 2012.

[125] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van Der Wijngaart, and T. Mattson. A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pages 108–109, Feb 2010.

[126] Po-Chin Hu, Zhi-Li Zhang, and Mostafa Kaveh. Channel condition arq rate control for real-time wireless video under buffer constraints. 2:124–127 vol.2, Sept 2000.

[127] Jia Huang, Andreas Raabe, Kai Huang, Christian Buckl, and Alois Knoll. A framework for reliability-aware design exploration on mpsoc based systems. *Design Automation for Embedded Systems*, 16(4):189–220, 2012.

[128] Kai Huang, Gang Chen, C. Buckl, and A. Knoll. Conforming the runtime inputs for hard real-time embedded systems. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 430–436, June 2012.

[129] K.H. Huang and J.A. Abraham. Algorithm-based fault tolerance for matrix operations. In *IEEE Transactions on Computers*, 1984.

[130] IEEE. IEEE Standard 802.1AS-2011 - Timing and Synchronization for Time-Sensitive Applications in Bridged Local Area Networks, September 2011.

[131] IEEE. IEEE P802.3bp 1000BASE-T1 PHY Task Force, September 2014.

[132] Infineon. Aurix - safety joins performance, 2014. retrieved 21.05.2014.

[133] Intel Corporation. *Intel® Threading Building Blocks*, October 2011.

[134] International Electrotechnical Commission. Fault tree analysis. - iec 61025, 2006. Edition 2, ISBN 2-8318-8918-9.

[135] International Electrotechnical Commission (IEC). Functional safety of electrical / electronic / programmable electronic safety-related systems ed2.0, 2010.

[136] International Organization for Standardization (ISO). Iso/fdis 26262: Road vehicles – functional safety, 2011.

[137] Internet Engineering Task Force. *RFC 791 Internet Protocol - DARPA Inernet Programm, Protocol Specification*, September 1981.

[138] ISO. Iso 9001, quality managment systems, 2008.

[139] ISO/IEC. Iso/iec 15504, 2012.

[140] V. Izosimov, P. Pop, P. Eles, and Zebo Peng. Synthesis of fault-tolerant embedded systems with checkpointing and replication. In *Proc. of Int. Workshop Electronic Design, Test and Applications*, 2006.

[141] Marek Jersak. *Compositional Performance Analysis for Complex Embedded Applications*. PhD thesis, TU Braunschweig, 2005.

[142] Marek Jersak, Rafik Henia, and Rolf Ernst. Context-aware performance analysis for efficient embedded system design. In *Prof. of DATE*, pages 59–72. Springer, 2008.

[143] Michel C Jeruchim, Philip Balaban, and K Sam Shanmugan. *Simulation of communication systems: modeling, methodology and techniques*. Springer, 2000.

[144] D. Jewett. Integrity s2: a fault-tolerant unix platform. In *Fault-Tolerant Computing, 1991. FTCS-21. Digest of Papers., Twenty-First International Symposium*, pages 512–519, June 1991.

[145] Arshad Jhumka, Martin Hiller, Vilgot Claesson, and Neeraj Suri. On systematic design of globally consistent executable assertions in embedded software. *SIGPLAN Not.*, 37(7):75–84, June 2002.

[146] Changli Jiao, Loren Schwiebert, and Bin Xu. On modeling the packet error statistics in bursty channels. In *Local Computer Networks, 2002. Proceedings. LCN 2002. 27th Annual IEEE Conference on*, pages 534–541. IEEE, 2002.

[147] M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5):390–395, 1986.

[148] Laveen N Kanal and ARK Sastry. Models for channels with memory and their applications to error control. *Proceedings of the IEEE*, 66(7):724–744, 1978.

[149] Charles Keating, Ralph Rogers, Resit Unal, David Dryer, Andres Sousa-Poza, Robert Safford, William Peterson, and Ghaith Rabadi. System of systems engineering. *Engineering Management Journal*, 15(3), 2003.

[150] Achim Klenke. *Probability theory: a comprehensive course*. Springer, 2007.

[151] G.A. Klutke, P.C. Kiessler, and M.A. Wortman. A critical look at the bathtub curve. *Reliability, IEEE Transactions on*, 52(1):125–129, March 2003.

