Dipl.-Ing. Tobias Peter Scheipel, BSc

Advances in Dynamic and Reconfigurable Embedded Systems Design

DOCTORAL THESIS
to achieve the university degree of
Doktor der technischen Wissenschaften

submitted to
Graz University of Technology

Supervisor
Univ.-Prof. Dipl.-Inf. Univ. Dr. rer. nat. Marcel Baunach
Institute of Technical Informatics
Embedded Automotive Systems Group

Graz, December 2022

Don't adventures ever have an end? I suppose not.
Someone else always has to carry on the story.

– Bilbo Baggins, by J.R.R. Tolkien

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material that has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present doctoral thesis.

In several parts of the dissertation, text passages from my publications are used without explicit citation. The list of publications can be found in Chapter 7[p95].

Date

Signature

Acknowledgements

A dissertation, even if written by a single individual, can never come into existence without the inspiration, support, and influence of others. Not only do people in the academic and professional environment have an important role to play, but impressions from private life also determine the time during which a dissertation is written. This is the place where I would like to explicitly mention some of the people who have accompanied me along the adventure and express my heartfelt thanks to them.

First, I would like to mention my supervisor **Prof. Marcel Baunach**, with whom I could already write my bachelor's and master's thesis. Thanks for your continuous support and ideas in research, teaching, and other matters. Your feedback, ideas, and many great discussions shaped and molded this dissertation into what it is today. The same applies to all of my dear colleagues of the Embedded Automotive Systems working group and the Institute of Technical Informatics at Graz University of Technology. I want to mention explicitly and thank **Renata Martins Gomes**, **Leandro Batista Ribeiro**, and **Maja Malenko**[1] with whom I spent most of my time in the same 28.49 m^2. The same applies to **Michael Stocker** and **Bernhard Großwindhager**, who were a vital part of the crew.

Next, I would like to thank all my friends. Special mention goes to **Paul Baumgartner**, **Christoph Weixlberger**, **Richard Močnik**, **Claudia Schönhart**, and **Anna Lallitsch**, with whom I spent a lot of leisure time – also during the challenging times of the pandemic.

Finally, I would like to thank my family, who has always supported me. Without you, there would not be a document I call my dissertation today.

This dissertation is for you!

Graz, December 2022
Tobias Peter Scheipel

[1] In the chronological order in which I met them.

Abstract

Embedded systems are electronic devices consisting of hardware and software that are embedded in a particular context. While the software can commonly be changed through updates, the hardware is usually hardwired. The hardware consists of various components, including one or more microcontroller units to run the software on a fixed circuit board. However, not only are the wires and components on the printed circuit board fixed, but the microcontroller also contains a fixed integrated circuit, making it impossible to change any part of the hardware.

As sustainability and dependability play an increasingly important role in future embedded systems, hardware must also become more flexible. Thus, the possibility of subsequent hardware modifications is particularly important: On the one hand, manufacturers want to hit the market as quickly as possible while still having the option of correcting hardware errors after deployment. On the other hand, they want to minimize maintenance efforts while producing less e-waste.

As hardware updates are not possible nowadays, this thesis proposes two main concepts to bring more flexibility and sustainability to embedded hardware design: an automatic process to generate the system stack below the application software dynamically from the application requirements, and a runtime reconfigurable soft processor that even allows adoption of its logic for bug fixing and adding new functionality.

In the first concept, starting with the application software requirements, the remaining parts of the embedded system are automatically generated. This generation includes the modules of the printed circuit board and their interconnections, but also parts of the operating system layer. The focus is on the selection and composition of hardware modules and the generation of the printed circuit boards.

In the second concept, on reconfigurable soft processors, the main research focus is on the design of a flexible microcontroller architecture. The architecture's internals must be designed to be partially reconfigurable at runtime, and the actual reconfiguration must even be possible from within the system. Hence, the microcontroller and the operating system must be carefully co-designed to facilitate the reconfiguration process based on the system state. Therefore, another focus is on the conceptual design of related operating system features and the performance measurement of individual hardware and software components.

These concepts significantly improve flexibility in embedded system development and long-term maintenance. This way, embedded systems can stay operational for longer, thus improving overall sustainability as less e-waste is produced.

Kurzfassung

Eingebettete Systeme sind elektronische Geräte, die aus Hardware und Software bestehen und in einen bestimmten Kontext eingebettet sind. Während die Software normalerweise mittels Updates nach Belieben geändert werden kann, ist die Hardware in der Regel fest verdrahtet. Die Hardware besteht aus verschiedenen Komponenten, darunter ein oder mehrere Mikrocontroller zur Ausführung der Software auf einer festen Leiterplatte. Aber nicht nur die Leiterbahnen und Komponenten auf der Leiterplatte sind unveränderlich, sondern auch der Mikrocontroller enthält einen festen integrierten Schaltkreis, so dass Änderungen an der Hardware unmöglich sind.

Da Nachhaltigkeit und Zuverlässigkeit in zukünftigen eingebetteten Systemen eine immer wichtigere Rolle spielen, muss auch die Hardware flexibler werden. Daher ist die Möglichkeit von nachträglichen Hardwareänderungen besonders wichtig: Einerseits wollen die Hersteller neue Geräte so schnell wie möglich auf den Markt bringen und gleichzeitig die Möglichkeit haben, Hardwarefehler nachträglich zu korrigieren. Andererseits soll der Wartungsaufwand minimiert und gleichzeitig weniger Elektroschrott produziert werden.

Da Hardwareupdates momentan noch nicht möglich ist, werden in dieser Arbeit zwei Hauptkonzepte vorgeschlagen, um mehr Flexibilität und Nachhaltigkeit in das Design eingebetteter Hardware zu bringen: ein automatischer Prozess, um den Systemstack unterhalb der Anwendungssoftware dynamisch aus den Anwendungsanforderungen zu generieren, und ein zur Laufzeit rekonfigurierbarer Softprozessor, der sogar die Anpassung seiner eigenen Logik zur Fehlerbehebung und zum Hinzufügen neuer Funktionen ermöglicht.

Beim ersten Teilkonzept werden, ausgehend von den Anforderungen der Anwendungssoftware die restlichen Teile des eingebetteten Systems automatisch generiert. Diese Generierung umfasst sowohl die Module der Leiterplatte und deren Verbindungen, als auch Teile der Betriebssystemschicht. Der Schwerpunkt liegt dabei auf der Auswahl und Zusammenstellung von Hardwaremodulen und der Leiterplattengenerierung.

Im zweiten Konzept, rekonfigurierbaren Softprozessoren, liegt der Schwerpunkt der Forschung auf dem Entwurf einer flexiblen Mikrocontrollerarchitektur, deren interne Logik so gestaltet sein soll, dass sie zur Laufzeit teilweise rekonfigurierbar ist. Die eigentliche Rekonfiguration muss dabei sogar von innerhalb des Systems aus möglich sein. Daher müssen der Mikrocontroller und das Betriebssystem sorgfältig zusammen entworfen werden, um den Rekonfigurationsprozess auf der Grundlage des Systemzustands zu unterstützen. Weitere Schwerpunkte liegen deshalb auf der Konzeption von zugehörigen Betriebssystemfunktionen und der Leistungsmessung einzelner Hardware- und Softwarekomponenten.

Die vorgestellten Konzepte verbessern die Flexibilität bei der Entwicklung und langfristigen Wartung eingebetteter Systeme erheblich. Auf diese Weise können eingebettete Systeme länger in Betrieb bleiben, was die Nachhaltigkeit insgesamt verbessert, da weniger Elektroschrott produziert wird.

Contents

1 Introduction **1**
- 1.1 Problem Sources and Motivation . . . 2
- 1.2 Research Questions and Contributions of this Thesis . . . 4
- 1.3 Thesis Structure and Organization . . . 7

2 Background and Terminology **9**
- 2.1 Embedded Systems . . . 9
- 2.2 Application Software . . . 10
- 2.3 Operating Systems . . . 11
  - 2.3.1 General Operating System Concepts . . . 11
  - 2.3.2 Peripheral Access . . . 12
- 2.4 Hardware . . . 13
  - 2.4.1 Printed Circuit Boards . . . 13
  - 2.4.2 Processors and Microcontrollers . . . 14
  - 2.4.3 Field-Programmable Gate Arrays . . . 16

3 Related Work **19**
- 3.1 Dynamic Electronics Generation . . . 19
  - 3.1.1 Automatic Hardware Generation . . . 19
  - 3.1.2 Embedded Systems Prototyping and Requirements Engineering . . . 20
  - 3.1.3 Annotations and Design Space Exploration . . . 22
  - 3.1.4 Constraint Solving . . . 24
- 3.2 Reconfigurable Systems Design . . . 24
  - 3.2.1 Flexible Microcontroller Architectures . . . 24
  - 3.2.2 Partial Reconfiguration at Runtime . . . 26
  - 3.2.3 Reconfigurable Computing Platforms . . . 28
  - 3.2.4 Operating System Support of Flexible Hardware and Reconfigurable Computing Platforms . . . 30
  - 3.2.5 Performance Monitoring . . . 32

4 Dynamic Electronics Generation **35**
- 4.1 Main Idea of *papageno*X . . . 36
  - 4.1.1 Application Software Analysis . . . 37
  - 4.1.2 Reconfigurable Logic Generation . . . 38
  - 4.1.3 Printed Circuit Board Generation . . . 38

4.2 System Description Format . . . 39
4.2.1 Module Definitions and Design Blocks . . . 40
4.2.2 Interface Definitions . . . 41
4.2.3 System Definition . . . 42
4.3 Generation of System Configurations from Requirements . . . 43
4.3.1 Mathematical Definitions and Terminology . . . 43
4.3.2 Implementation . . . 44
4.4 Generation of Schematics and Board Layouts from System Configurations . . . 46
4.4.1 Connection Establishment and Pin Assignment . . . 46
4.4.2 File Generation for Schematics and Board Layouts . . . 47
4.5 Evaluation . . . 48
4.5.1 Proof of Concept . . . 48
4.5.2 Evaluation of the File Generation Process for Schematics and Board Layouts . . . 55

5 Reconfigurable Systems Design 59
5.1 Main Idea and Architecture Overview . . . 60
5.2 *more*MCU Design . . . 61
5.2.1 Pipeline Design to enable flexible Instruction Sets . . . 62
5.2.2 Microcontroller Design to enable flexible Peripherals . . . 63
5.2.3 Partitioning and Partial Reconfiguration at Runtime . . . 63
5.2.4 Data Structures and Bitstream Repository . . . 64
5.2.5 Runtime Reconfiguration Controller . . . 65
5.2.6 Performance Monitoring Unit . . . 66
5.3 Application Software Interface . . . 71
5.4 Extended Operating System Concepts . . . 72
5.4.1 Flexible Instruction Set Handling . . . 73
5.4.2 Runtime Reconfiguration Handling . . . 74
5.5 Evaluation . . . 75
5.5.1 Flexible ISA Performance Gain . . . 75
5.5.2 Runtime Reconfiguration Evaluation . . . 84
5.5.3 General Observations and Limitations . . . 87

6 Conclusion and Future Work 91

7 Publications 95
7.1 E-I: *papageno*PCB: An Automated Printed Circuit Board Generation Approach for Embedded Systems Prototyping . . . 99
7.2 E-II: Towards an Automated Printed Circuit Board Generation Concept for Embedded Systems . . . 107
7.3 E-III: *papageno*X: Generation of Electronics and Logic for Embedded Systems from Application Software . . . 119
7.4 E-IV: *papageno*ReQ: Generation of Embedded Systems from Application Code Requirements . . . 127

7.5 R-I: System-Aware Performance Monitoring Unit for RISC-V Architectures . . 135
7.6 R-II: A Hardware/Software Concept for Partial Logic Updates of Embedded Soft Processors at Runtime 145
7.7 R-III: *Smart*OS: An OS Architecture for Sustainable Embedded Systems 155
7.8 R-IV: *more*MCU: Runtime-reconfigurable RISC-V Platform for Sustainable Embedded Systems 167

Bibliography **177**

List of Figures **192**

List of Tables **193**

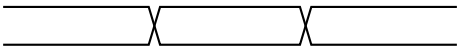
List of Listings **195**

List of Abbreviations **197**

CHAPTER 1

Introduction

This chapter introduces modern embedded systems and explains the sources of problems and errors in the design of such systems. The motivation for this thesis and the related research questions are derived from these problems. From each individual research question, the contributions of this thesis are explicitly stated. The chapter ends with an explanation of the overall thesis structure and organization.



In our hectic and fast-paced world, where efficiency and performance are increasingly important, small, often invisible electronic devices are playing an ever more significant role: electronic systems embedded in their respective contexts, whether they are integrated into washing machines, cars or other home appliances, are already facilitating and automating a wide range of life situations for us humans. Although they are often invisible or concealed from the eye, they are ubiquitous, and it is almost inconceivable to imagine our daily lives without them. As the evolution of the state of technology advances at breakneck speed, the level of complexity of the respective embedded systems is also constantly increasing. This fact is also reflected in an overall increase in the difficulty of the associated design processes. Since these systems can become very complex, it is vital to incorporate the right amount of expertise during the development process without unnecessarily involving too many or too few engineers (and their different opinions). The industry's high time and financial pressures to develop a flawless device add to this need.

"Too many cooks spoil the broth."
– unknown

From these observations, common problems and sources of errors during the development and runtime of embedded systems can be derived, and this dissertation contributes to their solution. The main focus here is, on the one hand, to design the systems as dynamically as possible so that they can meet all the requirements and, on the other hand, to ensure at design time that the systems support options for subsequent modifications in the delivered production device. As a result, embedded systems will stay operational for longer and thus be used more sustainably. All these capabilities must be maintained, while the process must also remain inexpensive and flexible to react adequately to any unexpected changes during design and development.

1.1 Problem Sources and Motivation

Recent developments like automated driving, where most innovations in modern cars are based on software and electronics [1], or the enormous growth of the Internet of Things (IoT) [2] lead to a demand for a faster design process with reduced error probabilities within the embedded systems domain. The established design processes for these embedded systems are bottom-up processes (e.g., the V-model [3] in the automotive industry). This means that based on a hardware Printed Circuit Board (PCB), consisting of a computing platform and other electronic components (cf. Figure 1.1), the implementation of the software is initiated in a second step. If shortcomings in hardware are discovered during software development, they are first tried to be circumvented in software since changing the hardware post-design is both expensive and time-consuming. Shortcomings may be found in electronics (e.g., inadequately selected components, bus overload) but also in the logic of the computing platform (e.g., missing instructions in the processor). A significant problem appears when the requirements for the final system change during development, although the hardware design and production have already been completed. It is even worse for the logic in computing platforms: Usually, it can only be adapted during the next revision of the design. If bugs occur in the logic design, their effects on the system's behavior can only be countered with software workarounds. This applies both to the design time and to the maintenance in the field. Since this is not always so easy, devices are usually replaced and disposed of very quickly when a new and updated device revision hits the market. With an expected total of 25 billion interconnected electronic systems in the IoT by 2030 [4], this throw-away culture is becoming an increasing problem, as e-waste already poses a significant negative impact on the environment.

In summary, this thesis proposes solutions for problems from two different fields that arise from the established bottom-up design process, where "software follows hardware". Mostly, the (i) electronics are designed before the software development is started, using electronic components, including (ii) computing platforms with hardwired logic. As motivated before, both categories of hardware – the electronics and the logic – are only rarely changed during the development process. The following paragraphs define the specific sources of problems in both fields and motivate how to tackle the resulting problems.

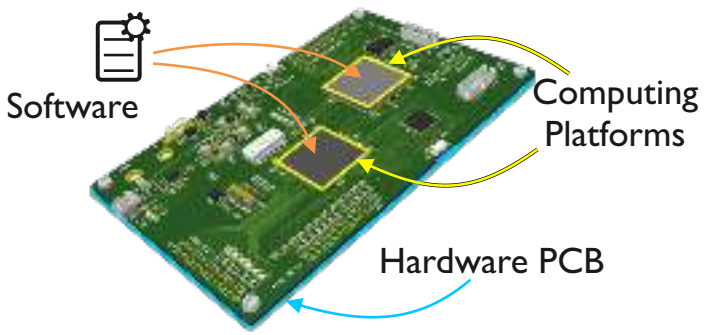


Figure 1.1: A high-level view of the parts of an embedded system.

(i) Towards dynamic system generation: While the system design process usually starts with defining the requirements for the final system, software development only begins once the final or prototype hardware exists. Hence, the hardware components and modules are composed before or in parallel with the development of the software. If new requirements arise or change during software development, this can lead to the inevitability of a hardware redesign. This inevitability is mainly due to inadequacies or limitations concerning specific new functionalities of the existing hardware system.

(ii) Towards reconfigurable systems design: When it comes to the usage of computing platforms, the usual embedded systems design process does not involve redesigning such platforms. Hardwired computing platforms in the form of Integrated Circuits (ICs) are commonly used (e.g., Microcontroller Units – MCUs or Central Processing Units – CPUs), making software the only way to work around deficiencies in the hardware. Supposing the software-based approach fails to achieve the desired success, switching to an alternative or improved platform is inevitable. This necessity is because the internal logic of the computing platform usually cannot be changed after production – it is hardwired. The switch is even necessary when only small parts of the logic become obsolete or contain bugs. There are many well-known examples of such hardware obsolescences in the past: The cryptographic hash algorithm SHA1 [5] was marked outdated and deprecated in 2011. This left the hardware acceleration unit, built into many processors at the time (e.g., in Intel CPUs [6]), useless and obsolete. While the hardwired processors were still in use, this part of the logic was also present but was not supposed to be used; however, it still took up power and space on the chip. A few years later, further security-critical bugs were found in computing cores and caused quite a stir in the media. These are the Meltdown [7] security vulnerabilities and the Spectre [8] attack scenarios for several different computing architectures, which allow unauthorized access to memory areas that should be protected and isolated. The provided patches within operating systems lead to a significant decrease in the computing performance of the systems. As dependability [9] becomes increasingly important in developing embedded systems – whether in the IoT, automotive, or other application areas – it is expected that hardwired computing hardware that cannot be modified after deployment will become less viable in the future. Nevertheless, not only bugs in the hardware can be a problem. Legal requirements [10] often change during a device's lifetime, which also impacts the algorithms and acceleration units to be used. However, modification of logic within non-reconfigurable computing platforms (including their acceleration units) is impossible. It gets even more complicated when rebootless hardware and logic updates at runtime are required due to safety regulations [11], which makes the look into reconfigurable embedded systems inevitable [12].

All these sources of problems motivate why it is exciting to look into the design process of embedded systems and introduce novel concepts toward more flexibility. While the software was and continues to be flexible anyways, this is not the case when it comes to hardware.

1.2 Research Questions and Contributions of this Thesis

Out of the described problems in (i) and (ii), we can derive two main Research Questions (RQs) for this thesis alongside their sub-questions. The individual contributions in this dissertation provide answers to these questions. Figure 1.2 shows the alignment along the embedded systems stack (cf. Section 2.1[p9]).

Research Question RQ 1

How can application software be used to automatically generate all the remaining parts of an embedded system?

RQ 1 deals with the first introduced problem source (i). Contrary to the established bottom-up design process, where the system software follows or adjusts to its hardware, we propose a novel top-down approach. To do so, we invert the embedded systems design process towards "hardware follows software". This inversion means that the system development starts with the Application Software (ASW) rather than the hardware. The remainder of the embedded system is generated automatically from this software and its requirements. This generation includes the entire hardware and parts of the low-level software and the Operating System (OS). The arrows for RQ 1 in Figure 1.2 sketch that the roots of the requirements lie within the application layer, out of which the OS layer and the hardware layer (electronics and logic) are derived. The main research focus, however, is on the dynamic generation of module-based electronics. This generation involves a thorough requirements analysis of the ASW followed by the derivation of the remaining system parts. This derivation contains a selection of a suitable computing platform, the surrounding hardware modules, and the interconnection between them to form a PCB that matches all requirements.

Further, the derivation step must compile the basic software modules, including the drivers for all hardware modules. The selection process also involves answering the questions *"What information about functional and non-functional requirements is needed to generate PCBs from ASW automatically?"* and *"How can we derive the needed operating system software features from ASW?"*. In addition, another question arises: *"How can we extract application-specific logic and automatically map it to reconfigurable computing platforms?"*. This question remains unanswered within this context, but it directly leads to the next RQ 2.

The main contribution toward answering RQ 1 presented by this thesis is a holistic system generation concept that uses application requirements and dynamically generates module-based electronics hardware. It can be roughly split into requirements matching and PCB generation. The name for this concept is ***papageno***X, and Chapter 4[p35] named "Dynamic Electronics Generation" shows a detailed explanation with references to the corresponding publications.

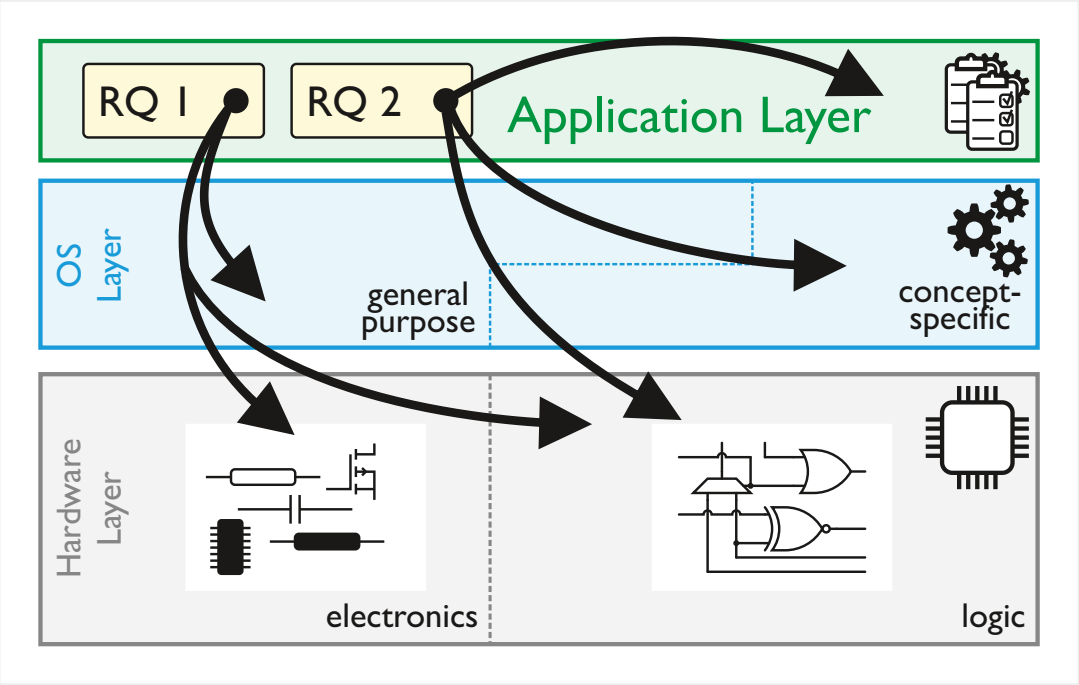


Figure 1.2: The two main research questions, out of which context they originate, and what they mainly address.

Research Question RQ 2

How can an embedded computing platform change its own logic at runtime?

RQ 2 addresses the second source of problems (ii), which originates in hardwired computing platform internals. In the proposed concept, we aim to change the design of embedded computing platforms as we anticipate that future processor cores will need to support post-production logic updates, for example, to extend the system's life by adapting to new software requirements or simply to fix bugs. If the hardware can be updated similarly to the software, this would also contribute to a more dynamic and sustainable system design, as devices can be maintained for much longer, reducing the environmental impact. This update capability also leads to more efficient use of hardware since keeping all mutually exclusive resources in reserve is not necessary. It must be ensured that while the logic modification process is running, the execution of the other parts of the system is not affected. Therefore, this process must be performed parallel to the rest of the system to guarantee a rebootless completion. The concept for the approach features not only hardware design principles but also embedded OS design methodologies for the support of flexible logic within the underlying processor. The subsequent research questions in this field are: *"How can we design future embedded systems so that they can make more sustainable use of their underlying hardware logic resources for long-term operation?"* and *"How can future OS architectures support flexible and changing software and hardware to keep systems operational in the long run?"*.

The main contribution towards answering RQ 2 is an approach for modifying embedded soft microcontrollers implemented on Field-Programmable Gate Arrays (FPGAs) at runtime so that software-like flexibility can also be achieved in hardware. With our approach, user-defined instructions can be inserted directly into the pipeline of the processor core executing the software itself. The required update is done on the fly, without the need to halt or reset the system. These updates are managed and supported by a specifically tailored OS running on the reconfigurable computing platform. This includes triggering and managing the hardware modification process and supporting flexible hardware when executing ASW. The software can still be compiled with standard, unmodified compilers, and the operating system takes care of the modifications introduced into the system. The name of the designed MCU is *more*MCU, and Chapter 5[p59] named "Reconfigurable Systems Design" shows a detailed explanation with references to the corresponding publications.

While such a process inversion towards "hardware follows software" requires a high degree of acceptance in the industry, this thesis shows the possibilities of such a paradigm shift. This applies to both hardware fields: (i) the electronics and (ii) the logic design.

1.3 Thesis Structure and Organization

The remainder of the thesis is structured as follows:

Chapter 2: Background and Terminology[p9] starts with some general background information that is necessary to understand the content of this thesis correctly.

The subsequent **Chapter 3: Related Work**[p19] gives an overview and summarizes existing related work, structured in every different aspect that is relevant to the thesis.

In the **Chapters 4: Dynamic Electronics Generation**[p35] and **5: Reconfigurable Systems Design**[p59], the two main topics of this thesis are motivated, described, and evaluated. These two chapters form the main content and the scientific contributions of this dissertation.

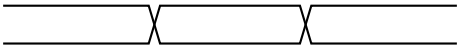
The thesis is concluded and summarized in **Chapter 6: Conclusion and Future Work**[p91], where also an outlook to a possible future continuation of the concepts is illustrated.

The scientific full text of the papers that were published over the course of this dissertation can be found in **Chapter 7: Publications**[p95]. The thesis is based on the publications shown in here. Additionally, related publications, theses and projects are also listed in this chapter.

CHAPTER 2

Background and Terminology

This chapter features all the necessary information to understand the approach within this thesis. It gives an overview of the basic terminology used within the embedded systems domain and introduces hardware concepts for printed circuit boards and reconfigurable logic, respectively.



We use many different terms when it comes to electronic systems that are invisible to the user (in contrast to, e.g., personal computers or smartphones). These systems are commonly used to perceive environment (e.g., temperature, humidity) and interact with their environment by controlling various actuators (e.g., motors, heaters, magnets). According to Marwedel [13], "Embedded systems are information processing systems embedded into enclosing products". This definition already gives an idea of the context in which those systems exist. "Embedded Systems" often become a "Cyber-Physical System" (CPS) through interaction with the environment (e.g., through mechanical components). The physical context is emphasized explicitly within the term CPS, as the interaction between the mostly digital device and the analog environment is explicitly stated. When we move on to a globally interconnected network, where a vast number of heterogeneous embedded systems collaborate on the Internet to reach a common goal [14], they become the "Internet of Things" (IoT). The fact that intelligent, networked, electronic devices have become part of everyday life is often referred to as "ubiquitous computing" or "pervasive computing".

This thesis focuses on "Embedded Systems", which resemble those depicted in Figure 2.1. These embedded systems can then be used to build CPSs, various systems of systems, and are also applicable in the IoT.

2.1 Embedded Systems

Embedded systems are usually designed for a particular target application in a dedicated domain and are therefore mostly not suitable for general-purpose applications [15]. They commonly have limited resources and power supply, small form factors, and little computing

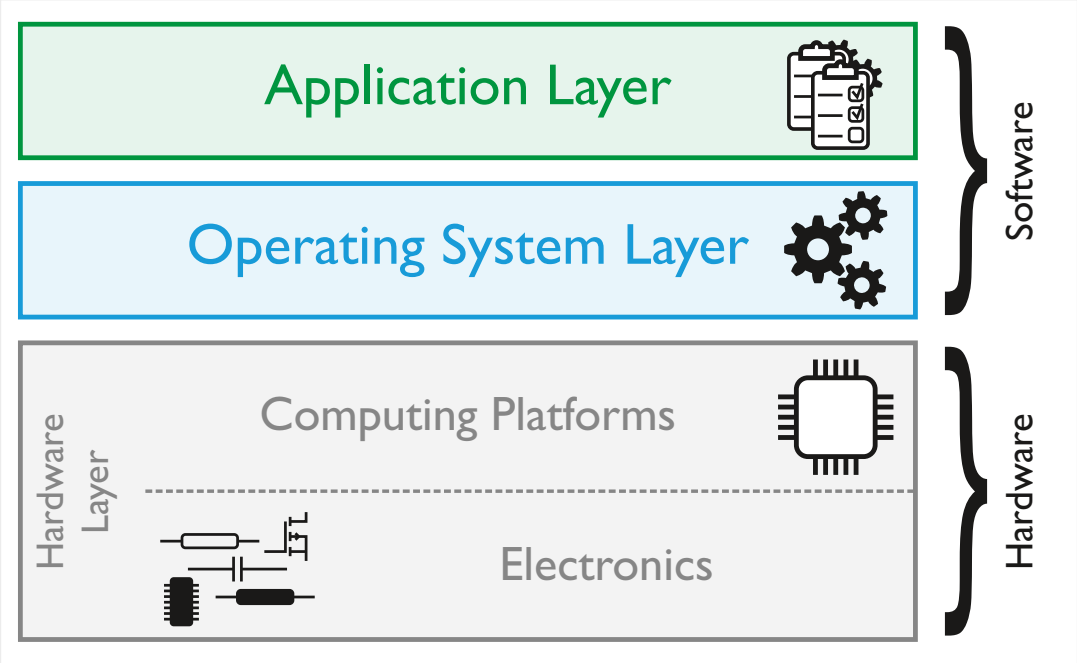


Figure 2.1: An embedded system's main parts depicted as a stack.

power and must still meet the application requirements. Hence, embedded systems must be designed meticulously to make them as efficient as possible in their application context.

When it comes to their architecture, embedded systems most commonly feature a structure similar to the one depicted in Figure 2.1. Throughout the thesis, this is referenced to as the (embedded) systems stack. The bottom-most part or layer is the hardware. It consists of an electronics PCB where all the necessary electronic components are soldered onto. A common PCB features a power supply, one or more computing platforms (e.g., an MCU, an FPGA), memories, and peripherals to interact with the surrounding environment including the necessary supporting components (e.g., capacitors, crystals, resistors). The upper two layers both consist of software, divided into the operating system or system software layer and the application layer on top. It must be said that the software layering is optional, as so-called "bare metal" programs can also be executed. However, it is highly recommended for state-of-the-art embedded systems design [16].

The following sections explain all three layers concerning their relevance to the thesis.

2.2 Application Software

The Application Software (ASW) layer is the software part, as the name implies, in which the main part of the application use case is implemented. Thus, the ASW is the layer originating the main requirements, which the other layers must also contribute to fulfill. While the rest of the system is more generically usable, the application software defines what purpose the system should serve.

Several software tasks within this layer implement different parts of the application. The tasks can collaborate, share resources, and communicate with each other by using features of the OS and its hardware abstractions. In the end, the application software is fully managed by the underlying OS layer [16]. As a software development engineer, writing ASW is most likely the point to start developing functionality for an embedded system.

2.3 Operating Systems

Just below the application software layer of an embedded system, the Operating System (OS) is situated. Although it is an optional part of the system, it brings many benefits. It provides the ASW with a multitude of functionalities and abstractions and is also responsible for managing the underlying hardware. This thesis uses the term OS to subsume all system management software provided to the application engineer. Following this definition for the OS layer, it consists of a kernel and several libraries, services, and drivers; hence, it includes all software that connects the ASW and the hardware layer. The automotive industry, amongst others, calls this layer Basic Software (BSW), whereas terms like System Software or Middleware are also widely used (cf. [16, 13, 17]) – depending on the domain. Another widely used naming for this kind of software within embedded systems is Real-Time Operating System (RTOS), which explicitly marks OSs with inherent features to handle the real-time requirements of a system. An embedded OS, therefore, can, but it does not have to be an RTOS.

2.3.1 General Operating System Concepts

Most embedded systems feature an OS with a microkernel or a monolithic kernel [18], where the main functionality of the OS is situated. This piece of software is responsible for managing all tasks, resources, memories, and peripherals, as well as the interaction between them and the hardware. Depending on the implementation, it also takes care of timing, scheduling, and other low-level software duties. Since all concepts in this thesis use *Smart*OS (cf. [19], Sections 5.4[p72] and 7.7[p155]), the most relevant general concepts of an embedded operating system with respect to this thesis are explained below using this example. General explanations [16] are also provided when necessary.

Time: *Smart*OS amongst others has an inherent notion of time. An internal timeline driven by a hardware clock allows tasks to, e.g., define deadlines or simply sleep for a specified amount of time. Temporal semantics is provided through kernel functions and can be used in an absolute (e.g., points in time) or a relative (e.g., intervals) way. This inherent notion of time also helps when implementing real-time critical tasks and their interaction.

Tasks: A task is an entity managed by the OS that holds the functionality of the ASW. The OS, however, must take care of managing all tasks within the system and their interaction. Within this thesis, every task has its individual stack and Task Control Block (TCB), whereas there is one additional stack for the kernel. Each task has a unique priority to implement

proper interleaving and synchronization between tasks. If available on the target computing platform, tasks always run in the least privileged mode (often called user mode). In contrast, the kernel is executed in the most privileged mode (kernel mode or machine mode).

Interrupt Handling: As our systems are embedded in a physical environment, they must also be able to react to certain external events. To handle these, the system must support so-called interrupts. Sources of interrupts can be, e.g., an external digital signal applied to a pin of the MCU, an internal timer that elapses, the processor detects illegal behavior of the software, or even a software-triggered interrupt via special instructions. Whenever the system is interrupted, the OS has to deal with it. The kernel provides an Interrupt Service Routine (ISR) as software entry points for every possible Interrupt Request (IRQ) of the system. Mainly, soft interrupts are used in an OS-based system, where the actual interrupt is caught by the kernel and then forwarded to whatever is registered to the interrupt source.

System Calls: Whenever the ASW needs to access privileged functionality of the OS, system calls or syscalls are provided as a well-defined interface between the ASW and the OS. On every call of a syscall, the kernel is entered. Now, the OS can provide otherwise isolated functionality in a managed matter.

Resources: To mutually exclude the access to hardware and software resources within a set of tasks, the OS provides a resource concept. These resources can be used to exclusively allocate shared objects like memory areas, on-chip or off-chip peripherals, driver data structures, and so on. Using these data structures and specific resource management protocols like the Highest Locker Protocol (HLP), the Priority Ceiling Protocol (PCP), or the Priority Inheritance Protocol (PIP), the OS can manage prioritized accesses to the shared objects in a well-defined way. So, whenever a task needs access to a shared object, it requests a resource via a syscall. If the resource is available, it will be assigned to the task. Otherwise (due to occupancy by another task), it must wait until the resource is free again, as every resource can only be occupied by one task at a time.

Events: Tasks often need to communicate with other tasks in the system. For this kind of Inter-Task-Communication (ITC), events come in handy. Events are a concept within the OS that can be used to signal other tasks in case, e.g., data processing is finished. To do so, one task sets an event and thus informs one or a queue of already waiting tasks about the signal. This concept can also be used to map IRQs to the responsible tasks.

2.3.2 Peripheral Access

Access from a task to on-chip or off-chip peripherals of the computing platform is usually done via services or libraries that are included in the OS and expose a well-defined Application Programming Interface (API) to the ASW. Hence, the ASW does not directly interact with any of the Input/Output (I/O) devices but only utilizes the OS's functions for shared and arbitrated access [15].

In embedded systems, access to the peripheral devices is typically done via memory-mapped registers of the I/O device. This means that certain registers within a device are mapped into the system's memory layout at a predefined location. Since the addresses are known to the OS, interaction can take place via these addresses. Independent of the addressing, it is irrelevant how the peripheral devices are connected to the core. They can be connected via, e.g., a complex interconnect network, a bus system with shared wires, a Network on Chip (NoC), or directly via the data interface. More complex interconnection protocols are mainly used to enable access to off-chip peripherals. Some examples of very common types of connection are, e.g., Wishbone [20], Advanced eXtensible Interface (AXI) [21], Inter-Integrated Circuit (I^2C) [22], Serial Peripheral Interface (SPI) [23]. In addition, interfaces like Ethernet [24], Universal Serial Bus (USB) [25] or Universal Asynchronous Receiver Transmitter (UART) [26] can be used to connect to the outer world (e.g., within the IoT). However, it must always be ensured that the memory-mapped address space is uniform and transparent for all participants, regardless of how they are connected.

2.4 Hardware

Embedded systems hardware usually consists of a PCB with at least one computing platform and other electronic components. The computing platform again consists of one or more cores within an MCU or a CPU. The corresponding silicon device containing these cores is either a general-purpose IC, an Application-Specific Integrated Circuit (ASIC), a System on Chip (SoC) or a dedicated device. It is also possible to use an FPGA that is configured to contain a softcore MCU or CPU. This computing platform (as well as all other components) can be powered by different kinds of power sources (e.g., battery, hard-wired power supply, on-system energy harvesting) and has wired connections to a set of off-chip peripherals. The term "hardware" in this sense can either mean dedicated electric components on a PCB or logic within a FPGA. Figure 2.2 shows the different levels of granularity within embedded hardware.

2.4.1 Printed Circuit Boards

Printed Circuit Boards (PCBs) form the basis of every embedded system. It is the physical board where all the electrical and electronic devices are soldered on to build the hardware of the system [16]. All the electrical wires are "printed" in conducting lines onto one or several stacked layers of a non-conducting material (usually fiberglass). The design of a PCB starts with creating a schematic of all the components necessary for the application. This is mostly done in a modular approach, where dedicated groups like the power supply, the input and output devices, the computing device or platform, its memories, and interconnects between the devices are composed into the system. This way, the PCB can be easily clustered in certain areas which are designed individually (cf. right side of Figure 2.2). As interfaces between components, the mentioned connection types in Section 2.3.2 are used.

After creating the schematic, a board layout must be designed, where more emphasis is on non-functional characteristics such as signal propagation times or voltage drops on components. The actual copper lines between the components must be created suitably for

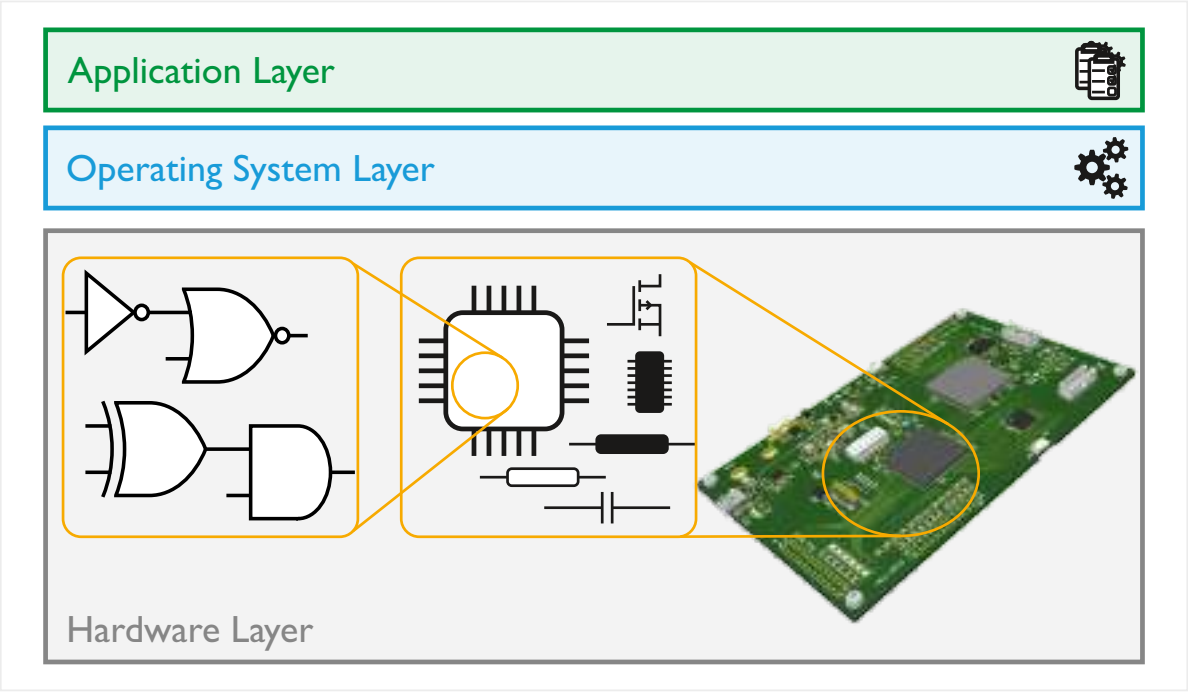


Figure 2.2: Hardware module granularity from the PCB on the right to a logic gate leftmost.

the system to work correctly. The difference between the schematic, board layout, and the actual PCB can be seen in Figure 2.3.