[152] J. C. Knight and N. G. Leveson. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Trans. Softw. Eng.*, 12(1):96–109, January 1986.

[153] J.C. Knight. Safety critical systems: challenges and directions. In *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*, pages 547–550, May 2002.

[154] Steffen Kollman, Victor Pollex, Kilian Kempf, Frank Slomka, Matthias Traub, Torsten Bone, Jurgen Becker, et al. Comparative application of real-time verification methods to an automotive architecture. In *Proceedings of the 18th International Conference on Real-Time and Network Systems*, pages 89–98, 2010.

[155] H. Kopetz and G. Grunsteidl. Ttp - a time-triggered protocol for fault-tolerant real-time systems. In *Fault-Tolerant Computing, 1993. FTCS-23. Digest of Papers., The Twenty-Third International Symposium on*, pages 524–533, June 1993.

[156] Hermann Kopetz, Astrit Ademaj, Petr Grillinger, and Klaus Steinhammer. The time-triggered ethernet (tte) design. In *Object-Oriented Real-Time Distributed Computing, 2005. ISORC 2005. Eighth IEEE International Symposium on*, pages 22–33. IEEE, 2005.

[157] Hermann Kopetz and G. Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126, Jan 2003.

[158] Adam Kostrzewa, Sebastian Tobuschat, Philip Axer, and Rolf Ernst. Supervised sharing of virtual channels in networks-on-chip. In *In Proc. of SIES*, Pisa, Italy, June 2014.

[159] Ondrej Kotaba, Jan Nowotsch, Michael Paulitsch, Stefan M Petters, and H Theilingx. Multicore in real-time systems temporal isolation challenges due to shared resources. In *Proc. of Workshop on Industry-Driven Approaches for Cost-effective Certification of Safety-Critical, Mixed-Criticality Systems (at DATE Conf.)*, 2013.

[160] T. Kranich and M. Berekovic. Noc switch with credit based guaranteed service support qualified for gals systems. In *Digital System Design: Architectures, Methods and Tools (DSD), 2010 13th Euromicro Conference on*, pages 53–59, Sept 2010.

[161] C.M. Krishna and A.D. Singh. Reliability of checkpointed real-time systems using time redundancy. *Reliability, IEEE Transactions on*, 42(3):427 –435, September 1993.

[162] C. LaFrieda, E. Ipek, J. F. Martinez, and R. Manohar. Utilizing dynamically coupled cores to form a resilient chip multiprocessor. In *Proc. of Int. Conf. Dependable Systems and Networks*, pages 317–326, 2007.

[163] K. Lakshmanan, S. Kato, and R. Rajkumar. Scheduling parallel real-time tasks on multi-core processors. In *Proc. of. RTSS*, pages 259 –268, 30 2010-dec. 3 2010.

[164] Karthik Lakshmanan, Ragunathan Rajkumar, and John Lehoczky. Partitioned fixed-priority preemptive scheduling for multi-core processors. In *Proc. of. ECRTS*, pages 239–248, Washington, DC, USA, 2009.

[165] Kai Lampka, Kai Huang, and Jian-Jia Chen. Dynamic counters and the efficient and effective online power management of embedded real-time systems. In *Proceedings of the Seventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '11, pages 267–276, New York, NY, USA, 2011. ACM.

[166] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems (TOCS)*, 10(4):265–310, 1992.

[167] L.B. Le, E. Hossain, and M. Zorzi. Queueing analysis for gbn and sr arq protocols under dynamic radio link adaptation with non-zero feedback delay. *Wireless Communications, IEEE Transactions on*, 6(9):3418–3428, September 2007.

[168] Jean-Yves Le Boudec and Patrick Thiran. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Springer-Verlag, Berlin, Heidelberg, 2001.

[169] Edward A. Lee. Cyber physical systems: Design challenges. Technical report, Center for Hybrid and Embedded Software Systems, EECS, University of California, Berkeley, 2008.

[170] P. A. Lee and T. Anderson. *Fault Tolerance: Principles and Practice*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2nd edition, 1990.

[171] Wen-Shing Lee, DL Grosh, Frank A Tillman, and Chang H Lie. Fault tree analysis, methods, and applications - a review. *Reliability, IEEE Transactions on*, 34(3):194–203, 1985.

[172] J. Lehoczky. Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines. *Proc. 11th RTSS*, pages 201–209, Dec 1990.

[173] Hyung-Taek Lim, Kay Weckemann, and Daniel Herrscher. Performance study of an in-car switched ethernet network without prioritization. In *Proc. of international conference on Communication technologies for vehicles*, pages 165–175, Berlin, Heidelberg, 2011. Springer-Verlag.

[174] ARM Limited. Amba open specifications. online, 2013.

[175] S. Lin and D. J. Costello, Jr. *Error Control Coding: Fundamentals and Applications*. Prentice Hall, Englewood Cliffs, NJ, 2nd edition, 2004.

[176] Shu Lin, D. Costello, and M. Miller. Automatic-repeat-request error-control schemes. *Communications Magazine, IEEE*, 22(12):5–17, 1984.

[177] LIN Steering Group. LIN Specification Package. Rev. 2.1. online, November 2006.

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

[179] Jane W.S. Liu. *Real-time systems*. Prentice Hall, 2000.

[180] Jacques Losq. A highly efficient redundancy scheme: self-purging redundancy. *Computers, IEEE Transactions on*, 100(6):569–578, 1976.

[181] R.E. Lyons and W. Vanderkulk. The use of triple-modular redundancy to improve computer reliability. *IBM Journal of Research and Development*, 6(2):200–209, April 1962.

[182] Aamer Mahmood, Dorothy M Andrews, and Edward J McClusky. *Executable assertions and flight software*. Center for Reliable Computing, Computer Systems Laboratory, Department of Electrical Engineering and Computer Science, Stanford University, 1984.

[183] T.C. Maxino and P.J. Koopman. The effectiveness of checksums for embedded control networks. *Dependable and Secure Computing, IEEE Transactions on*, 6(1):59–72, Jan 2009.

[184] Joseph W McPherson. Reliability challenges for 45nm and beyond. In *Proceedings of the 43rd annual Design Automation Conference*, pages 176–181. ACM, 2006.

[185] Aloysius K Mok and Deji Chen. A multiframe model for real-time tasks. volume 23, pages 635–645. IEEE, 1997.

[186] Boris Motruk, Jonas Diemer, Philip Axer, Rainer Buchty, and Mladen Berekovic. Safe virtual interrupts leveraging distributed shared resources and core-to-core communication on many-core platforms. In *In Proc. of PRDC*, 2013.

[187] Boris Motruk, Jonas Diemer, Rainer Buchty, Rolf Ernst, and Mladen Berekovic. Idamc: A many-core platform with run-time monitoring for mixed-criticality. In *High-Assurance Systems Engineering (HASE), 2012 IEEE 14th International Symposium on*, pages 24–31. IEEE, 2012.

[188] G. P. Mullery. Core - a method for controlled requirement specification. In *Proceedings of the 4th International Conference on Software Engineering*, ICSE '79, pages 126–135, Piscataway, NJ, USA, 1979. IEEE Press.

[189] S. Murali, T. Theocharides, N. Vijaykrishnan, M.J. Irwin, L. Benini, and G. De Micheli. Analysis of error recovery schemes for networks on chips. *Design Test of Computers, IEEE*, 22(5):434–442, Sept 2005.

[190] Nicolas Navet, Y-Q Song, and Françoise Simonot. Worst-case deadline failure probability in real-time applications distributed over controller area network. *Journal of systems Architecture*, 46(7):607–617, 2000.

[191] Mircea Negrean, Simon Schliecker, and Rolf Ernst. Response-Time Analysis of Arbitrarily Activated Tasks in Multiprocessor Systems with Shared Resources. In *Proc. of the Design, Automation and Test in Europe Conference and Exhibition (DATE)*, Nice, France, April 2009.

[192] G. Nelissen, V. Berten, J. Goossens, and D. Milojevic. Techniques optimizing the number of processors to schedule multi-threaded tasks. In *Proc. of ECRTS*, pages 321 –330, july 2012.

[193] M. Neukirchner, M. Negrean, R. Ernst, and T.T. Bone. Response-time analysis of the flexray dynamic segment under consideration of slot-multiplexing. In *Industrial Embedded Systems (SIES), 2012 7th IEEE International Symposium on*, pages 21–30, June 2012.