2.4.2 Processors and Microcontrollers

When it comes to the computing platform of an embedded system (cf. middle and left of Figure 2.2), CPUs or MCUs are used. This term not only names the logic responsible for the computing but is commonly also used synonymously for the physical device placed within an IC. An MCU consists of one or more processor cores, (embedded) memories, and peripherals, whereas the term CPU only refers to the computing "brain" of a system. For embedded systems, integrated CPUs that are integrated within an MCUs on a SoC are the most common computing platforms, whereas stand-alone CPUs mainly are utilized in desktop or high-performance computing. The difference between MCU and stand-alone CPU is depicted in Figure 2.4.

Within modern systems, the registers of a processor are usually 32 Bits wide, but especially for low-power systems, 16 Bit machines can still be found [15]. A more recent development directs towards 64 Bits, especially for systems that need a lot of memory or high performance. Most cores also feature a pipelining concept, where different stages of the execution of an instruction can be parallelized. Slightly modified versions of pipelines and their stages can be found in different architectures in the field.

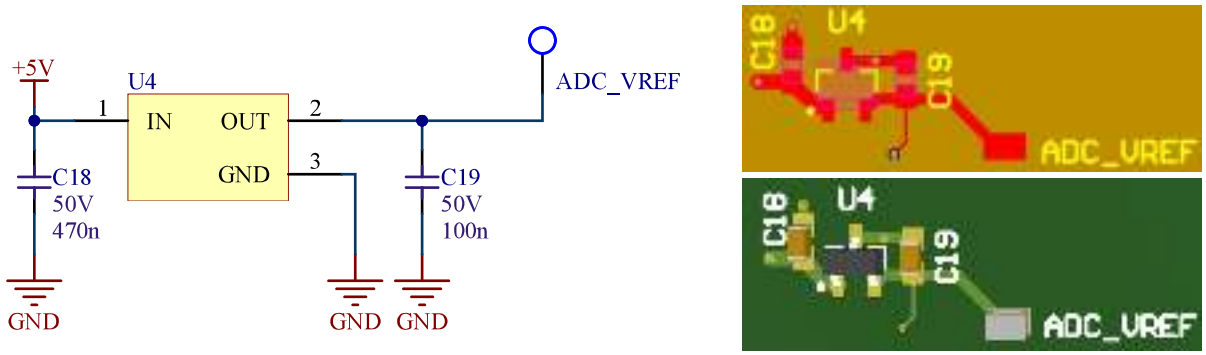


Figure 2.3: Schematic plan (left), board layout (top right) and 3D rendering (bottom right) of the PCB of a power supply.

RISC vs. CISC

When it comes to the Instruction Set Architecture (ISA), cores can be roughly differentiated between Complex Instruction Set Computing (CISC) and Reduced Instruction Set Computing (RISC) machines. Widely used CISC machines are the Intel or AMD x86 processors [27], whereas popular RISC processor architecture examples are ARM [28], RISC-V [29], PowerPC [30], MIPS [31], Hitachi SH-4 [32], PA-RISC [33] or SPARCv8 [34]. While the actual differentiation is very difficult to make, the most prominent historical difference between the two types of ISA is the instruction length. While typically RISC uses a fixed instruction length (e.g., 32 Bits for the standard RISC-V core instructions), CISC instructions can be of variable length [15]. The latter enables a more efficient code size, while the first can be optimized for speed [13]. Within a CISC architecture, there also exist complex operations that can be composed out of several instructions, whereas this is not the case for RISC [16]. Nowadays, RISC processors are becoming more and more complex, while CISC processor designs aim at being more efficient. Thus, the differentiation between RISC and CISC becomes less and less relevant.

Softcore Microcontrollers

Softcore MCUs are MCUs that are fully designed in a Hardware Description Language (HDL) and can be implemented utilizing hardware logic synthesis. They usually consist of a set of (Intellectual Property – IP) libraries alongside source files and can thus be easily altered for the specific use case [35]. While typically hardwired MCUs are tuned for performance, softcore MCUs enable modifications in a straightforward way. The result of the logic synthesis can then be programmed onto some piece of reconfigurable logic like an FPGA. This approach is widely used in prototyping but can also be found in production devices.

RISC-V

RISC-V is an open and free ISA defined in 2010 by the University of California, Berkeley, by Waterman et al. in [29] and [36] and is inspired by established RISC architectures as mentioned

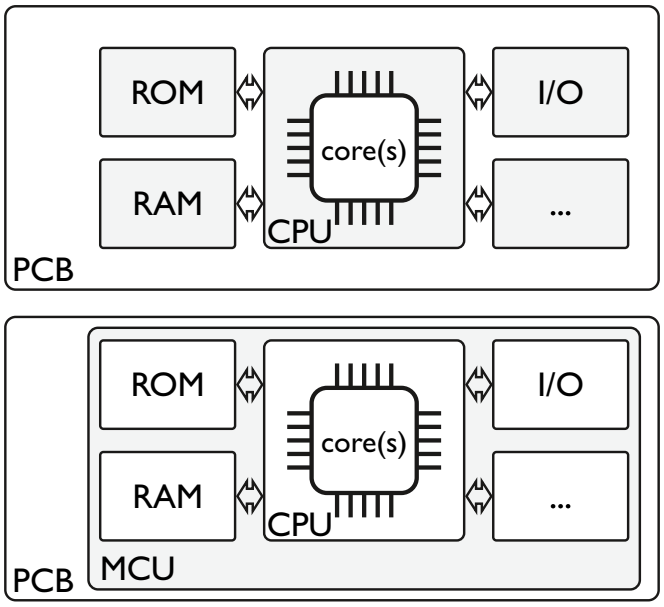


Figure 2.4: The difference between a stand-alone CPU and a CPU integrated into an MCU, both with typical surroundings.

above. However, unlike many proprietary instruction sets, RISC-V is freely available in terms of use and modification. The ISA is designed to be easily extensible, and its standard can be implemented in different variants with different extensions. A CPU or MCU implementing the RISC-V ISA must at least contain 32 basic registers alongside some standard instructions to fulfill the minimum specification. Many different implementations exist, both hardcore and softcore CPUs and MCUs. Amongst the variants, proprietary cores can be found alongside open-source RISC-V designs in industry and academia. Examples to be mentioned are the Rocket [37] from UC Berkeley, the CV32E40P [38] from the OpenHW Group[1] or the SiFive[2] processor family.

2.4.3 Field-Programmable Gate Arrays

An FPGA is an electronic device featuring generic logic cells that can be reconfigured and rewired at will. This way, application-specific logic can be designed without having to produce an ASIC device. In principle, it features three different groups of components being programmable logic elements, I/O blocks, and the interconnect between those [39]. In a modern FPGA (e.g., from Xilinx [40]) the logic elements feature Configurable Logic Blocks (CLBs) containing mainly Look-Up Tables (LUTs) and Flip-Flops (FFs), Digital Signal Processor (DSP) cells and Block Random-Access Memory (BRAM) cells. LUT and DSP cells can process data (combinatorial logic), whereas FF and BRAM primitives can store data (sequential logic). The logic cells are grouped in certain clock domains and arranged in a column hierarchy.

[1] `https://www.openhwgroup.org/`
[2] `https://www.sifive.com/`

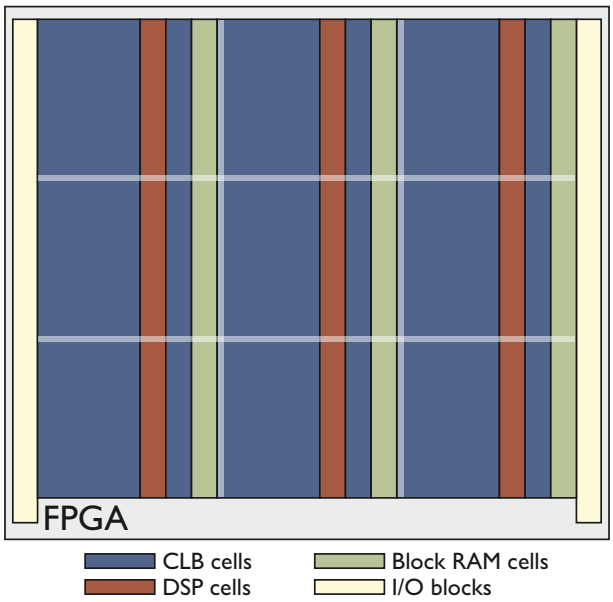


Figure 2.5: The floorplan of an FPGA. The grid indicates clock domains.

Figure 2.5 shows the general layout of an FPGA. This layout makes interconnecting of the logic elements [41] easier. Therefore, the programmable features include the "function" of each logic element and the interconnection network of the FPGA. The logic configuration itself is stored in the configuration memory of the device. This way, different implementations of logic can be easily synthesized out of HDL code and programmed onto the logic device. Most vendors enable this functionality by flashing a proprietary bitstream onto the configuration memory.

When comparing FPGAs with ASICs, it is essential to keep in mind that FPGAs usually have both higher power consumption, a lower effective clock, and a larger package. This is due to an FPGA's flexibility, as the connections between the individual logic gates are only configured afterward. Contrary, for ASICs, these characteristics can be optimized; the logic circuits, however, are hardwired. Theoretically, any logic circuit can be realized with both, but producing an ASIC is significantly more expensive and takes longer. Therefore, ASICs are usually only produced when they are needed in large numbers. With an FPGA, on the other hand, the circuit can be implemented almost immediately and can also be changed again. However, this flexibility implies the disadvantages mentioned above.

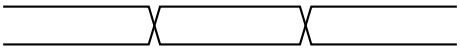
Dynamic Partial Reconfiguration

If a bitstream does not update the entire configuration of an FPGA, we talk about partial reconfiguration. The terms used for features like this are Dynamic Partial Reconfiguration (DPR) or Dynamic Function eXchange (DFX) [42]. The partial reconfiguration can either be done offline by accessing the FPGA externally or online (if available) by accessing the internal reconfiguration port (e.g., Internal Configuration Access Port – ICAP for Xilinx devices). This way, specific parts of the system can be changed on-the-fly without influencing the operation of the rest of the system [43].

CHAPTER 3

Related Work

This chapter features scientific and industrial contributions that are related to the work conducted in this thesis. The contributions are organized by relevance to the two research questions and their relevant aspects and provide a detailed insight into certain concepts and approaches. Furthermore, it is stated to what extent this thesis differs from the works in the literature.



Since this thesis can be roughly split into two topics, the related work in academia and industry will also be divided into Section 3.1 “Dynamic Electronics Generation”, and Section 3.2 “Reconfigurable Systems Design”. The subsections reflect the relevant aspects contributing to finding answers to the research questions, and these aspects are subsequently used in Chapters 4[p35] and 5[p59]. The publications in Chapter 7[p95] and their corresponding related work sections form the basis for the following text.

3.1 Dynamic Electronics Generation

The first topic dealing with dynamic electronics generation has many influences from a diverse research field. Beginning with approaches towards the automatic generation of hardware, prototyping platforms and requirements engineering for embedded systems must be considered. For the composition of electronic devices from ASW, source code annotation approaches are of interest. Design space exploration and the subsequent requirement constraint solving also highly influenced this work.

3.1.1 Automatic Hardware Generation

For automatic hardware generation, holistic approaches are scarce. To the best of our knowledge, no concept automatically generates hardware from ASW. However, there exist some loosely related papers indicated in the following. Although these concepts do not generate hardware from software, they have influenced the present thesis. Swinkels and Hafer [44]

show a method that generates schematics using expert systems in a way that makes them more easily human-readable. They partition the schematics generation process into smaller problems, including component ordering, placement, and routing. Visual representation of crossings, feedback loops, and the direction of the signal flows are essential to make the schematics more pleasant to the human eye.

Singh et al. [45] show an approach where the schematics of a circuit can be generated automatically by extracting connectivity data from the netlists of the system. The devices are categorized into groups, on which placement constraints are placed subsequently. Schematics can be generated from the resulting tree instances that look similar to analog schematics.

The devicetree [46] data structure within Linux-based systems [47] also uses tree-based data structures. The Linux kernel uses its intrinsic data structure to handle hardware components like processors, memories, internal or external buses, as well as peripherals of the system. This way, a holistic system description can be generated easily to enable the usage of one compiled OS kernel with several slightly different hardware configurations in, e.g., SoCs.

Relevance and Distinction: These works influenced this thesis regarding how the interconnections between and the grouping of hardware modules are done. As the main distinction, this thesis uses a module-based system description that is generated out of ASW requirements rather than only reflecting the already produced hardware, as in [46]. While the mentioned approaches use existing network information for their generation, we create the connections between the modules while composing the hardware systems. They neither extract requirements from nor are they even aware of the ASW. A pleasant visual representation of plans or tree-based schematics generation is not a part of the present thesis.

3.1.2 Embedded Systems Prototyping and Requirements Engineering

Prototyping plays a significant role in developing automotive Electronic Control Units (ECUs). With *rCube2*, Kouba et al. [48] show a rapid prototyping platform for a 1-D gas dynamics model for a real engine control strategy. The hardware system consists of two independent Infineon TC1797 [49] MCUs and several peripheral co-processors to control the ignition and injection of the engine. An isolated shared memory area facilitates communication between the dedicated controllers.

Similarly, Eichberger et al. [50] show a generic engine control platform based on Infineon's single-core TC1796 [51] or TC1798 [52] MCUs. This platform's flexibility lies in its adaptability to different combustion engine systems. However, electric and hybrid powertrain management can also be implemented. The PCB for the prototyping platform is a mainboard that can be equipped with several different, application-specific extension boards. These extension boards feature automotive-grade ASICs controlled by the central MCU. This approach allows near-to-production prototyping to minimize the Time To Market (TTM) when implementing control algorithms for new engine systems.

Relevance and Distinction: While many prototyping platforms for embedded systems exist in academia and industry, hardware changes always require a full or partial redesign of PCBs whenever requirements change. The main problem with those solutions is that even though they offer high performance and come with complete toolchains, their hardware is often very different from a series device. If a prototype is then to be transferred to a series product, the hardware must be completely redesigned. This thesis shows an approach where the hardware is automatically generated from the ASW requirements; thus, a redesign (in this case: the actual hardware generation) happens implicitly and automatically during the development phase.

The requirements for the system are usually derived in an initial Requirements Engineering (RE) process, where both functional and non-functional requirements are extracted and processed. Works like Broy et al. [53] and Sikora et al. [54] explain the fundamental approaches to RE, focusing on industrial needs in the latter. They focus on using natural language and the corresponding requirements models, support for high system complexity, quality assurance, and the interrelation of RE and the architectural design for embedded system requirements to derive a good understanding of the envisioned system. The RE process of large international companies from different embedded system domain branches was compared and investigated.

Pereira et al. [55] take it a step further and introduce a metamodel for an RE process to define requirements systematically. This model defines concepts and relationships to be considered when designing an embedded system. Their systematic literature research found lacks in the established RE tools for embedded systems that highly influenced the creation of their metamodel, serving as a resource model for the significant characteristics within the embedded systems domain. Out of this metamodel, they created a sketch for a guided process to derive the requirements of embedded systems.

Models for embedded systems can also be described in an object-oriented way. For hybrid systems, Elmqvist et al. [56] show an object-oriented modeling approach, where a mixture of continuous and discrete components can be modeled. The particular focus is on the reusability of the models independently to their respective environments.

Another concept derived from a software engineering structure – namely design patterns – is the requirements pattern work presented by Konrad and Cheng [57]. They explore how object-oriented models can represent requirement patterns within embedded systems. To do so, structural and behavioral information form requirement patterns to facilitate reusability in embedded systems design. Within their requirement patterns, they focus on functional and non-functional requirements with the possibility to include design and implementation details that can also have architectural and design patterns as well as a precise specification of the distinction between hardware and software parts of the system. Their subsequent work [58] extends this concept by adding a model checker. Within their object analysis pattern, they can check whether the constraints of the derived requirements are satisfied in the context of a given pattern. These constraints include a formal specification of the respective properties. The object analysis pattern is applied in the analysis phase of embedded systems development and aims to guide the construction of a system's conceptual model.

Regarding the composition of embedded systems, Stankovic et al. [59] show an exciting approach applied in real-time systems within the avionics industry. They try to ensure that hidden dependencies of domain-specific components do not cause failures while the non-functional properties are still met. Their tailored toolkit provides dependency checks for these components and their corresponding properties within the distributed embedded systems domain.

Relevance and Distinction: While some works focus on a specific field within the embedded systems domain, we generalize our concept. Contrary to the shown works, we extract the requirements of the ASW by forcing the engineer to provide more information already at development time. While the needed hardware module must not be selected within the ASW, the parameters of the used function calls require particular behavior (e.g., a specific data rate requirement). In this regard, the requirements extraction is influenced by component and module-based RE and the composition of the embedded system. While the approaches in the literature stop after extracting requirements or models, we use the ASW requirements to select suitable modules and generate the corresponding hardware.

3.1.3 Annotations and Design Space Exploration

The requirements mentioned above and the respective RE process must be derived from some aspects of the corresponding system. We have looked at some work towards annotations and design space exploration to facilitate this.

Annotations are often used to include information about the functional and non-functional properties of the embedded system directly into its source code. Examples that deal with Worst-Case Execution Time (WCET) are shown by Schnerr et al. [60] and Schommer et al. [61]. Both of the works use back-annotations to include WCETs in the system. Since the WCET is not computable, the first work utilizes a cycle-accurate simulation of the system's software with static execution time analysis. Based on the results, timing information can be integrated into the system description to get more accurate timing information for non-statically analyzable system parts. These parts include data dependencies as well as branch prediction and caching.

The second work [61] focuses on safety-critical and real-time systems, where WCET plays a vital role. The authors create an annotation mechanism that allows annotating C source code and reason about C variables instead of using code addresses and processor registers. The corresponding toolchain passes the annotation into the executable without external annotation files to tightly link the annotations and the binary.

Chakravarty et al. [62] show a work towards performance and power modeling. They propose a host-compiled simulation with a close to a cycle-accurate estimation of timing and energy characteristics using an architecture description language and corresponding models. The accuracy is very high due to their path-dependent, pairwise and simulation-based characterization at the basic block level.

Relevance and Distinction: The approaches that use annotations rely on estimates and measurements that are back-annotated into the system to allow the developer to understand better how the system will perform in a particular use case. The works presented have influenced the present thesis because some of the requirements for the system are also extracted from the source code this way. In addition, the proposed approach in this thesis uses an enhanced function parameter concept to extract more requirements that the development engineer can provide. The final set of requirements is then used to compose system configurations and generate the hardware for the embedded system.

After the extraction of the requirements, design space exploration techniques can be applied to find components and modules with particular properties that match these requirements. Pimentel [63] gives an introduction to design space exploration within the embedded systems design domain. Within this work, the basic concept where, e.g., performance, energy consumption, and cost are considered simultaneously is illustrated in detail. This concept helps to find an optimal solution for the embedded system's components using decision variables. Out of the design space, potential solutions are generated and optimized towards these variables with a Pareto optimization. Part of this work also includes searching the design space as well as different workload models.

Künzli [64] shows other general design space exploration techniques, where the main emphasis is on the decision-making in different abstraction levels of the system. As the design space and the number of derived alternative solutions is usually quite large, different strategies for finding reasonable solutions are discussed exhaustively in this work.

Herrera et al. [65] propose a methodology for design space exploration that combines model-driven engineering approaches with electronic system-level design. The set of possible solutions is graphically displayed and can be used to generate the application components' code automatically. The corresponding design framework is modular and easily extensible, improving model reusability.

Saxena and Karsai [66] present a survey of several different automated design space exploration works. They created a meta-framework for design space exploration that can be meta-programmed to work for a class of design space exploration problems independent of the domain. The reusability and flexibility of the framework are of utmost importance here.

Relevance and Distinction: Design space exploration greatly influenced designing the system generation concept for this dissertation concerning how hardware modules are selected according to requirements. While the present work is no design space exploration framework per se, many ideas from the literature were incorporated. The ASW requirements span a selection space over suitable hardware modules in this work, and these hardware modules are subsequently used to generate suitable hardware for the embedded system.

3.1.4 Constraint Solving

Constraint solving is of interest when matching general constraints or requirements to properties of individual components or modules to composite holistic systems. MiniZinc [67] is a language that enables the modeling of constraint satisfaction and optimization problems independent of the solver. The language allows modeling close to the mathematical formula of the underlying problem. Picat [68] is a similar language that integrates features like logic and functional programming for combinatorial search problems, including constraint programming.

Regarding concepts specifically for embedded systems design, Kuchcinski [69] shows that constraint programming is helpful. This work discusses the main advantages and drawbacks of such an approach alongside its essential features. The previously mentioned systems are included here as well as satisfiability solvers or integer programming solvers, which are not limited to constraint programming.

Relevance and Distinction: The work in this dissertation did not use the systems mentioned above in any form. However, ideas and algorithms are adapted and integrated into the concept when the ASW requirements are matched to the properties of hardware modules. Since the proposed approach already follows a similar direction, constraint programming could optimize the module selection process in this work.

3.2 Reconfigurable Systems Design

The second topic that deals with reconfigurable systems design is situated more in the processor architecture and reconfigurable logic domain. To design a system that is reconfigurable at runtime, several aspects are of interest in combination: design of flexible MCU architectures, partial reconfiguration of logic at runtime and its application on reconfigurable computing platforms. Additionally, all aspects must be supported by the OS as well. For all of these aspects, it is beneficial to have a monitoring capability for the behavior of the system's components and their performance in order to obtain detailed runtime information. The combination of the aspects is extensively explained in Chapter 5[p59].

3.2.1 Flexible Microcontroller Architectures

The envisioned MCU architecture for this dissertation is a softcore MCU architecture for embedded systems. Hence, design decisions and features from related flexible MCU architectures are interesting. While most modern designs in this field are based on the RISC-V ISA [29], different architectures also exist. To learn more about this topic, see Section 2.4.2[p15]. The architectures shown here are designed to be easily extensible at **design time**. In this sense, the extensibility covers not only on-chip peripherals but also internal structures of the core, like pipeline stages or other parts of the logic.

While most RISC-V implementations are already designed to be flexible at design time, some are worth mentioning. The PULP platform [70] developed at ETH Zürich has three different core implementations: the CV32E40P [38] (formerly RI5CY [71]), the Ibex [72], and the CVA6 [73]. Several of PULP's MCU platforms incorporate one or more of these cores alongside different on-chip peripherals. From those MCUs, some are adapted by the OpenHW Group (cf. Section 2.4.2[p15]). The CV32E40P core is the basis for the MCU developed for the concept proposed in this thesis. Apart from general-purpose MCUs, many application-specific cores and controllers exist in academia and industry.

Verbeure [74] shows the capabilities of VexRiscv that was developed by Charles Papon. It is an FPGA-optimized RISC-V core implemented in SpinalHDL[1]. The corresponding language features can already be used to facilitate modifications very easily. It is fully object-oriented and does not follow the standard practices by which logic is usually designed. However, this language does not introduce overhead, as everything is translated into Register Transfer Level (RTL).

With QuantumRISC, Alkim et al. [75] show a compact custom RISC-V ISA extension to support post-quantum cryptography implementation with hardware/software co-design approaches. Their main application use case is resource-constrained processors in devices such as smartcards, based on the previously shown VexRiscV. Their custom instructions drastically reduce the code size without introducing runtime penalties while only creating a small logic overhead. These instructions are excellent examples that show how hardware implementations of specific algorithms can improve the system performance without being too costly concerning hardware.

Menon et al. [76] show another RISC-V implementation that introduces security features into their RISC-V core. Particular emphasis is placed on an enhanced Application Binary Interface (ABI) for software layers to protect sensitive data throughout the system. While the authors claim to reduce metadata storage overhead, the solution only adds a minor amount of hardware resource overhead.

Relevance and Distinction: All the shown works illustrated how softcore MCUs are used for specific application domains and aim at modularity, but only at design time. Once the MCU is deployed, no modifications of the logic are possible. Contrary to this, the proposed MCU of this thesis has a flexible architecture tailored explicitly for runtime reconfigurability – with all the known problems and advantages. The proposed concept uses a generic way to extend the ISA at runtime with minimal modifications to the pipeline. Thus, the management overhead can be kept very low, simplifying subsequent hardware updates.

[1] `https://spinalhdl.github.io/SpinalDoc-RTD`

3.2.2 Partial Reconfiguration at Runtime

When moving from flexible MCU architectures towards (runtime-) reconfigurable computing platforms, a part of the hardware must handle reconfigurability at runtime. Modern FPGAs have an internal configuration port (cf. Section 2.4.3[p16]) for this. While this port usually only provides the actual technical implementation of the reconfiguration at runtime, a separate controller must provide the corresponding data for the reconfiguration process.

Pezzarossa et al. [12] provide an experimental investigation of whether DPR can be used beneficially within embedded and real-time systems. They extensively research how the hot-swapping of hardware accelerators affects the system's WCET and hardware resource utilization whenever the application needs them. Their results show that using DPR for computationally intensive tasks can lead to more efficient use of the FPGA at a comparable computational performance compared to having many hardware accelerators.

A similar survey concerning partial reconfiguration was carried out by Papadimitriou et al. [77]. They, however, elaborate on the performance of the factors that influence the actual reconfiguration speed within a system. They conducted experiments with an FPGA-based architecture to obtain a cost model for partial reconfiguration. Their approach helps to decide if introducing partial reconfiguration to the system can be beneficial, with an estimation of the overall overhead it would introduce.

When looking at DPR controllers that enable a system to reconfigure itself at runtime, Cardona and Ferrer [79] provide a fundamental approach. They improve how to use the ICAP of modern Xilinx FPGAs [40]. They connect their AC_ICAP controller to a MicroBlaze [90] MCU via a Processor Local Bus (PLB) [91], a Fast Simplex Link (FSL) [92], or an AXI [93] bus connection. Their work focuses on achieving a speedup compared to the XPS_HWICAP [87] while still having a small hardware footprint.

Table 3.1: Comparison of different reconfiguration controllers in literature.

Reconfiguration Controller	Interface	Software	Domain
ZyCAP [78]	AXI	drv	academic
AC_ICAP [79]	PLB/direct	-	academic
RT-ICAP [80]	direct	drv	academic
RV-CAP [81]	AXI	drv	academic
D²PR [82]	direct	-	academic
Vipin et al. [83]	PLB/AXI	-	academic
ICAP-I [84]	direct	-	academic
DPRM [85]	PLB/XPS/dir.	-	academic
AXI_HWICAP [86]	AXI	-	industrial
XPS_HWICAP [87]	XPS	-	industrial
PRC [88]	AXI	-	industrial
***more*MCU-RC** [89]	WB/direct	OS	academic

With RT_ICAP, Pezzarossa et al. [80] provide a lightweight reconfiguration controller for systems with real-time capabilities. The controller makes the reconfiguration process time-predictable while still having a small hardware footprint. Further, it supports compressed bitstreams to minimize the memory needed. A holistic toolchain to facilitate the rich feature set for bitstream handling and reconfiguration time analysis is also included.

Kohn [94] illustrates a reference design for Zynq-7000 SoCs [95] for enabling reconfiguration access to the two dedicated processors of the system. This so-called Processor Configuration Access Port (PCAP) enables the reconfiguration of the programmable logic portion of the SoC from the ASIC portion; hence, the system can reconfigure itself.

A similar controller for RISC-V-based softcore MCUs is RV-CAP, introduced by Charaf et al. [81]. They use an open-source Ariane [73] core connected to their controller and the corresponding Reconfigurable Partitions (RPs) via an AXI crossbar. A rich software API aids in accessing the features of the system. The partial bitstreams can be stored on an external Double Data Rate (DDR) memory device to provide enough space.

Table 3.1 shows an overview of the named and some further reconfiguration controllers. The applied metrics in this regard are the hardware interface, software support, and domain from which it arose. The last line features our proposed controller, *more*MCU-RC (cf. Chapter 5 [p59]).

Nevertheless, not only partial reconfiguration controllers can be found in the literature. Some approaches, like the one provided by Beckhoff et al. [96], show their framework GoAhead, where a bitstream repository provides partial bitstreams to accelerate a set of algorithms. The tool supports portability and migration functionalities from and to different FPGA families. The concept is very generic, and there is no requirement for any processor core on the device to use partial reconfiguration. It uses the external interface Xilinx provides to partially reconfigure the FPGAs rather than the internal reconfiguration port. However, the idea of the bitstream repository can also be adapted for runtime-reconfigurable systems with a reconfiguration controller.

Relevance and Distinction: The literature presented in this section shows different approaches where partial reconfiguration was used at runtime. This thesis proposes a similar reconfiguration controller. The main difference to the controllers presented in the literature is that *more*MCU-RC is co-designed to the other parts of the system. In conjunction with the OS, it can actively manage the reconfiguration of parts of the pipeline and the peripherals at runtime. This modification process is transparent and happens in parallel to the actual system execution. This dissertation only addresses the technology and the available hardware to achieve runtime reconfiguration within embedded systems. In practice, however, much persuasion is needed before partial reconfiguration is accepted in industry and academia as a suitable solution.

3.2.3 Reconfigurable Computing Platforms

Contrary to the two sections before dealing with flexible MCU architectures and partial reconfiguration, this section looks at reconfigurable computing platforms in general. Reconfigurable means that the internal logic of the computing platform can, in principle, be modified to adapt to new use cases. This modification can happen either at design time or at runtime. Technology-wise, softcore MCUs are included here, and combinations of dedicated processor cores with reconfigurable logic hardware are covered.

One of the most influential works in this regard is the MOLEN polymorphic processor, detailed by Vassiliadis et al. [97]. It involves using a general-purpose processor coupled with a reconfigurable hardware portion. This reconfigurable hardware is in charge of providing new instructions similar to a co-processor. The logic for these new instructions must be created at compile time of the software. For this purpose, short code sequences are extracted from the application code and mapped to synthesized instructions in the co-processor. While software running on the multipurpose processor can use these instructions as if they were regular instructions, the interconnection between processor and co-processor introduces some overhead in terms of performance. Moreover, the approach requires adjustments to the toolchain and does not allow reconfiguration of the system at runtime.

Another similar, very early work in this regard is PRISM (Processor Reconfiguration through Instruction-Set Metamorphosis), which was presented by Athanas and Silverman [98]. Their configuration compiler takes a program as input and generates images for software and hardware. While the software image is similar to an executable produced by a conventional compiler, the hardware image holds the physical specifications for a reconfigurable platform. Again, everything is designed to work at compile time. However, a performance gain can be achieved using application-specific hardware accelerators.

Alternative static approaches such as CHIMAERA, described by Ye et al. [99], use bus systems and programmable functional units connected to the host processors to achieve a flexible instruction set. However, it is found that this approach leads to performance degradation compared to a highly customized functional unit directly in the processor. This degradation is because the programmable units cannot be accessed directly within the pipeline as they are only attached via an execution control unit.

Pittman et al. [100] show a concept called eMIPS, where co-processing modules can be loaded into the pipeline while the system runs. Basic blocks within application code can be converted into an instruction. When the original code is executed and the corresponding instruction is reached, it is skipped over in software if the corresponding module is loaded into the pipeline. Instead, the hardware-accelerated version is executed. The eMIPS processor also notifies the OS whenever a new instruction is loaded; it does not let it decide on the process. The concept targets workstations within high-performance computing for on-demand performance improvements.

While softcore microprocessors were not widely available when the previously shown concepts were introduced, there are some later concepts, like the one shown by Damschen et al. [101]. With their i-Core, they introduce a runtime-reconfigurable processor core that

can add application-specific instructions either by microcode or reconfiguration. The authors base their work on a LEON3 [102] softcore processor.

Ordaz and Koch [103] carried out another work worth mentioning. They allow the partial reconfiguration of an FPGA in order to add new Single Instruction/Multiple Data (SIMD) instructions to a dual-core processor system. The SIMD engine, in their approach, is very tightly coupled to both dedicated scalar CPUs, where a claimed improvement of the area utilization on the FPGA can be achieved. Again, no modifications of the cores themselves were done.

Relevance and Distinction: Contrary to the above concepts, this thesis enables partial reconfiguration at runtime, not only in a static way at synthesis time. Furthermore, the functionality is to be integrated directly where it is supposed to be used (e.g., the processor's pipeline) instead of being located in some remote place (e.g., connected via a bus system). Another distinction to the presented concepts is the hardware/software co-designed approach, where the OS and the hardware can actively decide whether reconfiguration should occur. Furthermore, we retain the functionality of standard software compiler toolchains.

While most approaches either use an entire FPGA only or a single reconfigurable logic hardware part integrated alongside a hardwired processor system, embedded Field-Programmable Gate Arrays (eFPGAs) introduce a new, yet costly, facet into the world of partial reconfigurability. While these systems can still be designed static or dynamic, the granularity and performance benefits are beyond question. The SoCs presented in the following feature hardwired ASIC parts with an arbitrary number of embedded FPGA portions for reconfiguration purposes. This concept is similar to systems with dedicated processors alongside an FPGA portion, yet finer granular integrated. In the end, both parts form a kind of reconfigurable ASIC, which must be created in an application-specific way again, with all the benefits and drawbacks (cf. Section 2.4[p13]).

Schiavone et al. [104] show an example with their Arnold SoC. This SoC features a RISC-V-based computing module with integrated eFPGA portions. The eFPGA is directly connected to the processor core via a bus-interfaced memory controller. It is mainly used for preprocessing sensor data within IoT applications.

Dao et al. [105] describe yet another reconfigurable RISC-V core with their FlexBex. It forms a very influential yet parallelly developed work to this dissertation, as it also features runtime hot swapping of instructions. They show an open-source framework for adding eFPGAs to RISC-V processors, where both are directly connected.

Similarly, Neumann et al. [106] show an application-specific instruction set processor core that again uses an eFPGA for acceleration purposes.

Relevance and Distinction: Although all three approaches have the performance-wise advantage of being partially ASIC and partially FPGA, they lack the flexibility of a standard FPGA regarding design-time and runtime reconfigurability. Furthermore, creating an ASIC

with features like this is even more costly. Nevertheless, the idea is also interesting for the present thesis and could be easily ported to a system like this in the future. For the proposed concept, the static area of the MCU can be taped out as the ASIC part, whereas eFPGAs portions must be put where the dynamically reconfigurable partitions are. Modifications to the ASIC part become impossible, however.

3.2.4 Operating System Support of Flexible Hardware and Reconfigurable Computing Platforms

An important aspect that sets this thesis apart from other work, and thus needs to be illuminated, is the support for reconfigurability in the OS running on the embedded system. Reconfigurability here includes all the previously mentioned concepts and ideas that use reconfigurable computing platforms, use flexible hardware, or ISAs in some form. This applies to both design-time and runtime reconfigurability.

In 2016, Eckert et al. [107] published an overview article about the future challenges of reconfigurable computing. A particular emphasis is on operating systems that support the abstraction of the underlying reconfigurable system. Their review and survey work covers many different OS and the main features they should support. While it is not a complete list, some concepts are exciting and shall be pointed out in more detail in the following paragraphs, alongside other related papers.

With ReconOS, Agne et al. [108] show an OS for reconfigurable systems based on an RTOS, but they also feature Linux support within their second generation. ReconOS runs on a hardware architecture comprising a dedicated CPU connected with several reconfigurable areas called *slots*. The connection itself is implemented as a bus system. From the OS point of view, only software threads exist in the system, whereas behind the abstraction, hardware threads placed in these *slots* can be used to accelerate functionality. The OS hardware interface manages the hardware threads on the physical level and delegate software threads take over their corresponding roles within the OS. Synchronization and communication between hardware threads and software threads are also supported. While the name suggests otherwise, the concept is no stand-alone OS but rather a runtime environment that runs on top of a host OS.

Having a similar runtime concept on top of a Linux environment, Kelm and Lumetta [109] show their approach called HybridOS. This set of OS extensions claims to support fine-grained reconfigurable accelerators that are coupled with general-purpose processor cores. The application integration, communication overhead, and data movement between the cores and the reconfigurable accelerators are particularly emphasized. The concept aims to reduce the difficulty of integrating hardware accelerators within FPGA-based systems into embedded systems executing general-purpose operating systems. While it is no full-fledged OS, they call it an API to support reconfigurable accelerators.

An approach toward integrating reconfigurable systems into the real-time domain was made by Adetomi et al. [110] with their R3TOS (Reliable Reconfigurable Real Time Operating

System). This concept deals more with task offloading into hardware, namely having task bodies fully implemented in hardware. To do so, they partition the hardware resources into *computation regions* interconnected by an NoC. The concept identifies defective hardware regions to ensure reliability and automatically moves the corresponding reconfigurable modules to healthy areas. Additionally, R3TOS uses a particular version of an Earliest Deadline First (EDF) schedule that is aware of the reconfiguration times. The motivation for having reliability in this way is a use case in space, where radiation might affect certain areas of the FPGA fabric.

Dörflinger et al. [111] show yet another framework to support DPR within embedded systems running on FPGAs; in this case, a Zynq-7000 SoC [95]. Within this system, algorithms for image processing can be accelerated in hardware, and runtime-reconfiguration hot-swaps the implemented logic whenever needed. To do so, an embedded microkernel called Genode OS runs on the dedicated cores of the system and manages the AXI-connected FPGA portion within the SoC. The authors claim that in this way, DPR is not only available for the high-performance computing field but can also be facilitated for the embedded and safety-critical domain, as there is also a strong emphasis on the isolation of the system's individual components. The paper shows not only the implementation of the OS but proposes a holistic framework to achieve said functionality.

To handle partial reconfiguration within big FPGAs featuring multiple ICAPs, such as the Virtex-5 [112], Göhringer et al. [113] present their CAP-OS. CAP in this abbreviation stands for Configuration Access Port, hinting where the main emphasis of this work is placed, namely the synchronization between multiple of these ports. In particular, whenever a hardware task is to be executed (and programmed onto the FPGA fabric), the OS actively manages and synchronizes the transfer to the reconfiguration ports over an NoC. Furthermore, the scheduling and resource allocation within the system is also a responsibility of CAP-OS. Internally it is based on the Xilkernel RTOS [114], where it uses six system threads.

While there are still more OS-based systems, the works above form an excellent overview. However, some works integrate software support of flexible ISAs without the help of a full-fledged OS. Andrews et al. [115] show a discussion of computational models for systems that feature a CPU alongside an FPGA. Throughout their work, the authors present a unifying programming model where single POSIX threads [116] can be either compiled to run on the CPU or synthesized for execution on the FPGA. Their hardware thread interface provides an abstraction for the CPU/FPGA system. It looks like a unified multiprocessor architecture, where standard thread communication and synchronization are still available across the hardware/software boundary.

An approach that uses special instructions within the used ISA is presented by She et al. [117]. They use a partially reconfigurable decoder that is space optimized and a software-controlled bypass network to optimize operand encoding. This way, the flexible ISA within the embedded system can be designed very energy-efficiently.

Bergmann et al. [118] introduce a process model for hardware modules in reconfigurable SoCs. This work allows hardware co-processors to be developed and integrated into a software-like environment based on an embedded Linux OS.

Relevance and Distinction: All the works shown above show essential aspects and ideas on how reconfigurable systems can be managed by software or even by an OS running on top. The OS in most of the presented approaches runs on dedicated cores within an SoC, having only an interfaced FPGA portion for acceleration and reconfiguration. This is one of the main distinctions to this dissertation, as no holistic hardware/software co-designed system uses reconfigurability to alter the very core the OS itself is executed on while it is running. Also, flexible ISAs alongside reconfigurability – essential when designing a runtime-reconfigurable system backed by an OS – are not yet covered by the literature to the best of our knowledge.

3.2.5 Performance Monitoring

Most modern processors contain a generic Performance Monitoring Unit (PMU) that is usually accessible via registers or memory-mapped address areas. Commonly, they can be used to measure the occurrence of events or contain different time values that are incremented in a certain way. Within Intel CPUs, model-specific registers [119] hold these values, whereas RISC-V-based CPUs store them in their Control and Status Registers (CSRs) [36]. ARM CPUs allocate a particular memory block for these performance values [120].

The basics of performance monitoring hardware are extensively discussed by Sprunt [121]. Performance-monitoring hardware, as well as the corresponding software integration, is a common part of a modern computing platform and can be essential in improving overall performance. The measurable performance events are mentioned explicitly, and how they can form a basis for optimizing high-performance processors and the software executed.

Patil et al. [122] show a study of performance counters and profiling tools that can be used to monitor several performance characteristics of applications running on different CPUs, mainly on Intel's x86 architecture. This paper explains the architecture of the PMU within different processor variants in detail and a particular emphasis on the provided tools in several software libraries. The final comparison gives a good overview of which features are available in the x86 architecture and how they are used.