[194] Moritz Neukirchner. *Establishing Sufficient Temporal Independence Efficiently*. PhD thesis, TU Braunschweig, 2014.

[195] Moritz Neukirchner, Philip Axer, Tobias Michaels, and Rolf Ernst. Monitoring of workload arrival functions for mixed-criticality systems. In *RTSS*, 2013.

[196] Moritz Neukirchner, Tobias Michaels, Philip Axer, Sophie Quinton, and Rolf Ernst. Monitoring arbitrary activation patterns in real-time systems. In *RTSS*, 2012.

[197] Moritz Neukirchner, Sophie Quinton, Tobias Michaels, Philip Axer, and Rolf Ernst. Sensitivity analysis for arbitrary activation patterns in real-time systems. In *Proc. of Design Automation and Test in Europe (DATE)*, mar 2013.

[198] NIST/SEMATECH. Engineering statistics handbook, 2003.

[199] Dionisio de Niz, Karthik Lakshmanan, and Ragunathan Rajkumar. On the scheduling of mixed-criticality real-time task sets. In *Proc. of RTSS*, RTSS '09, pages 291–300, Washington, DC, USA, 2009. IEEE Computer Society.

[200] Christer Norstrom, Anders Wall, and Wang Yi. Timed automata as task models for event-driven systems. In *Real-Time Computing Systems and Applications, 1999. RTCSA'99. Sixth International Conference on*, pages 182–189. IEEE, 1999.

[201] R. Obermaisser, C. El Salloum, B. Huber, and H. Kopetz. The time-triggered system-on-a-chip architecture. In *Proc. of Int. Symp. Industrial Electronics ISIE 2008*, pages 1941–1947, 2008.

[202] Jens-Rainer Ohm and Hans Dieter Lüke. *Signalübertragung*. Springer-Verlag Berlin Heidelberg, 2007.

[203] OMG. Marte specification, 2008.

[204] OMG. Omg systems modeling language (omg sysml), 6 2012.

[205] OpenMP Architecture Review Board. *OpenMP Application Program Interface*, 3.1 edition, July 2011.

[206] Ronald O'Rourke. Coast guard deepwater program: Background, oversight issues, and options for congress. Technical report, Congressional Research Service, 2007.

[207] OSEK VDX. OSEK VDX: open systems and the corresponding interfaces for automotive electronics. `http://www.osek-vdx.org`.

[208] J. C. Palencia and M.G. Harbour. Offset-based response time analysis of distributed systems scheduled under EDF. In *Proc. of ECRTS*, pages 3–12, 2003.

[209] Matthias Pätzold, Matthias Patzold, and Mattias Paetzold. *Mobile fading channels*. John Wiley England, 2002.

[210] Simon Perathoner, Ernesto Wandeler, Lothar Thiele, Arne Hamann, Simon Schliecker, Rafik Henia, Razvan Racu, Rolf Ernst, and Michael González Harbour. Influence of different abstractions on the performance analysis of distributed hard real-time systems. *Design Automation for Embedded Systems*, 13(1-2):27–49, 2009.

[211] Jason A. Poovey, Thomas M. Conte, Markus Levy, and Shay Gal-On. A benchmark characterization of the eembc benchmark suite. *IEEE Micro*, (5):18–29, 2009.

[212] P. Pop, V. Izosimov, P. Eles, and Zebo Peng. Design optimization of time- and cost-constrained fault-tolerant embedded systems with checkpointing and replication. *IEEE Trans. on VLSI*, 17(3):389–402, 2009.

[213] T. Pop, P. Eles, and Zebo Peng. Holistic scheduling and analysis of mixed time/event-triggered distributed embedded systems. In *Hardware/Software Codesign, 2002. CODES 2002. Proceedings of the Tenth International Symposium on*, pages 187–192, 2002.

[214] David Powell, Jean Arlat, Ljerka Beus-Dukic, Andrea Bondavalli, Paolo Coppola, Alessandro Fantechi, Eric Jenn, Christophe Rabéjac, and Andy Wellings. Guards: A generic upgradable architecture for real-time dependable systems. *Parallel and Distributed Systems, IEEE Transactions on*, 10(6):580–599, 1999.

[215] L.L. Pullum. *Software fault tolerance techniques and implementation*. Artech House Publishers, 2001.