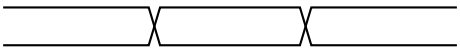
Very similar work is presented by Singh et al. [123], where the same PMUs are discussed concerning the *perf* and *perf_events* subsystem within Linux-based systems. This work examines the software utilization of the performance monitoring hardware in detail. It delivers exciting insights into the design decisions of the latter, as well as the different types of monitoring: counting and event-based sampling.

Some softcore architectures feature tailored units for energy consumption or overall system performance monitoring. Ambrose et al. [124] show a scalable performance monitoring concept. They integrate their event monitoring unit into a softcore Multiprocessor System-on-Chip (MPSoC). The event counting can be individually programmed to facilitate runtime adaptation and verification of the system while being hardware resource effective.

Regarding implementations on softcore MCU architectures, Ho et al. [125] introduce a PMU for LEON3 [102] single- and multi-core platforms. The hardware/software co-designed infrastructure can be used to monitor concurrent events within the system that runs on an FPGA. It

offers seamless integration into the standard Linux performance measurement functionality, including the profiling tools. The monitored events include partial reconfiguration times, influencing this work considerably.

Relevance and Distinction: While modern high-performance processors already feature a PMU, this is not yet the case for every embedded MCU or is only starting to be used on a large scale. Since the part of this thesis dealing with the design of reconfigurable systems is heavily dependent on performance measurements, ideas from the presented works are used to implement a system-aware PMU for an embedded use case. While most works in the literature are built to measure events, there are only a few methods to monitor tasks or parts of the ASW in hardware. However, the PMU designed within this dissertation features an in-depth knowledge of the entire system. It is aware of the OS data structures and the internal structure of the core and its pipeline. Another distinction to existing works is that the hardware counters within our PMU can be reused at runtime, enabling more sustainable use of the hardware resources.

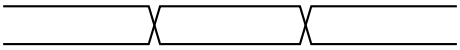


In contrast to the works presented, this thesis proposes a holistic co-design of the self- and runtime-reconfigurable embedded MCU and the OS running on top of it. The OS running on the main processor, as well as the reconfiguration controller, can reconfigure the MCU directly. Hence, the OS can directly exploit the changed hardware. An explicit focus is on embedded devices with inherent real-time features that can benefit from rebootless runtime reconfiguration.

CHAPTER 4

Dynamic Electronics Generation

This chapter deals with the scientific approach that answers the first research question: "How can application software be used to automatically generate all the remaining parts of an embedded system?". For this purpose, the overarching idea is first introduced, and the individual parts are presented in detail. Subsequently, the system description is dealt with to explain the generation of system configurations and the final electronics generation. The chapter concludes with an evaluation of the implementation of the concept.



The first part of this dissertation deals with the dynamic generation of the remaining layers of the embedded system stack out of the requirements of the ASW code. First, the system description format developed for this purpose is introduced (cf. Section 4.2). Then, the generation of system configurations from the requirements taken from the application software is discussed in detail (cf. Section 4.3). This can be achieved by matching the initial requirements to hardware module properties and the subsequent generation of suitable system configurations. The concept is concluded by describing the generation of schematics and board layouts for a final PCB from the previously generated system configurations (cf. Section 4.4). To do so, modules within the system configuration must be interconnected, and their respective pins must be assigned before the actual schematics. From these schematics, board layouts can be generated. A detailed evaluation concludes the chapter (cf. Section 4.5).

Publications E-I, E-II, E-III, and E-IV included in Chapter 7 [p95] propose ***papageno***X and are the foundation of the text in this chapter. The abbreviation means **P**rototyping **AP**plication-based with **A**utomatic **GEN**eration **O**f **X**, where **X** is the envisioned final system. Figure 4.1 shows the logo of the concept. While the text presented here concentrates on the readability and comprehensibility of the concept, some details have been omitted. In this case, the reader is referred to the complete documentation in the respective publications. Figures and further material are adapted from these publications without explicit citation.

4.1 Main Idea of *papagenoX*

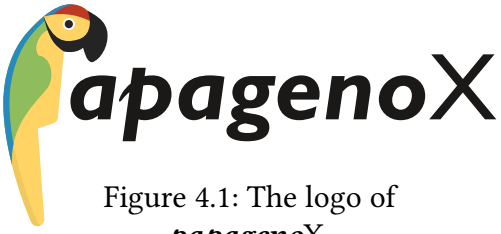


Figure 4.1: The logo of ***papageno***X.

The established design process for embedded systems (especially within prototyping) follows a paradigm where hardware is designed before the software development starts. Within this bottom-up process, where "software follows or adjusts to hardware", an engineer builds the software according to the given requirements but is limited by the properties of the previously designed hardware platform. Whenever a deficiency within the hardware platform is detected during software design, workarounds within the flexible part – which only the software usually is – are the only way to go. Hardware in this development phase is rarely changed, as this means additional effort and costs. In prototyping, software adaptations for hardware deficiencies also often violate established standards. An example in the automotive domain are the Complex Device Drivers (CDDs) in the AUTOSAR standard [126]. Although these are specified in the standard, they are often used to bypass it and implement functionalities that do not comply with it. However, if a prototype is transferred to a series product, this product must fully comply with these standards. This non-compliance again makes new adaptations necessary.

To enable more flexible and future-oriented prototyping, we invert the established prototyping process towards a methodology where "hardware follows software". Consequently, we propose ***papageno***X as a concept that starts with the development of the ASW and generates the remainder of the embedded system out of the ASW code. The concept includes a set of tools and methods that can be used to analyze ASW code, obtain the relevant requirements, and generate the underlying PCB automatically. It also maps parts of the software to reconfigurable computing platforms within FPGAs. The overall goal of ***papageno***X is to enable system engineers to reduce the TTM, as changes within the ASW automatically create or update the underlying embedded system stack. The concept can also be used to fine-tune the overall system configuration in the development phase, simplifying the transition from prototype to series production and helping to judge whether or not field-deployed hardware is still suitable for updated software.

When working with ***papageno***X, the following "hardware follows software" process is to be followed:

1. ASW development
2. in-depth analysis of ASW concerning functional and non-functional requirements
3. creation of a selection space for suitable components and modules
4. filtering of the selection space concerning the requirements
5. generation of potential configurations from components
6. evaluation and optimization to select best fitting configuration(s)
7. mapping and synthesis of functions or algorithms to the reconfigurable logic
8. file generation of schematics and board layouts for the final PCB

Item 7 of this list is not explained in this chapter but is central to the second part of this dissertation (cf. Chapter 5[p59]). To summarize, the approach to the proposed methodology contains three major parts, also shown in Figure 4.2:

(4.1.1) ASW analysis → creates selection space
(4.1.2) FPGA generation → maps functions to reconfigurable logic
(4.1.3) PCB generation → module-based generation of suitable PCBs

The subsequent Subsections 4.1.1 to 4.1.3 introduce all these parts and give a short overview, while the further sections go into detail.

4.1.1 Application Software Analysis

During software development, all the ASW requirements must be identified to answer parts of the question *"What information about functional and non-functional requirements is needed to generate PCBs from ASW automatically?"*. The entire process is iterative. This means that the hardware can be generated not only after the software is final. Instead, it is constantly refined during the ASW development. Within our concept, the ASW is executed on top of an embedded OS that we base our implementation on: *Smart*OS (cf. Section 2.3[p11]). The provided toolchain, when using this OS, already compiles and links all the relevant libraries automatically at compile time [127]. Hence, when using *Smart*OS, the question on *"How can we derive the needed operating system software features from ASW?"* is answered implicitly. When using a driver or hardware-dependent library, requirements can be easily retrieved by analyzing function parameters. To get an impression, the following ASW code example shows how to store data permanently.

```
store_data(&data, StoreType.Permanent, 10000000);
```

The storage device is not defined, but the access data rate must be greater or equal to 10 MB/s (→ requirement).

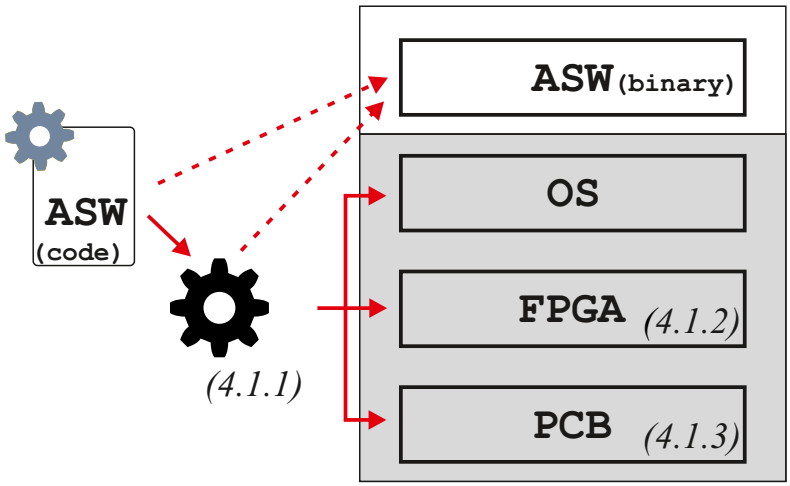


Figure 4.2: The major parts of ***papageno***X.

It can already be seen here that the libraries must be designed, so that information about the requirements is explicitly specified in the function parameters. Explicit requirements specification also applies to the request for resources by the ASW. Using OS functionalities (cf. resource management in Section 2.3[p11]), deadlines and timeouts (→ requirements) can be specified for these requests. In conjunction with annotations, the requirements can be easily extracted and forwarded to the following parts of the concept.

A general conclusion is that it is beneficial to develop the ASW so that more information must be explicitly specified already during the development process. The more explicit information is available, the easier it is to derive the requirements of the ASW.

4.1.2 Reconfigurable Logic Generation

The question *"How can we extract application-specific logic and automatically map it to reconfigurable logic devices?"* aims to extract pieces of the ASW and map them to application-specific logic parts within an FPGA. This hardware acceleration of software pieces can either be facilitated by interfacing a softcore MCU as a coprocessor that introduces new peripherals or directly within the ISA of the system's main CPU. The second part of this thesis proposes a holistic concept in this regard (cf. Chapter 5[p59]).

4.1.3 Printed Circuit Board Generation

This part is the most extensively researched part of the dynamic electronics generation approach. It answers the question *"What information about functional and non-functional requirements is needed to generate PCBs from ASW automatically?"* and *"How can this information be used to generate a PCB prototype matching all ASW requirements?"*. To answer these questions, we divide the approach into two sub-parts and use dedicated hardware modules as an abstraction. The first sub-part translates the requirements of the ASW into constraints and spans a selection space of possible system configurations (cf. Section 4.3). These system configurations are described in the specifically designed system description format, explained in Section 4.2. Out of all feasible system configurations, the most suitable one(s) must be selected according to functional and non-functional requirements. However, no electrical properties have been dealt with up to now. These are considered in the second sub-part, where the previously generated system configurations act as an input for the PCB generation (cf. Section 4.4). There, the interconnection between the dedicated hardware modules on the electrical level is established, and schematic and board layout plans for the final PCB form the output. The result of the generation, an automatically generated PCB, can either be a motherboard, where the dedicated hardware modules can be plugged into slots, or a single integrated PCB where all the needed electrical and electronic components are directly soldered.

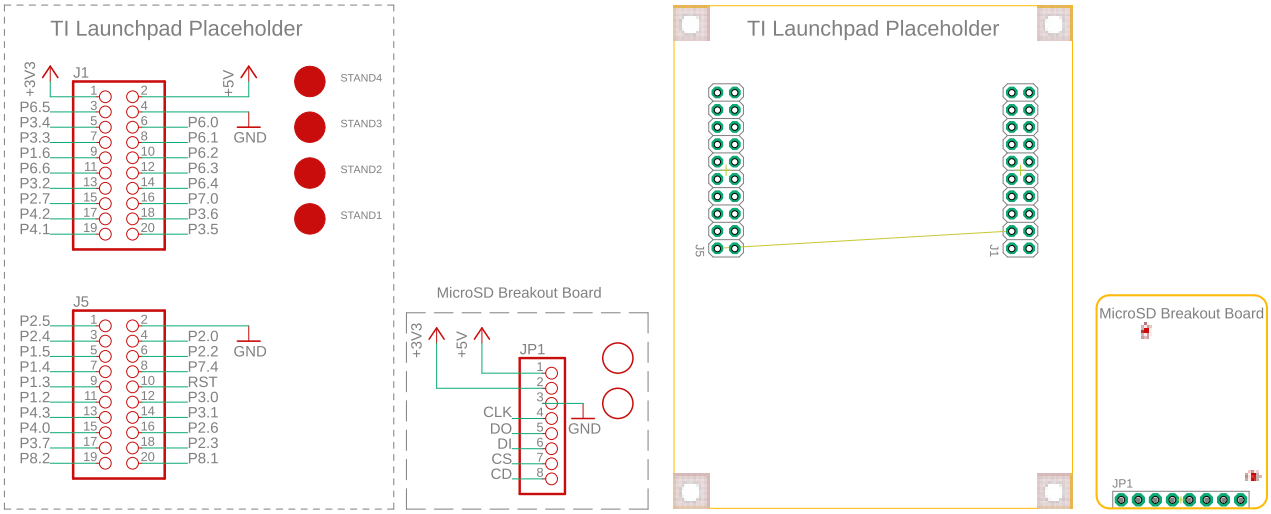


Figure 4.3: Schematics and board layouts of two placeholder design blocks for a LaunchPad and a MicroSD Breakout Board.

4.2 System Description Format

The specifically tailored system description format is a generic, JavaScript Object Notation (JSON)-based [128] format[1] designed to be easily adaptable for different scenarios. As the system description within this approach is based on a module library, the definition of every available module is inevitable before interconnecting them into a system. Interfaces must also be defined to compose a system out of these modules. The implemented solution yields three different definitions (clustered in individual JSON files) that are needed in order to generate electronic systems dynamically:

(4.2.1) Module Definition: One single file defines the hardware module, its interfaces, and pins, and a second file contains the design block for creating schematics and board layouts concerning this module.

(4.2.2) Interface Definition: Generic definition of several different interface types to interconnect modules with each other; new types can be easily implemented and included within this file.

(4.2.3) System Definition: Contains modules and connections between these; is abstractly wired with certain interface types.

The following exemplary explanations include module information of a Texas Instruments LaunchPad™ ([130], “LaunchPad”) and an Adafruit MicroSD Breakout Board ([131], “MicroSD Breakout Board”).

```
{
    name: "LaunchPad", type: "ComputingPlatform",
    design: "LaunchPad.dbl",
    properties: { cpu_frequency: 25000000 },
    interfaces: [{
            name: "SPI0", type: "SPI",
            pins: { MISO: "P3.1", MOSI: "P3.0",
                SCLK: "P3.2", CS: 'any@["P2.0", "P2.2"]' }
        }, {
            name: "SPI1", type: "SPI",
            pins: { MISO: "P4.5", MOSI: "P4.4",
                SCLK: "P4.0", CS: any }
        }, {
            name: "I2C0", type: "I2C",
            pins: { SDA: "P3.0", SCL: "P3.1" }
        }, {
            name: "I2C1", type: "I2C",
            pins: { SDA: "P4.1", SCL: "P4.2" }
        }],
    pins: [ "P6.5", "P3.4", "P3.3", "P1.6", "P6.6", "P3.2", "P2.7",
    "P4.2", "P4.1", "P6.0", "P6.1", "P6.2", "P6.3", "P6.4", "P7.0",
    "P3.6", "P3.5", "P2.5", "P2.4", "P1.5", "P1.4", "P1.3", "P1.2",
    "P4.3", "P4.0", "P3.7", "P8.2", "P2.0", "P2.2", "P7.4", "RST" ,
    "P3.0", "P3.1", "P2.6", "P2.3", "P8.1"]
}
```

Listing 4.1: Module definition of a LaunchPad with two SPI and two I^2C interfaces.

4.2.1 Module Definitions and Design Blocks

When looking at Listings 4.1 and 4.2, a module definition always contains a *name*, a *type*, the path to its corresponding design block file, and a list of properties in the beginning. The design block file property refers to an EAGLE [132] design block file that acts as a schematic and board layout placeholder within the final composed system (cf. Figure 4.3). The board placeholders in this example only consist of the pins of the corresponding hardware module. This way, a motherboard can be created, where the actual hardware module can be plugged atop with external connectors (e.g., a pin header). The placeholder, therefore, only serves as an interface layout to the fully assembled module PCB.

Furthermore, the module definition for a hardware module must contain all its interfaces and pins. The pins are simply an array of strings with the names of the pins of the hardware module. To guarantee a consistent design and interconnection between the modules, the names of the pins must be coherent between this module definition and the design block file. The power supplies of the modules follow a consistent naming format to avoid discrepancies between modules. This means that voltage potentials that are to be connected and have the same value must also use the same name across all modules (e.g., `'V+'` for every 12 V

[1]While any other object notation format (e.g., eXtensible Markup Language (XML) [129]) could have been chosen, JSON provides the most freedom while being the leanest.

```
1 {
2   name: "MicroSD_Breakout_Board", type: "Peripheral",
3   design: "MicroSD_Breakout_Board.dbl",
4   properties: { can_store_data_non_volatile: true,
5                 can_store_data_detachable: true },
6   interfaces: [
7       {
8           name: "SPI1", type: "SPI",
9           pins: { MISO: "DO",  MOSI: "DI",
10                  SCLK: "CLK",   CS: "CS" }
11      }
12  ],
13  pins: ["CLK", "DO", "DI", "CS", "CD"]
14 }
```

Listing 4.2: Module definition of a MicroSD Breakout Board with an SPI interface.

potential). The wiring of voltage potentials is done automatically in the background to ensure the correct power supply for every module. Hence, no explicit modeling within the system definition is needed.

The interfaces are represented by the corresponding array within the definition and may contain different interface types (cf. Section 4.2.2), reflecting the module's I/O features. Besides a *name*, the property *type* – as its name suggests – determines the interface type. When looking at Line 6 in Listing 4.1, the LaunchPad includes (amongst other interfaces) an SPI interface at the corresponding pins one line later. As illustrated in Line 8, pins within interfaces can be either directly assigned to dedicated hardware pins (e.g., `SCLK: "P3.2"`) or automatically assigned (e.g., `'any@["P2.0", "P2.2"]'`) by the generation concept, if more than one pin is possible for the interface. Any open pin can be used if no array is given after the **`any`** statement. This way, e.g., Chip Select (CS) wires can be easily placed based on the given requirements.

4.2.2 Interface Definitions

The interfaces that can be used are defined in a single generic interface definition per module library. Listing 4.3 shows an exemplary structure of how such a file can look like. This example shows one definition of an SPI interface. As the definition format is generic, any wire-based interface type can be integrated into this system. The keywords **`master`** and **`slave`** represent the role a partner within the communication takes, whereas **`bus`** is used for a common wire one can connect to.

When looking at Lines 11 and 12 in Listing 4.3, special treatment for two corner cases is displayed. While an SPI master has CS wires for every slave selection, the selected slave only has a single CS wire. This is reflected with the keywords **`wiremultiple`** for a master and **`wiresingle`** for a slave. The concept then maps multiple CS wires to open pins according to the module definition shown before.

```
{
  interfaces: [
      {   type: "SPI",
          connections: [
              { "master.MOSI" : "bus.MOSI" },
              { "master.MISO" : "bus.MISO" },
              { "master.SCLK" : "bus.SCLK" },
              { "slave.MOSI"  : "bus.MOSI" },
              { "slave.MISO"  : "bus.MISO" },
              { "slave.SCLK"  : "bus.SCLK" },
              { "master.CS"   : "wiremultiple" },
              { "slave.CS"    : "wiresingle" }
          ]
      }, { ... }, { ... }, ...
  ]
}
```

Listing 4.3: Interface definition file containing SPI.

4.2.3 System Definition

Now that both a library of module definitions and an interface definition exist, hardware can be composed out of these modules. System definitions, therefore, contain hardware modules and the connections between them. Listing 4.4 shows the definition of a system composed of two instances of the previously explained modules, a LaunchPad and a MicroSD Breakout Board. The instantiation happens within the *modules* array, where a unique *name* for each instance must also be given. Once instantiated, the interconnection between the modules can be established within the *connections* array. This interconnection is done by utilizing the previously defined interfaces. In this example, the LaunchPad is connected to the MicroSD Breakout Board via an SPI connection, in which the instance `LP1` is the master, and the instance `SD1` is the slave. The system definitions can either be done manually or automatically generated using the concept shown in Section 4.3.

```
{
  modules: [
      { name: "LP1", type: "LaunchPad"},
      { name: "SD1", type: "MicroSD_Breakout_Board"}
  ],
  connections: [
      { name: "SPI_Connection1", type: "SPI",
        participants: [ { name: "LP1", role: "master" },
                        { name: "SD1",  role: "slave" } ] }
  ]
}
```

Listing 4.4: A system model containing two modules connected via SPI.

4.3 Generation of System Configurations from Requirements

This section explains the part of the approach where the already assumed requirements of the ASW are translated into constraints, and a selection space of possible system configurations is spanned. In the following, we first give the necessary mathematical definitions and the terminology on which the system configuration generation is based. Second, the implementation and some design decisions of the approach are illustrated.

4.3.1 Mathematical Definitions and Terminology

The most important terms within the approach are “module”, “system”, “requirement”, and “property”. All these four terms are explicitly defined in the following.

$$M = M_c \cup M_p = \{c_1, c_2, ..., c_n, p_1, p_2, ..., p_m\}, \text{ where } M_c \cap M_p = \varnothing \tag{4.1}$$

M denotes the set of all feasible modules being the union set of M_c and M_p. M_c contains all computing modules (n modules c_i[2]), and M_p contains all peripheral modules (m modules p_i).

From the set of all theoretically existing modules M, we form a sub-set $M_{library} \subseteq M$. This sub-set is necessary since not all conceivable components might be available in a development engineer’s library. Subsequently, after applying this filter upfront, the hardware components that match the requirements of the ASW, $M_{matching} \subseteq M_{library}$, can be selected. $M_{matching}$ is a set of modules, where every module satisfies at least one of the initial requirements. The modules in this set are then used to compose a number of o feasible, not yet interconnected system configurations $s_{nc,i}$ in the set $S_{nc} = \{s_{nc,1}, s_{nc,2}, ..., s_{nc,o}\}$, where nc stands for “not connected”. Thus, every not-yet-interconnected system configuration $s_{nc,i}$ is a composition of an arbitrary number of hardware modules. In this case, every module must satisfy at least one of the initial requirements.

Per definition, every system configuration $s_{nc,i}$ must contain at least one computing module c_i and can have an arbitrary number of peripheral modules p_i. If the system contains more than one module – no matter which kind – these modules are tried to be interconnected into feasible system configuration candidates $S_c \subseteq S_{nc}$. Figure 4.4 depicts an example of four system candidates with different modules and related properties. All these properties of the modules combined form the overall properties of the system candidate, which can be matched against requirements.

The interconnectable final system configurations $S_{matching} \subseteq S_c$, which match all the initial requirements of our wanted system, form the output of this part, and are selected due to their overall properties.

[2]This explanation uses i as a generic index.

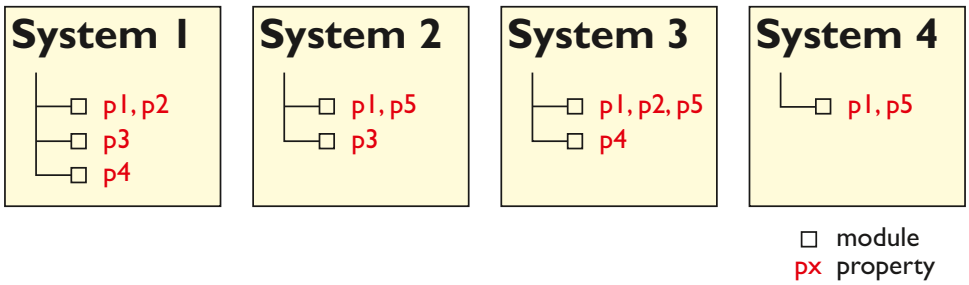


Figure 4.4: Example for a set S_c of system configuration candidates, their modules, and properties px.

4.3.2 Implementation

The already roughly sketched approach is implemented in a divide-and-conquer methodology with the following steps:

1. **Module Library Creation**: The input requirement string extracted from the ASW is the basis for the library generation. Based on these requirements, the set of modules is pre-filtered and forms the library for the following steps.
2. **System Configuration Creation**: Based on the mathematical definitions outlined above, the remaining modules form a power set of feasible system configurations without yet interconnected modules. Out of these not-yet-interconnected modules, interconnectable system configurations are formed.
3. **Requirement Matching**: After having all the feasible system configurations at hand, our requirement matching algorithm steps in and exports the interconnectable final set of system configurations ($S_{matching}$) for the initial requirements. These are then forwarded to the following part of the approach, explained in Section 4.4.

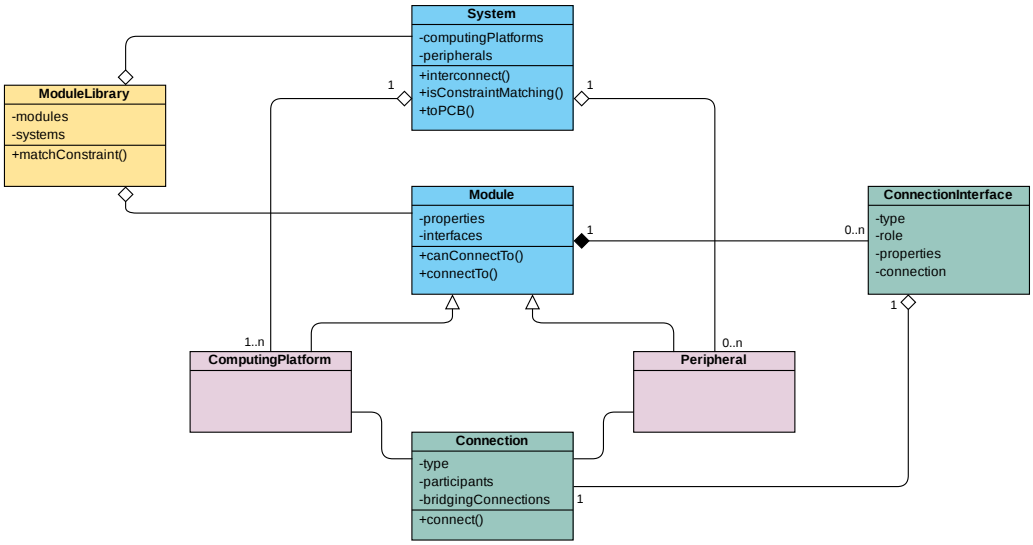


Figure 4.5: Class diagram of the approach on how to obtain system configurations.

Table 4.1: Example decision table for different system configuration candidates with their properties px.

	p1	p2	p3	p4	p5
System 1	*true*	100	*true*	3.2	*NULL*
System 2	*true*	*NULL*	*true*	*NULL*	250
System 3	*true*	50	*NULL*	7.1	700
System 4	*true*	*NULL*	*NULL*	*NULL*	100

We chose the object-oriented methodology based on the class diagram depicted in Figure 4.5 implemented in Java to achieve this functionality. One singleton instance of the class `ModuleLibrary` holds the module library, which contains a set of modules alongside the final system configurations. The class `System` represents these system configurations, and the class `Module` the modules, respectively. A single instance of a module consists of the related properties and a set of instances of the class `ConnectionInterface` forming the module's interfaces. Module instances can be either a `ComputingPlatform`, or a `Peripheral`. To connect one module with another, the methods `canConnectTo()` and `connectTo()` provide the functionality. Within the class `ConnectionInterface`, the type of the connection (e.g., I^2C, SPI) and the corresponding role (e.g., master, slave, none) are reflected. Furthermore, specific properties and an instance of `Connection` are included. The latter is responsible for connecting the interfaces of modules with each other, thus forming a connection.

An instance of the class `System`, including at least one computing platform and a set of peripherals, represents a system configuration. Its method `interconnect()` tries to interconnect all the system configuration modules. To do so, every `connect()` method of every connection within a module is called. To match all system configuration properties against a set of requirements, `isConstraintsMatching()` provides said functionality.

Now, to generate the system configurations in the system description format, the singleton instance of `ModuleLibrary` contains the method `matchConstraints()`, having the initial requirement string as an input. It then filters and combines all relevant modules and tries to deliver interconnected system configurations as a result. The requirements matching algorithm is based on decision tables, where Table 4.1 depicts an example generated from Figure 4.4.

The module library generates a decision table where all system configuration candidates are included. This table holds only the relevant properties (p1 - p5) concerning the initial requirements (r_1 - r_5). The system configuration candidates, as previously explained, can only contain modules with properties px that satisfy at least one of the initial requirements r_i. Whenever a system configuration does not contain any module with a property that satisfies a particular requirement, this is indicated by `NULL` (cf. System 4, properties p2, p3, and p4 in Table 4.1). The properties can be functional or non-functional but must be quantifiable. They can be of arbitrary number format (e.g., boolean, integer, real numbers) with or without units. The requirement strings are extracted from the ASW, as explained in Section 4.1.1. Non-quantifiable properties are not supported by the concept as of now.

Equations 4.2, 4.3, and 4.4 illustrate the mapping of requirement strings to sets of system configuration candidates.

$$(r_1 == true)\ \&\&\ (r_2 >= 100)\ \rightarrow\ \{\text{System 1}\} \tag{4.2}$$

$$(r_1 == true)\ \&\&\ (r_5 > 100)\ \rightarrow\ \{\text{System 2, System 3}\} \tag{4.3}$$

$$(r_2 > 75)\ ||\ (r_4 >= 7.0)\ \rightarrow\ \{\text{System 1, System 3}\} \tag{4.4}$$

The implementation of this functionality uses the decision table as a relational database and runs queries in it. We use a file-based database structure and SQL [133] as a query language. Finally, supposing a system configuration makes it all the way into the final results. In that case, `toPCB()` generates the corresponding PCB for the system configuration (cf. Section 4.4). The algorithm is a non-deterministic polynomial-time hard problem. For large systems, the runtime might therefore increase drastically. However, the shown module-based approach can reduce the complexity to a certain extent, providing adequate performance in practice. The number of constraints per module is sufficiently small that an exhaustive search can solve the problem.

4.4 Generation of Schematics and Board Layouts from System Configurations

This section describes the part of the approach where the previously generated system configurations act as an input for the schematics and board layout generation explained below. This generation approach is split into two steps, the connection establishment and pin assignment, as well as the final schematic and board layout generation. Again, the approach is implemented in Java, and the explanation continues the example given in Section 4.2[p39].

4.4.1 Connection Establishment and Pin Assignment

This step takes an interconnectable system definition as input, establishes the electrical wire connections, and assigns dedicated pins of the hardware modules accordingly. The approach includes iterating over all connections between the modules, the mapping to their corresponding interface type, and the final wiring on the electrical level. Every connection has – apart from its type – a finite number of participants with different roles, interfaces, and pins. To connect the participants, wires that map to dedicated pins are introduced, where each wire belongs to a particular connection.

As introduced in the system description format, a wire can either be a bus wire, a `wiresingle`, or a `wiremultiple`. A `wiresingle` is the most common wire type, as it can be assigned to any of the free pins of a module. Due to their property of being the most general connection wires, they must be assigned to their pins as the very last step.

As some hardware modules have dedicated pins for particular bus types, bus wires must be connected directly to the corresponding pin. Hence, these pins must be matched with the

connection's wires directly. The interface definition must match roles and pins accordingly to ensure correctly interconnected participants of every connection.

Lastly, the type `multiplewire` takes care of wires connecting multiple components. For SPI, e.g., a master needs to have the same number of CS wires as slaves in the system. The wire can clone itself as often as needed to achieve this functionality and is assigned to a free pin of the master module. The selection to which SPI connection the wire belongs is already decided in the previous part explained in Section 4.3.

When connecting the modules, these wires are inserted into the system description and form a holistic interconnected description of the final system. The next step converts the wires to *nets* within the electrical circuit plans.

4.4.2 File Generation for Schematics and Board Layouts

Having generated the interconnected system description, this step generates the corresponding electrical circuit schematic and board layout plans in an XML-based file format [134] that can be used by the EAGLE [132] Electronic Design Automation (EDA) tool. While the concept is not tied to any particular tool, EAGLE was chosen over another EDA for implementation in this work, as it is still widely used in both industry and academia for electronics design. This is the part of the concept where the design blocks must be loaded and placed alongside the application of the previously established connections. To generate both the schematic and the board layout, the following three steps must be executed:

1. **Module Instantiation**: The module's design blocks are loaded and instantiated. As a design block can be instantiated an arbitrary number of times, the signal names of each instance must be adapted to avoid ambiguities or faulty connections between pins. This is handled by introducing signal name suffixes according to the name of the instance (cf. *_LP1* in Figure 4.6a, whereas not in Figure 4.3[p39]).
2. **System Interconnection**: Based on the connections generated before, the pins of an instantiated module must be connected to the corresponding wire within the system. In this case, the wires get prefixes according to the connection's name. This is displayed, e.g., as *SPI_CONNECTION1.* in Figure 4.6a.
3. **Placement**: The last and computationally most expensive step is the placement of all the instances on both plan types. It involves XML parsing and generating, as well as two-dimensional translations along the plans. This step results in two XML files for both the schematic and the board layout.

After all the steps were applied, two generated XML files form the result. Figure 4.6a depicts the schematic, and Figure 4.6b the board layout. These files can then be opened within EAGLE to perform the final step: routing the board layout. As routing a PCB is a non-trivial task, this step is not done by ***papageno***X automatically but is left to the engineer. However, EAGLE (amongst other EDA tools) includes a sophisticated auto-routing functionality that can also be used. In addition to Electromagnetic Compatibility (EMC) characteristics, attention must be paid to the placement of power supply circuits, communication link and bus lane wires, and the physical dimensions of the components. Figure 4.6c shows the final PCB for our example system.

4.5 Evaluation

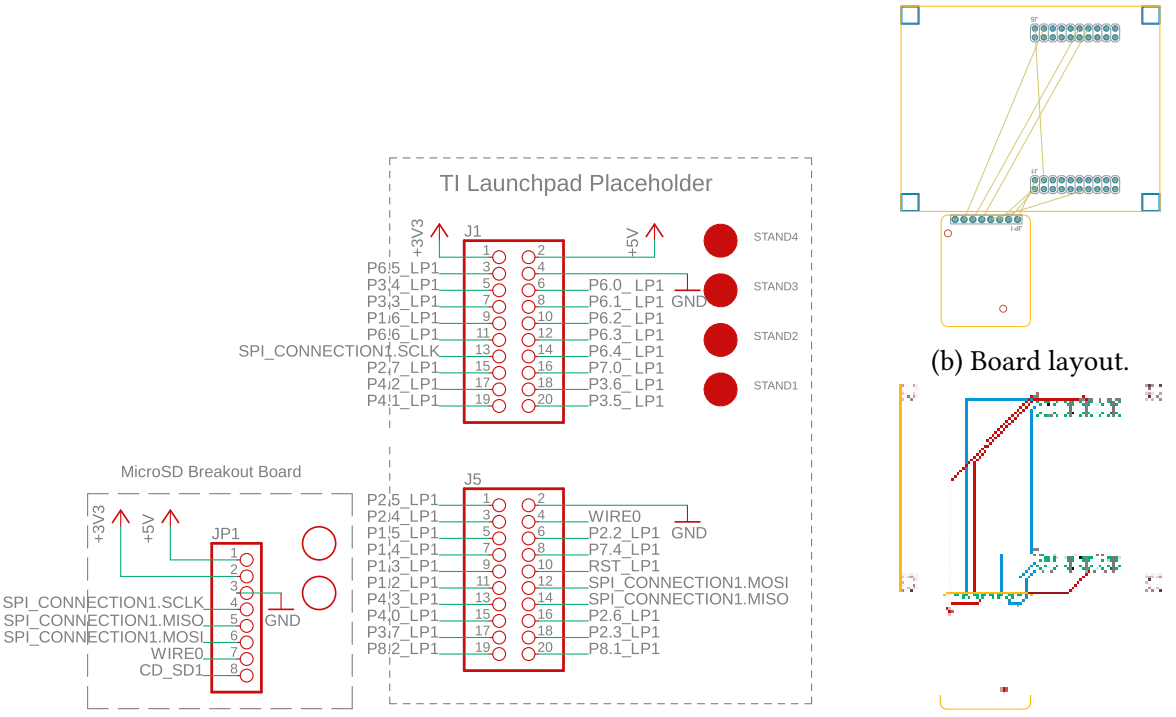
This section evaluates the approach taken within ***papageno***X to generate PCBs from the application's requirements automatically. First, some use cases are illustrated to show how the concept can be applied. Second, a detailed runtime evaluation of the computationally expensive file generation for schematics and board layouts after the system configuration generation is carried out.

4.5.1 Proof of Concept

An overall number of three different use cases show the functionality of ***papageno***X. The extension of a given system configuration is also shown from the first to the second use case. Additionally, the PCB for the last use case was manufactured entirely and equipped with electronic components for demonstration purposes.

Simple Use Case

Assuming we have an embedded system that is designed to take some measurements in its environment, process them, and store the results on a removable storage medium. This is a common case for embedded systems within sensor networks or the IoT.



(a) Interconnected schematic.

(b) Board layout.

(c) Routed board layout.

Figure 4.6: Wiring diagrams for a very simple example (cf. Section 4.2).

```
10  OS_TASKENTRY(task1) {
11    // [...]
12    while(1) {
13      // [...]
14      input_value = measure_voltage();
15      process_data(&data, input_value);
16      store_data(&data, StorageType.Removable);
17      // [...]
18    }
19  }
```

Listing 4.5: Example ASW for the simple use case.

Following the initial example in Section 4.1.1, the code in Listing 4.5 shows the ASW for this use case. The ASW has one task, which periodically measures voltage values, processes them, and stores them in a removable storage device. After the creation of the ASW, the following requirement string can be extracted:

$$(can_measure_voltage_values == true)\ \&\&\ (can_store_data_removable == true)\ \&\&\ (cpu_frequency >= 2MHz).$$

For the module's properties in our module library, we use the abbreviations p1 - p6, which have the following meaning:

- p1: number of modules
- p2: can store data removable (*can_store_data_removable*)
- p3: can store data non-volatile
- p4: can provide voltage values (*can_provide_voltage_values*)
- p5: can measure voltage values (*can_measure_voltage_values*)
- p6: CPU frequency (*cpu_frequency*)

The execution of the system configuration generation starts with pre-filtering the entire module library and yields the decision table depicted in Table 4.2. The pre-filtering is the reason why only boolean values of **`true`** are in the table, as no module with the value **`false`** made it to the decision table (also due to the requirement string). Many properties in the table

Table 4.2: Decision table for the simple use case.

	p1	p2	p3	p4	p5	p6
System 0	2	*true*	*true*	*NULL*	*NULL*	$25MHz$
System 1	3	*true*	*true*	*NULL*	*true*	$25MHz$
System 2	1	*NULL*	*NULL*	*NULL*	*NULL*	$25MHz$
System 3	2	*NULL*	*NULL*	*NULL*	*true*	$25MHz$

```
{
 modules: [
   { name: "MSP2", type: "MSP430_device" },
   { name: "MIC3", type: "MicroSD_Board" },
   { name: "16B1", type: "16Bit_I2C_ADC" }
 ],
 connections: [
   {
     name: "I2C_CONNECTION_0", type: "I2C",
     participants: [ { name: "MSP2", role: "master" },
                     { name: "16B1", role: "slave" }
     ]
   },
   {
     name: "SPI_CONNECTION_1", type: "SPI",
     participants: [ { name: "MSP2", role: "master" },
                     { name: "MIC3", role: "slave" }
     ]
   }
 ]
}
```

Listing 4.6: System description of the final system generated for the simple use case.

are marked with `NULL`, meaning that the modules in the respective system do not represent them. This fact shows well how important it is to specify information about the modules explicitly, as missing information might yield wrong results. When matching the initial requirements to this decision table, the best fitting system configuration is System 1, as it meets all the requirements.

Listing 4.6 shows the intermediate system definition of this simple use case, consisting of three hardware modules. The computing module is an MSP430 [135], a MicroSD board is responsible for the removable measurement storage, and the measurements are generated by a 16 Bit I^2C Analog-to-Digital Converter (ADC). Figure 4.7 shows the corresponding generated (and manually routed) PCB.

Extension of the Simple Use Case

While developing the software, new requirements arise: now, we additionally need to provide an output pin's voltage within the ASW, as illustrated in the updated Listing 4.7. The extracted requirements change to:

$$(can_measure_voltage_values == true)\ \&\&\ (can_provide_voltage_values == true)\ \&\&\ (can_store_data_removable == true)\ \&\&\ (cpu_frequency >= 2MHz).$$

```
10    OS_TASKENTRY(task1) {
11      // [...]
12      while(1) {
13        // [...]
14        input_value = measure_voltage();
15        output_value = process_data(&data, input_value);
16        store_data(&data, StorageType.Removable);
17        provide_voltage(output_value);
18        // [...]
19      }
20    }
```

Listing 4.7: Example ASW for the simple use case.