[216] S. Punnekkat and A. Burns. Analysis of checkpointing for schedulability of real-time systems. In *Proc. of Int. Workshop Real-Time Computing Systems and Applications*, pages 198–205, 1997.

[217] Alain Pétrissans, Stéphane Krawczyk, Lorenzo Veronesi, Gabriella Cattaneo, Nathalie Feeney, and Cyril Meunier. Design of future embedded systems toward system of systems. Technical report, IDC, 2012.

[218] Manar Qamhieh, Frédéric Fauberteau, Serge Midonnet, et al. Performance analysis for segment stretch transformation of parallel real-time tasks. In *Proceedings of the 5th Junior Researcher Workshop on Real-Time Computing (JRWRTC 2011)*, pages 29–32, 2011.

[219] S. Quinton, M. Hanke, and R. Ernst. Formal analysis of sporadic overload in real-time systems. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, pages 515–520, March 2012.

[220] R. Racu, Li Li, R. Henia, A. Hamann, and R. Ernst. Improved response time analysis of tasks scheduled under preemptive round-robin. In *Proc. 5th IEEE/ACM/IFIP Int Hardware/Software Codesign and System Synthesis (CODES+ISSS) Conf*, pages 179–184, 2007.

[221] Razvan Racu. *Performance Characterization and Sensitivity Analysis of Real-time Embedded Systems*. PhD thesis, TU Braunschweig, 2008.

[222] Razvan Racu, Li Li, Rafik Henia, Arne Hamann, and Rolf Ernst. Improved response time analysis of tasks scheduled under preemptive round-robin. In *Proc. of CODES+ISSS*, pages 179–184, 2007.

[223] Radio Technical Commission for Aeronautics (RTCA). Do-254: Design assurance guidance for airborne electronic hardware, 2000.

[224] Radio Technical Commission for Aeronautics (RTCA). Do-178b: Software considerations in airborne systems and equipment certification, 2001.

[225] R. Rajkumar, L. Sha, and JP Lehoczky. Real-time synchronization protocols for multiprocessors. *Real-Time Systems Symposium, 1988., Proceedings.*, pages 259–269, 1988.

[226] Ragunathan Rajkumar. *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, Norwell, MA, USA, 1991.

[227] Eberle A Rambo, Alexander Tschiene, Jonas Diemer, Leonie Ahrendts, and Rolf Ernst. Failure analysis of a network-on-chip for real-time mixed-critical systems. In *In Proc. of DATE*, pages 1–4, March 2014.

[228] M. Rebaudengo, M.S. Reorda, Marco Torchiano, and M. Violante. Soft-error detection through software fault-tolerance techniques. In *Defect and Fault Tolerance in VLSI Systems, 1999. DFT '99. International Symposium on*, pages 210–218, Nov 1999.

[229] G.A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D.I. August. Swift: software implemented fault tolerance. In *Code Generation and Optimization, 2005. CGO 2005. International Symposium on*, pages 243–254, March 2005.

[230] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, David I. August, and Shubhendu S. Mukherjee. Software-controlled fault tolerance. *ACM Transactions on Architecture and Code Optimization*, 2:366–396, 2005.

[231] Tom Richardson and Ruediger Urbanke. *Modern coding theory*. Cambridge University Press, 2008.

[232] Kai Richter. *Compositional scheduling analysis using standard event models*. PhD thesis, TU Braunschweig, 2005.

[233] Robert Bosch GmbH. CAN Specification version 2.0, 1991. Postfach 30 02 40, D-70442 Stuttgart.

[234] David S. Rosenblum. A practical approach to programming with assertions. *Software Engineering, IEEE Transactions on*, 21(1):19–31, 1995.

[235] Mark V. Rosenker. Safety recommendation a-08-53 through -55. Technical report, National Transportation Safety Board, 2008.

[236] Jonas Rox and Rolf Ernst. Exploiting inter-event stream correlations between output event streams of non-preemptively scheduled tasks. In *Proc. Design, Automation and Test in Europe (DATE 2010)*, mar 2010.

[237] Jonas Rox and Rolf Ernst. Formal timing analysis of full duplex switched based ethernet network architectures. In *SAE World Congress*, volume System Level Architecture Design Tools and Methods (AE318), Detroit, MI, USA, Apr 2010. SAE International.

[238] SAE. As5506 - architecture analysis & design language (aadl)., 9 2012.

[239] A. Saifullah, K. Agrawal, C. Lu, and C. Gill. Multi-core real-time scheduling for generalized parallel task models. In *Proc. of RTSS*, pages 217 –226, 29 2011-dec. 2 2011.

[240] S. Schliecker, M. Negrean, and R. Ernst. Response Time Analysis on Multicore ECUs with Shared Resources. *IEEE Transactions on Industrial Informatics*, 5(4):402–413, November 2009.

[241] S. Schliecker, J. Rox, M. Ivers, and R. Ernst. Providing accurate event models for the analysis of heterogeneous multiprocessor systems. In *Proc. of CODES-ISSS*, pages 185–190, October 2008.

[242] Simon Schliecker. *Performance Analysis of Multiprocessor Real-Time Systems with Shared Resources*. Cuvillier Verlag, 2011.

[243] Simon Schliecker, Jonas Rox, Rafik Henia, Razvan Racu, Arne Hamann, and Rolf Ernst. Formal performance analysis for real-time heterogeneous embedded systems. In *Model-Based Design of Heterogeneous Embedded Systems*, chapter 3, pages 57–92. CRC Press, nov 2009.

[244] M. Schmidt, M Rau, E Dr. Helmig, and B Dr. Bauer. Funktionale sicherheit – umgang mit unabhängigkeit, rechtlichen rahmenbedingungen und haftungsfragen. Technical report, SGS TÜV Saar, 2011.

[245] Martin Schoeberl. A time-triggered network-on-chip. In *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pages 377–382. IEEE, 2007.

[246] Michael Schulze, Philipp Werner, Georg Lukas, and Jörg Kaiser. Afp-an adaptive fragmentation protocol supporting large datagram transmissions. *Journal of Communications*, 6(3):240–248, 2011.

[247] Jörg Schäuffele and Thomas Zurawka. *Automotive Software Engineering*. Vieweg+Teubner, 2010.

[248] M. Sebastian and R. Ernst. Reliability Analysis of Single Bus Communication with Real-Time Requirements. In *Proc. of PRDC*, pages 3–10, 2009.

[249] Maurice Sebastian, Philip Axer, and Rolf Ernst. Utilizing hidden markov models for formal reliability analysis of real-time communication systems with errors. In *Proceeding of the 17th IEEE Pacific Rim International Symposium on Dependable Computing*, dec 2011.

[250] Maurice Sebastian, Philip Axer, Rolf Ernst, Nico Feiertag, and Marek Jersak. Efficient reliability and safety analysis for mixed-criticality embedded systems. *SAE System Level Architecture Design Tools and Methods*, April 2011.

[251] Sol M Shatz and Jia-Ping Wang. Models and algorithms for reliability-oriented task-allocation in redundant distributed-computer systems. *IEEE Transactions on Reliability*, 38(1):16–27, 1989.

[252] Zheng Shi and A. Burns. Real-time communication analysis for on-chip networks with wormhole switching. In *Networks-on-Chip, 2008. NoCS 2008. Second ACM/IEEE International Symposium on*, pages 161–170, 2008.

[253] Abraham Silberschatz, Peter B Galvin, Greg Gagne, and A Silberschatz. *Operating system concepts*, volume 4. Addison-Wesley Reading, 1998.

[254] Jeffrey S Slack. *Finite State Markov Models for Error Bursts on the Land Mobile Satellite Channel*. PhD thesis, Brigham Young University, 1996.

[255] J. C. Smolens, B. T. Gold, J. Kim, B. Falsafi, J. C. Hoe, and A. G. Nowatryk. Fingerprinting: bounding soft-error-detection latency and bandwidth. *IEEE Micro*, 24(6):22–29, 2004.

[256] 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. of Int. Computer Architecture Symp.*, pages 123–134, 2002.

[257] Marco Spuri. Analysis of deadline scheduled real-time systems. 1996. Technical Report.

[258] Steffen Stein. *Allowing Flexibility in Critical Systems: The EPOC Framework*. PhD thesis, TU Braunschweig, 2012.