Consequently, the pre-filtered decision table also changes (cf. Table 4.3), as a new property is of interest now. Due to this new property, more system configurations that were not considered in the example before become relevant. The new best-fitting system configuration is System 3 in this table, and its system definition is given in Listing 4.8. While System 3 would also satisfy the requirement string for the simple use case, it is not included in Table 4.2. This is because the system would be overkill since it also includes modules that are not needed in the first use case. When comparing it to Listing 4.6, a 12 Bit I^2C Digital-to-Analog Converter (DAC) that is responsible for providing output voltages is newly added to the system. The changes in the naming of the modules are due to different generation runs and are only essential to avoid ambiguities in the concept and can be ignored here. Figure 4.8 shows the corresponding generated (and manually routed) PCB, where – contrary to Figure 4.7 – a module placeholder is added to the left-most part of the PCB.

These two use cases show how the concept yields PCBs from requirement strings. Additionally, it shows the concept's capabilities to adapt to changing requirements during development time.

Table 4.3: Decision table for the extended simple use case.

	p1	p2	p3	p4	p5	p6
System 0	2	*NULL*	*NULL*	*true*	*NULL*	$25MHz$
System 1	3	*true*	*true*	*true*	*NULL*	$25MHz$
System 2	3	*NULL*	*NULL*	*true*	*true*	$25MHz$
System 3	4	*true*	*true*	*true*	*true*	$25MHz$
System 4	2	*true*	*true*	*NULL*	*NULL*	$25MHz$
System 5	1	*NULL*	*NULL*	*NULL*	*NULL*	$25MHz$
System 6	3	*true*	*true*	*NULL*	*true*	$25MHz$
System 7	2	*NULL*	*NULL*	*NULL*	*true*	$25MHz$

```
{
 modules: [
   { name: "MSP4", type: "MSP430_device" },
   { name: "16B4", type: "16Bit_I2C_ADC" },
   { name: "MIC3", type: "MicroSD_Board" },
   { name: "12B6", type: "12Bit_I2C_DAC" }
 ],
 connections: [
   {
     name: "SPI_CONNECTION_1", type: "SPI",
     participants: [ { name: "MSP4", role: "master" },
                     { name: "MIC3", role: "slave" }
     ]
   },
   {
     name: "I2C_CONNECTION_3", type: "I2C",
     participants: [ { name: "MSP4", role: "master" },
                     { name: "16B4", role: "slave" },
                     { name: "12B6", role: "slave" }
     ]
   }
 ]
}
```

Listing 4.8: System description of the final system generated for the extended simple use case.

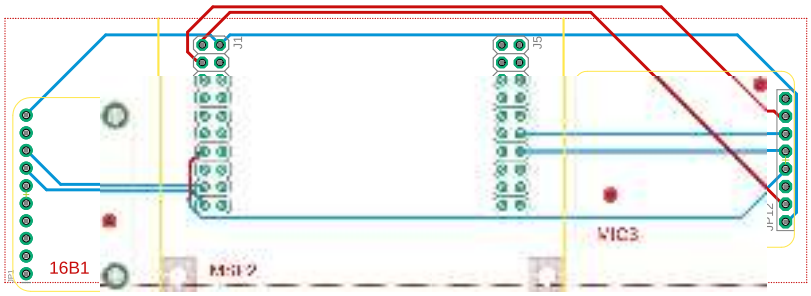


Figure 4.7: Generated board layout of the simple use case.

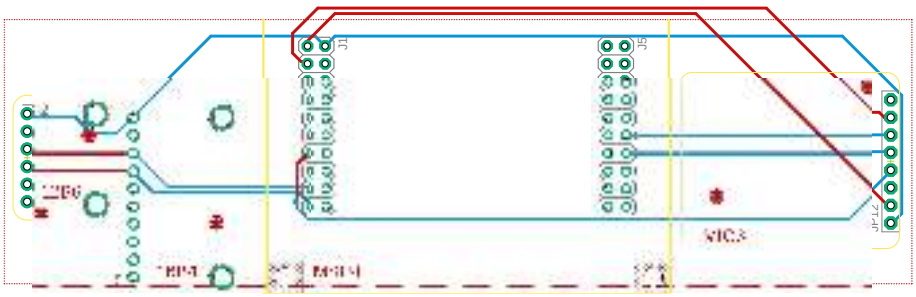


Figure 4.8: Slightly adapted generated board layout of the extended simple use case.

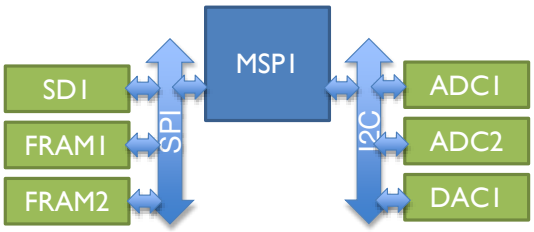


Figure 4.9: The block diagram of all modules in the control system use case.

Control System Use Case

While the two previous use cases show the functionality of the overall concept, this use case shows how initial requirements result in an actual manufactured PCB at the end that runs the developed ASW. To show this, we want to create a control system that interacts with its environment through

- measuring several analog voltage values,
- providing analog voltage values on output pins, and
- storing a data log permanently in two different ways:
  - removable, and
  - stored with low energy consumption, yet non-volatile and redundant.

While we do not specify the code for this use case, the extracted requirement string looks like this:

$$(can_measure_voltage_values == true)\ \&\&\ (can_provide_voltage_values == true)\ \&\&\ (number_of_voltage_inputs >= 8)\ \&\&\ (can_store_data_non_volatile == true)\ \&\&\ (can_store_data_removable == true)\ \&\&\ (number_of_storage_devices >= 3)\ \&\&\ (cpu_frequency >= 2MHz).$$

When matching these requirements to the module library, a possible system configuration may consist of

- an MCU to execute the control algorithms
  → MSP430 on a LaunchPad,
- two 4-channel ADCs to measure voltage values
  → Adafruit 4-channel Breakout Board featuring an ADS1115 ADC [136],
- a DAC to provide voltage values
  → Adafruit 12-bit DAC board featuring an MCP4725 DAC [137],
- a MicroSD card module to log data in a removable way
  → MicroSD Breakout Board, and
- two Ferroelectric Random Access Memory (FRAM) [138] modules to store data in a low-power, non-volatile, redundant way
  → Adafruit SPI FRAM Breakout Board, featuring an MB85RS64V FRAM [139].

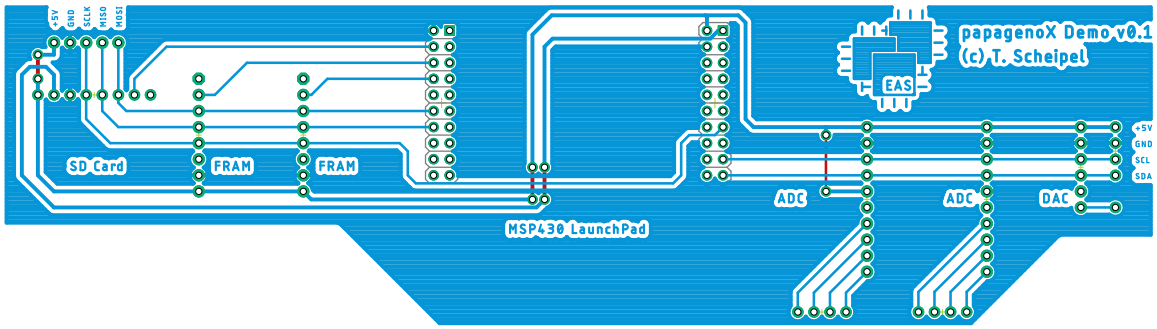


Figure 4.10: Manufacture-ready PCB board layout with an MSP430 LaunchPad connected to three SPI and I^2C modules (manually routed).

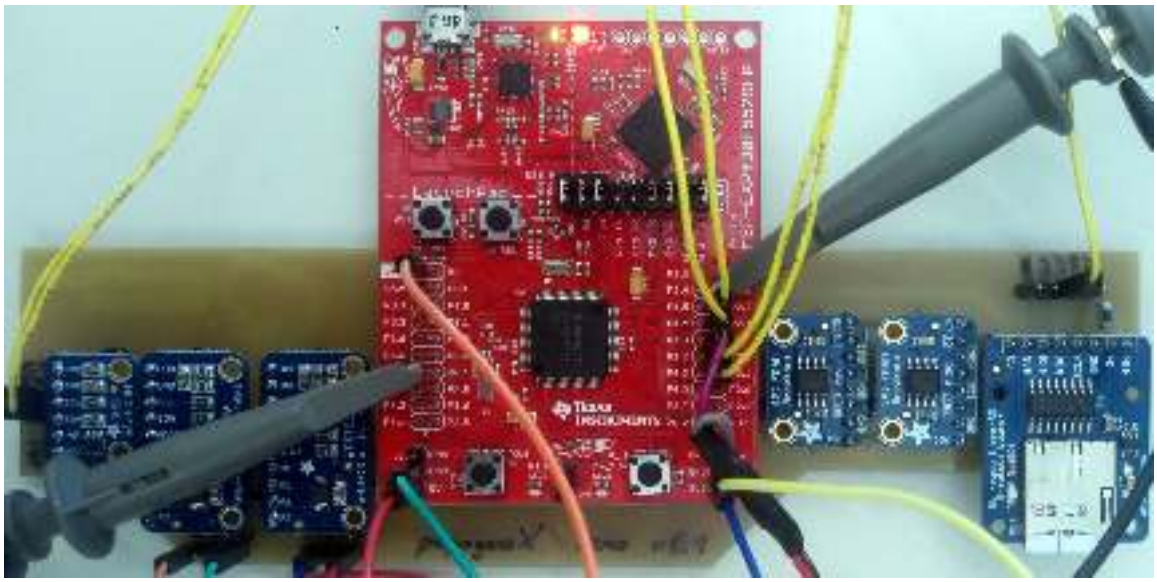


Figure 4.11: Fully equipped prototype board with debug wires.

```
{
 modules: [
   { name: "MSP1",  type: "MSP430F5529_Launchpad" },
   { name: "ADC1",  type: "Adafruit_ADS1115_16Bit_I2C_ADC" },
   { name: "ADC2",  type: "Adafruit_ADS1115_16Bit_I2C_ADC" },
   { name: "DAC1",  type: "Adafruit_MCP4725_12Bit_I2C_DAC" },
   { name: "FRAM1", type: "Adafruit_FRAM_SPI" },
   { name: "FRAM2", type: "Adafruit_FRAM_SPI" },
   { name: "SD1",   type: "MicroSD_BreakoutBoard" } ],
 connections: [{
     name: "I2C_Connection1", type: "I2C",
     participants: [ { name: "MSP1", role: "master" },
                     { name: "ADC1", role: "slave"  },
                     { name: "ADC2", role: "slave"  },
                     { name: "DAC1", role: "slave"  }
   ] }, {
     name: "SPI_Connection1", type: "SPI",
     participants: [ { name: "MSP1",  role: "master" },
                     { name: "FRAM1", role: "slave"  },
                     { name: "FRAM2", role: "slave"  },
                     { name: "SD1",   role: "slave"  }
   ] } ]
}
```

Listing 4.9: The system definition of the prototype.

Figure 4.9 depicts the corresponding block diagram and shows the interconnection between the hardware modules. Listing 4.9 shows the system definition, containing all module instances and both connections with types, roles, and participants. Figure 4.10 depicts the manufacture-ready board layout, whereas Figure 4.11 shows the fully equipped PCB of the embedded system, with the modules already plugged into the mainboard.

4.5.2 Evaluation of the File Generation Process for Schematics and Board Layouts

This section describes measurements and investigations related to the performance of the file generation process for schematics and board layouts. It is an independent evaluation of the proof of concept shown before. This step of the concept uses the system configurations generated before as input. It only generates the files from finished intermediate system descriptions. No requirements matching is performed in this step. However, as many string operations are involved here, this step is the most computationally expensive of the concept for systems with only a few modules. For very large systems, the execution time of this algorithm is likely to explode, as the system configuration generation is a non-deterministic polynomial-time hard problem.

All the discussed evaluations use the same setup as a reference. The application was run using a Java 10 virtual machine on an Intel Core i7 7500U@2.7 GHz with 16 GB Random-

Access Memory (RAM). Table 4.4 shows the mean execution time and combined XML output file size of the generation process for different test case scenarios discussed below. All test cases had a different and constellation of participants, consisting of masters and slaves with different connection types.

Each test scenario is based on the example described in Section 4.2 but with different constellations regarding the number and type of participants and connections. All test cases were executed 100 times. In total, there are four scenarios with seven test cases each: The first scenario included only one SPI connection, with an increasing number of slaves per test case. The second scenario included two SPI connections again with an increasing number of slaves. Test scenario three had one I^2C connection similar to scenario one, while scenario four included SPI and I^2C connections to a single master with an increasing number of slaves. Figure 4.12 shows all test scenarios' average execution time (in ms).

The observed trend is relatively similar when comparing all scenarios: All performance graphs show a linear evolution with a very small additive logarithmic component, as the curve is steeper at the beginning and flattens out. The linear component is due to the linear increase in test case complexity. The additive logarithmic increase can be explained as Java has optimizations in place (e.g., the string constant pool) when processing the same strings repeatedly. These similar processing operations happen when doing the connection reasoning and the generation of XML schematic and board layout files. This is less relevant for smaller systems, but as system complexity increases, these optimizations become more and more effective. This logarithmic evolution is also why doubling the numbers in the first test case resulted in much higher values than in the second test case. Test scenario four is the only one that shows a steeper curve. This is because the combination of different connection types in this scenario makes the generation more complex.

Since the overall file size increased linearly, no correlation was found between file size and execution time.

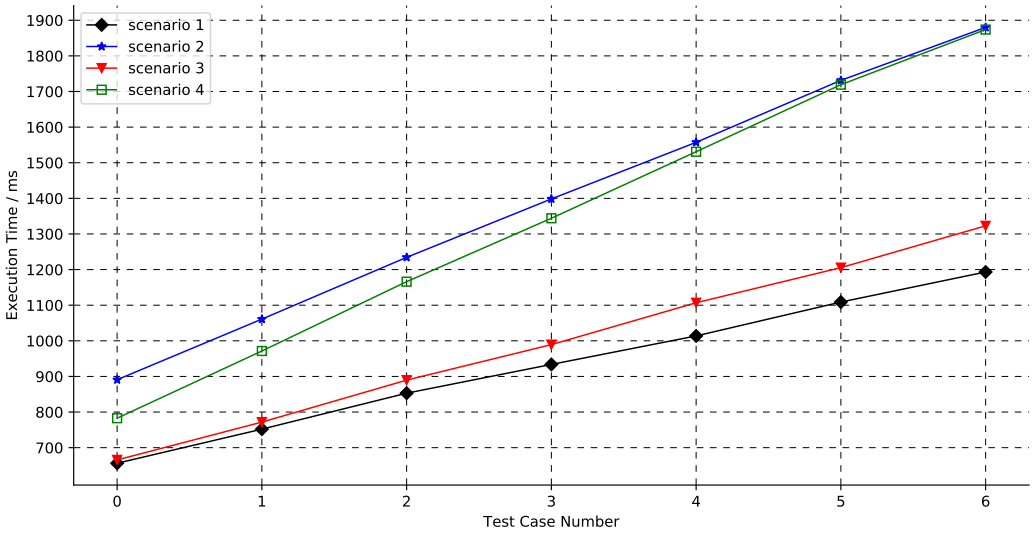


Figure 4.12: Performance graph for different test cases in all four scenarios.

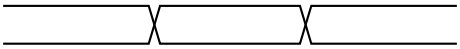
Table 4.4: Mean execution times and output file sizes for different scenarios.

#	*test scenario description*	*execution time*	*file size*
	1 SPI connection (scenario 1)		
0	2 participants (1 master, 1 slave)	656.22 ms	94 KiB
1	3 participants (1 master, 2 slaves)	751.98 ms	111 KiB
2	4 participants (1 master, 3 slaves)	852.95 ms	128 KiB
3	5 participants (1 master, 4 slaves)	933.71 ms	144 KiB
4	6 participants (1 master, 5 slaves)	1013.67 ms	161 KiB
5	7 participants (1 master, 6 slaves)	1108.81 ms	178 KiB
6	7 participants (1 master, 7 slaves)	1193.25 ms	194 KiB
	2 SPI connections (scenario 2)		
0	4 participants (1 master and 1 slave each)	890.77 ms	160 KiB
1	6 participants (1 master and 2 slaves each)	1060.91 ms	192 KiB
2	8 participants (1 master and 3 slaves each)	1234.36 ms	226 KiB
3	10 participants (1 master and 4 slaves each)	1398.30 ms	259 KiB
4	12 participants (1 master and 5 slaves each)	1557.30 ms	293 KiB
5	14 participants (1 master and 6 slaves each)	1731.26 ms	326 KiB
6	14 participants (1 master and 7 slaves each)	1879.93 ms	360 KiB
	1 I^2C connection (scenario 3)		
0	2 participants (1 master, 1 slave)	665.61 ms	106 KiB
1	3 participants (1 master, 2 slaves)	771.77 ms	136 KiB
2	4 participants (1 master, 3 slaves)	889.56 ms	167 KiB
3	5 participants (1 master, 4 slaves)	989.33 ms	198 KiB
4	6 participants (1 master, 5 slaves)	1106.84 ms	228 KiB
5	7 participants (1 master, 6 slaves)	1205.48 ms	259 KiB
6	7 participants (1 master, 7 slaves)	1332.82 ms	290 KiB
	1 I^2C and 1 SPI connection (scenario 4)		
0	3 participants (1 master, 1 slave each)	782.70 ms	127 KiB
1	5 participants (1 master, 2 slaves each)	971.42 ms	174 KiB
2	7 participants (1 master, 3 slaves each)	1166.01 ms	222 KiB
3	9 participants (1 master, 4 slaves each)	1344.08 ms	269 KiB
4	11 participants (1 master, 5 slaves each)	1530.92 ms	317 KiB
5	13 participants (1 master, 6 slaves each)	1719.09 ms	364 KiB
6	13 participants (1 master, 7 slaves each)	1873.92 ms	411 KiB

CHAPTER 5

Reconfigurable Systems Design

This chapter deals with the scientific approach that answers the second research question: "How can an embedded computing platform change its own logic at runtime?". For this purpose, the overarching idea is first introduced, followed by detailed descriptions of every layer of the embedded systems stack. It deals with the microcontroller architecture and its reconfigurability at runtime. Subsequently, a description of the application software interface and the extended concepts within the operating system layer follows. The chapter concludes with an evaluation of the implementation of the concept.



The second part of this dissertation deals with designing an embedded system platform that can be reconfigured at runtime. First, the design of a runtime-reconfigurable MCU platform is outlined in detail (cf. Section 5.2). Particular emphasis is placed on how to design the MCU so that the pipeline provides a flexible ISA and allows for flexible on-chip peripherals. Another major focus is on the partial reconfiguration and the partitioning of the underlying reconfigurable computing platform. The reconfiguration controller will also be addressed alongside how to enable performance monitoring within the system. Then, the application software interface is introduced (cf. Section 5.3), followed by an explanation of the underlying OS and its extended concepts (cf. Section 5.4). Here, the handling of the MCU's flexible ISA is relevant, and managing and triggering the partial reconfiguration of the logic is dealt with. A detailed evaluation of every part of the hardware/software co-designed concept with extensive use cases concludes the chapter (cf. Section 5.5).

Publications R-I, R-II, R-III, and R-IV included in Chapter 7 [p95] are the foundation of the text in this chapter. While the text presented here concentrates on the readability and comprehensibility of the concept, some details have been omitted. In this case, the reader is referred to the complete documentation in the respective publications. Figures and further material are adapted from these publications without explicit citation.

5.1 Main Idea and Architecture Overview

As motivated in Chapter 1[p1], the main idea within the reconfigurable systems design part of this thesis is to enable embedded systems to use and reuse their logic gates more sustainably, i.e., to extend their life cycles. Furthermore, these systems can execute specific software functionalities more efficiently by adding hardware accelerators at runtime. The presented concept overarches all three layers of the embedded systems stack to achieve this. In detail, we compose the concept out of ASW programming paradigms, extended OS features and concepts, MCU design considerations, and partial reconfiguration methodologies. The goal is to provide a platform that enables embedded systems developers to use the runtime reconfigurability capabilities of modern reconfigurable computing platforms, such as FPGAs. Hence, the proposed concept is suitable for softcore MCUs or hybrid systems with a reconfigurable hardware part. The software side of the approach to this concept is based on our general-purpose RTOS, *Smart*OS [19]. *Smart*OS is an abbreviation of **S**ustainable **m**odular **a**daptive **r**eal-**t**ime **O**perating **S**ystem, and Figure 5.3 shows its logo. The hardware part takes a CV32E40P processor core [38] as a basis and integrates a modified version into a specifically tailored MCU called *more*MCU. This abbreviation stands for **m**odular **O**S-aware **r**untime-reconfigurable **e**mbedded MCU, and Figure 5.2 depicts the MCU logo.

Figure 5.2: The logo of *Smart*OS.

Figure 5.3: The logo of *more*MCU.

As a reference, a block diagram of the entire embedded system structure can be seen in Figure 5.1. In contrast, Figure 5.4 shows the envisioned architecture within the hardware.

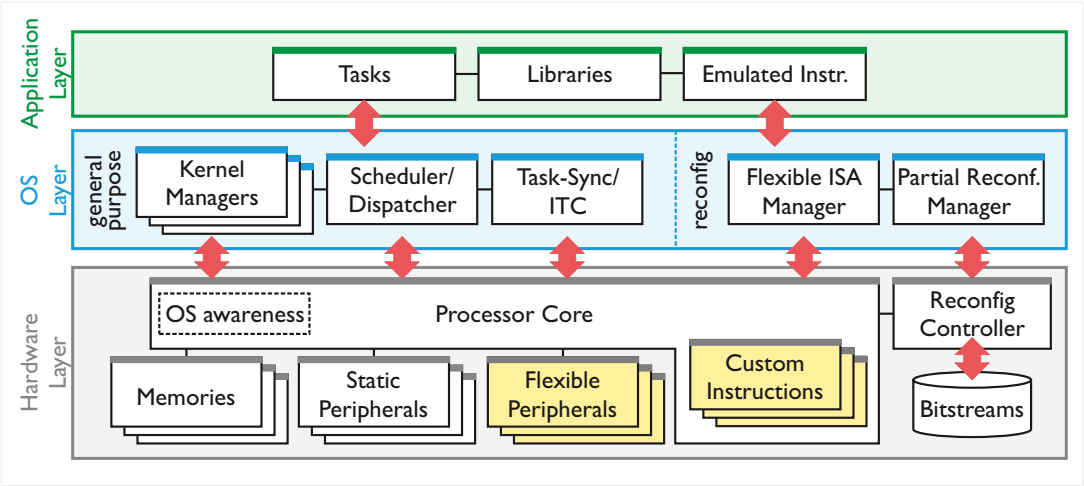


Figure 5.1: The overall system architecture and layers of the embedded systems stack. The runtime-reconfigurable parts are marked in yellow.

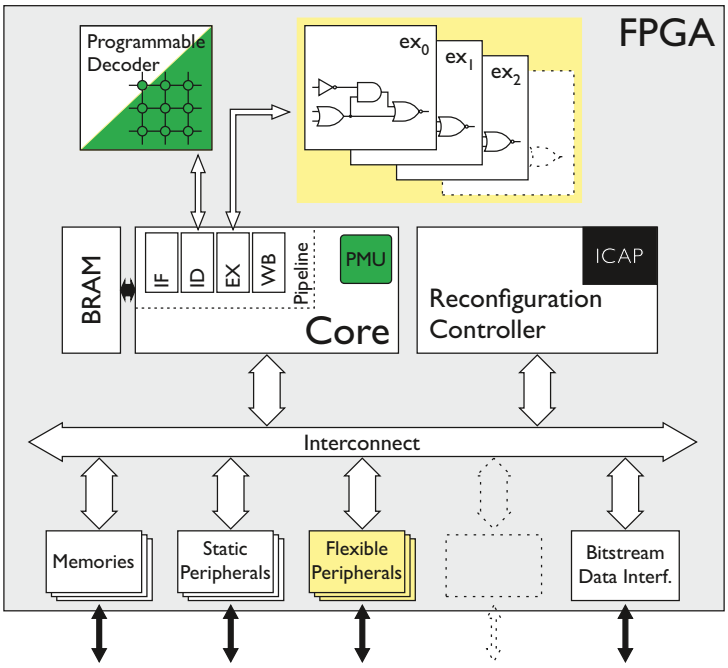


Figure 5.4: The hardware block diagram of *more*MCU. Dynamically runtime-reconfigurable parts are marked in yellow, and runtime-programmable parts are marked in green.

To summarize, the proposed methodology contains three major parts, explained in the corresponding sections:

- *(5.2)* The design of the actual MCU and its flexible parts alongside a process to enable its partial reconfiguration, and
- *(5.3)* an ASW interface to the
- *(5.4)* specifically tailored OS and its extended features for managing and triggering the reconfiguration process according to the (changing) ASW requirements.

5.2 *more*MCU Design

The *more*MCU is the central part of this concept to answer the question, *"How can we design future embedded systems so that they can make more sustainable use of their underlying hardware logic resources for long-term operation?"*.

Its main features are a modular pipeline to enable a flexible ISA (cf. Section 5.2.1) within the MCU's core(s) and the support of flexible on-chip peripherals (cf. Section 5.2.2). Flexible in this context means that instructions and peripherals can be flexibly added to and removed from the hardware at runtime. The OS and the reconfiguration controller (cf. Section 5.2.5) jointly manage and trigger the reconfiguration. Whenever an instruction is not directly available in the ISA, the OS emulates it (cf. Section 5.4.1). Thus, there are two ways to execute an instruction: directly within the ISA and emulated by the OS. For unavailable peripherals,

the system designer can implement a procedure for adding them according to the current requirements. This concept allows the ASW to use and benefit from flexible instructions and peripherals. To enable partial reconfiguration in the sense of the proposed architecture, it is essential to separate the static part of the core from the dynamically runtime-reconfigurable part for custom instructions and peripherals. Section 5.2.3 shows this partitioning of the FPGA into a static area and dynamically reconfigurable partitions (RPs), which plays a vital role in the entire approach. To facilitate proper reconfiguration at runtime, the MCU contains a reconfiguration controller (cf. Section 5.2.5) and a PMU (cf. Section 5.2.6) to profile parts of the system. While the first is responsible for the technologically proper partial reconfiguration at runtime, the second helps determine whether a reconfiguration is reasonable.

5.2.1 Pipeline Design to enable flexible Instruction Sets

Figure 5.4 shows the pipeline of the MCU, which has a static pipeline portion within the core. In addition, the two pipeline stages, Instruction Decode (ID) and Execute (EX), are statically connected to their corresponding dynamic areas:

- The programmable decoder allows to introduce new opcodes to the ID stage.
- The reconfigurable area for adding new logic is divided into several partitions (marked as ex_0, ex_1, ex_2, ... in the yellow upper block) connected to the EX stage.

The programmable decoder is a part of the actual decoder within the ID stage that is responsible for indicating whether or not an instruction is valid. It consists of a set of registers that map RPs of the EX stage to opcodes. A memory-mapped interface for manipulation can be used from the OS or the reconfiguration controller to manage these opcodes. The corresponding partitions ex_n within the partially reconfigurable EX stage are statically connected to the rest of the pipeline to make a flexible ISA possible. This static interface consists of signals for instruction words, operands, internal clock signals, function description flags, or even memory access channels to exchange data between the static and dynamic parts of the pipeline. Since the pipeline infrastructure remains intact and is also used by the new instructions, multi-cycle instructions can also be added, assuming the static interface allows it. To avoid illegal states within the processor core, it is of utmost importance to keep coherence between the programmable decoder and the RPs. As the OS and the reconfiguration controller can manipulate these two parts of the MCU, they are also in charge of housekeeping. Whenever new functionality is added to the core, **first**, the logic in the EX stage must be reconfigured; **second**, the reprogramming of the ID stage must happen (cf. Section 5.2.3). This order ensures that the illegal instruction signal of the pipeline can still be used correctly by the OS to emulate instructions (cf. Section 5.4.1) in case they are not already or are no longer present in the hardware. When stripping functionality, removing the opcode from the programmable decoder is sufficient. As the reconfiguration is a relatively slow process, this order also ensures that the system can still operate normally while the modification takes place. After the updating procedure, a newly added instruction is directly added to the pipeline as a native functionality with no execution time overhead compared to initially available instructions. This results in a performance gain over the software-emulated versions of the instruction.

5.2.2 Microcontroller Design to enable flexible Peripherals

The design process for supporting flexible on-chip peripherals is very similar, although significantly less critical than changing parts of the pipeline. As these peripherals need an interface to the peripheral bus rather than the pipeline, these interfaces can be put on the edge of the core. In our case, the access to on-chip peripherals is designed to be memory mapped via a Wishbone bus [20], also acting as a processor interconnect (cf. Figure 5.4). While any other interconnect (e.g., AXI) could have been chosen, Wishbone is a suitable and widely used processor interconnect within the open-source hardware community and is easy to be used in conjunction with RISC-V. To enable partial reconfiguration, it must only be ensured that the bus interfaces to the flexible peripheral partitions are static within the FPGA. The development engineer can make further interface-related design decisions. Not even the peripheral interface or the reconfiguration management is predefined but can be designed according to the current requirements. Depending on these requirements, the reconfiguration can be managed by, e.g., the peripheral interface, a privileged driver, the OS, or the reconfiguration controller. However, it is vital to consider possible invalid states within the peripheral access individually.

5.2.3 Partitioning and Partial Reconfiguration at Runtime

The partial reconfiguration of the FPGA is the most crucial step to provide new instructions and peripherals to the software at runtime. The technical process features the programming of partial bitstreams [42] into the FPGA on-the-fly. While the clocks of the FPGA remain active and the soft MCU keeps running, this ensures the retainment of the configuration of the static cells.

*Smart*OS keeps track of how often an application uses an emulated instruction and triggers the ISA update in case of expected benefits (cf. Section 5.4). The same applies to accesses to peripherals that are not present in hardware yet. The reconfiguration controller takes care of the modification process and does not introduce any delays or interruptions to the running system. While the process happens in parallel to normal system execution, two aspects are relevant not to disturb the operation of the MCU:

Placement and interfacing of the reconfigurable partitions: Initially, it is essential to mention that the partial reconfiguration process changes all cells in affected RPs of the FPGA. Thus, all connections between the cells in the RPs are reset and newly routed according to the updated logic description. Connections in other areas of the FPGA, especially in the static area, remain unchanged. While there is no general recipe for where to place the different areas, this section gives a few hints on how to succeed.

Cells like FFs and BRAMs can store data (sequential logic), while other cells like LUTs and DSPs can process data (combinatorial logic). During reconfiguration, all logic cells in the RPs will be put into a newly defined state. In contrast, the ones in the static area must retain their data/configuration under all circumstances. The interaction between the static area and the RPs will only continue if there is a hardwired interface between them. This interface is provided by adding hardwired LUTs at the edge of the static area. The information about their exact locations is known to the update mechanism and its data structures (cf. Section 5.2.4),

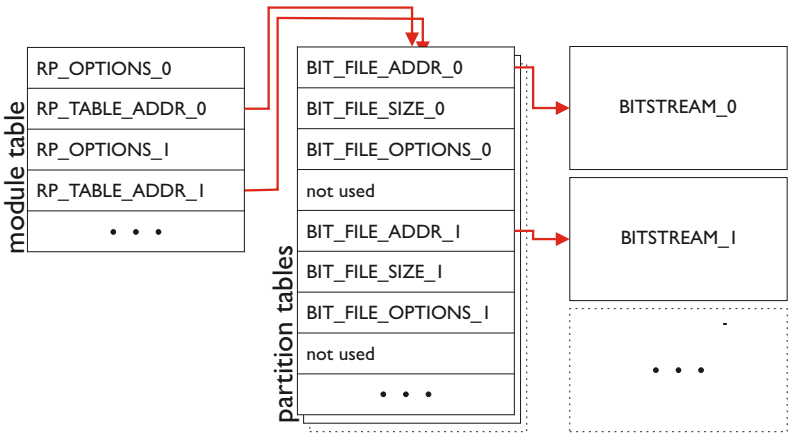


Figure 5.5: Data structure of the partial bitstream repository of *more*MCU.

so they can be used as docking points for new logic in the RPs. Figure 5.17[p88] shows the location of these LUTs in the grey boxes within the RPs. A resulting limitation to our concept is that – despite the runtime flexibility within the RPs – the sizes and locations of both the static area and the RPs in the FPGA cannot be changed at runtime. Additionally, there can be, at most, as many instructions or peripherals available in hardware as there are RPs.

Reconfiguration order: Apart from the partitioning and interfacing, it is also essential to reconfigure and reprogram the contained parts in the correct order: The most crucial signal in this regard is the illegal instruction signal of the pipeline, as it indicates whether or not the processor core supports an instruction. The ID stage generates this signal while decoding the current instruction word and preparing the execution in the EX stage (as well as processing in other stages). If the logic in this stage detects an undefined instruction, an illegal instruction exception triggers the OS to execute a software emulation (cf. Section 5.4.1). After adding the instruction's logic to the RP within the reconfigurable area of the pipeline, the illegal instruction signal must indicate a valid instruction. Hence, the programmable decoder that generates this signal is the last to be updated during the modification process (cf. Section 5.2.1). Updating the programmable decoder before providing the instruction logic itself would lead to undefined states and unpredictable behavior.

While this description only shows the case for the flexible ISA, it is highly application-specific for flexible peripherals, as explained before. However, one must ensure that the corresponding peripheral is only marked as valid and available **after** its logic is updated.

5.2.4 Data Structures and Bitstream Repository

As the reconfiguration controller (cf. Section 5.2.5) uses the data structures in the bitstream repository (cf. Figure 5.1), the following paragraphs highlight all the relevant design decisions. The bitstream repository contains partial bitstreams for every runtime-reconfigurable module (instruction or peripheral) and every RP (in the dynamic areas marked in yellow in Figure 5.4) on the *more*MCU. Figure 5.5 depicts the data structure that allows easy management of the

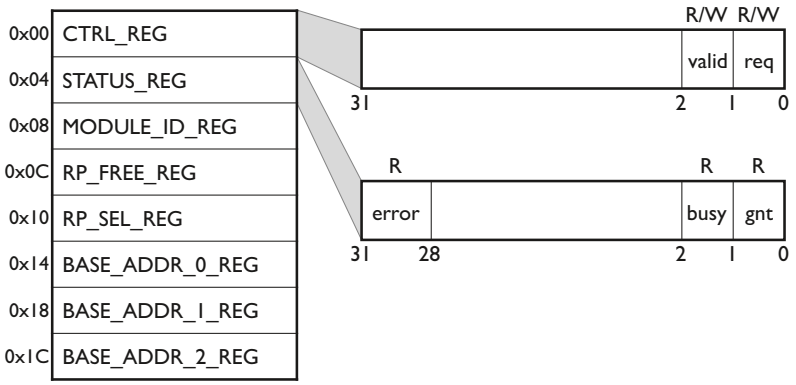


Figure 5.6: The control and status registers of the reconfiguration controller of *more*MCU.

bitstreams. All data words, if not otherwise mentioned, are 32 Bit wide. The structure consists of a module table containing RP placement options (`RP_OPTIONS_n`) and a table address (`RP_TABLE_ADDR_n`) for every RP. The table address points to the corresponding partition table that holds the relevant information about all the possible partial bitstreams that fit into this partition. The bitstream information holds a bitstream address (`BIT_FILE_ADDR_n`) pointing to the actual partial bitstream that can be loaded into an RP alongside its size (`BIT_FILE_SIZE_n`) and some options (`BIT_FILE_OPTIONS_n`) like compression state for every module loadable to a certain partition. The structure has a "not used" memory word to prepare for future adaptations. As the reconfiguration controller supports decompression while programming partial bitstreams, a run-length-encoding-based compression algorithm saves space on the memory device. A bitstream data interface (cf. Figure 5.4) connects the bitstream repository's memory device to *more*MCU. This way, different memory technologies, e.g., Static Random-Access Memory (SRAM) or flash memory, can be used interchangeably.

Since the overall concept is hardware/software co-designed, both the reconfiguration controller and the OS can use these data structures. To prevent race conditions, the OS accesses the data in read-only mode; write operations can be performed via the reconfiguration controller.

5.2.5 Runtime Reconfiguration Controller

The reconfiguration controller is responsible for partially reconfiguring the *more*MCU at runtime and uses the data structures explained in Section 5.2.4. Partial bitstreams are loaded from the bitstream repository, decompressed, and programmed into the corresponding partition of the configuration memory[1] via the ICAP of the FPGA on-the-fly. During this process, all clocks remain active, and the current logic on the FPGA remains operational, resulting in the continued operation of all software. The reconfiguration controller acts in parallel to the remaining MCU and thus does not influence its operation. It is aware of all the relevant data structures used by the OS and the current state of the FPGA and its RPs. It can actively decide whether a piece of logic can be placed in an RP or not (due to, e.g., current occupation) and

[1]The configuration memory – as its name indicates – holds the current configuration of all logic cells of the FPGA.

is also responsible for taking care of the programmable decoder (cf. Section 5.4.1) in case of a pipeline reconfiguration. Whenever a partial reconfiguration happens at runtime, all cells in the corresponding RP are reset and newly connected through the configuration memory to follow the intended, updated hardware description. The static cells, however, stay untouched.

The control and status registers and their relative addresses depicted in Figure 5.6 form the interface that the reconfiguration controller exposes on the MCU interconnect. It can be used in a memory-mapped environment to control the controller's behavior with a piece of software. Section 5.4.2 explains the required program flow used in the OS. In addition to the interconnect bus interface, a direct port can be used to perform reconfiguration from another hardware component. When both interfaces are in use, differentiation between the requested bitstream repository base addresses can be made with the registers `BASE_ADDR_n_REG`. For use in software, it is sufficient to set only `BASE_ADDR_0_REG` to the starting address of the bitstream repository explained in Section 5.2.4.

5.2.6 Performance Monitoring Unit

The Performance Monitoring Unit (PMU) is the part of the *more*MCU that is capable of monitoring the performance of hardware and software parts of the system. It is co-designed to the MCU and the OS and is aware of the system it is embedded in. This awareness means that it has a deep knowledge of the hardware structure of the MCU as well as the internal data structures of the OS and can even manipulate them. Hence, it can also, e.g., monitor internal signals of the MCU and parts of the OS. Its main profiling features include measuring, e.g., all tasks' execution times or even the execution time between two specific program counter values. While being a standalone PMU proposed in R-II, the *more*MCU uses it to measure, e.g., execution times or call counts of emulated and native instructions.

The PMU contains features like several programmable counter registers triggerable by hardware or software, like through a change of the task pointer or the program counter, interrupts, or external events. Hence, there must be different kinds of counters for general usage, task-aware counters, and counters belonging to a particular task. The PMU is programmable at runtime in terms of the number of counters itself and the number of configuration[2] registers per counter. All task-aware counters offer the possibility to buffer counter values to the RAM at a task switch without delay. The counter registers are 64 Bit wide, whereas the configuration registers take the MCU's word length of 32 Bit. As stated before, the PMU is integrated directly into the pipeline (cf. Figure 5.4) module, as it needs access to pipeline status information. In addition, a direct connection to the RAM and the CSR file is necessary. The memory link is needed to buffer values directly to the RAM and be able to reuse performance values later on.

Configuration of the PMU

The PMU consists of the main configuration register to define every counter's type. Within this register, every counter has an 8 Bit value representing its type. Additionally, every counter

[2]The term "configuration" in this context refers to the configuration of the PMU, not the FPGA.

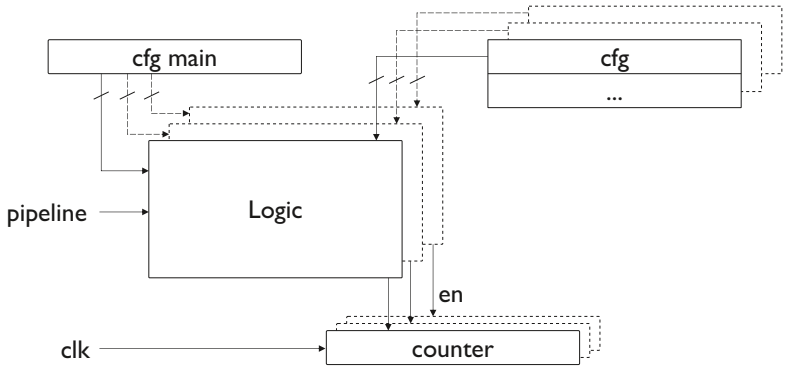


Figure 5.7: A simplified block diagram of the combinatorial logic to enable a counter.

has its own two 32 Bit configuration registers to apply further configurations to the particular type of counter.