[259] Steffen Stein, Jonas Diemer, Matthias Ivers, Simon Schliecker, and Rolf Ernst. On the Convergence of the SymTA/S analysis. Technical report, Technische Universität Braunschweig, Germany, Nov. 2008.

[260] Luca Sterpone and Massimo Violante. An analysis of seu effects in embedded operating systems for real-time applications. In *Industrial Electronics, 2007. ISIE 2007. IEEE International Symposium on*, pages 3345–3349. IEEE, 2007.

[261] Thilo Streichert and Matthias Traub. *Elektrik/Elektronik-Architekturen im Kraftfahrzeug*. VDI-Buch. Springer Berlin Heidelberg, 2012.

[262] R Stroph and T Clarke. Dynamic acceptance tests for complex controllers. In *Euromicro Conference, 1998. Proceedings. 24th*, volume 1, pages 411–417. IEEE, 1998.

[263] J.K. Strosnider, J.P. Lehoczky, and Lui Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. volume 44, pages 73–91, Jan 1995.

[264] Andrew Tanenbaum. *Computer Networks*. Prentice Hall Professional Technical Reference, 4th edition, 2002.

[265] R. Teodorescu, J. Nakano, and J. Torrellas. Swich: A prototype for efficient cache-level checkpointing and rollback. *IEEE Micro*, 26(5):28–40, 2006.

[266] George R Terrell. *Mathematical statistics: A unified introduction*. Springer, 1999.

[267] Daniel Thiele, Philip Axer, and Rolf Ernst. Improving formal timing analysis of switched ethernet by exploiting fifo scheduling. In *Design Automation Conference (DAC)*, San Francisco, CA, USA, June 2015.

[268] Daniel Thiele, Philip Axer, Rolf Ernst, Jonas Diemer, and Kai Richter. Cooperating on real-time capable ethernet architecture in vehicles. In *Proc. of Internationaler Kongress Elektronik im Fahrzeug*, oct 2013.

[269] Daniel Thiele, Philip Axer, Rolf Ernst, and Jan R. Seyler. Improving formal timing analysis of switched ethernet by exploiting traffic stream correlations. In *Proc. of CODES+ISSS*, New Delhi, India, October 2014.

[270] Daniel Thiele, Jonas Diemer, Philip Axer, Rolf Ernst, and Jan Seyler. Improved formal worst-case timing analysis of weighted round robin scheduling for ethernet. In *In Proc. of CODES+ISSS*, Montreal, Canada, September 2013.

[271] Daniel Thiele, Johannes Schlatow, Philip Axer, and Rolf Ernst. Formal timing analysis of can-to-ethernet gateway strategies in automotive networks. *Real-Time Systems*, 2015.

[272] L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. In *Proc. IEEE International Symposium on Circuits and Systems (ISCAS)*, volume 4, pages 101–104, 2000.

[273] K. Tindell and A. Burns. Guaranteeing message latencies on control area network (can). In *Proceedings of the 1st International CAN Conference*. Citeseer, 1994.

[274] K. W. Tindell, A. Burns, and A. J. Wellings. An Extendible Approach for Analyzing Fixed Priority Hard Real-Time Tasks. *Real-Time Systems*, 6(2):133–151, 1994.

[275] Yoshiharu Tosaka, Shigeo Satoh, Toru Itakura, Hideo Ehara, Toshimitsu Ueda, Gary A Woffinden, and Stephen A Wender. Measurement and analysis of neutron-induced soft errors in sub-half-micron cmos circuits. *Electron Devices, IEEE Transactions on*, 45(7):1453–1458, 1998.

[276] TU Dresden OS Group. L4/Fiasco.OC microkernel. `http://www.tudos.org/fiasco`, 2012.

[277] Amos Tversky and Daniel Kahneman. Availability: A heuristic for judging frequency and probability. *Cognitive Psychology*, 5(2):207 – 232, 1973.

[278] Spyridon Vassilaras. A cross-layer optimized adaptive modulation and coding scheme for transmission of streaming media over wireless links. *Wireless Networks*, 16(4):903–914, 2010.

[279] R. Vemu and J. A. Abraham. Ceda: control-flow error detection through assertions. In *Proc. 12th IEEE Int. On-Line Testing Symp. IOLTS 2006*, 2006.