In total, there are three different groups of counter types:

- **Global counters** measure execution times and events independently of the current task. This configuration is globally valid and does not need to be buffered. An example is a counter that records the time a **single** task is running.
- **Task-aware counters**, measuring execution times and events for **every** running task with only one hardware counter. Therefore, the counter values are different from task to task. Hence, the counter values must be buffered in the corresponding TCB at every task switch to guarantee valid values for each task. However, the configuration is persistent (e.g., a single counter that measures every task's execution time and buffers the values on task switches). Its configuration can be set once and is globally valid.
- **Counters belonging to a task** measure differently for every task. Counter and configuration values must be buffered in this case since the configuration differs from task to task. Hence, the configuration must be set within every task and, therefore, saved and restored on each task switch. This way, e.g., code sequences of a task can be profiled individually.

The configuration registers for the counters are used to apply further configuration. Supposing one needs a counter to measure between two program counter values. In that case, these two addresses must be written into the counter's two configuration registers. All registers, including the counter values, have their individual CSR address for access via special operations defined in the ISA [29]. The module's access address ranges and logic scale with the configuration of the PMU. The number of counters and registers within the current hardware configuration determines the addresses. The module handles it with separate address logic blocks for reading and writing rather than the present CSR file.

Combinatorial logic of the counters

A single counter unit consists of a byte within the main configuration register, two dedicated configuration registers, a counter register, and a combinatorial logic block to enable the current incrementation of the counter register. Figure 5.7 shows a simplified block diagram.

The logic block generates an enable flag to enable the sequential clock-triggered increment of the counter value. Inputs for the logic block are the configuration registers and the current processor information gathered by inputs directly from the pipeline module. As this happens only by reading from specific wires, the system is not interfered with by this approach. The logic block itself contains sub-blocks for every different counter type and its constraints. To implement an extensible structure, a logic of combinatorial blocks has a final or-gate with multiple inputs, as shown in Figure 5.8. It illustrates the part of the logic for generating a counter's enable flag (`cnt_en[i]`) with `PMU_CNT_TYPE_0` as counter type in its main configuration (`cfg[0]`). Hence, this counter increments if the current task pointer has the programmed value in `pmu_cfg[i][0]`, which is the first configuration register of the current counter `i`. Future counter types must be added at the inputs of the final or-gate at the end.

The different counter types divided into three groups are:

- For **global** counters, there can be: a counter that continuously increments (i.e., a system clock; cf. *cnt 4* in Figure 5.9), a counter for the processor user-mode execution time, a single task execution time counter (cf. *cnt 1* in Figure 5.9), an overall task execution time counter, a counter for all but a specific task, an overall interrupt counter, a counter for missed interrupts by a specific task, and an overall external port pattern match counter.
- The group of **task-aware** counters, which save all counter values separately for every task in memory, contains: an overall task execution time counter (cf. *cnt 2* in Figure 5.9; the different colors indicate the profiling of different tasks), a task interrupt counter, and an external port pattern match counter.
- There is only one counter that needs buffering of counter and configuration values: The counter that measures if the program counter is between two values (cf. *cnt 3* in Figure 5.9) has another unique feature. In order to start the counter at a particular program counter value and end it at another one, the PMU uses the least significant bit of the configuration registers to mark whether or not the program counter has already reached the programmed value.

Buffering of counter and configuration values

To ensure the explained functionality, buffering of counter and configuration values is compulsory. Hence, there has to be a memory interface within the logic, and the hardware has to be aware of the OS' task concept. In particular, the current task pointer must be known to

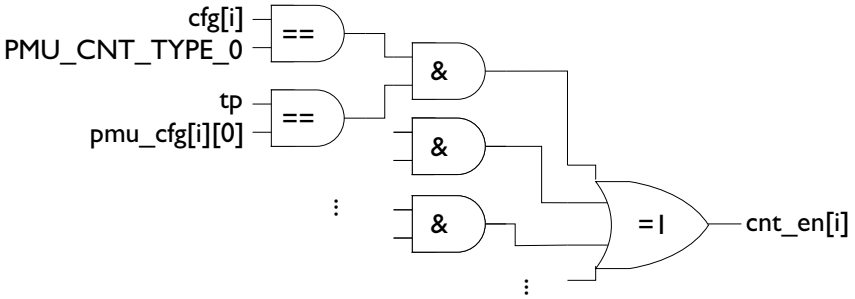


Figure 5.8: A generalized logic schematic to derive the enable flag of a counter.

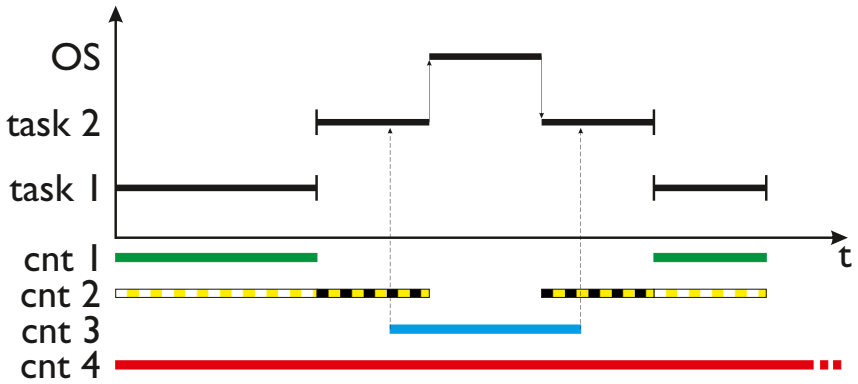


Figure 5.9: Performance monitoring of two tasks with four hardware counters. The colored lines indicate when a counter actually measures.

integrate task awareness into the unit. On a task switch, all the register's values must be saved somewhere belonging to the suspended task in the RAMs, and the buffered values of the new task must be loaded into the registers, respectively.

One of the main problems concerning buffering is that memory access collisions must be avoided. As the present hardware structure uses a dual port RAM, one of the two memory ports is reserved for PMU purposes. Consistent values are guaranteed as the memory allows access to a memory cell from the two channels simultaneously.

The current task pointer of the OS must also be available to the PMU, which allows the PMU to detect task switches and initiate the configuration swap process on its own by a state machine. As long as the unit's memory access cycle is faster than the task context switch of the software system, valid values in the registers can be ensured.

With this knowledge, the module can calculate the memory addresses of the used counter within the TCB by adding specific offsets. These calculations take the hardware configuration into account. The state machine handles the memory swap by swapping out all the current register values into the TCB of the to-be-suspended task and reading data from the new task's TCB into the PMU registers. Task-aware and task-belonging measurements must be stopped during this process. To avoid losing measuring cycles during this work, every counter type must have its own compensation, as a task continues to run before the measurements can be started again. It is also essential to differentiate between counter types, as far as overwriting of configuration and counter values are concerned. For example, a task-aware counter configuration must not be overwritten during this activity. However, the swap from the registers into the TCB is made, no matter which counter type is programmed. This swapping ensures consistent data in the memory for the OS to use valid data of a suspended task from within another task to profile it.

PMU interface to the software

Since the PMU knows where the OS manages the pointer to the TCB of the currently running task, a simple interface between hardware and software can be created. Certain measurements must be started immediately after a task switch, so the OS must set this value first during a

context switch. This allows the PMU to start its swapping process while the OS executes the rest of its register context switch.

Additionally, the system's data structure must be equivalent to the one used in the buffering process of the PMU to ensure consistent and valid data storage. Due to this, the OS must be well aware of the hardware configuration of the unit and vice versa, as far as the number of counters and configuration registers is concerned.

Specific addresses within the CSR file represent the registers, so they can also be accessed via simple instructions defined by the RISC-V ISA [29]. Because the addresses of the counter values are aligned with the RISC-V definition for hardware performance counters [36], the keywords `hpmcounterN` and `hpmcounterNh` can be used where N is defined within the interval $[3, 31]$. The user must know that the PMU starts measuring at the cycle the main configuration register is set to a valid value. To stop and reset the unit, `0x0000` must be set. Additionally, the configuration of counters belonging to a task must be done within the context of the corresponding task to ensure proper measurements. All the accesses to registers provide the current values in the counter registers of the currently running task. For every task, however, the corresponding TCB contains the counter values of the last swapping process. Hence, these values provide snapshots of the previous task execution. The values can be used to easily profile all tasks from within another dedicated task or the OS itself to gain information concerning execution times or events.

Relevance for the dynamic runtime-reconfiguration of *more*MCU

While the PMU is a general-purpose hardware unit, it also has many features that the dynamic runtime reconfiguration of the *more*MCU can benefit from. As explained in Section 5.4, the OS provides profiling information from the PMU to the developer to facilitate the decision of whether having a hardware accelerator for a particular functionality is beneficial. In the current implementation, the OS uses the PMU for three different purposes:

1. Instruction call count: This counter increments whenever a custom instruction is called. It is a global counter and provides the overall call count of an instruction.
2. Instruction execution time: This counter is of type counter belonging to a task, similar to *cnt 3* depicted in Figure 5.9. It measures the time between the program counter of the call of the instruction and the next program counter. For emulated instructions, this yields the emulation execution time; for natively supported instructions, the counter yields the instruction execution time (cf. Figure 5.10).
3. Reconfiguration time: This counter measures the time the reconfiguration controller is busy. It is a global counter and provides the reconfiguration time for a particular RP.

Depending on the first two counters, the OS can determine whether a hot swap of an instruction is beneficial to the system. The third counter is only relevant for profiling the actual reconfiguration process.

5.3 Application Software Interface

As previously explained, the hardware of the *more*MCU is now designed to allow partial reconfiguration at runtime. It can also monitor the performance of certain hardware and software parts with its PMU. Now it is necessary to determine how the ASW can beneficially use these features.

Algorithms in many use cases can benefit from application-specific hardware acceleration that may not be available in all target processors. While compilers will still generate binaries (from, e.g., C code) for the target architecture, we take a different approach by simply calling an assumed processor instruction `cinsi` that shall provide the required acceleration. Listing 5.1, line 102 shows an example: The target processor might or might not natively support the `cinsi` instruction. Figure 5.10 depicts the different ways of executing the instruction: If not (yet) supported, the MCU will raise an illegal instruction exception. Then, the functionality is (a) emulated by a code sequence (e.g., a C function) injected into the execution flow by the OS. Once available (b), the instruction will execute like any other instruction of the original ISA. In any case, the emulating function or the required logic comes with the application code, and the code execution is transparent to the application. However, while it yields the same result, e.g., the timing behavior might differ. Although this interface provides a potential entry point for attacks, security is not addressed in this dissertation. One concept that deals with protection mechanisms in hardware/software co-designed platforms in this context is proposed by Malenko et al. [140].

In this work, we execute Listing 5.1 on a RISC-V architecture. Within its original ISA [29, 36], the `cinsi` instruction (meaning **c**ustom **ins**truction **i**mmediate) in line 102 is unknown so that we can explain our concept as well as the behavior of hardware and software. In order to provide a consistent API to application developers, the OS offers functionality to register emulation code for a custom instruction. Listing 5.2 shows an example C code to make this clear. Line 16 registers the emulated instruction. The parameter `0x1B` is the opcode of the new instruction, `cinsi_emu` is a pointer to the emulating function in line 10, and `CINSI_ID` is the module index in the bitstream repository (cf. Section 5.2.4) for partial reconfiguration. The parameter `instr` of the emulating function in line 10 is the corresponding instruction word of the instruction that triggers the emulation. This instruction word (cf. [29]) includes information like the opcode, the source, and destination registers. By passing this information, the emulating function knows where to find the parameters and how to provide the return values. After registering the instruction, a task can use the instruction as sketched in line 20 by generating the required encoding for the binary. This line in the C code corresponds to

```
100    [...]
101    addi  t4, zero, 12
102    cinsi t3, t4,   10
103    [...]
```

Listing 5.1: Example assembly code with the custom instruction `cinsi`.

```
10  void cinsi_emu( uint32_t instr ) {
11    // [... code for emulating the cinsi functionality ...]
12  }
13
14  OS_TASKENTRY(task1) {
15    // [...]
16    addEmulInstr(0x1B, cinsi_emu, CINSI_ID);
17    // [...]
18    while(1) {
19      // [...]
20      asm(".insn i 0x1B, 0, t3, t4, 10");
21      // [...]
22    }
23  }
```

Listing 5.2: Example C code using the custom **`cinsi`** instruction.

line 102 in the assembly code in Listing 5.1. The syntax represents a custom instruction call within a RISC-V compiler. With this approach, standard RISC-V compilers can be used without modification. The new functionality is only introduced within the system's application code. Therefore, code execution on non-altered standard-compliant RISC-V cores remains possible, as illegal instruction exceptions must be supported by every type of core [36], and the OS handles these.

5.4 Extended Operating System Concepts

As the *more*MCU platform is a hardware/software co-designed platform, the OS is of vital importance for exploiting its features. For this case, *Smart*OS is extended to be capable of managing the *more*MCU and supporting its reconfigurability. This allows to answer the question, *"How can future OS architectures support flexible and changing software and hardware to keep systems operational in the long run?"*.

From an application perspective, the OS provides API functions to register instructions for emulation in case they are not natively supported. The actual mapping to partial bitstreams in the bitstream repository is facilitated by linking a module index to the emulating function (cf. line 16 in Listing 5.2). The reconfiguration controller automatically chooses a suitable partial bitstream when it performs the partial reconfiguration at runtime. The Partial Reconfiguration Manager of the OS layer (cf. Figure 5.1[p60]) is responsible for triggering the actual reconfiguration (cf. Section 5.4.2).

There is also some profiling using the built-in PMU in place to gather information about the relevant instructions (cf. Section 5.2.6). The OS can use this measured information to facilitate the decision of whether the system would profit from hardware acceleration for an instruction or a peripheral or not. While the *more*MCU provides measurements in

this regard, analyzing the gathered values, decision-making, and replacement strategies are highly use-case-dependent. Thus, the concept only provides the values and leaves the actual implementation to the engineer.

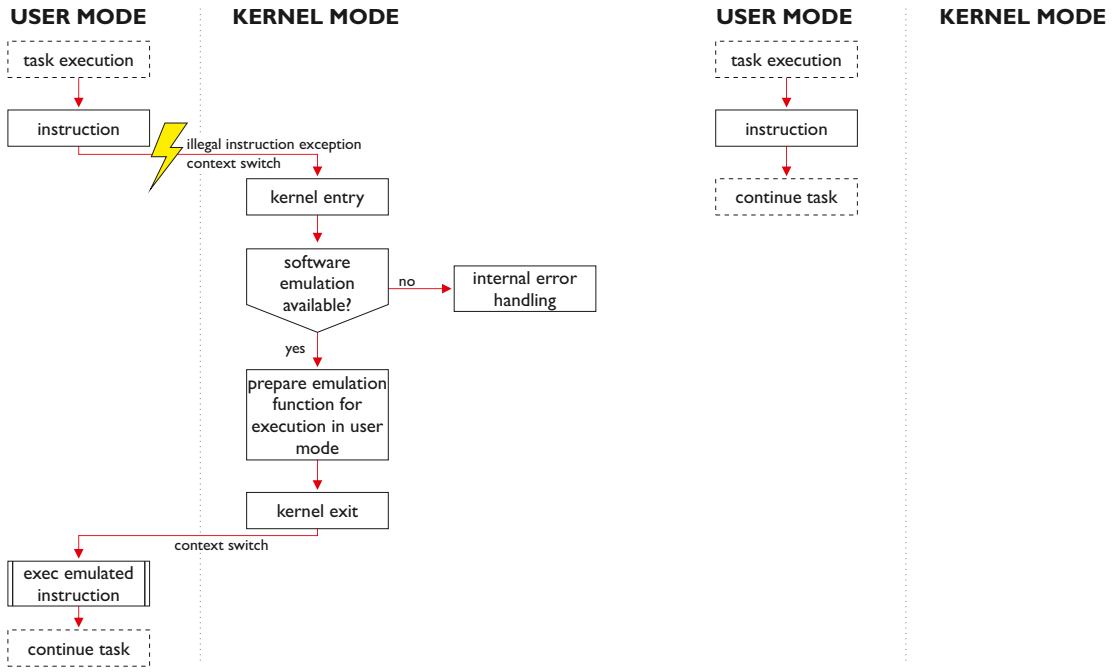
The approach ensures that standard-compliant RISC-V compilers can still be used. This compliance is because the emulation handling only uses well-defined behavior (cf. illegal instruction exceptions [36]) to which every RISC-V core must adhere in any case. Conversely, this also means that the entire software stack can be executed on any non-altered, standard-compliant RISC-V MCU without restrictions. The software runs slower in this case since there is no possibility of hardware acceleration. Moreover, access to the reconfiguration controller is only possible if it is also available within the MCU.

5.4.1 Flexible Instruction Set Handling

To support a flexible ISA, *Smart*OS must know how to handle initially unknown instructions within the ASW. To do so, it contains a Flexible ISA Manager module (cf. Figure 5.1). An illegal instruction exception occurs (cf. Figure 5.10a[p74]) within the core whenever an instruction is not natively available in the ISA. While this exception usually indicates corrupted software, the OS uses it and triggers the execution of the missing instruction's emulation functionality. If the application wants the OS to emulate an instruction using this technique, then it must be registered beforehand. For the application code in Listing 5.1, we thus follow two different software execution flows (cf. Figure 5.10):

(a) To **emulate** `cinsi`, *Smart*OS follows the execution flow depicted in Figure 5.10a: Right after fetching and decoding the instruction, the processor throws an illegal instruction exception, as it is unaware of the `cinsi` opcode. The OS catches this exception for consistent handling across all applications. Therefore, the application is interrupted, and the OS enters kernel mode. Then, the OS processes the binary encoding of the current instruction, extracts essential information, e.g., opcode and operands, and checks its internal data structures for a corresponding emulated function that was registered before. If the lookup fails, *Smart*OS treats the instruction as illegal and internal error handling proceeds (e.g., terminate the application). If an emulation for the instruction exists, the corresponding function is prepared for execution in the calling task's context. The registered function is executed in user mode upon kernel exit and restoration of the calling task's context. The task can continue normally at its privilege level, and finally, the task directly continues further code execution.

(b) **Natively** supported instructions in the ISA follow the software execution flow from Figure 5.10b: Whenever a previously emulated instruction is added to the processor's pipeline, the application does not need to enter the kernel. Instead, the MCU executes the new instruction like any other instruction. The difference to the emulated case is that no kernel overhead is introduced, as the processor core of the MCU executes the instruction (cf. Section 5.2) as any other instruction in the ISA.



(a) *Smart*OS tries to emulate the instruction, as it is illegal or unknown to the processor.

(b) The instruction is natively supported by the processor.

Figure 5.10: The different software flow diagrams depending on whether an instruction is unknown or illegal, or natively available within the ISA.

5.4.2 Runtime Reconfiguration Handling

After deciding[3] that a modification is beneficial to the system, the OS triggers the actual reconfiguration of the logic. For this purpose, the Partial Reconfiguration Manager (cf. Figure 5.1) of the OS indicates to the reconfiguration controller in hardware whenever a custom instruction or flexible on-chip peripheral is to be introduced to or stripped from the system at runtime. This *Smart*OS decision-making can be based on different metrics, e.g., instruction call count and duration. It can be adapted according to the system's use case. Once it decides to reconfigure *more*MCU, it uses the interface of the reconfiguration controller (cf. Figure 5.6[p65]) as follows: At the beginning, the controller must be requested. This request is made by setting the `req` bit in the control register `CTRL_REG`. Access is granted once `gnt` in the status register `STATUS_REG` is set. Subsequently, the index of the selected runtime-reconfigurable module must be set in `MODULE_ID_REG` alongside the preferred RP index in `RP_FREE_REG`. By setting the `valid` bit in the control register, the reconfiguration is requested to start. Whenever `busy` is reset again, the process is completed or has failed, according to the `error` flags. `RP_SEL_REG` shows the actual selected RP – also, in case the reconfiguration was triggered by a hardware device. Since these are all memory-mapped registers, the typical protection mechanisms can be used to avoid unauthorized access.

[3] As explained before, the decision-making depends on the use case and is left for the developer to implement.

5.5 Evaluation

This section evaluates the different features of the *more*MCU platform. It starts with evaluating the flexible ISA and its hardware acceleration performance gain capabilities by examining several use case examples. Subsequently, the runtime reconfiguration capabilities of the *more*MCU are evaluated. In this regard, the MCU itself and its reconfiguration controller are examined, and a proof of concept is also provided. Finally, general observations and limitations of the entire concept are sketched.

5.5.1 Flexible ISA Performance Gain

The experimental setup for this evaluation comprises a *more*MCU at 50 MHz clock speed running on a Xilinx XC7A35T FPGA [40] on a Basys3 board [141]. For I/O, the MCU contains a UART interface and a General-Purpose Input/Output (GPIO) module. The UART of the MCU directly connects to a USB socket, and some GPIO pins to onboard LEDs – both used for debugging and status messages. For timing measurement with a PicoScope 2205 MSO [142] or a Keysight MSOS254A 2.5 GHz oscilloscope [143], further GPIOs come into play. We use Xilinx Vivado 2020.1 WebPACK running on Ubuntu 18.04.6 LTS for hardware simulation and synthesis.

The MCU runs *Smart*OS, and the test application contains one task using OS features and the PMU to monitor different runtime aspects (e.g., execution times for code sequences or counters for calls to application-specific instructions).

Use Case 1: Multiply and Accumulate

A very common operation within digital signal processing for, e.g., control systems is the multiply-and-accumulate operation. It calculates sums of products of arbitrary, bounded values, and can follow an equation like

$$y = \sum_{i=1}^{n} x \cdot i, \tag{5.1}$$

where x and n are parameters of the function y that the `cinsi` instruction will later compute. For further evaluation, the values $x = 12$, and $n = 15$ are assumed to achieve reproducible and comparable measurements. When implemented as a regular loop, the calculation is computationally expensive, resulting in $\mathcal{O}(n)$ complexity. As the final execution time is not constant but depends on n, this might raise issues for real-time systems that rely heavily on such functions.

Listing 5.3 shows the assembly code for executing the instruction for this use case. The binary encoding of the I-type instruction [29] in line 101 is depicted in Figure 5.11.

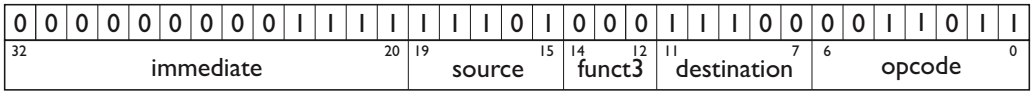


Figure 5.11: The binary encoding of the instruction in line 101 in Listing 5.4.

```
100    [...]
101    cinsi t3, t4, 15
102    [...]
```

Listing 5.3: Assembly code for executing the instruction in Use Case 1.

The binary encoding of the instruction contains the opcode of `0b0011011` (`0x1B`) alongside the source (`0b11101` for register `x29` / `t4`) and destination (`0b11100` for register `x28` / `t3`) registers and the immediate value `0b000000001111` (`0x0F` / 15). The parameter and return value passing happen through these registers and the immediate. The field *funct3* is unused in this context. Listing 5.4 shows the custom instruction in Line 27 within a *Smart*OS task. The emulating function in line 10 illustrates the multiply-and-accumulate operation implemented in software. As the original ISA of *more*MCU does not support vector instructions, this must be implemented in a loop. This function is designed to be called every time a new value x must be processed. This value x represents, e.g., a new sensor reading within the system.

This evaluation addresses potential performance improvements through the flexible ISA concept. The computationally expensive algorithm follows three implementation options, A to C, to show these improvements.

```
10   void cinsi_emu( uint32_t instr ) {
11     // code for mapping registers
12     // from instr to variables x, y, and n
13     uint32 y = 0;
14     uint32 i;
15     for(i = 1; i <= n; i++)
16     {
17       y += x * i;
18     }
19     // code for return value passing
20   }
21   OS_TASKENTRY(task1) {
22     // [...]
23     addEmulInstr(0x1B, cinsi_emu, CINSI_ID);
24     // [...]
25     while(1) {
26       // code for aquiring new sensor values, etc.
27       asm(".insn i 0x1B, 0, t3, t4, 15");
28       // [...]
29     }
30   }
```

Listing 5.4: The execution of the instruction in Listing 5.3 in the context of a *Smart*OS task alongside its emulating function.

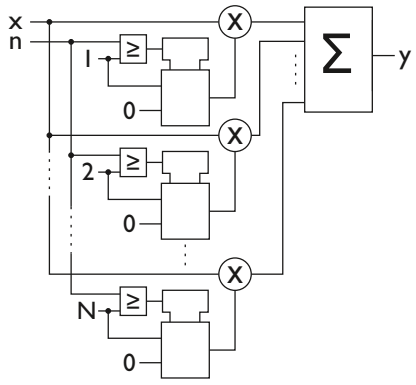


Figure 5.12: Logic for calculating a sum of products in hardware.

A) Implementation in software in the traditional way, i.e., without our hardware modifications.
B) Implementation as emulated instruction as explained in Section 5.4 (cf. Listing 5.4).
C) Implementation in hardware by adding the instruction to the ISA at runtime, as explained in Section 5.2. Figure 5.12 shows the logic of the new instruction.

For implementation options B and C, the same application code can be used for calling the instruction (cf. Listing 5.1), while implementation option A needs separate coding.

Implementation option B requires the application software to provide the emulation as a function (to be executed within the illegal instruction exception), and implementation option C requires the application to provide the corresponding logic for partial reconfiguration. The partial modification mechanism triggered by the OS adds the new instruction to the ISA. The partial modification comprises two steps:

S1 The new logic is placed into an RP of the EX stage of the pipeline. The logic is added as a new function slice comparable to other Arithmetic Logic Unit (ALU) instructions.
S2 The new opcode is introduced to the programmable decoder, located in the ID stage of the pipeline. This procedure marks the new opcode as "legal"; therefore, the stage does not throw an illegal instruction exception.

The logic reconfiguration in S1 is carried out on-the-fly (in parallel to ongoing MCU operation) and therefore does not introduce interruptions or delays to the system. When step S2 finishes, the application software automatically uses the newly introduced instruction. Section 5.5.1 shows the consequences regarding the execution time. The significant advantage of implementing Equation 5.1 in hardware is the possible parallelism. Depending on the available logic cells in the FPGA, m multiplications can be parallelized, and (together with the adder) the entire function might require just one clock cycle ($T_{instruction} = 1\,\text{cycle}$) in the EX stage. As long as $n \leq m$, the execution time reduces to 1 cycle. The effective speedup (cf. Section 5.5.1) for this case is

$$S_{eff} = \frac{T_{loopbody} \cdot n}{T_{instruction} \cdot \lceil n/m \rceil} = n \cdot \frac{T_{loopbody}}{1\,\text{cycle}}. \tag{5.2}$$

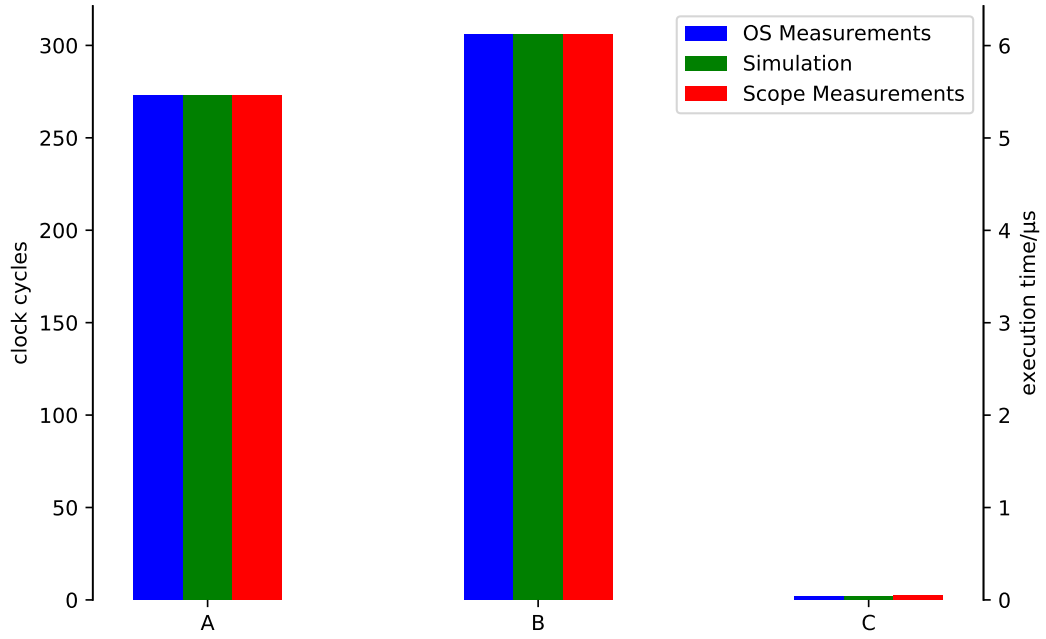


Figure 5.13: Runtimes of the implementation options A-C for Equation 5.1.

Thus, the developer needs to trade execution time against logic consumption on the FPGA; and can still revise the decision at runtime. However, data type restrictions may apply, depending on which logic elements are available on the used FPGA.

The proof of concept evaluated in the following sections takes the following progression:

(i) Implementation option A is the baseline of our evaluation.

(ii) Implementation option B adds flexibility but has an expectedly slight increase in the execution time due to OS overhead for emulation.

(iii) Implementation option C adds the instruction logic to the pipeline and expectedly removes the OS overhead while significantly reducing the execution time compared to B and A.

Use Case 1: Measurements and Evaluation

This section discusses the runtime effects of our approach, followed by an analysis of the resource consumption of both software and hardware. Both parts refer to the scenario sketched in Section 5.5.1.

Execution Time Table 5.1 and Figure 5.13 show the runtimes of the implementation options A-C for Equation 5.1. The measurements were carried out in three different ways: by pure simulation of the processor logic, system self-observation through OS profiling in the actual hardware, and attaching an oscilloscope to the debug interface of the same hardware. The cycle count is determined by measuring how long a certain program counter remains in the EX stage of the pipeline. Hence, the PMU is directly attached to the pipeline stage register. This value is also wired to the debug interface for the oscilloscope measurements.

Table 5.1: Runtime Evaluation.

#	Implementation Option	t_{exec}	cycle #
	OS profiling (cycle-accurate PMU measurements)		
0	A – traditional software	≈5.46 µs[a]	273
1	B – instruction emulation	≈6.12 µs[a]	306
2	C – instruction in hardware	≈20 ns[a]	1
	Simulation[b]		
3	A	5.46 µs	273
4	B	6.12 µs	306
5	C	20 ns	1
	Oscilloscope measurements[c] (using debug ports/pins)		
6	A	5.461 µs	≈273[a]
7	B	6.122 µs	≈306[a]
8	C	21.667 ns	≈1[a]

[a]calculated values using $f_{clk} = 50\,\text{MHz}$ and $T_{clk} = 20\,\text{ns}$.
[b]Xilinx Vivado 2020.1 WebPACK.
[c]Keysight MSOS254A 2.5 GHz [143] measurement.

The values show different effects – not only in simulation but also in the running system: First, the emulation of the instruction (option B) takes 33 cycles[4] more when compared to the baseline implementation (option A). However, the actual OS code injection overhead that is added to the pure functionality of the emulation cannot be seen by this calculation. This is since the function in option A also has a certain amount of overhead due to its prologue and epilogue. The actual OS overhead sums up to 83 cycles, as elaborated in Table 5.6. Thus, the increased ISA flexibility is initially paid with performance loss. However, a significant improvement in execution time can be reached when using option C. This option removes the OS overhead entirely and simultaneously ensures a constant runtime of 1 cycle. We also want to point out explicitly that the simulation results are equal for the self-measurements with internal profiling features and for the scope measurements that confirm our simulation. Figure 5.13 shows the similarity of the three bars for every implementation option. The exact match between simulation and OS profiling was to be expected, as the PMU uses the same clock frequency as the core. This enables the system to perform cycle-accurate self-measurements. It also matches the calculated values from the oscilloscope measurements. In fact, it is required by design that all these values are identical since deviating numbers would indicate an error in the system. The following part of the evaluation only depicts the PMU measurements.

[4]These numbers differ from Publication R-II, as optimizations were applied after publication.

Resource consumption: Table 5.2 shows the resource consumption of different hardware variants of the *more*MCU on our target FPGA. Certain features like I/Os, BRAMs, and clocks are not considered, as they stay the same throughout the variants.

- **Variant I** is the basic CV32E40P [38] core without further modification, thus not supporting the reconfiguration concept. It executes implementation option A only.
- **Variant II** directly adds the `cinsi` instruction to the ISA of Variant I, but without our concept for adding further ISA extensions. This variant only shows the resource consumption in a static system with hardware support for `cinsi`.
- **Variant IIIa** supports our concept of partial reconfiguration, but initially without the logic for the `cinsi` instruction.
- **Variant IIIb** equals Variant IIIa, but after adding the `cinsi` instruction at runtime into the corresponding RP.

Thus, variants I and II come with a static ISA. In contrast, variants IIIa and IIIb support a flexible ISA and refer to implementation options B and C.

Table 5.2: Resource Consumption on the FPGA.

	static ISA		flexible ISA	
	Variant I	**Variant II**	**Variant IIIa**	**Variant IIIb**
Latches	2628	2628	2496	2496
LUTs	7597	8685	6929	7913
MUX	430	430	486	486
Carries	234	378	222	366

As one can see, Latches and Multiplexers are the same for I and II (static ISA) and for IIIa and IIIb (flexible ISA). This equality can be explained as the added logic is purely combinatorial and only creates signal lines driven by one source. However, LUTs and Carries increase from I to II and IIIa to IIIb, as they implement the added `cinsi` logic.

When comparing the variants I and IIIa, it is interesting to see that introducing the flexible ISA through partitions on the FPGA initially leads to even less resource usage (apart from the multiplexers). A possible explanation is that a more efficient way of logic cell placement can be found during synthesis when partitioning the system into different parts. The subsequent routing of these separate parts can then be achieved with fewer resources. All values were obtained by using Xilinx Vivado 2020.1 WebPACK[5].

Use Case 2: Moving Average

The moving average filter is a widespread algorithm to process sensor values on embedded systems. It calculates the arithmetic average over the last N values and is defined[6] as

$$y = \frac{1}{N}\left(x + \sum_{i=1}^{N-1} x_i\right) = \frac{1}{N}\cdot x + \sum_{i=1}^{N-1} \frac{1}{N}\cdot x_i, \tag{5.3}$$

[5] `www.xilinx.com/products/design-tools/vivado/vivado-webpack.html`

[6] For the sake of simplicity, the implementation for this evaluation uses 32 Bit integers only. Hence, multiplications and divisions yield round-off errors.

where x is the only variable parameter of y that the **cinsi** instruction will later compute.

In order to give specific values in time a certain weight, the weighted moving average filter is used. It is defined in an equation like

$$y = w_0 \cdot x + \sum_{i=1}^{N-1} w_i \cdot x_i, \tag{5.4}$$

where x is again the only variable parameter, as the weights w_i are hard-coded within the hardware. A common case is a "linear weighted moving average", where the most recent value is weighted highest, and the weight distribution decreases linearly. For the sake of this use case that only addresses performance gain, the values of the individual weights are not relevant.

The source code for calling **cinsi** in this use case is the same as in Listing 5.4 – with the updated algorithm in the emulating function, of course. Contrary to the stateless multiply-and-accumulate algorithm, the moving average algorithm is a stateful one. Hence, it needs the history of the past $N-1$ values x_i for future calculations of y. In software, this is facilitated by an array of length $N-1$. In hardware, a shift register with the length $N-1$ is used to implement the algorithms, as depicted in Figures 5.14 and 5.15. It should be explicitly noted that this is not a vector instruction taking a whole series of values as input. Only the most current value is passed and then stored internally. Of course, dealing with memory components (e.g., flip-flops, registers) introduces the importance of data consistency when updating the logic, meaning that the contents of the memories must either be retained or dealt with in a retaining manner – depending on the requirements of the use case.

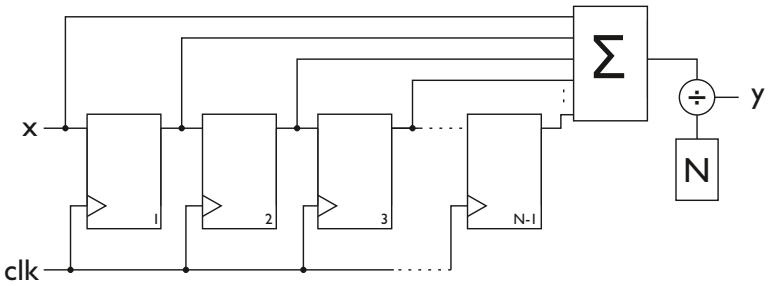


Figure 5.14: Hardware structure of a moving average filter.

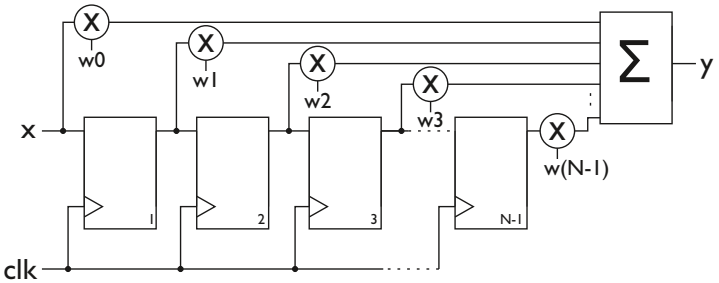


Figure 5.15: Hardware structure of a weighted moving average filter.

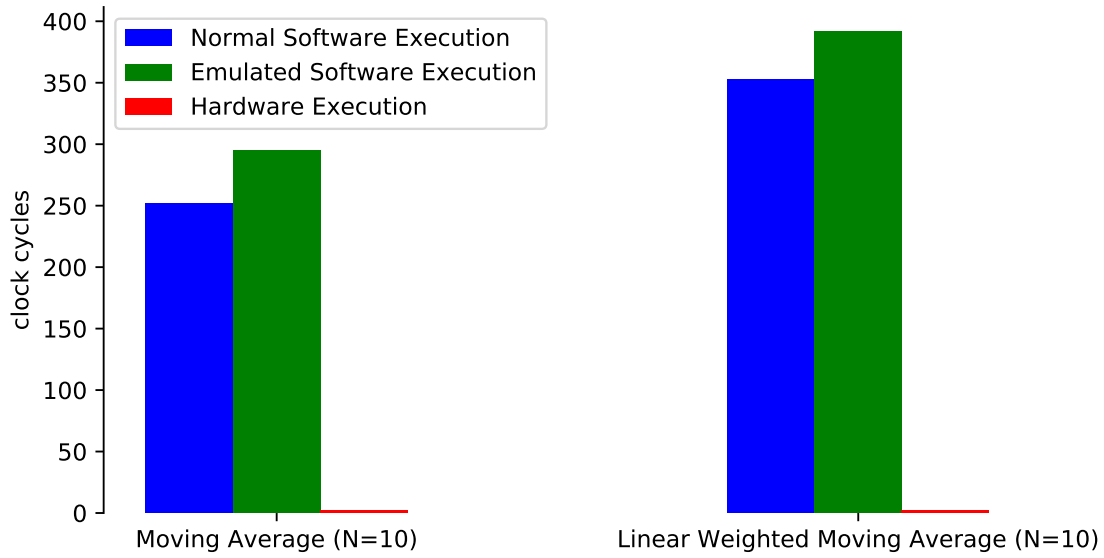


Figure 5.16: Runtimes of the different implementation options for both moving average (cf. Equation 5.3) and linear weighted moving average filter (cf. Equation 5.4).

This use case shows another prominent feature of our concept: maintenance/update of a previously added instruction in the field. Imagine a progression in the system's lifetime, where an algorithm was initially introduced in software. After some time, a hardware implementation of the functionality was added as a new instruction to the ISA. Even later, the implementation shows to be incorrect or needs an update due to changed application requirements. Therefore, an update of the instruction within our ISA is necessary to switch from the moving average to the *weighted* moving average filter to overcome the deficiencies in the already deployed system.

Use Case 2: Measurements and Evaluation

Similar to Section 5.5.1, this section analyzes Use Case 2 concerning execution time improvements and resource consumption.

Execution time: Figure 5.16 shows the runtimes of the different implementations (normal, emulated, and hardware execution) of the two algorithms, Equations 5.3 and 5.4. For the evaluation, we assume $N = 10$.

Compared to the use case before, emulating the instruction introduces a similar overhead, but the implementation in hardware once more yields a massive performance improvement. The first two implementations have a complexity of $\mathcal{O}(N)$. In contrast, the implementation in hardware again executes in 1 cycle. Again, OS measurements, simulation, and scope measurements show the same results.

Resource consumption: Table 5.3 outlines the resource consumption of three different hardware implementations. The corresponding logic modules can be put into the corresponding RP:

- **Normal MA**: The logic for implementing a standard moving average filter (cf. Equation 5.3).
- **Weighted MA**: The extended logic for the implementation of a linear weighted moving average filter (cf. Equation 5.4).
- **Combined**: Both modules mentioned before are combined into one module, making two different instructions available for systems where both algorithms are required.

Table 5.3: Module Resource Consumption on FPGA.

	Normal MA	Weighted MA	Combined
Latches	288	288	576
LUTs	947	1066	1937
MUX	0	0	0
Carries	128	161	289

As shown in Table 5.3, the number of required latches stays the same for both normal and weighted moving average, as both modules need to save the same amount of data. For 32 Bit datatypes, this is $(N-1)\cdot 32\,\text{Bit} = 9\cdot 32\,\text{Bit} = 288\,\text{Bit}$. The combined module requires exactly the sum of both individual variants. LUT requirements, however, increase dramatically from the normal to the weighted module. This is because this variant requires performing a number of additional multiplications with the weights (cf. Figure 5.15) to calculate the final result value y. The combined module, in this case, can optimize and reuse some LUTs for either algorithm, as their count is slightly less than the sum of both modules. None of the modules uses any multiplexers. Carries increase a bit from one module to the other; the combined module, however, requires the sum of both modules again. The values for the resource consumption were obtained in the same way as in Section 5.5.1.

Observations and General Conclusions

From these two use cases, it can generally be concluded that flexible ISAs can be nicely used to accelerate parallelizable algorithms. Of course, one is still limited by the utilized hardware resources (both in number and type). For example, an RP of a specific size can only contain a certain number of multipliers for hardware acceleration. The achievable speedup S depends on the complexity of the algorithm as well as its parallelizability p. The effective parallelizability p_{eff} is derived from p and the available hardware resources. If the entire structure cannot be completely realized in hardware, sequential calls to the instruction can still be used to speed up the algorithm. This results in an effective speedup

$$S_{eff} = \frac{T_{loopbody} \cdot n}{T_{instruction} \cdot \lceil n/p_{eff} \rceil}. \tag{5.5}$$

The time $T_{loopbody}$ corresponds to the execution time of one loop iteration. In contrast, $T_{instruction}$ is the execution time of the `cinsi` instruction.

For Use Cases 1 and 2, where $n \leq p_{eff}$ as long as $n \leq N$ (N being the number of hardware modules for the acceleration) and $T_{instruction} = 1\,\text{cycle}$, this equation yields an effective speedup $S_{eff} = n \cdot {}^{T_{loopbody}}/_{1\,\text{cycle}}$. If p_{eff} is reduced to ${}^{n}/_{2}$, the effective speedup is also reduced to

$$S_{eff} = \frac{T_{loopbody} \cdot n}{T_{instruction} \cdot \lceil \frac{n}{n/2} \rceil} = \frac{T_{loopbody} \cdot n}{T_{instruction} \cdot 2} = \frac{T_{loopbody}}{1\,\text{cycle}} \cdot \frac{n}{2}. \tag{5.6}$$

This speedup calculation does not consider the overhead imposed on the algorithm but only the pure functionality of the implementation.

5.5.2 Runtime Reconfiguration Evaluation

This section evaluates the *more*MCU and its runtime reconfigurability features. Contrary to the evaluation before, the reconfiguration controller and the bitstream repository are also evaluated here. Hence, the following experimental setup moves to a Nexys4-DDR board [144] with a Xilinx XC7A100T FPGA [40] as a target. This move is because this board has more hardware logic resources and a bigger onboard flash memory for the bitstream repository to evaluate the runtime reconfiguration. All other tools and configurations for obtaining measurements and values stay the same as in Section 5.5.1.

As the *more*MCU platform can be used in a modular way, the following configuration is chosen for the evaluation: A *more*MCU with three RPs for custom instructions (each with 2400 LUTs and 4800 FFs) as well as two RPs for flexible peripherals (each with 400 LUTs and 800 FFs) – both connected to the main Wishbone interface. The bitstream repository is located in an onboard 16 MiB Quad SPI (QSPI) flash memory and is accessed by the bitstream data

Table 5.4: Resource Utilization of *more*MCU.

	Module	Resource Utilization LUT	FF	BRAM
full *more*MCU	top[†]	8310	3509	17
	Core	6626	2539	0
	EX stage	1132	118	0
	Decoder	165	21	0
	Programmable Decoder	20	21	0
	Peripheral Controller (including memories)	312	151	16
	Reconfiguration Controller and Wishbone interface	513	494	1
	Bitstream Data Interface	577	289	0
w/o extens.	top[†]	6857	2641	16
	Core	6509	2418	0
	EX stage	1067	118	0
	Decoder	124	0	0
	Peripheral Controller (including memories)	279	151	16

[†]In top, the core is connected with all on-chip peripherals and memories.

interface on the FPGA side. Memory-wise, 32 KB RAM and ROM are allocated, respectively, located directly within the FPGA in BRAM cells.

*more*MCU resource utilization

Table 5.4 shows the resource utilization on the FPGA of the top modules of the *more*MCU in the configuration explained above compared to a *more*MCU without the proposed extensions. In addition, some essential submodules of the corresponding top module are explicitly described. Modules that are not shown are irrelevant for the comparison because they are equal in both cases (such as the interconnect and some peripherals).

It can be seen that the extensions within the RISC-V core only require 1.7 % more LUTs and 5 % more FFs. As expected, these increases are attributable to the pipeline's EX stage (+6 % LUTs) and the decoder (+35 % LUTs and 21 total FFs). The decoder adds 20 LUTs and all of the 21 FFs within the newly introduced programmable decoder. The remaining difference is attributed to the interfacing. The peripheral controller connecting the core and the memory-mapped components (e.g., on-chip peripherals, interconnects) also adds 33 LUTs compared to the non-extended MCU. This resource consumption is because interfacing to the flexible peripherals must also be done here. Overall, it can be said that the resource overhead concerning LUTs is 21 % and concerning FFs is 33 %. The partitioning of the MCU has little to no effect on resource utilization. The values for the resource consumption were obtained in the same way as in Section 5.5.1.

Reconfiguration Controller Evaluation

As several reconfiguration controllers are already available for different systems, Table 3.1[p26] gives a comparison. Table 5.5 adds one line to this table to show how the reconfiguration controller of the proposed *more*MCU (*more*MCU-RC) performs. In addition to resource utilization,

Table 5.5: *more*MCU Reconfiguration controller key figures in comparison with Table 3.1[p26].

Reconfiguration Controller	Resource Utilization LUT	FF	BRAM	Interface	Software
ZyCAP [78]	620	806	0	AXI	drv
AC_ICAP [79]	1286	1193	22	PLB/direct	-
RT-ICAP [80]	190	88	0	direct	drv
RV-CAP [81]	420	909	0	AXI	drv
D²PR [82]	249	112	0	direct	-
Vipin et al. [83]	586	672	8	PLB/AXI	-
ICAP-I [84]	177	303	0	direct	-
DPRM [85]	109	77	0	PLB/XPS/dir.	-
AXI_HWICAP [86]	546	741	2	AXI	-
XPS_HWICAP [87]	741	745	3	XPS	-
PRC [88]	1171	1203	0	AXI	-
***more*MCU-RC**	513	494	1	WB/direct	OS

comparison criteria include the interface that is provided and the software support, indicating how easy it is to incorporate the solution into existing systems.

From the numbers, one can see that some implementations are using fewer resources than the proposed one. However, most of them do not include the resources of the interface controller in their numbers, or they only feature a direct, register-based access interface without hardware overhead in this regard. From the bus-connected implementations, only RV-CAP [81] has similar numbers (but significantly more FFs), even though no OS-awareness is included in this solution. From a software point of view, only a few implementations feature simple driver concepts (drv), and not a single one is fully backed by an OS implementation – *more*MCU's reconfiguration controller is. Its Wishbone bus interface is unique among the solutions, where AXI dominates, followed by PLB.

*more*MCU proof of concept

This proof of concept uses the previously introduced moving average filter instructions shown in Section 5.5.1 in a demonstration scenario. Therefore, the OS overhead measurements also stay roughly the same as previously discussed.

The evaluation application software on top of *Smart*OS registers three instructions for emulation and hardware acceleration. The OS measures the number of calls and the duration of the execution of the emulated functions. Introducing and stripping custom instructions in and from hardware is triggered periodically for demonstration and evaluation purposes. The same applies to flexible peripherals. Figure 5.17 depicts the partitioning of the FPGA for this use case. The most exciting metrics for the evaluation are the reconfiguration time and the OS overhead, as well as the performance benefit from hardware acceleration by introducing instructions in hardware.

As Table 5.6 shows, the reconfiguration of a flexible peripheral partition takes a total time of 37.957 ms, whereas the reconfiguration of a custom instruction needs 112.422 ms. These times consist of the access of the reconfiguration controller to the flash and its writing to the ICAP. Furthermore, the times depend on how large the target RP is and whether the bitstream

Table 5.6: Reconfiguration time, OS overhead and runtime advantages.

	Measurement	clock cycles[a]	time
reconf.	reconfiguration of flexible peripherals	1 897 838	37.957 ms[b]
	reconfiguration of custom instructions	5 621 104	112.422 ms[b]
	reconfiguration OS overhead	16	320 ns[c]
emul.	instruction emulation OS overhead	83	1.660 μs[c]
	traditional software	659	13.180 μs[c]
	OS emulated instruction	692	13.840 μs[c]
	native instruction	1	20 ns[c]

[a]PMU measurement.
[b]Picoscope 2205 MSO [142] measurement.
[c]calculated values using $f_{clk} = 50MHz$ and $T_{clk} = 20ns$.

is compressed or not. The bottleneck here is the QSPI flash that can only work at 25 MHz and must transfer the bitstream data serially.

As the reconfiguration process happens in parallel to the regular operation of the MCU, no execution delay is introduced to normal code execution. Since the OS starts the reconfiguration (which is then executed in parallel), it is essential to ensure that this start only happens when potentially no other task is interrupted. The start of the reconfiguration introduces an OS management overhead of 320 ns or 16 cycles.

For instruction emulation handling, the OS overhead is 83 cycles. We evaluated a use case with a weighted moving average algorithm that can be executed in software in a "traditional" way within 659 cycles. When choosing the emulated approach, it can thus be executed in 692 cycles. The native execution as a custom instruction drastically improves the cycle amount to 1 cycle. Again, OS overhead and traditional execution do not sum up precisely to the emulated version, as function calls also add a certain amount of overhead to the functionality.

5.5.3 General Observations and Limitations

Instruction types and performance: The proposed approach is neither conceptually nor technologically limited to the presented type of instruction that may be supported. However, some types of instructions can benefit more from hardware support than others. These are mainly based on massively parallelizable algorithms that can be processed concurrently in hardware. The massive parallelization also helps to invalidate the argument of slow clock speeds of FPGAs compared to ASICs. The parallelization helps to compensate for the lower clock speed and leads to an increase in performance.

If multicycle instructions are inserted, no negative consequences must be expected since the relevant structure of the static pipeline portion (e.g., pipeline registers) is used. If used correctly, the pipeline automatically takes care of, i. e., all data dependencies. However, as the instruction complexity increases, the static interface of the pipeline may also have to become more extensive. Since the concept does not specify this interface, this is entirely feasible, but one must consider the possible consequences. While the pipeline's structure can still be used, it is essential to consider the implications of the newly added instruction on other instructions. The acceptance of such an approach for hardware accelerations depends on various factors, e.g., usability.

Long-term operation and maintenance: Since the expected lifetime extension achieved by the proposed concept cannot be described quantitatively in general, a qualitative description is given. Using an FPGA or reconfigurable logic makes it possible to adapt the internal logic at any time. However, the FPGA hardware will become outdated at some point, meaning it might not be possible to make currently relevant functions work. However, it is to be expected that the same logic blocks will continue to be used in the future to build digital circuits. With these existing hardware resources, modifications can still be made if they are urgently needed.

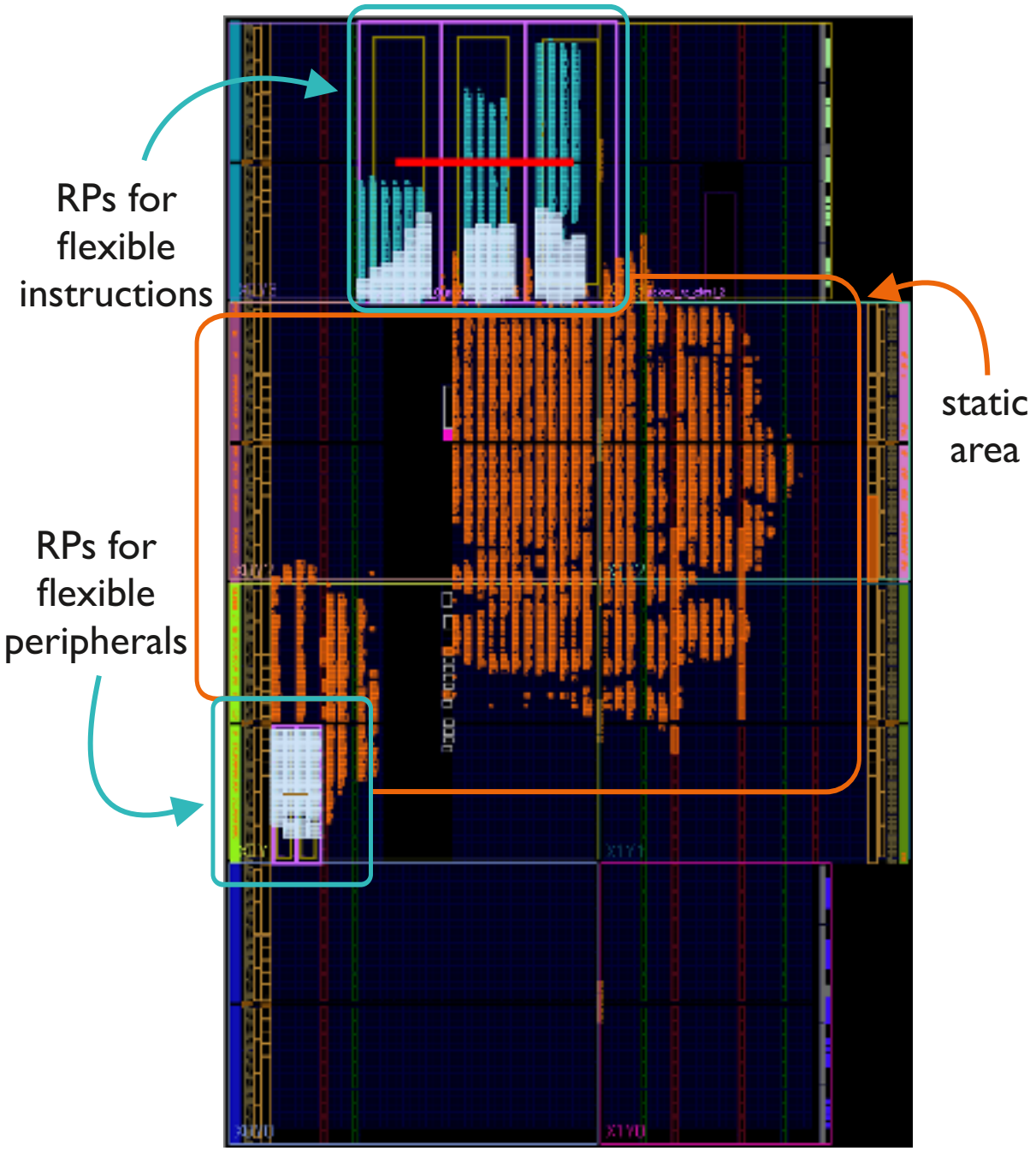


Figure 5.17: The floorplan of *more*MCU partitioned for the use case running on a Xilinx XC7A100T FPGA [40]. In this snapshot, the partitions are already populated by instructions and peripherals.

However, the number of logic cells represents a limit here. Furthermore, it is to be expected that FPGAs will also clock faster in the future. Supposing the requirements of the use case develop towards significantly faster clocking over the lifetime. In that case, one could also run into limits here.

Finally, FPGAs are also subject to an aging process. However, the question is, will the hardware become deficient before the software becomes outdated? Software updates are often no longer distributed as soon as the hardware becomes insufficient. Supposing the hardware lasts longer than the usefulness of the software. In that case, the software can be updated, and the hardware upgraded (considering the limits described before) when using the proposed concept.

Flexibility limits: The RP size and the total number of logic blocks limit flexibility. Also, if the pipeline's static interface explodes or is designed excessively large, many unnecessary resources are tied up. Supposing the logic within an RP requires only a few logic resources. In that case, the unused resources within the RP cannot be used and remain wasted. Conversely, if the logic is too large, it will not fit into an RP and thus cannot be applied. In this case, combining several RPs could be considered, but this requires a new static interface between the RPs.

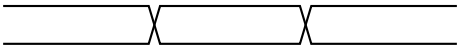
System reset: Resets – whether intentional or unintentional – occur quite regularly in embedded systems. The configuration memory of the used Xilinx FPGAs (amongst many others) is a volatile memory. Hence, it does not retain its contents during a reset. FPGAs normally load the configuration from a non-volatile boot memory in this case.

In most cases, this is sufficient because it restores the default MCU. The accelerated instructions would then automatically be added during the system's runtime. It becomes more complex if the configuration memory's state before the reset needs to be recovered. In this case, the reconfiguration controller must adapt the boot memory configuration at every runtime reconfiguration. The current version of the reconfiguration controller is quite capable of doing this. It can even restore the values stored in the individual hardware registers. For this purpose, however, the stored image must be kept up to date regularly. The selected behavior in the event of a reset must therefore be implemented for each specific application, since the requirements for the behavior can vary.

CHAPTER 6

Conclusion and Future Work

This chapter concludes the thesis and gives an outlook on possible future adaptations. It features a summary of all the work carried out throughout the dissertation and explicitly highlights the novelty once more.



The contributions of this thesis can be divided into two sub-parts: advances in dynamic electronics generation and reconfigurable systems design. The two parts propose concepts that make the design of embedded systems more flexible and sustainable. This is also facilitated by allowing subsequent hardware modifications in the field. The enhanced long-term operation and maintenance capabilities improve the overall sustainability as systems can remain operational for longer. The proposed solutions address problems arising from the established bottom-up embedded systems design process, where "software follows hardware". In this process, the hardware is rarely changed during development. Contrary, the proposed paradigm shift, where "hardware follows software", starts with the development of the application software rather than the hardware.

The approach to advance the **dynamic electronics generation** proposes ***papageno***X, a novel, top-down concept to develop embedded systems. Within this concept, software development marks the first step towards a "hardware follows software" design process. The corresponding PCB for the embedded system can then be generated from a set of requirements the ASW provides. The entire approach is module-based, i.e., systems are composed of pre-defined hardware modules with specific properties that match the requirements. After composing the systems out of these modules, the final PCBs are created from automatically generated intermediate system descriptions. This way, the embedded system's hardware must not be created beforehand but can be automatically generated while or after developing the ASW. Hence, ***papageno***X inverts the established embedded systems design process and introduces a "hardware follows software" paradigm in which the software is no longer required to compensate for the deficiencies of the hardware. The process also allows checking whether a hardware redesign is necessary in case the ASW requirements change during development

or during updates in the field. All the features of ***papageno***X are used to answer the question, "How can application software be used to automatically generate all the remaining parts of an embedded system?". The final result of the concept is not only a suitable electronics PCB of the system but also the software stack is compiled accordingly.

The functionality of the approach is evaluated by a proof of concept, including a manufactured prototype PCB for a control system use case. In addition, the performance of the file generation process for schematics and board layouts was analyzed concerning scalability.

As a general conclusion, it requires quite an amount of information about the ASW's requirements to generate the hardware automatically. However, explicitly specifying this information in the ASW development facilitates the process immensely. Furthermore, the granularity of the hardware modules also plays a significant role in the generation process concerning the complexity of the process. In the future, it would be interesting to take a closer look at the process that extracts requirements from the ASW and to optimize the design process further. It is also interesting to explore how more detailed information can be integrated into the ASW code without demanding the developer to have extensive prior information about all requirements. According to our findings, this can significantly simplify the development, and it might even increase the acceptance of such a process. From a technical perspective, it is also possible to extend the concept to support more PCB design tools.

The approach to advance the **reconfigurable systems design** proposes the *more*MCU, supported by *Smart*OS, a co-designed OS/MCU platform based on RISC-V that enables partial reconfiguration at runtime for embedded systems. It enables flexible ISAs and flexible on-chip peripherals within the logic of the computing platform. These features are possible by creating a carefully hardware/software co-designed platform that can change its hardware at runtime, thus answering the initial research question, "How can an embedded computing platform change its own logic at runtime?". The system design takes advantage of the runtime reconfiguration capabilities of a modern FPGA. For this purpose, the FPGA is divided into a static area and several dynamically reconfigurable partitions for custom instructions and flexible on-chip peripherals that can hold the corresponding logic. All of the contained logic within the RPs can be hot swapped. The OS must be capable of supporting the emulation of initially unknown instructions; otherwise, an illegal instruction exception might lead to unforeseen circumstances. Once the reconfiguration process is started, the partial reconfiguration itself happens on-the-fly without reboots, resets, or halts of the system. This reconfiguration is carried out by the reconfiguration controller included in *more*MCU. The entire approach also contributes to more sustainable embedded systems, as improved long-term maintenance can be achieved when the computing platform's logic modifications are possible. This increase in sustainability is because the existing chip area can be used and re-used for various purposes at runtime. The modification and replacement of logic components within *more*MCU can be actively triggered by *Smart*OS. The concept can be used with standard compilers and toolchains, as it only adds to the RISC-V standard and does not modify it. Furthermore, the concept can be used on all FPGAs with an internal component that enables runtime reconfiguration.

The evaluation discusses the resource overhead in software and hardware, which is also extensively shown by evaluating several use cases. The evaluation includes a discussion of the achievable performance gain when implementing an algorithm in hardware rather than in software. Apart from the acceleration, the runtime reconfiguration is evaluated concerning resource utilization of the MCU and the reconfiguration controller. A proof of concept of the entire system concludes the evaluation. This proof of concept shows how the partial reconfiguration at runtime performs and how an exemplary system in this regard could look like.

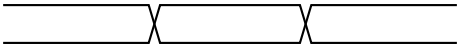
As a general conclusion, partial reconfiguration at runtime can result in benefits for embedded systems, provided that certain conditions apply. These conditions are described in more detail in the evaluation. While the concept has its limits, the benefits in terms of, e.g., resource utilization are substantial. The demonstrated ability to update hardware after deployment also provides a significant advantage regarding long-term operation and maintenance. In the future, it is interesting to take a closer look at the OS replacement and defragmentation strategies for reconfigurable partitions in order to optimize the concept even further. It is also feasible to move towards multi-core applications, where entire cores can be added and removed at runtime, making even more flexibility possible. However, this would also require the examination of corresponding on-chip networks. To overcome the limitations mentioned regarding flexibility, it is also imaginable to design future FPGAs in such a way that they fit the concept better. It would be advantageous to make RPs more flexible and enable the static interface to be designed more effectively.

This dissertation shows two ways to add more flexibility to the design of embedded systems. One way uses the ASW to generate system electronics dynamically; the other way adds runtime-reconfigurability to the logic design process and the long-term hardware maintenance. Hence, both ways yield advances in dynamic and reconfigurable hardware and software for more sustainable embedded systems design.

CHAPTER 7

Publications

This chapter shows an overview of all papers published and related theses co-supervised throughout this doctorate. For the most important and relevant publications, the full text is printed. Every reprint includes a fact sheet of the corresponding publication, where basic information can be found.



The following list shows all relevant publications for this thesis and where to find them in the following. Papers related to dynamic electronics design have the prefix **E**. Publications related to reconfigurable systems design are marked with **R**. Figure 7.1 maps them to the embedded systems stack and shows the corresponding paper's main focus.

E-I ***papageno*PCB: An Automated Printed Circuit Board Generation Approach for Embedded Systems Prototyping**, ICONS 2019 → Section 7.1[p99]

E-II **Towards an Automated Printed Circuit Board Generation Concept for Embedded Systems**, SysMea 2019 → Section 7.2[p107]

E-III ***papageno*X: Generation of Electronics and Logic for Embedded Systems from Application Software**, SENSORNETS 2020 → Section 7.3[p119]

E-IV ***papageno*ReQ: Generation of Embedded Systems from Application Code Requirements**, ICECCE 2021 → Section 7.4[p127]

R-I **System-Aware Performance Monitoring Unit for RISC-V Architectures**, DSD 2017 → Section 7.5[p135]

R-II **A Hardware/Software Concept for Partial Logic Updates of Embedded Soft Processors at Runtime**, DSD 2021 → Section 7.6[p145]

R-III ***Smart*OS: An OS Architecture for Sustainable Embedded Systems**, FGBS 2022 → Section 7.7[p155]

R-IV ***more*MCU: Runtime-reconfigurable RISC-V Platform for Sustainable Embedded Systems**, DSD 2022 → Section 7.8[p167]

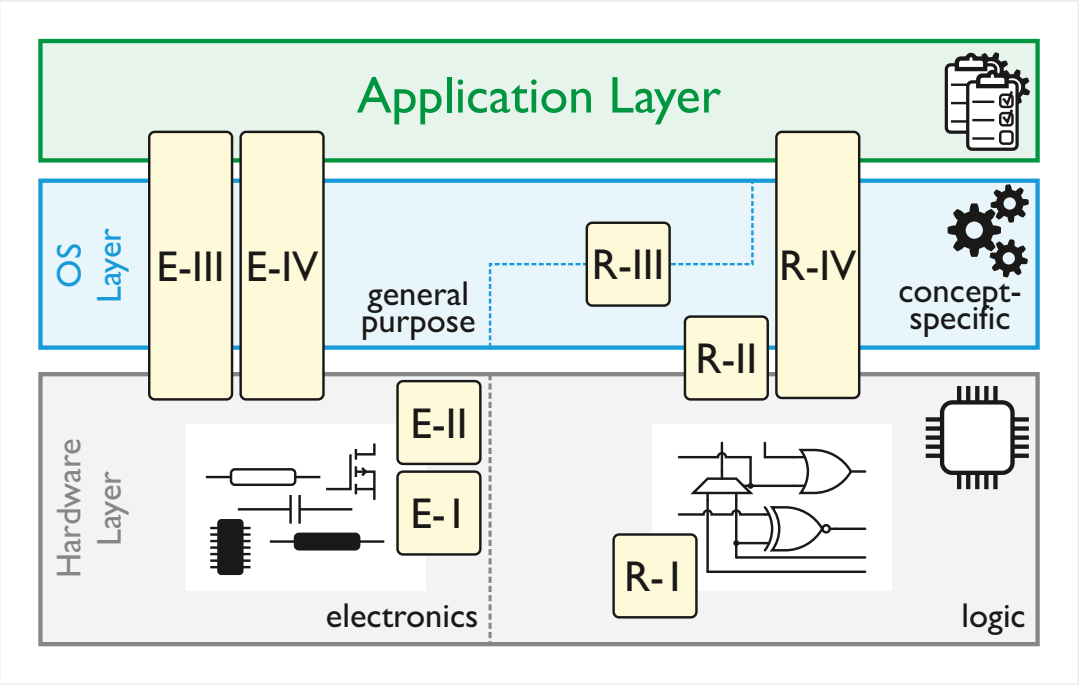


Figure 7.1: All publications in context of the embedded systems stack.

Other Publications (4) The following list contains scientific publications related to the work conducted but not included in the dissertation.

2021 Vimal Sivashanmugam, Tobias Scheipel, Marcel Baunach, and Bhargav Adabala
A Conversion Concept for a Legacy Software Model towards AUTOSAR Compliance – 1st International Conference on Computing and Applied Engineering (ICCAE 2021), Goa, India. August 2021. → [145]

2020 Gernot Fiala, Tobias Scheipel, Werner Neuwirth, and Marcel Baunach
FPGA-Based Debugging with Dynamic Signal Selection at Run-Time – 17th Workshop on Automotive Software Engineering (ASE 2020), Innsbruck, Austria. February 2020. → [146]

2018 Marcel Baunach, Renata Martins Gomes, Maja Malenko, Fabian Mauroner, Leandro Batista Ribeiro, and Tobias Scheipel
Smart Mobility of the Future – A Challenge for Embedded Automotive Systems – e&i Elektrotechnik und Informationstechnik Special Issue "Automated Driving" (2018), Springer. → [147]

2017 Tobias Scheipel, Fabian Mauroner, and Marcel Baunach
Einheit zur anwendungsbezogenen Leistungsmessung für die RISC-V-Architektur – "Logistik und Echtzeit", ECHTZEIT 2017, Boppard, Germany. November 2017. → [148]

Student Theses (17) The following list contains bachelor's and master's theses, seminars, and projects related to the dissertation and were conducted under the co-supervision of the thesis author.

in progress Master's Thesis: **SmartOS goes Multicore: A RISC-V Adventure** by Nikolaus Sifferlinger

in progress Master's Thesis: **Dynamic partial re-configurable hardware management module for a RISC-V based Microcontroller** by Florian Angermair

in progress Master's Project: **Dynamic Reconfiguration of On-chip Peripherals - A Case Study** by Florian Angermair

in progress Bachelor's Thesis: **Real-time Audio Synthesis on an FPGA using an open-source Toolchain** by Hannes Haberl

in progress Bachelor's Thesis: **Wishbone Flash Programming in Dependable Systems** by Florian Riedl

in progress Master's Project: **An Overview of Multi-Core Systems and their Implementations** by Nikolaus Sifferlinger

2022 Master's Project: **Construction of a demonstrator for fuel cell stack monitoring** by Christian Zingl

2022 Master's Thesis: **Design of a heterogenous, automotive multi-core system** by Kristóf Kanics

2021 Bachelor's Thesis: **A Debug Module for an Embedded RISC-V Platform** by Philip Ortlieb

2021 Bachelor's Thesis: **PapagenoPCB Module Design** by David Rösel

2021 Master's Project: **Design of a heterogenous, automotive multi-core system** by Kristóf Kanics

2021 Bachelor's Thesis: **Generic Driver Management System for MCSmartOS** by Alexander Deibel

2020 Bachelor's Thesis: **Driver Development for RTOS: SPI Driver for RISC-V** by Lukas Fuchs

2020 Master's Project: **Driver Development for RTOS** by Harald Kogler

2019 Master's Thesis: **Conversion of Control Unit Software towards AUTOSAR Compliance** by Vimal Sivashanmugam

2019 Master's Project: **Performance analysis and performance enhancement of AVL THDA™ Fuel Cell Monitoring System** by Mojca Kolšek

2018 Master's Seminar: **Porting the MosartMCU to a new RISC-V Core** by Florian Angermair

7.1 E-I: *papageno*PCB: An Automated Printed Circuit Board Generation Approach for Embedded Systems Prototyping

◇ Authors: Tobias Scheipel, and Marcel Baunach
◇ Date: March, 2019
◇ ISBN: 978-1-61208-696-5
◇ Reference in bibliography: [149]
◇ Presented by myself at the 15th International Conference on Systems (ICONS'19), Valencia, Spain.
◇ Published in the proceedings of ICONS'19 (IARIA).

Summary ***papageno***PCB is a concept that enables automatic Printed Circuit Board (PCB) generation from an intermediate system description language. It can be used to generate schematics and layouts for PCBs based on predefined hardware modules and their properties that match the requirements of an Application Software (ASW).

Author's Contribution I am the main author of this paper, and I have entirely envisioned, designed, and implemented the concept of the paper myself. My supervisor Marcel Baunach supported me through discussions and in the composition of the paper.

Copyright ©2019 by IARIA. The version included in this thesis is reprinted, with permission, from the published version stated above, which can be accessed in the IARIA ThinkMind Digital Library:
https://www.thinkmind.org

*papageno*PCB: An Automated Printed Circuit Board Generation Approach for Embedded Systems Prototyping

Tobias Scheipel and Marcel Baunach

Institute of Technical Informatics
Graz University of Technology
Graz, Austria
E-mail: {tobias.scheipel, baunach}@tugraz.at

***Abstract*—Designing an embedded system from scratch is becoming increasingly challenging these days. In fact, the design process requires extensive manpower, comprising engineers with different fields of expertise. While most design principles for embedded systems start with a search for a suitable computing platform and the design of proper hardware to meet certain requirements, our novel vision involves using a completely different approach: The remainder of the embedded system can be automatically generated if the application software is available. In order to achieve automatic Printed Circuit Board (PCB) generation, we describe an approach, *papageno*PCB, in this paper, which is a part of a holistic approach called *papageno*X. *papageno*PCB provides a way to automatically generate schematics and layouts for printed circuit boards using an intermediate system description language. Therefore, the scope of the present work was to develop a concept, which could be used to analyze the embedded software and automatically generate the schematics and board layouts based on predefined hardware modules and connection interfaces. To be able to edit the plans once they have been generated, a file format for common electronic design automation applications, based on Extensible Markup Language (XML), was used to provide the final output.**

Keywords–embedded systems; printed circuit board; design automation; hardware/software codesign; systems engineering.

I. Introduction

Embedded systems are relevant in almost every part of our society. From the simple electronics in dishwashers to the highly complex electronic control units in modern and autonomous cars – today, daily life is nearly inconceivable without those systems. As the technology improves, the complexity of embedded systems inevitably and steadily increases. A whole team of engineers usually plans, designs, and implements a novel system in several iteration steps. An example of such a process in the automotive industry is shown in Figure 1.

Designing an embedded system can be prone to errors due to a multitude of possible sources of such errors. This presents one major challenge when designing such a system: The challenge of how to eliminate error sources and make design processes more reliable and, therefore, cheaper. Nowadays, most design paradigms choose a bottom-up approach. This means that a suitable computing platform is chosen after defining all requirements with respect to these explicit requirements, prior experience, or educated guesses. Then, software development can either start based on an application kit of a computing platform, or some prototyping hardware must be built beforehand. If the requirements change during the development process, major problems could possibly arise,

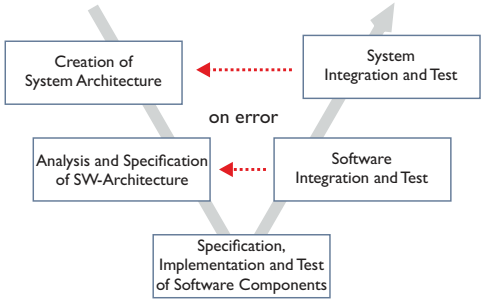


Figure 1. Automotive design process according to the V-Model [1].

e.g., new software features cannot be implemented due to computing power restrictions or additional devices cannot be interfaced because of hardware limitations. Another problem could arise if connection interfaces or buses become overloaded with too much communication traffic after the hardware has already been manufactured.

To tackle these problems, we propose a holistic approach, *papageno*X, and a sub-approach, *papageno*PCB, which are discussed in detail in the following sections. *papageno*X is a novel approach that has been developed for use while creating embedded systems with a top-down view. Therefore, it uses application source code to automatically generate the whole embedded system in hardware and software. One part of the concept behind this approach is *papageno*PCB. This concept handles the automatic generation of schematics and board layouts for printed circuit board design with standardized XML-based [2] output from intermediate system description models. To do so, a module-based description of the system hardware and software needs to be made. Furthermore, connections between the hardware modules on wire level are done automatically. The concept and its related challenges were the main topics of the work described in this paper, whereas software analysis and model generation is part of work that will be conducted in the future with *papageno*X.

The paper is organized as follows: Section II includes a summary of related work. The rough idea of the holistic vision of *papageno*X (this paper includes a detailed description of the first part of this concept) is illustrated in Section III, whereas Section IV starts with the system description format within *papageno*PCB. In Section V, an explanation is given of the necessary steps taken to create the final output, and Section VI includes a proof of concept example. The results of

an analysis on the scalability and performance of the developed generator are presented in Section VII. The paper concludes with Section VIII, in which the steps that need to be taken to achieve a final version of *papageno*X are described.

II. Related Work

As this work dealt with the automatic generation of hardware and extensively utilized hardware definition models, it was influenced by existing solutions such as devicetree, which is used, e.g., within Linux [3]. The devicetree data structure is used by the target Operating System's (OS) kernel to handle hardware components. The handled components can comprise processors and memories, but also the internal or external buses and peripherals of the system. As the data structure is a description of the overall system, it must be created manually and cannot be generated in a modular way. It is mostly used with System-on-Chips (SoCs) and enables the usage of one compiled OS kernel with several hardware configurations. Different approaches have been taken to use annotated source code to extract information about the underlying system. Annotations can be used to analyze the worst-case execution times [4][5] of software in embedded systems. Other approaches that have been taken have used back-annotations to optimize the power consumption simulation [6]. These annotations have allowed researchers to gain a better idea of how the system works in a real-world application, meaning that the annotated information is based on estimations or measurements. As far as the automatic generation of schematics and board layouts is concerned, few solutions have been developed towards design automation. Some authors have dealt with the question of how to generate schematics using expert systems so their appearance is more pleasing to human readers [7]. Some work has even been carried out on the generation of circuit schematics by extracting connectivity data from net lists [8]. These approaches are all based on various kinds of network information and cannot be used to extract system data out of – or are even aware of – application source code or system descriptions.

All the approaches mentioned above have some advantages and inspired this work, as no solution has yet been proposed for how to automatically generate PCBs from source code.

III. Main Idea of *papageno*X

*papageno*X stands for **P**rototyping **AP**plication-based with **A**utomatic **GEN**eration **O**f **X**; it prospectively contains a toolchain that can be used to automatically generate the software, reconfigurable logic, and hardware of the final prototype of system X by simply using application software source code. In this context, system X could be an automotive Electronic Control Unit (ECU), a Cyber-Physical System (CPS), or an Internet-of-Things (IoT) device. After generating X, *papageno*X should also be able to check whether a new application version is still compatible with previously designed systems, or if it is not, which restrictions apply.

As depicted in Figure 2, *papageno*X uses Application Software (ASW) to generate software code that includes Basic Software (BSW) and an executable ASW, reconfigurable logic code in some hardware description language for Field Programmable Gate Arrays (FPGAs), as well as schematics

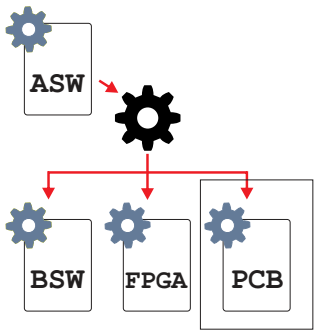


Figure 2. The main idea behind the *papageno*X approach.

and layouts for PCBs. In this context, the term BSW subsumes operating systems with, e.g., drivers, services. Even though the *papageno*X approach envisions generation of reconfigurable logic, it differs from, e.g., SystemC [9], because it also generates hardware on the PCB level.

In this paper, the very first step taken to generate a PCB from an intermediate system model (prospectively extracted from source code) is described.

IV. System Description Format

The system description format in *papageno*PCB is module-based. This means that every possible module, e.g., a Microcontroller Unit (MCU) board or different peripherals must be defined before they are connected with each other. The whole description and modeling approach taken is generic, which enables its easy adaptation to different use cases. The structure was defined according to a JavaScript Object Notation (JSON) [10] format, and three different kinds of definition files were established:

A. Module Definition: One single file that defines the hardware module, its interfaces and its pins, and a second file that contains the design block for creating schematics and board layouts concerning this module.
B. Interface Definition: Generic definition of several different interfaces between modules.
C. System Definition: Contains modules and connections between these; is abstractly wired with certain interface types.

All three types will be explained below. The example modules show footprints of a Texas Instruments (TI) LaunchPad™ [11] with a 16-bit, ultra-low-power MSP430F5529 MCU [12].