[280] Rajesh Venkatasubramanian, John P Hayes, and Brian T Murray. Low-cost on-line fault detection using control flow assertions. In *On-Line Testing Symposium, 2003. IOLTS 2003. 9th IEEE*, pages 137–143. IEEE, 2003.

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

[282] E. Wandeler. *Modular performance analysis and interface-based design for embedded real-time systems*. PhD thesis, SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZURICH, 2006.

[283] E. Wandeler, A. Maxiaguine, and L. Thiele. Performance analysis of greedy shapers in real-time systems. In *Proc. of DATE*, volume 1, pages 6 pp.–, 2006.

[284] Cheng-Xiang Wang and Wen Xu. Packet-level error models for digital wireless channels. In *Communications, 2005. ICC 2005. 2005 IEEE International Conference on*, volume 4, pages 2184–2189. IEEE, 2005.

[285] Don Ward. Avsi's system architecture virtual integration program: Proof of concept demonstrations. Presentation to the INCOSE MBSE Workshop, 2013.

[286] Jiesheng Wei, Anna Thomas, Guanpeng Li, and Karthik Pattabiraman. Quantifying the accuracy of high-level fault injection techniques for hardware faults. In *Dependable Systems and Networks (DSN), 2014 44rd Annual IEEE/IFIP International Conference on*, 2014.

[287] V. Wiels, R. Delmas, D. Doose, P.-L. Garoche, and G. Durrieu J. Cazin. Formal verification of critical aerospace software. *Aerospace Lab*, Issue 4, 2012.

[288] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem-overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7:36:1–36:53, May 2008.

[289] Shiying Xiong, Jeffrey Bokor, Qi Xiang, Philip Fisher, Ian M Dudley, and Paula Rao. Gate line-edge roughness effects in 50-nm bulk mosfet devices. In *SPIE's 27th Annual International Symposium on Microlithography*, pages 733–741. International Society for Optics and Photonics, 2002.

[290] Gulay Yalcin, Osman S Unsal, Adrian Cristal, and Mateo Valero. Fimsim: A fault injection infrastructure for microarchitectural simulators. In *Computer Design (ICCD), 2011 IEEE 29th International Conference on*, pages 431–432. IEEE, 2011.

[291] Y.C. Yeh. Safety critical avionics for the 777 primary flight controls system. In *Digital Avionics Systems, 2001. DASC. 20th Conference*, volume 1, pages 1C2/1–1C2/11 vol.1, Oct 2001.

[292] Ti-Yen Yen and Wayne Wolf. Performance estimation for real-time distributed embedded systems. *Parallel and Distributed Systems, IEEE Transactions on*, 9(11):1125–1136, 1998.

[293] Patrick Meumeu Yomsi, Dominique Bertrand, Nicolas Navet, and RI Davis. Controller area network (can): Response time analysis with offsets. In *Factory Communication Systems (WFCS), 2012 9th IEEE International Workshop on*, pages 43–52. IEEE, 2012.

[294] M. Yoshimoto, T. Takine, Y. Takahashi, and T. Hasegawa. Waiting time and queue length distributions for go-back-n and selective-repeat arq protocols. volume 41, pages 1687–1693, 1993.

[295] Homayoun Yousefi'zadeh and Hamid Jafarkhani. Statistical guarantee of qos in communication networks with temporally correlated loss. In *Global Telecommunications Conference, 2003. GLOBECOM'03. IEEE*, volume 7, pages 4039–4043. IEEE, 2003.

[296] Sergio Yovine. Model checking timed automata. In *In European Educational Forum: School on Embedded Systems*, pages 114–152. Springer-Verlag, 1998.

[297] Fengxiang Zhang and Alan Burns. Schedulability analysis for real-time systems with edf scheduling. *Computers, IEEE Transactions on*, 58(9):1250–1258, 2009.

[298] Dakai Zhu and Hakan Aydin. Reliability-aware energy management for periodic real-time tasks. *Computers, IEEE Transactions on*, 58(10):1382–1397, 2009.

[299] Dimitrios Ziakas, Allen Baum, Robert A Maddox, and Robert J Safranek. Intel® quickpath interconnect architectural features supporting scalable system architectures. In *High Performance Interconnects (HOTI), 2010 IEEE 18th Annual Symposium on*, pages 1–6. IEEE, 2010.