A. Module Definitions and Design Blocks

The module definition of a TI LaunchPad™ is shown in Figure 3. Apart from a name and a design block file property, this definition consists of an array of interfaces and pins. The design block file property refers to an EAGLE [13] design block file, comprising of a schematic placeholder (cf. Figure 4), and a board layout placeholder (cf. Figure 5). These placeholders will later be placed on the output schematics and board layouts. The array of interfaces may contain several different interface types of which the module is capable. The property *type* determines the corresponding interface type. In the case presented, two Serial Peripheral Interfaces (SPIs) are present. Both contain a name, the type *SPI*, and several pins.

```
1  {
2    name: "MSP430F5529_LaunchPad",
3    design: "MSP430F5529_LaunchPad.dbl",
4    interfaces: [{
5        name: "SPI0",
6        type: "SPI",
7        pins: { MISO: "P3.1", MOSI: "P3.0",
8          SCLK: "P3.2", CS: 'any@["P2.0", "P2.2"]'  }
9      }, {
10       name: "SPI1",
11       type: "SPI",
12       pins: { MISO: "P4.5", MOSI: "P4.4",
13         SCLK: "P4.0", CS: any }
14     }
15   ],
16   pins: ["P6.5", "P3.4", "P3.3", "P1.6",
17   "P6.6", "P3.2", "P2.7", "P4.2", "P4.1",
18   "P6.0", "P6.1", "P6.2", "P6.3", "P6.4",
19   "P7.0", "P3.6", "P3.5", "P2.5", "P2.4",
20   "P1.5", "P1.4", "P1.3", "P1.2", "P4.3",
21   "P4.0", "P3.7", "P8.2", "P2.0", "P2.2",
22   "P7.4", "RST" , "P3.0", "P3.1", "P2.6",
23   "P2.3", "P8.1"]
24 }
```

Figure 3. Module definition of a TI LaunchPad™ with two SPI interfaces.

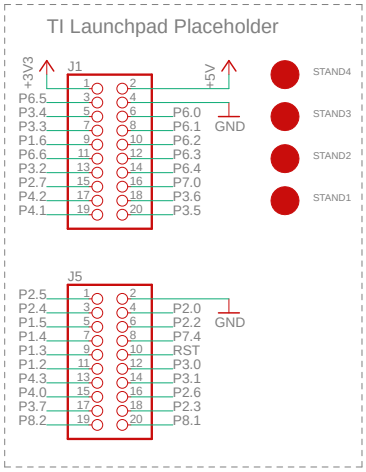


Figure 4. Schematics of a placeholder design block for a TI LaunchPad™ [11].

Pins within interfaces can either be directly assigned to hardware pins (e.g., `MISO: "P3.1"` in line 7) or left for automatic assignment (e.g., `CS: any` in line 13). It is also possible to automatically assign a wire from a dedicated pool by using `any@somearray` (cf. line 8) syntax. Each module definition file is associated with its corresponding design block. It is of utmost importance that pin names are coherent in both module representations, as coherence of naming later ensures that proper interconnections are made between modules. Furthermore, a standard format for power supply connections must be used to avoid creating discrepancies between modules. The bus speed of the SPI was not taken into account in this work and will be addressed in future developments. As depicted in Figure 5, the board layout of a module only consists of its pins. The main idea here was to create a motherboard upon which modules can be placed using their exterior connections (e.g., pin headers or similar connectors). Therefore, the placeholder serves as interface layout between fully assembled PCB modules, such as the LaunchPad™, and can then be connected to other modules through interfaces.

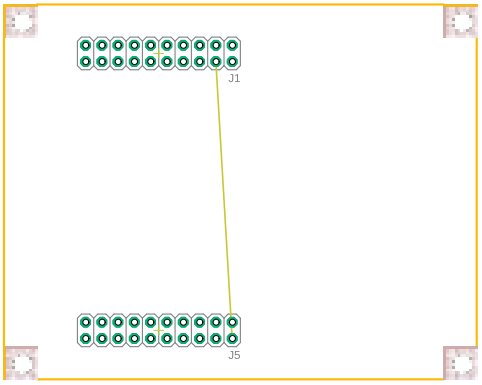


Figure 5. Board layout of a placeholder design block for a TI LaunchPad™ [11].

```
1  {
2    interfaces: [
3      {
4        type: "SPI",
5        connections: [
6          { "master.MOSI" : "bus.MOSI" },
7          { "master.MISO" : "bus.MISO" },
8          { "master.SCLK" : "bus.SCLK" },
9          { "slave.MOSI"  : "bus.MOSI" },
10         { "slave.MISO"  : "bus.MISO" },
11         { "slave.SCLK"  : "bus.SCLK" },
12         { "master.CS"   : "wiremultiple" },
13         { "slave.CS"    : "wiresingle" }
14       ]
15     }
16   ]
17 }
```

Figure 6. Interface definition containing SPI.

B. *Interface Definitions*

After defining the modules, the generic interfaces must be defined. The interface definition collection is centralized in a single file, and its structure is shown in Figure 6. In this example, only SPI [14] has been defined with its standard connections. As the format is generic, other interface types, e.g., Inter-Integrated Circuit (I^2C, [15]) or even a Controller Area Network (CAN, [16]), are also feasible. It also shows how masters and slaves within this communication protocol are connected to the bus wires. As the SPI also has so-called Chip Select (CS) wires for every slave selection, special treatment must be used here: A slave only has one CS wire, which is marked with *wiresingle* (cf. line 13), whereas a master has as many CS wires as it has slaves connected to it (marked with *wiremultiple*; cf. line 12).

C. *System Definition*

The final step taken was to define the system itself, which was built from modules and the connections between them. To do so, a single project file must be created, as illustrated in Figure 7. Initially, all necessary modules are imported and named accordingly within the *modules* array. Once defined, they can be interconnected using the previously defined interface definitions. In our example, LaunchPad™ *MSP1* was connected to a microSD board *SD1* of type "MicroSD Breakout Board" [17] via SPI. This particular SPI connection is called *SPI_Connection1* of type *SPI* and has two participants with different roles: *MSP1* as a master and *SD1* as a slave. This system definition will prospectively be generated and extracted

```
{
  modules: [
    { name: "MSP1",
      type: "MSP430F5529_LaunchPad"},
    { name: "SD1",
      type: "MicroSD_BreakoutBoard"}
  ],
  connections: [
    {
      name: "SPI_Connection1",
      type: "SPI",
      participants: [
        { name: "MSP1", role: "master" },
        { name: "SD1", role: "slave" }
      ]
    }
  ]
}
```

Figure 7. A system model containing two modules connected via SPI.

out of the ASW code by *papageno*X. The *papageno*PCB approach is taken to generate PCBs only.

V. Implementation of PCB Generation

After having defined the modules, interfaces, and implemented a system definition, PCB generation can start. The generation consists of two major steps: (*A.*) establishing connection wires based on predefined module and system definitions, and assigning dedicated pins and (*B.*) generating XML-based schematic files from its output. The final step (*C.*), which is carried out to deal with the final layout of the schematics, must be done (in part manually) afterwards. The generator is developed as a Java command line application to maintain platform-independence and ensure that it can be integrated into standard tool chains and build management tools.

A. Connection Establishment and Pin Assignment

During this first step, JSON data structure analysis presents the main challenge. The whole system must be interconnected appropriately using the previously explained definition files. To do so, all connections within the system definition must be matched at the beginning of the process. This task subsumes the discovery of connections between modules, their mapping to certain interface types, and the final wire allocation required to interconnect all participants. Specifically, each connection has a type and a finite number of participants with different roles, interfaces, and pins. These pins must then be connected to the newly introduced wires, belonging to the communication. Several different types of wires can be used to connect the participants with each other:

The easiest wires to use are common wires, which can be assigned to a pool of free pins of the module. These wires are marked with *wiresingle* within the interface definition. Due to the fact that all unused General-Purpose Input/Output (GPIO) pins of a module can be used for this purpose, they need to be assigned last.

Furthermore, every participant can connect itself directly to bus wires via its dedicated pins, depending on, e.g., the type of MCU used. In the case of an MSP430 MCU, certain pins are electrically connected to an interface circuit, as defined in its module definition (cf. Figure 3). These pins must, therefore, be matched with the connection's wires (cf. Figure 6). The interface definition must match roles and pins accordingly to correctly interconnect the participants of each connection.

Another type of wires that can be used are multiple wires. If we take SPI as an example, the master needs to have as many chip-select wires as slaves with which it wants to communicate. Therefore, this type of wire – marked with *wiremultiple*, as previously defined – must clone itself to obtain the number of wires needed.

These different types of wires must be connected to the pins of the modules to establish a proper connection or *net* according to the interface definition. The interconnected modules with their nets form a holistic JSON-based description of the system.

B. Schematic and Board Layout Generation

Utilizing the interconnected system description, schematics and board layouts can be generated. In our case, EAGLE's XML data structure [2] was used to form a dedicated output file for schematics and board layouts. To generate those plans, (1) design blocks for each module must be loaded, (2) the previously found connections must be applied and (3) the connected design blocks must be placed on an empty schematic plan or board layout.

(1) In this step, each module has to be instantiated by loading the corresponding design block of its type.
(2) This step must carried out to form the whole system according to the JSON-based holistic description. Therefore, pins of each module must be assigned to the wires of a connection within the system. To do so, each connection again must be applied separately to each participant. As the system description already contains information, as to which pin of a module must be connected to which wire, this can be done quite easily.
(3) This step, which is the computationally most expensive step, must be carried out to merge the connected instances of each module into an empty plan, as a great deal of XML parsing is required here. To create consistent plans, the design blocks must be prepared well beforehand to avoid, e.g., inconsistencies within board layers or signal names. To keep the modules from overlapping, a two-dimensional translation of each module must be executed as part of each merge procedure as well. In total, two merging steps are required for each module – one for the schematic and one for the board layout. As this approach generates connection PCBs ("motherboards") where one can plug in modules, only placeholders are used.

Finally, the two generated XML structures are exported and saved into different files for further usage.

C. Routing Generated Schematics and Board Layouts

As layouting and routing of PCBs is a non-trivial task, and engineers need a great deal of experience when performing a task like this, *papageno*PCB cannot be used to produce final variants of a board. It is recommended to use EAGLE's autorouting functionality or manual routing to finalize the already well-prepared layouts.

VI. Proof of Concept

The proof of concept comprises the generation of the system definition as shown in Figure 7. As mentioned before, the system created consists of two modules interconnected with

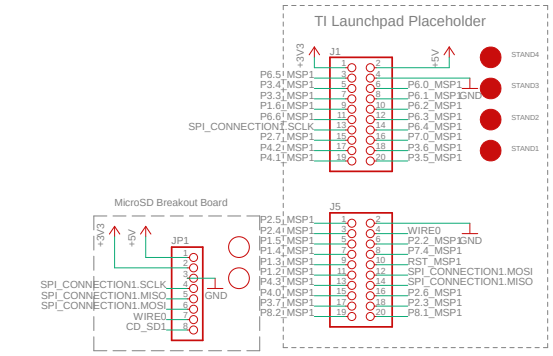


Figure 8. Raw output of the schematics generated as displayed in EAGLE.

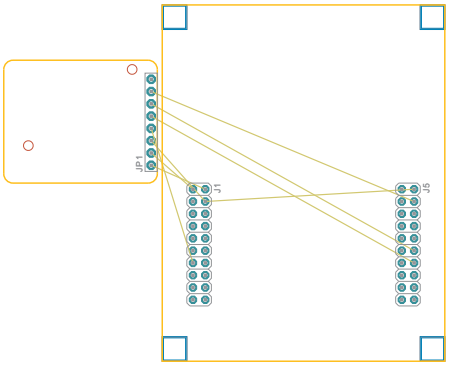


Figure 9. Raw output of the board layout generated as displayed in EAGLE.

one SPI bus, whereas the processor board serves as master. The schematics generation step yields in the drawing depicted in Figure 8.

Compared with the LaunchPad™'s design block shown in Figure 4, one can see the differences in the net names. As examples, *P2.0* has been replaced with *WIRE0*, and *P3.0* is now assigned to *SPI_CONNECTION1.MOSI*. These wires connect to pins 7 and 6 of the MicroSD Breakout Board on the left, respectively. Also, each unconnected pin gets a suffix describing its module (cf. *_MSP1*). These newly introduced net names are the results of the wire generation explained in Section V-A. As the reusability of schematic plans is an important aspect, the feature of non-overlapping module placement can be emphasized as well. The result of the board layout generation step is shown in Figure 9, as described in Section V-B. The fine lines show non-routed connections between the pins. As the plan will be manufactured as a real hardware PCB, no part can overlap. Routing of the board has to be either performed manually or by using a design tool's built-in auto router. A feasible layout variant is presented in Figure 10. EAGLE can also be used to check the correctness of the XML file format.

VII. Scalability and Performance

In this Section, we describe measurements and investigations that concern the performance of the PCB-generating process. All discussed evaluations use one setup as a reference. The application was executed with a Java 10 virtual machine on an Intel Core i7 7500U@2.7GHz with 16 gigabytes of RAM. Table I shows the mean execution time and the

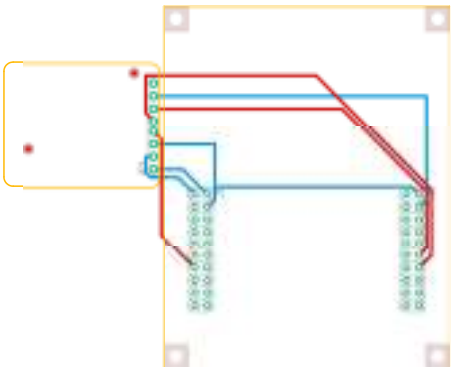


Figure 10. Board layout after auto-routing in EAGLE.

combined output XML file size of the generation process of different test case scenarios, which are explained below. All test cases featured a different number of participants (part.) which consisted of masters (M) and slaves (S) with different connection types.

TABLE I. MEAN EXECUTION TIMES FOR DIFFERENT SCENARIOS.

#	*test scenario description*	*execution time*	*file size*
	1 SPI conn. (scenario 1)		
0	2 part. (1 M, 1 S)	678.98 ms	94 KiB
1	3 part. (1 M, 2 S)	793.56 ms	100 KiB
2	4 part. (1 M, 3 S)	893.89 ms	105 KiB
3	5 part. (1 M, 4 S)	983.56 ms	110 KiB
4	6 part. (1 M, 5 S)	1 060.70 ms	116 KiB
5	7 part. (1 M, 6 S)	1 151.37 ms	122 KiB
	2 SPI conn. (scenario 2)		
0	4 part. (1 M and 1 S each)	910.93 ms	115 KiB
1	6 part. (1 M and 2 S each)	1 079.78 ms	126 KiB
2	8 part. (1 M and 3 S each)	1 242.88 ms	137 KiB
3	10 part. (1 M and 4 S each)	1 388.39 ms	149 KiB
4	12 part. (1 M and 5 S each)	1 500.44 ms	160 KiB
5	14 part. (1 M and 6 S each)	1 613.93 ms	171 KiB
	1 I^2C conn. (scenario 3)		
0	2 part. (1 M, 1 S)	693.04 ms	105 KiB
1	3 part. (1 M, 2 S)	813.64 ms	111 KiB
2	4 part. (1 M, 3 S)	918.96 ms	117 KiB
3	5 part. (1 M, 4 S)	1 010.40 ms	123 KiB
4	6 part. (1 M, 5 S)	1 107.56 ms	129 KiB
5	7 part. (1 M, 6 S)	1 198.17 ms	135 KiB
	1 I^2C and 1 SPI conn. (scenario 4)		
0	3 part. (1 M, 1 S each)	828.13 ms	127 KiB
1	5 part. (1 M, 2 S each)	1 020.54 ms	138 KiB
2	7 part. (1 M, 3 S each)	1 195.82 ms	150 KiB
3	9 part. (1 M, 4 S each)	1 337.50 ms	162 KiB
4	11 part. (1 M, 5 S each)	1 493.75 ms	173 KiB
5	13 part. (1 M, 6 S each)	1 612.04 ms	185 KiB

Each test case is based on the example described in Section VI but with different constellations concerning the numbers and types of participants and connections. All test cases were executed 100 times. Four types of test scenarios with six test cases each were conducted: Within the first scenario, just one SPI connection was present, with a varying number of slaves each test case. The second scenario comprised two SPI connections with an increasing number of slaves. Test scenario three had one I^2C connection and was similar to scenario one, whereas scenario four included SPI and I^2C connections to a single master with an increasing number of slaves. The devolution of the mean execution time (in ms) in all test scenarios is shown in Figure 11. When comparing all scenarios, the trend observed is quite similar: All performance graphs show a linear devolution with an additive, logarithmic-

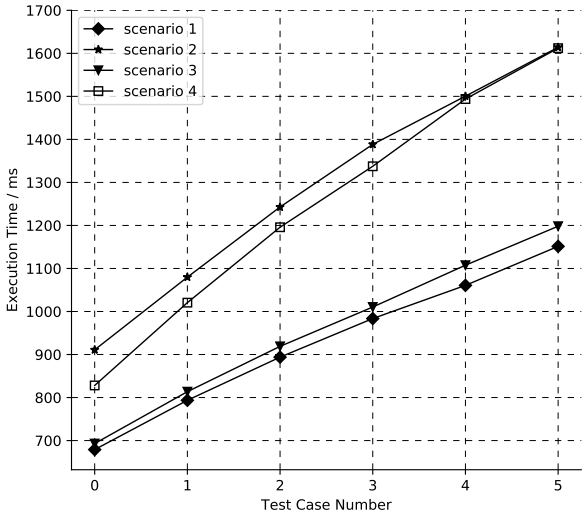


Figure 11. Performance graph for different test cases.

like component. The linear component is due to the linear increase in the complexity of the test cases. The logarithmic-like growth observed can be explained by the decreasing, additive overhead of the linear component when processing similar connection reasoning, as well as the XML schematic and layout data. This is also the reason why doubling the numbers in the first test case yielded in much higher values than in test case two. Test scenario four is the only one that displays a steeper curve. This is due to the combination of different connection types, yielding less-optimal algorithm executions. As XML processing is quite costly, some further optimizations are needed. As the overall file size displayed linear growth, no correlation was observed between file size and execution time.

VIII. Conclusion and Future Work

In conclusion, *papageno*PCB represents a novel, top-down approach that can be taken to develop an embedded system. With just having a model-based system description at hand, it is possible to use *papageno*PCB to generate hardware schematics and board layouts accordingly. This opens up a numerous new possibilities on higher abstraction levels. It is feasible to carry out automatic bus balancing or bandwidth engineering before building the hardware. The use of these concepts requires the availability of in-depth information about the electrical and mechanical characteristics of all parts of a PCB, so that the hardware can be optimized regarding non-functional metrics such as bandwidth or power consumption. Due to the generic design, new models can be integrated easily, and it will be even possible to take a non-module-based approach on the device level (if the proper definitions are available).

In the future, work must be carried out to extract system models from ASW source code. Therefore, we are working on introducing annotations into our operating system environment [18], which will enable us to automatically generate system definition files. These annotations can either be introduced into the code as compiler keywords (e.g., pragmas, defines) or as comments. As some work is already being conducted to improve the automatic portability of real-time operating systems [19], the proposed approach could be used to build a system for which only the application code must be programmed. The rest of the system can then be generated automatically. Even suitable and application-optimized processor architectures [20] could be created by taking this approach. The ultimate goal is to establish *papageno*X as a universal embedded systems generator using only annotated ASW code.

Acknowledgment

This research project was partially funded by AVL List GmbH, the Austrian Federal Ministry of Education, Science and Research (bmbwf), and Graz University of Technology.

References

[1] J. Schäuffele and T. Zurawka, *Automotive Software Engineering*, ser. ATZ/MTZ-Fachbuch. Springer Fachmedien Wiesbaden, 2016.

[2] Autodesk, Inc., *EAGLE XML Data Structure 9.1.0*, 2018.

[3] devicetree.org, *Devicetree Specification*, Dec. 2017, release v0.2.

[4] J. Schnerr, O. Bringmann, A. Viehl, and W. Rosenstiel, "High-performance Timing Simulation of Embedded Software," in *Proc. of the 45th Annual Design Automation Conference*, June 2008, pp. 290–295.

[5] B. Schommer, C. Cullmann, G. Gebhard, X. Leroy, M. Schmidt, and S. Wegener, "Embedded Program Annotations for WCET Analysis," in *WCET 2018: 18th Int'l Workshop on Worst-Case Execution Time Analysis*, Barcelona, Spain, Jul. 2018, pp. 8:1–8:13.

[6] S. Chakravarty, Z. Zhao, and A. Gerstlauer, "Automated, Retargetable Back-annotation for Host Compiled Performance and Power Modeling," in *Int'l Conference on Hardware/Software Codesign and System Synthesis*, Piscataway, NJ, USA, 2013, pp. 1–10.

[7] G. M. Swinkels and L. Hafer, "Schematic generation with an expert system," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 9, no. 12, pp. 1289–1306, Dec 1990.

[8] B. Singh *et al.*, "System and method for circuit schematic generation," US Patent US7 917 877B2, 2011.

[9] IEEE Standards Association, *IEEE 1666-2011 - IEEE Standard for Standard SystemC Language*, Sep. 2012.

[10] ECMA International, *ECMA-404: The JSON Data Interchange Syntax*, 2nd ed., Dec. 2017.

[11] Texas Instruments, *MSP430F5529 LaunchPad™ Development Kit (MSP--EXP430F5529LP)*, Apr. 2017.

[12] ——, *MSP430x5xx and MSP430x6xx Family User's Guide*, Mar. 2018, [retrieved: Jan, 2019]. [Online]. Available: http://www.ti.com/lit/ug/slau208q/slau208q.pdf

[13] Autodesk, Inc., "EAGLE," [retrieved: Jan, 2019]. [Online]. Available: https://www.autodesk.com/products/eagle/

[14] S. Hill *et al.*, "Queued serial peripheral interface for use in a data processing system," US Patent US4 816 996, 1989.

[15] NXP Semiconductors, Inc., *UM10204: I2C-bus specification and user manual*, Apr. 2014, rev. 6.

[16] International Organization for Standardization, *ISO 11898: Road vehicles – Controller area network (CAN)* , 2nd ed., Dec. 2015.

[17] Adafruit Industries, *Micro SD Card Breakout Board Tutorial*, Jan. 2019, [retrieved: Jan, 2019]. [Online]. Available: https://cdn-learn.adafruit.com/downloads/pdf/adafruit-micro-sd-breakout-board-card-tutorial.pdf

[18] R. Martins Gomes, M. Baunach, M. Malenko, L. Batista Ribeiro, and F. Mauroner, "A Co-Designed RTOS and MCU Concept for Dynamically Composed Embedded Systems," in *Proc. of the 13th Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, 2017, pp. 41–46.

[19] R. Martins Gomes and M. Baunach, "A Model-Based Concept for RTOS Portability," in *Proc. of the 15th Int'l Conference on Computer Systems and Applications*, Oct. 2018, pp. 1–6.

[20] F. Mauroner and M. Baunach, "mosartMCU: Multi-Core Operating-System-Aware Real-Time Microcontroller," in *Proc. of the 7th Mediterranean Conference on Embedded Computing*, Jun. 2018, pp. 1–4.

7.2 E-II: Towards an Automated Printed Circuit Board Generation Concept for Embedded Systems

◇ Authors: Tobias Scheipel, and Marcel Baunach
◇ Date: December, 2019
◇ ISSN: 1942-261x
◇ Reference in bibliography: [150]
◇ Published in the International Journal on Advances in Systems and Measurements, Volume 12, Number 3 and 4, 2019 (IARIA).
◇ Invited journal publication, as E-I was the best paper runner-up in ICONS'19.

Summary This work is an extended version of the previous paper and sketches a concept to generate Printed Circuit Boards (PCBs) from Application Software (ASW) code. A top-down approach called ***papageno***X is proposed in this regard. The generation portion of the concept called ***papageno***PCB is described in detail. Goal is to generate layouts and schematics for mainboards that interconnect predefined hardware modules according to the needs of the software.

Author's Contribution I am the main author of this paper, and I have entirely envisioned, designed, and implemented the concept of the paper myself. My supervisor Marcel Baunach supported me in the composition of the paper.

Copyright ©2019 by IARIA. The version included in this thesis is reprinted, with permission, from the published version stated above, which can be accessed in the IARIA Jourals Digital Library:
https://www.iariajournals.org

Towards an Automated Printed Circuit Board Generation Concept for Embedded Systems

Tobias Scheipel and Marcel Baunach

Institute of Technical Informatics
Graz University of Technology
Graz, Austria
E-mail: {tobias.scheipel, baunach}@tugraz.at

***Abstract*—Future embedded systems will need to be generic, reusable and automatically adaptable for the rapid advance development of a multitude of different scenarios. Such systems must be versatile regarding the interfacing of electronic components, sensors, actuators, and communication networks. Both the software and the hardware might undergo a certain evolution during the development process of each system, and will significantly change between projects and use cases. Requirements on future embedded systems thus demand revolutionary changes in the development process. Today these processes start with the hardware development (bottom-up). In the future, it shall be possible to only develop application software and generate all lower layers of the system automatically (top-down). To enable automatic Printed Circuit Board (PCB) generation, the present work deals mainly with the question *"How to automatically generate the hardware platform of an embedded system from its application software?"*. To tackle this question, we propose an approach termed *papageno*PCB, which is a part of a holistic approach known as *papageno*X. This approach provides a way to automatically generate schematics and layouts for printed circuit boards using an intermediate system description format. Hence, a system description shall form the output of application software analysis and can be used to automatically generate the schematics and board layouts based on predefined hardware modules and connection interfaces. To be able to edit and reuse the plans after the generation process, a file format for common electronic design automation applications, based on Extensible Markup Language (XML), was used to provide the final output files.**

Keywords–embedded systems; printed circuit board; design automation; hardware/software codesign; systems engineering.

I. Introduction

Embedded systems are of relevance in virtually every area of our society. From the simple electronics in dishwashers to the highly complex electronic control units in modern and autonomous cars – daily life today is nearly inconceivable without those systems. As the technology improves, the complexity of embedded systems inevitably and steadily increases. A whole team of engineers usually plans, designs, and implements a novel system in several iteration steps. An example of such a process in the automotive industry is shown in Figure 1.

Designing an embedded system can be prone to errors due to a multitude of possible error sources. This presents one major challenge when designing such a system: The challenge of how to eliminate error sources and make design processes more reliable and, therefore, cheaper. Most design paradigms today choose a bottom-up approach. This means that a suitable computing platform is chosen after defining all requirements with respect to these explicit requirements, prior experience, or educated guesses. Then, software development can either start based on an application kit of a computing platform, or some prototyping hardware must be built beforehand. If the requirements change during the development process, major problems could possibly arise, e.g., new software features cannot be implemented due to computing power restrictions or additional devices cannot be interfaced because of hardware limitations. Another problem could arise if connection interfaces or buses become overloaded with too much communication traffic after the hardware has already been manufactured.

To tackle these problems, we already proposed a holistic approach, ***papageno***X, and a sub-approach, ***papageno***PCB in [1]. In the course of this work, we intend to further discuss and extend our approach in more detail in the following sections. ***papageno***X is a novel approach that has been developed for use while creating embedded systems with a top-down view. Therefore, it uses application source code to automatically generate the whole embedded system in hardware and software. One part of the concept behind this approach is ***papageno***PCB. This concept handles the automatic generation of schematics and board layouts for PCB design with standardized XML-based [2] output from intermediate system description models. To do so, a module-based description of the system hardware and software needs to be made. Furthermore, connections between the hardware modules on wire level are done automatically. The concept and its related challenges were the main topics of the work described in this paper, whereas software analysis and model generation is part of work that will be conducted in the future with ***papageno***X.

The paper is organized as follows: Section II includes a summary of related work. The rough idea of the holistic vision of ***papageno***X (this paper includes a detailed description of the first part of this concept) is illustrated in Section III, whereas Section IV starts with the system description format within ***papageno***PCB. In Section V, an explanation is given of the necessary steps taken to create the final output, and Section VI includes a proof of concept example, an analysis of the scalability and performance for the developed generator and a use case with a manufactured prototype. The paper concludes with Section VII, in which the steps that need to be taken to achieve a final version of ***papageno***X are described.

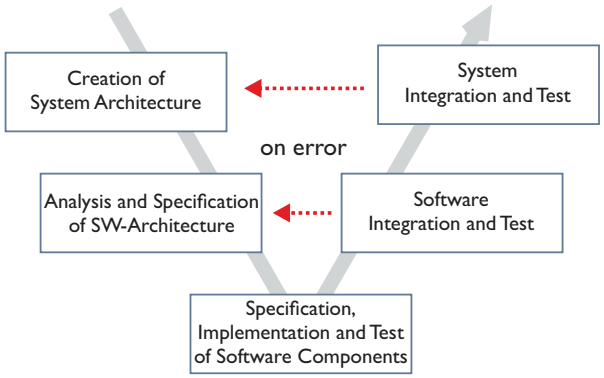


Figure 1. Automotive design process according to the V-Model [3].

II. State of the Art and Related Work

This section gives an overview on how embedded systems are developed nowadays and how hardware can be generated automatically in different types of systems. Additionally, some approaches towards software annotations and design space exploration are shown to provide an overview.

A. Embedded Systems Prototyping Approaches

Conventional embedded systems prototyping makes use of very specialized hardware platforms, capable of executing a vast variety of use cases typical for the field of deployment (e.g., an automotive Electronic Control Unit (ECU), a Cyber-Physical System (CPS), an Internet-of-Things (IoT) device).

In the context of ECU prototyping platforms, one approach is *rCube2* [4], based on two powerful independent TC1797 [5] Microcontroller Units (MCUs). The two processors can interact via shared memory, but are completely isolated during execution. AVL RPEMS [6] is a generic engine control platform provided as a highly flexible and configurable engine management system for the development and optimization of conventional and new combustion engines, power and emissions optimization, and the realization of hybrid and electric powertrains. The current version consists of a single-core automotive MCU (TC1796 [7] or TC1798 [8]) with different variants for diesel and gasoline engine control applications. These different PCBs are equipped with automotive-compliant Application-specific Integrated Circuits (ASICs) and a head-mounted MCU board, which allows prototyping as close as possible to series production. It was designed to offer engineers utmost flexibility when developing new control algorithms for non-standard engines, or standard engines with new components.

There are other similar prototyping platforms from commercial suppliers, but they also lack in flexibility and adaptability when it comes to hardware changes. The main problem of those commercial solutions is that even though they offer high performance and come with complete toolchains, their hardware is very different from a series device, as overcompensation takes place. Since the components cannot be easily changed when a prototype is turned into a commercial product, a complete redesign has to take place.

When the need for hardware changes after deployment is taken into account, reconfigurable logic is mostly mentioned in literature. This can reach from pure Field Programmable Gate Arrays (FPGAs) to System on Chips (SoCs), which include an FPGA alongside other MCU cores (e.g., Zynq-7000 [9]). The main advantage of those systems is that one does not have to change the physical hardware, but can easily adapt features like on-chip peripherals within the logic without the need of manufacturing an ASIC. However, it is not possible to change physical hardware features after deployment with those devices.

B. Automatic Hardware Generation

As this work is concerned with the automatic generation of hardware and extensively utilized hardware definition models, it was influenced by existing solutions such as devicetree, which is used, e.g., within Linux [10]. The devicetree data structure is used by the target Operating System's (OS) kernel to handle hardware components. The handled components can comprise processors and memories, but also the internal or external buses and peripherals of the system. As the data structure is a description of the overall system, it must be created manually and cannot be generated in a modular way. It is mostly used with SoCs and enables the usage of one compiled OS kernel with several hardware configurations. As far as automatic generation of schematics from software is concerned (top-down), there are a few solutions towards design automation. Some papers deal with the question, how to generate schematics, so that these look nice for a human reader by using expert systems [11]. Some work on the generation of circuit schematics has even been done by extracting connectivity data from net lists [12]. These approaches all have some kind of network information as a basis and do not extract system data out of – or are even aware of – application source code or system descriptions.

C. Annotations and Design Space Exploration

Different approaches have been taken to use annotated source code to extract information about the underlying system. Annotations can be used to analyze the worst-case execution times [13][14] of software in embedded systems. Other approaches that have been taken have used back-annotations to optimize the power consumption simulation [15]. These annotations have allowed researchers to gain a better idea of how the system works in a real-world application, meaning that the annotated information is based on estimations or measurements. Introduction of annotations can be achieved by simple source code analysis or more sophisticated approaches such as, e.g., creating add-ons or introducing new features into source code compilers.

To generate systems out of application software, annotations can be used to extract requirements. These requirements can then be utilized to apply design space exploration [16] by, e.g., modeling constraints [17]. In [18], different types for design space explorations are shown and categorized, also mentioning language-based constraint solvers featuring, e.g., MiniZinc [19]. By using approaches like these, a design space model can easily be translated into a mathematical model for optimization.

All the approaches and concepts mentioned above have some advantages and inspired this work, as no solution has yet been proposed for how to automatically generate PCBs from source code.

III. Main Idea of *papageno*X

The main idea of ***papageno***X consists of an application driven electronics generation and the inversion of the state of the art "software follows or adjusts to hardware" paradigm in embedded systems development, where the design starts with the hardware architecture. Software is then built on selected components (e.g., automotive grade MCUs and PCBs).

Even when hardware deficits become visible during the software development process, the hardware is unlikely to see significant changes due to the high cost and many people or even companies involved. Thus, software developers try to compensate, e.g., by manual tuning and workarounds beyond automotive standards (e.g., AUTOSAR [20]). This violates compliance and is one reason why prototypes differ significantly from series devices, also complicating the transition and the subsequent maintenance in the field. Apart from this, future embedded systems will contain reconfigurable logic which is scarcely supported in current development processes due to both the lack of support and a fear of even more complexity (in addition to the software, electronics, and networks). This is why ***papageno***X is an abbreviation for **P**rototyping **AP**plication-based with **A**utomatic **GEN**eration **O**f **X**. The envisioned concept of it will prospectively contain a set of tools that can be used to automatically generate the software, reconfigurable logic, and hardware of the final prototype of system X by simply using application software source code. In this context, system X could be an automotive ECU, a CPS, an IoT device or some other embedded system. The goal is to support frequent changes to the Application Software (ASW) requirements by immediately reflecting them in the Basic Software (BSW), logic, and electronics – reducing time to market and efforts in development and maintenance. During development, the process will optimize the selection and configuration of BSW, on-board components, network interfaces, etc. for simplified transition to series production ("perfect fit"). After deployment, the process will help in the assessment of intended ASW chances to quantify the consequences on lower layers and thus to evaluate their feasibility and cost.

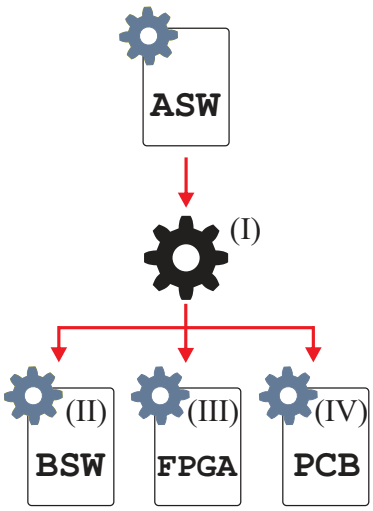


Figure 2. The main idea behind the ***papageno***X approach.

As depicted in Figure 2, the starting point of ***papageno***X is some application as a model or in source code. This ASW is analyzed in order to get to know all necessary requirements for the underlying system layers. These requirements are then used to generate software code that includes BSW and an executable ASW, reconfigurable logic code in a hardware description language (e.g., VHDL [21], Verilog [22]) for FPGAs, as well as schematics and layouts for PCBs. In this context, the term BSW subsumes operating systems with, e.g., drivers, services, hardware abstraction. Even though the ***papageno***X approach envisions the generation of reconfigurable logic, it differs from, e.g., SystemC [23], because it also generates hardware on the PCB level.

The following steps are envisioned within ***papageno***X:

1) application software development
2) in-depth analysis of ASW with respect to functional and non-functional requirements (NFRs)
3) creation of a selection space over potential components
4) filtering of the selection space with respect to general design decisions (e.g., data retention time)
5) generation of potential configurations from components
6) evaluation and optimization towards NFRs to select a single or several final, best fitting configuration(s)
7) mapping of functions or algorithms to reconfigurable logic (FPGAs)

To get a simple overview, the following example sketches the envisioned process while developing an embedded system with our novel approach: a user wants to store data somewhere permanently; with a data rate $\geq 5MB/s$ by writing this line of code in the ASW:

```
store_data(&data, StoreType.Permanent, 5000000);
```

The follow-up analysis of the ASW yields in an exemplary selection space as depicted in Figure 3. The green filled boxes illustrate the final configuration selected by the concept.

So, apart from the running application on the topmost level, a BSW must be generated, supporting a FAT16 [24] file system on top of a SD card driver and its underlying Serial Peripheral Interface (SPI, [25]) module driver. But even more important for this work, the final embedded system must be composed of a computing platform and a storage device, interconnected with each other. Finally, the generated system structure must be manufactured on a PCB, still matching all requirements with its properties.

Based on this overview, ***papageno***X will contain four major parts (also depicted in Figure 2):

(I) ASW analysis → creates selection space
(II) BSW generation → derives, e.g., needed components, drivers, OS features
(III) FPGA generation → maps functions to reconfigurable logic
(IV) PCB generation → module-based generation of suitable PCBs

In this paper, however, the main focus is on (IV), where a very first step is taken to generate a PCB from an intermediate system model (prospectively extracted from source code). The attempt is made to answer the research questions "What information is needed to automatically generate PCBs from ASW?" and "How can this information be used to generate a PCB prototype matching all ASW requirements?". It is henceforth named ***papageno***PCB.

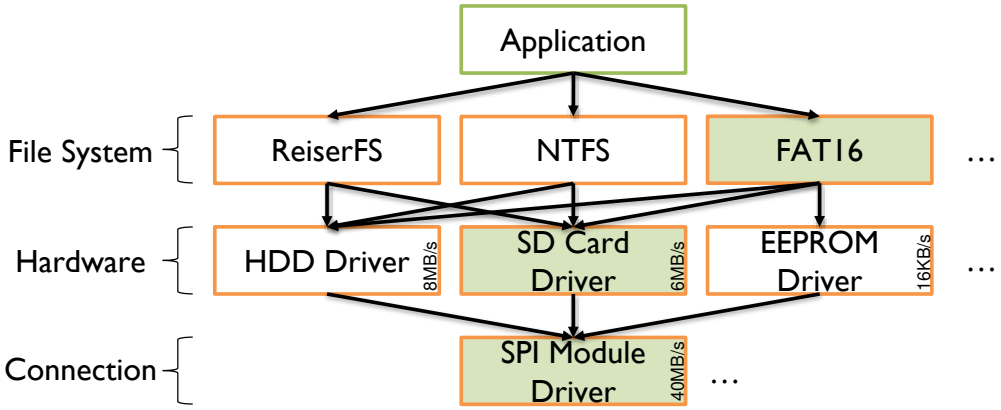


Figure 3. A system's requirements mapped to a corresponding exemplary selection space.

IV. System Description Format

The system description format in ***papageno***PCB is module-based. This means that every possible module, e.g., a MCU board or different peripherals must be defined before they are connected with each other. The whole description and modeling approach taken is generic, which enables its easy adaptation to different use cases. The structure was defined according to a JavaScript Object Notation (JSON) [26] format, and three different kinds of definition files were established:

A. Module Definition: One single file that defines the hardware module, its interfaces and its pins, and a second file that contains the design block for creating schematics and board layouts concerning this module.

B. Interface Definition: Generic definition of several different interface types to interconnect modules with each other; new types can be easily implemented and included within this file.

C. System Definition: Contains modules and connections between these; is abstractly wired with certain interface types.

All three types will be explained below. The example modules show footprints of (1) a Texas Instruments (TI) LaunchPad™ [27] with a 16-bit, ultra-low-power MSP430F5529 MCU [28] and (2) MicroSD card module of type "MicroSD Breakout Board" [29].

```
1  {
2    name: "MSP430F5529_LaunchPad",
3    design: "MSP430F5529_LaunchPad.dbl",
4    interfaces: [{
5        name: "SPI0",
6        type: "SPI",
7        pins: { MISO: "P3.1", MOSI: "P3.0",
8          SCLK: "P3.2", CS: 'any@["P2.0", "P2.2"]' }
9      }, {
10       name: "SPI1",
11       type: "SPI",
12       pins: { MISO: "P4.5", MOSI: "P4.4",
13         SCLK: "P4.0", CS: any }
14     } , {
15       name: "I2C0",
16       type: "I2C",
17       pins: { SDA: "P3.0", SCL: "P3.1" }
18     }, {
19       name: "I2C1",
20       type: "I2C",
21       pins: { SDA: "P4.1", SCL: "P4.2" }
22     }
23   ],
24   pins: ["P6.5", "P3.4", "P3.3", "P1.6",
25   "P6.6", "P3.2", "P2.7", "P4.2", "P4.1",
26   "P6.0", "P6.1", "P6.2", "P6.3", "P6.4",
27   "P7.0", "P3.6", "P3.5", "P2.5", "P2.4",
28   "P1.5", "P1.4", "P1.3", "P1.2", "P4.3",
29   "P4.0", "P3.7", "P8.2", "P2.0", "P2.2",
30   "P7.4", "RST" , "P3.0", "P3.1", "P2.6",
31   "P2.3", "P8.1"]
32 }
```

Figure 4. Module definition of a TI LaunchPad™ with two SPI and two I²C interfaces, both overlapping.

A. Module Definitions and Design Blocks

The module definition of (1) a TI LaunchPad™ is shown in Figure 4, whereas the definition of (2) a MicroSD Breakout Board can be seen in Figure 5. Apart from a name and a design block file property, this definition consists of an array of interfaces and pins. The design block file property refers to an EAGLE [30] design block file, comprised of a schematic placeholder (cf. Figures 6 and 7, respectively), and a board layout placeholder (cf. Figures 8 and 9, respectively). These placeholders will later be placed on the output schematics and board layouts. The array of interfaces may contain several different interface types of which the module is capable. The property *type* determines the corresponding interface type. In module (1) in Figure 4, two SPIs and two Inter-Integrated Circuit (I²C, [31]) interfaces are present. Both contain a name, the type (*SPI*, *I2C*), and several pins. Module (2) in Figure 5, on the contrary, is very simple, with only one SPI interface in total.

Pins within interfaces can either be directly assigned to hardware pins (e.g., `MISO: "P3.1"` in line 7, Figure 4) or left for automatic assignment (e.g., `CS: any` in line 13, Figure 4). It is also possible to automatically assign a wire from a dedicated pool by using `any@somearray` (cf. line 8, Figure 4) syntax. This syntax enables the placing of so-called Chip Select (CS) wires in a more detailed way, e.g., based on needs for shorter connection wires, module specifications or other PCB properties. In this case, `somearray` must, of course, be replaced by a JSON-compliant array of strings, being a subset of the pins of the module, cf. Equation (1).

$$somearray \subseteq pins \tag{1}$$

```
1  {
2    name: "MicroSD_BreakoutBoard",
3    design: "MicroSD_BreakoutBoard.dbl",
4    interfaces: [
5      {
6        name: "SPI1",
7        type: "SPI",
8        pins: { MISO: "DO", MOSI: "DI",
9          SCLK: "CLK", CS: "CS" }
10     }
11   ],
12   pins: ["CLK", "DO", "DI", "CS", "CD"]
13 }
```

Figure 5. Module definition of a MicroSD Breakout Board with an SPI interface.

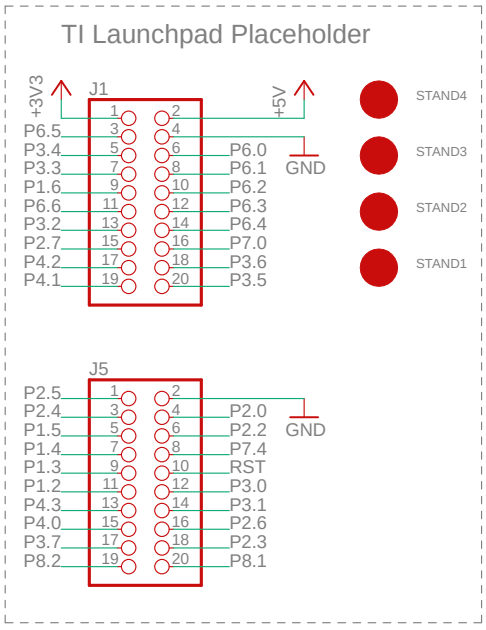


Figure 6. Schematics of a placeholder design block for a TI LaunchPad™ [27].

Each module definition file is associated with its corresponding design block. It is of utmost importance that pin names are coherent in both module representations, as the naming coherence later ensures that proper interconnections are made between modules. Furthermore, a standard format for power supply connections must be used to avoid creating discrepancies between modules. The bus speed of the SPI and the I²C was not considered in this work and will be addressed in future developments towards NFRs. As depicted in Figures 8 and 9, the board layout of a module only consists of its pins. The main idea here was to create a motherboard upon which modules can be placed using their exterior connections (e.g., pin headers or similar connectors). Therefore, the placeholder serves as interface layout between fully assembled PCB modules, such as the LaunchPad™or the Breakout Board, and can then be connected to other modules through interfaces.

B. Interface Definitions

After defining the modules, the generic interfaces must be defined. The interface definition collection is centralized in

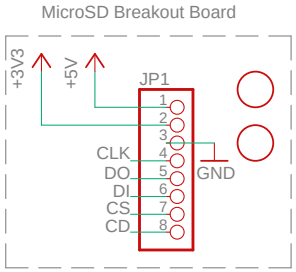


Figure 7. Schematics of a placeholder design block for a MicroSD Breakout Board [29].

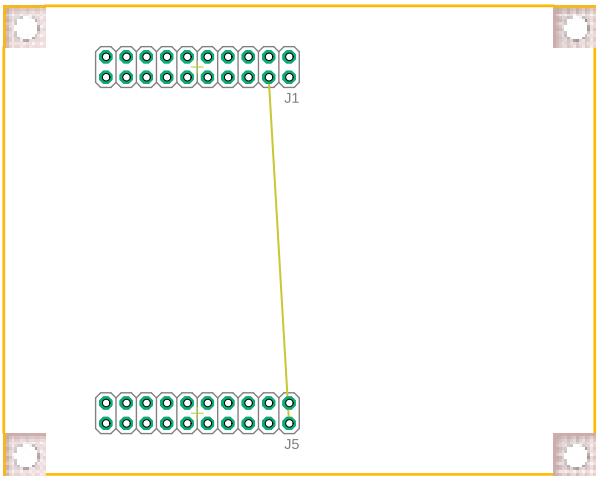


Figure 8. Board layout of a placeholder design block for a TI LaunchPad™ [27].

a single file, and its structure is shown in Figure 10. In this example, only SPI and I²C have been defined with its standard connections. As the format is generic, other interface types, e.g. Controller Area Network (CAN, [32]) or even Advanced eXtensible Interface Bus (AXI, [33]), are also feasible. It also shows how masters and slaves within this communication protocol are connected to the bus wires. As the SPI also has CS wires for every slave selection, special treatment must be used here: A slave only has one CS wire, which is marked with *wiresingle* (cf. line 14, Figure 10), whereas a master has as many CS wires as it has slaves connected to it (marked with *wiremultiple*; cf. line 13, Figure 10). Compared to SPI, the shown example of I²C is rather simple, as it only consists of two wires, with a master/slave concept as well. All participants are simply connected to the corresponding bus wires.

C. System Definition

The final step taken was to define the system itself, which was built from modules and the connections between them. To do so, a single project file must be created, as illustrated in Figure 11. Initially, all necessary modules are imported and named accordingly within the *modules* array. Once defined, they can be interconnected using the previously defined interface definitions. In our example, LaunchPad™ *MSP1* was connected to a MicroSD Breakout Board *SD1* via SPI. This particular SPI connection is called *SPI_Connection1* of type *SPI* and has two participants with different roles: *MSP1* as

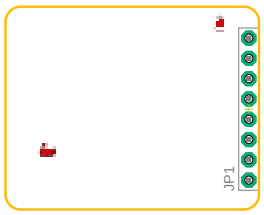


Figure 9. Board layout of a placeholder design block for a MicroSD Breakout Board [29].

```
{
  interfaces: [
    {
      type: "SPI",
      connections: [
        { "master.MOSI" : "bus.MOSI" },
        { "master.MISO" : "bus.MISO" },
        { "master.SCLK" : "bus.SCLK" },
        { "slave.MOSI"  : "bus.MOSI" },
        { "slave.MISO"  : "bus.MISO" },
        { "slave.SCLK"  : "bus.SCLK" },

        { "master.CS"   : "wiremultiple" },
        { "slave.CS"    : "wiresingle" }
      ]
    },
    {
      type: "I2C",
      connections: [
        { "master.SDA" : "bus.SDA" },
        { "master.SCL" : "bus.SCL" },
        { "slave.SDA"  : "bus.SDA" },
        { "slave.SCL"  : "bus.SCL" }
      ]
    }
  ]
}
```

Figure 10. Interface definition containing SPI and I^2C.

a master and *SD1* as a slave. This system definition will prospectively be generated and extracted out of the ASW code by the analysis step in ***papageno***X. The ***papageno***PCB approach is taken to generate PCBs only.

V. Implementation of PCB Generation

After having defined the modules, interfaces, and implemented a system definition, PCB generation can start. The generation consists of two major steps: (*A.*) establishing connection wires based on predefined module and system definitions, and assigning dedicated pins and (*B.*) generating XML-based schematic files from its output. The final step (*C.*), which is carried out to deal with the final layout of the schematics, must be done subsequently (in part manually). The generator is developed as a Java command line application to maintain platform-independence and ensure that it can be integrated into standard tool chains and build management tools.

```
{
  modules: [
    { name: "MSP1",
      type: "MSP430F5529_LaunchPad"},
    { name: "SD1",
      type: "MicroSD_BreakoutBoard"}
  ],
  connections: [
    {
      name: "SPI_Connection1",
      type: "SPI",
      participants: [
        { name: "MSP1", role: "master" },
        { name: "SD1", role: "slave" }
      ]
    }
  ]
}
```

Figure 11. A system model containing two modules connected via SPI.

A. Connection Establishment and Pin Assignment

During this first step, JSON data structure analysis presents the main challenge. The whole system must be interconnected appropriately using the previously explained definition files. To do so, all connections within the system definition must be matched at the beginning of the process. This task subsumes the discovery of connections between modules, their mapping to certain interface types, and the final wire allocation required to interconnect all participants. Specifically, each connection has a type and a finite number of participants with different roles, interfaces, and pins. These pins must then be connected to the newly introduced wires, belonging to the communication. Several different types of wires can be used to connect the participants with each other:

The easiest wires to use are common wires, which can be assigned to a pool of free pins of the module. These wires are marked with *wiresingle* within the interface definition. Due to the fact that all unused General-Purpose Input/Output (GPIO) pins of a module can be used for this purpose, they need to be assigned last.

Furthermore, every participant can connect itself directly to bus wires via its dedicated pins, depending on, e.g., the type of MCU used. In the case of an MSP430 MCU, certain pins are electrically connected to an interface circuit, as defined in its module definition (cf. Figure 4). These pins must, therefore, be matched with the connection's wires (cf. Figure 10). The interface definition must match roles and pins accordingly to correctly interconnect the participants of each connection.

Another type of wires that can be used are multiple wires. If we take SPI as an example, the master needs to have as many chip-select wires as slaves with which it wants to communicate. Therefore, this type of wire – marked with *wiremultiple*, as previously defined – must clone itself to obtain the number of wires needed.

These different types of wires must be connected to the pins of the modules to establish a proper connection or *net* according to the interface definition. The interconnected modules with their nets form a holistic JSON-based description of the system.

B. Schematic and Board Layout Generation

Utilizing the interconnected system description, schematics and board layouts can be generated. In our case, EAGLE's XML data structure [2] was used to form a dedicated output file for schematics and board layouts. To generate those plans, (1) design blocks for each module must be loaded, (2) the previously found connections must be applied and (3) the connected design blocks must be placed on an empty schematic plan or board layout. The basis of every schematic and board plan forms an empty EAGLE plan, on which the explained actions are performed.

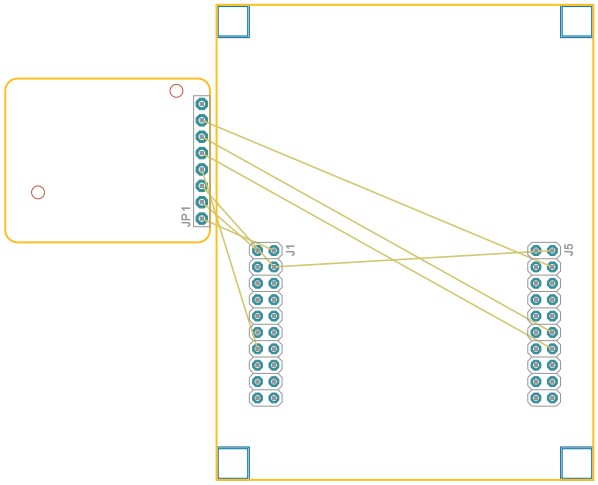


Figure 12. Raw output of the board layout generated as displayed in EAGLE.

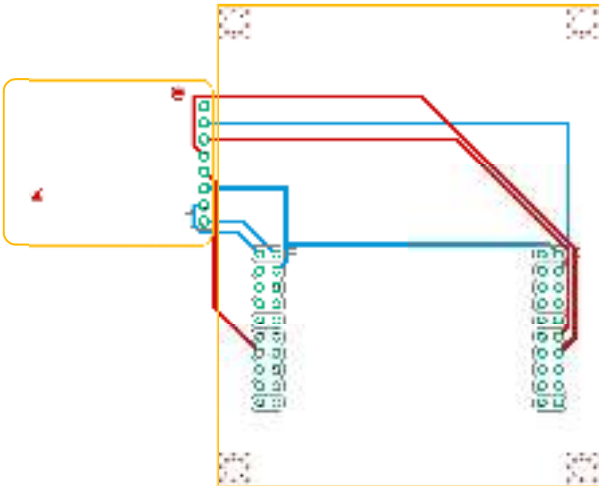


Figure 13. Board layout after auto-routing in EAGLE.

(1) **Instantiation:** In this step, each module must be instantiated by loading the corresponding design block of its type. To avoid overlapping of signal names and, therefore, unwanted connections between modules of the same type, suffixes are added according to the instance's name. For readability purposes, these suffixes are equal to the instance's name defined in the system definition file (e.g., **_MSP1*). This can be easily seen when comparing, e.g., Figures 6/7 and 14.

(2) **Interconnection:** This step must be carried out to form the whole system according to the JSON-based holistic description. Therefore, pins of each module must be assigned to the wires of a connection within the system. To do so, each connection again must be applied separately to each participant. As the system description already contains information, as to which pin of a module must be connected to which wire, this can be done quite easily. In this case, to avoid overlapping of signals, a dot notation style is used do distinguish between wires of different connection instances (e.g., *SPI_CONNECTION1.** in Figure 14).

(3) **Placement:** This step, which is the computationally most expensive step, must be carried out to merge the connected instances of each module into an empty plan, as a great deal of XML parsing is required here. To create consistent plans, the design blocks must be prepared well beforehand to avoid, e.g., inconsistencies within board layers or signal names. To keep the modules from overlapping, a two-dimensional translation of each module must be executed as part of each merge procedure as well. In total, two merging steps are required for each module – one for the schematic and one for the board layout. As this approach generates connection PCBs ("motherboards") where one can plug in modules, only placeholders are used.

Finally, the two generated XML structures are exported and saved into different files (one for the schematics, one for the board layout) for further usage.

C. Routing Generated Schematics and Board Layouts

As layouting and routing of PCBs is a non-trivial task, and engineers need a great deal of experience when performing a task like this, ***papageno***PCB cannot be used to produce final variants of a board. It is recommended to use EAGLE's auto-routing functionality or manual routing to finalize the already well-prepared layouts.

VI. Experiments and Evaluation

Within this section, the previously explained concept on how to define and create PCBs from a definition language is shown in different examples and evaluation. At first, a simple proof of concept is presented in Section VI-A, followed by some analysis and evaluation on scalability and performance of the algorithms in Section VI-B. Section VI-C shows a use case with a corresponding manufactured and equipped prototype PCB. In this case, a comparison with other approaches is not executed, as all related works go in different directions. Hence, there are no acceptable metrics for comparison provided.

A. Proof of Concept

The proof of concept comprises the generation of the system definition as shown in Figure 11. As mentioned above, the system created consists of two modules interconnected with one SPI bus, whereas the processor board serves as master. The schematics generation step yields in the drawing depicted in Figure 14. Compared with the LaunchPad™'s design block shown in Figure 6, one can see the differences in the net names. As examples, *P2.0* has been replaced with *WIRE0*, and *P3.0* is now assigned to *SPI_CONNECTION1.MOSI*. These wires connect to pins 7 and 6 of the MicroSD Breakout Board on the left, respectively. Also, each unconnected pin is given a suffix describing its module (cf. *_MSP1*). These newly introduced net names are the results of the wire generation explained in Section V-A. As the reusability of schematic plans is an important aspect, the feature of non-overlapping module placement can be emphasized as well. The result of the board layout generation step is shown in Figure 12, as described in Section V-B. The fine lines show non-routed connections between the pins. As the generated plan will, of course, be

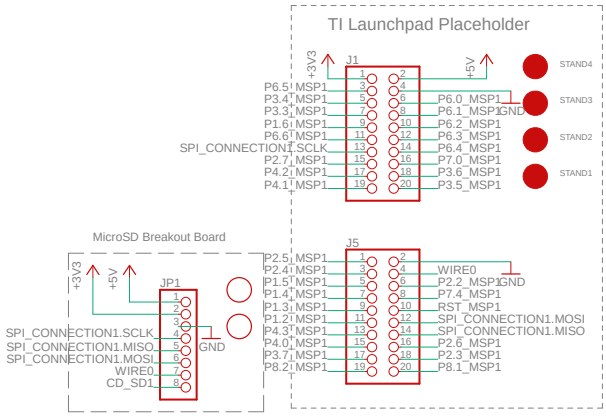


Figure 14. Raw output of the schematics generated as displayed in EAGLE.

manufactured as a real hardware PCB, no single component can overlap in the final layout. Routing of the board has to be either performed manually or by using a design tool's built-in auto router. A feasible layout variant is presented in Figure 13. EAGLE can also be used to check the correctness of the XML file format.

B. Scalability and Performance

In this section, we describe measurements and investigations that concern the performance of the PCB-generating process. All discussed evaluations use one setup as a reference. The application was executed with a Java 10 virtual machine on an Intel Core i7 7500U@2.7GHz with 16 gigabytes of RAM. Table I shows the mean execution time and the combined output XML file size of the generation process of different test case scenarios, which are explained below. All test cases featured a different number of participants (part.) which consisted of masters (M) and slaves (S) with different connection types. The experiment is based on the one presented in [1], but the generation tool version is more stable and optimized and the objective differs slightly, yielding different measurements.

Each test case is based on the example described in Section VI-A, but with different constellations concerning the numbers and types of participants and connections. All test cases were executed 100 times. Four types of test scenarios with seven test cases each were conducted: Within the first scenario, just one SPI connection was present, with a varying number of slaves each test case. The second scenario comprised two SPI connections with an increasing number of slaves. Test scenario three had one I^2C connection and was similar to scenario one, whereas scenario four included SPI and I^2C connections to a single master with an increasing number of slaves. The devolution of the mean execution time (in ms) in all test scenarios is shown in Figure 15. When comparing all scenarios, the trend observed is relatively similar: All performance graphs show a linear devolution with an additive, logarithmic-like component. The linear component is due to the linear increase in the complexity of the test cases. The logarithmic-like growth observed can be explained by the decreasing, additive overhead of the linear component when processing similar connection reasoning, as well as the XML schematic and layout data. This is also the reason why

TABLE I. MEAN EXECUTION TIMES FOR DIFFERENT SCENARIOS.

#	*test scenario description*	*execution time*	*file size*
	1 SPI conn. (scenario 1)		
0	2 part. (1 M, 1 S)	656.22 ms	94 KiB
1	3 part. (1 M, 2 S)	751.98 ms	111 KiB
2	4 part. (1 M, 3 S)	852.95 ms	128 KiB
3	5 part. (1 M, 4 S)	933.71 ms	144 KiB
4	6 part. (1 M, 5 S)	1 013.67 ms	161 KiB
5	7 part. (1 M, 6 S)	1 108.81 ms	178 KiB
6	7 part. (1 M, 7 S)	1 193.25 ms	194 KiB
	2 SPI conn. (scenario 2)		
0	4 part. (1 M and 1 S each)	890.77 ms	160 KiB
1	6 part. (1 M and 2 S each)	1 060.91 ms	192 KiB
2	8 part. (1 M and 3 S each)	1 234.36 ms	226 KiB
3	10 part. (1 M and 4 S each)	1 398.30 ms	259 KiB
4	12 part. (1 M and 5 S each)	1 557.30 ms	293 KiB
5	14 part. (1 M and 6 S each)	1 731.26 ms	326 KiB
6	14 part. (1 M and 7 S each)	1 879.93 ms	360 KiB
	1 I^2C conn. (scenario 3)		
0	2 part. (1 M, 1 S)	665.61 ms	106 KiB
1	3 part. (1 M, 2 S)	771.77 ms	136 KiB
2	4 part. (1 M, 3 S)	889.56 ms	167 KiB
3	5 part. (1 M, 4 S)	989.33 ms	198 KiB
4	6 part. (1 M, 5 S)	1 106.84 ms	228 KiB
5	7 part. (1 M, 6 S)	1 205.48 ms	259 KiB
6	7 part. (1 M, 7 S)	1 332.82 ms	290 KiB
	1 I^2C and 1 SPI conn. (scenario 4)		
0	3 part. (1 M, 1 S each)	782.70 ms	127 KiB
1	5 part. (1 M, 2 S each)	971.42 ms	174 KiB
2	7 part. (1 M, 3 S each)	1 166.01 ms	222 KiB
3	9 part. (1 M, 4 S each)	1 344.08 ms	269 KiB
4	11 part. (1 M, 5 S each)	1 530.92 ms	317 KiB
5	13 part. (1 M, 6 S each)	1 719.09 ms	364 KiB
6	13 part. (1 M, 7 S each)	1 873.92 ms	411 KiB

doubling the numbers in the first test case resulted in much higher values than in test case two. Test scenario four is the only one that displays a steeper curve. This is due to the combination of different connection types, yielding less-optimal algorithm executions. As XML processing is quite costly, some further optimizations are needed. As the overall file size displayed linear growth, no correlation was observed between file size and execution time.

C. Simple Use Case and Prototype Manufacturing

The simple use case in this section is a minimalistic, generic control system. In order to react to its environment, it must

- read several analog voltage values,
- output analog voltage values, and
- store a data log permanently in two different ways, such as it is on the one hand side
  - "detachable", and on the other hand side
  - stored with low energy consumption, yet non-volatile and redundant.

After manually spanning a selection space over the available equipment in our lab, those requirements yield in a system configuration with

- an MCU to execute the control algorithms (→ MSP430 on a corresponding LaunchPad™),
- two 4-channel analog-to-digital converters (ADCs) to read voltage values (→ Adafruit 4-channel Breakout Board featuring an ADS1115 ADC [34]),
- a digital-to-analog converter (DAC) to output voltage values (→ Adafruit 12-bit DAC board featuring a MCP4725 DAC [35]),

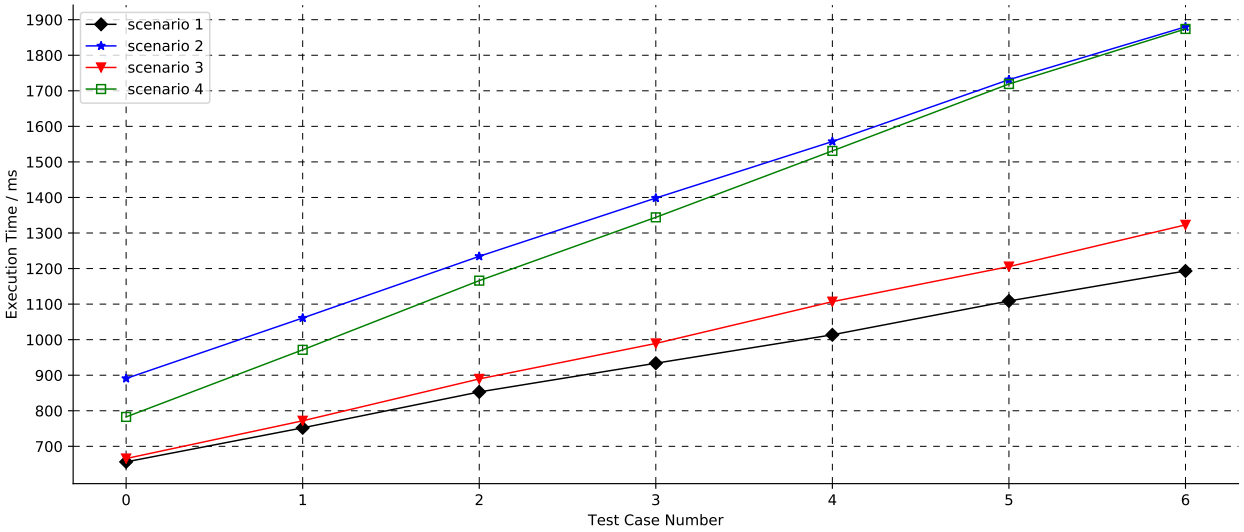


Figure 15. Performance graph for different test cases in all four scenarios.

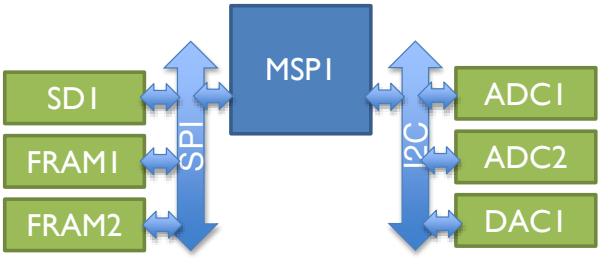


Figure 16. The block diagram of all modules in the example use case.

- a MicroSD card module to log data in a detachable way (→ MicroSD Breakout Board), and
- two Ferroelectric Random Access Memory (FRAM, [36]) modules to store data in a low-power, non-volatile, redundant way (→ Adafruit SPI FRAM Breakout Board featuring a MB85RS64V FRAM [37]).

The block diagram of this configuration, including its interconnection, is shown in Figure 16. It can be seen that a total of two different connection types must be used to interconnect all modules. The corresponding system definition is presented in Figure 17. It contains all module instances, both connections with their types, participants, and roles. The result of running the PCB generation and manually routing the board layout is depicted in Figure 18. With this result it is possible to manufacture an actual PCB, equip it with the hardware modules, flash the control system ASW and BSW, and run measurements. The software setup in this case consists of our own real-time operating system *MCSmart*OS [38][39] enriched with a modular driver management system and a simple test application. The equipped and running prototype is shown in Figure 19, where it is connected to several measurement devices (e.g., a PicoScope 2205 MSO [40] with digital and analog inputs) through debug wires and probes to observe and verify correct functionality.

```
{
  modules: [
    { name: "MSP1",  type: "MSP430F5529_Launchpad" },
    { name: "ADC1",
      type: "Adafruit_ADS1115_16Bit_I2C_ADC" },
    { name: "ADC2",
      type: "Adafruit_ADS1115_16Bit_I2C_ADC" },
    { name: "DAC1",
      type: "Adafruit_MCP4725_12Bit_I2C_DAC" },
    { name: "FRAM1", type: "Adafruit_FRAM_SPI" },
    { name: "FRAM2", type: "Adafruit_FRAM_SPI" },
    { name: "SD1",   type: "MicroSD_BreakoutBoard" }
  ],

  connections: [
    {
      name: "I2C_Connection1",
      type: "I2C",
      participants: [
        { name: "MSP1", role: "master" },
        { name: "ADC1", role: "slave" },
        { name: "ADC2", role: "slave" },
        { name: "DAC1", role: "slave" }
      ]
    },
    {
      name: "SPI_Connection1",
      type: "SPI",
      participants: [
        { name: "MSP1",  role: "master" },
        { name: "FRAM1", role: "slave" },
        { name: "FRAM2", role: "slave" },
        { name: "SD1",   role: "slave" }
      ]
    }
  ]
}
```

Figure 17. The system model of the prototype.

VII. Conclusion and Future Work

In conclusion, the present work based on the ***papageno***PCB approach represents a novel, top-down concept to develop an embedded system for a multitude of possible application scopes. Having only a model-based system description at hand, it is possible to use ***papageno***PCB to generate hardware schematics and board layouts accordingly. This opens up numerous new possibilities towards automatic system generation

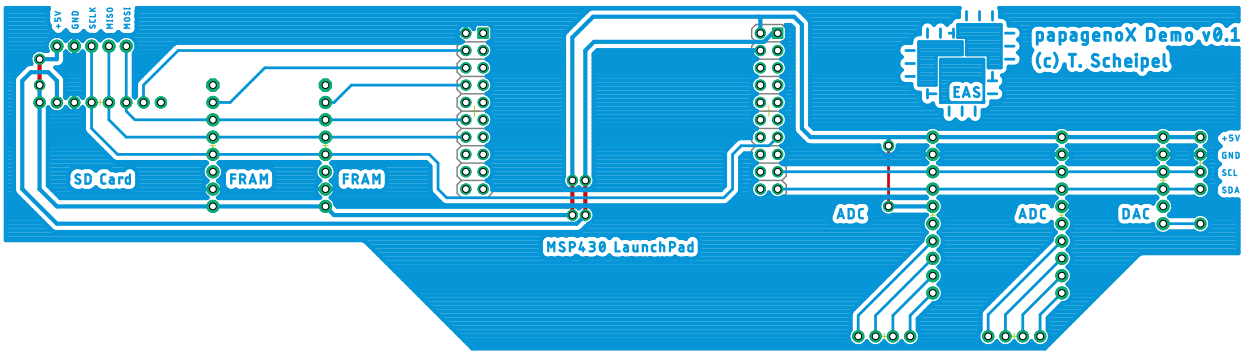


Figure 18. Prototype PCB layout with a MSP430 LaunchPad™ connected to three SPI and three I²C modules (manually routed).

Figure 19. Fully equipped prototype board with debug wires.

and across several abstraction levels including, e.g., automatic bus balancing, bandwidth engineering, optimization towards functional and non-functional hardware requirements. All these things can be carried out even before building the actual hardware for the system. The use of these concepts requires the availability of in-depth information about the electrical and mechanical characteristics of all parts of a PCB, so that the hardware can be optimized in terms of non-functional metrics such as bandwidth or power consumption. Generally speaking, the presented concept is able to optimize systems under development regarding different, user-defined metrics already at design level. Therefore, metrics to measure the overall improvement in general are hard to define, as they depend on the actual system's development process and its requirements and properties. Due to the generic design, new models can be integrated easily, and it will be even possible to take a non-module-based approach on the electrical device or component level, proper definitions presumed.

Concerning future work, a detailed extraction of system models from a profound ASW source code analysis is of utmost importance. Therefore, we are working on introducing annotations into our operating system environment [39], which will enable us to automatically generate system definition files. These annotations can either be introduced into the code as compiler keywords (e.g., pragmas, defines) or as comments. As some work is already being done to improve the automatic portability of real-time operating systems [41], the proposed approach could be used to build a system for which only the application code must be programmed. The rest of the system can then be generated automatically. Even suitable and application-optimized processor architectures [42] or application-specific logic components on reconfigurable computing platforms could be created and included by taking this approach. The ultimate goal is to establish ***papageno***X as a universal embedded systems generator, which uses only ASW source code or models as an input.

References

[1] T. Scheipel and M. Baunach, "papagenoPCB: An Automated Printed Circuit Board Generation Approach for Embedded Systems Prototyping," in *ICONS 2019 - The Fourteenth International Conference on Systems*, 3 2019, pp. 20–25.

[2] Autodesk, Inc., *EAGLE XML Data Structure 9.1.0*, 2018.

[3] J. Schäuffele and T. Zurawka, *Automotive Software Engineering*, ser. ATZ/MTZ-Fachbuch. Springer Fachmedien Wiesbaden, 2016.

[4] A. Kouba, J. Navratil, and B. Hnilička, "Engine Control using a Real-Time 1D Engine Model," in *VPC – Simulation und Test 2015*, J. Liebl and C. Beidl, Eds. Wiesbaden: Springer Fachmedien Wiesbaden, 2018, pp. 295–309.

[5] Infineon Technologies AG, "TC1797 – 32-Bit Single-Chip Microcontroller," 2014.

[6] B. Eichberger, E. Unger, and M. Oswald, "Design of a versatile rapid prototyping engine management system," in *Proceedings of the FISITA 2012 World Automotive Congress*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 135–142.

[7] Infineon Technologies AG, "TC1796 – 32-Bit Single-Chip Microcontroller," 2007.

[8] ——, "TC1798 – 32-Bit Single-Chip Microcontroller," 2014.

[9] L. H. Crockett, R. A. Elliot, M. A. Enderwitz, and R. W. Stewart, *The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc*. UK: Strathclyde Academic Media, 2014.

[10] devicetree.org, *Devicetree Specification*, Dec. 2017, release v0.2.

[11] G. M. Swinkels and L. Hafer, "Schematic generation with an expert system," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 9, no. 12, pp. 1289–1306, Dec 1990.

[12] B. Singh, D. O'Riordan, B. G. Arsintescu, A. Goel, and D. R. Deshpande, "System and method for circuit schematic generation," US Patent US7 917 877B2, 2011.

[13] J. Schnerr, O. Bringmann, A. Viehl, and W. Rosenstiel, "High-performance Timing Simulation of Embedded Software," in *2008 45th ACM/IEEE Design Automation Conference*, June 2008, pp. 290–295.

[14] B. Schommer, C. Cullmann, G. Gebhard, X. Leroy, M. Schmidt, and S. Wegener, "Embedded Program Annotations for WCET Analysis," in *WCET 2018: 18th International Workshop on Worst-Case Execution Time Analysis*. Barcelona, Spain: Dagstuhl Publishing, Jul. 2018, [retrieved: Nov, 2019]. [Online]. Available: https://hal.inria.fr/hal-01848686

[15] S. Chakravarty, Z. Zhao, and A. Gerstlauer, "Automated, retargetable back-annotation for host compiled performance and power modeling," in *9th Int'l Conference on Hardware/Software Codesign and System Synthesis*, Piscataway, NJ, USA, 2013, pp. 36:1–36:10, [retrieved: Nov, 2019]. [Online]. Available: http://dl.acm.org/citation.cfm?id=2555692.2555728

[16] A. D. Pimentel, "Exploring Exploration: A Tutorial Introduction to Embedded Systems Design Space Exploration," *IEEE Design Test*, vol. 34, no. 1, pp. 77–90, Feb 2017.

[17] F. Herrera, H. Posadas, P. Peñil, E. Villar, F. Ferrero, R. Valencia, and G. Palermo, "The COMPLEX methodology for UML/MARTE Modeling and design space exploration of embedded systems," *Journal of Systems Architecture*, vol. 60, no. 1, pp. 55 – 78, 2014, [retrieved: Nov, 2019]. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S138376211300194X

[18] T. Saxena and G. Karsai, "A meta-framework for design space exploration," in *2011 18th IEEE International Conference and Workshops on Engineering of Computer-Based Systems*, April 2011, pp. 71–80.

[19] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack, "MiniZinc: Towards a Standard CP Modelling Language," in *Principles and Practice of Constraint Programming – CP 2007*, C. Bessière, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 529–543.

[20] AUTOSAR, "Classic platform release 4.3.1," 2017.

[21] IEEE Standards Association, *IEEE 1076-2008 - IEEE Standard VHDL Language Reference Manual*, 2008.

[22] ——, *IEEE 1364-2005 - IEEE Standard for Verilog Hardware Description Language*, 2005.

[23] ——, *IEEE 1666-2011 - IEEE Standard for Standard SystemC Language*, Sep. 2011.

[24] B. Maes, "Comparison of contemporary file systems," *Citeseer*, 2012.

[25] S. C. Hill, J. Jelemensky, M. R. Heene, S. E. Groves, and D. N. Debrito, "Queued serial peripheral interface for use in a data processing system," US Patent US4 816 996, 1989.

[26] ECMA International, *ECMA-404: The JSON Data Interchange Syntax*, 2nd ed., Dec. 2017.

[27] Texas Instruments, *MSP430F5529 LaunchPad™ Development Kit (MSP--EXP430F5529LP)*, Apr. 2017.

[28] ——, *MSP430x5xx and MSP430x6xx Family User's Guide*, Mar. 2018, [retrieved: Nov, 2019]. [Online]. Available: http://www.ti.com/lit/ug/slau208q/slau208q.pdf

[29] Adafruit Industries, *Micro SD Card Breakout Board Tutorial*, Jan. 2019, [retrieved: Nov, 2019]. [Online]. Available: https://cdn-learn.adafruit.com/downloads/pdf/adafruit-micro-sd-breakout-board-card-tutorial.pdf

[30] Autodesk, Inc., "EAGLE," [retrieved: Nov, 2019]. [Online]. Available: https://www.autodesk.com/products/eagle/

[31] NXP Semiconductors, Inc., *UM10204: I2C-bus specification and user manual*, Apr. 2014, rev. 6.

[32] International Organization for Standardization, *ISO 11898: Road vehicles – Controller area network (CAN)* , 2nd ed., Dec. 2015.

[33] ARM Ltd., *AMBA AXI and ACE Protocol Specification*, 2017, [retrieved: Jul, 2019].

[34] Texas Instruments, *Ultra-Small, Low-Power,16-Bit Analog-to-Digital Converter with Internal Reference*, Oct. 2009, [retrieved: Nov, 2019]. [Online]. Available: http://www.ti.com/lit/ds/symlink/ads1114.pdf

[35] Microchip, *12-Bit Digital-to-Analog Converter with EEPROM Memory in SOT-23-6*, 2009, [retrieved: Nov, 2019]. [Online]. Available: https://cdn-shop.adafruit.com/datasheets/mcp4725.pdf

[36] H. Ishiwara, M. Okuyama, and Y. Arimoto, *Ferroelectric random access memories: fundamentals and applications*. Springer Science & Business Media, 2004, vol. 93.

[37] Fujitsu Semiconductor, *64KBit SPIMB85RS64V*, 2013, [retrieved: Nov, 2019]. [Online]. Available: https://cdn-shop.adafruit.com/datasheets/MB85RS64V-DS501-00015-4v0-E.pdf

[38] M. Baunach, "Advances in Distributed Real-Time Sensor/Actuator Systems Operation," Dissertation, University of Würzburg, Germany, Feb. 2013. [Online]. Available: http://opus.bibliothek.uni-wuerzburg.de/frontdoor/index/index/docId/6429

[39] R. Martins Gomes, M. Baunach, M. Malenko, L. Batista Ribeiro, and F. Mauroner, "A Co-Designed RTOS and MCU Concept for Dynamically Composed Embedded Systems," in *Proc. of the 13th Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, 2017, pp. 41–46.

[40] *PicoScope 2205 MSO Mixed Signal Oscilloscope*, Pico Technology, 2016. [Online]. Available: https://www.picotech.com/download/datasheets/PicoScope2205MSODatasheet-en.pdf

[41] R. Martins Gomes and M. Baunach, "A Model-Based Concept for RTOS Portability," in *Proc. of the 15th Int'l Conference on Computer Systems and Applications*, Oct. 2018, pp. 1–6.

[42] F. Mauroner and M. Baunach, "mosartMCU: Multi-Core Operating-System-Aware Real-Time Microcontroller," in *Proc. of the 7th Mediterranean Conference on Embedded Computing*, Jun. 2018, pp. 1–4.

7.3 E-III: *papageno*X: Generation of Electronics and Logic for Embedded Systems from Application Software

◇ Authors: Tobias Scheipel, and Marcel Baunach
◇ Date: February, 2020
◇ DOI: 10.5220/0009159701360141
◇ Reference in bibliography: [151]
◇ Presented by myself at the 9th International Conference on Sensor Networks (SENSORNETS'20), Valetta, Malta.
◇ Published in the proceedings of SENSORNETS'20 (INSTICC).

Summary ***papageno***X introduces an approach on how to generate all lower layers of an embedded system when having only the Application Software (ASW) and its requirements at hand. The concept includes an in-depth analysis of the ASW, the subsequent generation of the entire hardware platform, and the Basic Software (BSW) configuration that suits the ASW. When speaking of hardware, Printed Circuit Boards (PCBs) and Hardware Description Language (HDL) code for reconfigurable logic on Field-Programmable Gate Arrays (FPGAs) is considered.

Author's Contribution I am the main author of this paper, and I have entirely envisioned, designed, and implemented the concept of the paper myself. My supervisor Marcel Baunach supported me through discussions and in the composition of the paper.

Copyright ©2020 by SciTePress. The version included in this thesis is reprinted, with permission, from the published version stated above, which can be accessed in the SciTePress Digital Library:
https://www.scitepress.org

*papageno*X: Generation of Electronics and Logic for Embedded Systems from Application Software

Tobias Scheipel[a] and Marcel Baunach[b]
Institute of Technical Informatics, Graz University of Technology, Graz, Austria

Keywords: Embedded Systems, Printed Circuit Board, Design Automation, Hardware/Software Co-design, Systems Engineering, Reconfigurable Logic.

Abstract: Embedded systems development usually starts with hardware engineering based on specific requirements of the systems. These requirements are mainly derived from the needs of the not yet developed software to be executed on the system. This process is predictive and many iterations are thus needed, as new requirements often arise during the software development period. In the future, the market will demand more and more sophisticated embedded systems with a much reduced time to market. It will thus be inevitable that system prototypes and series products will need to be created as fast as possible. To enable this, we propose a top-down approach termed ***papageno***X, dealing with the question of "How to generate all layers X of the embedded systems stack including hardware and reconfigurable logic units from application software?". The present work is a work in progress and deals with the definition of the research questions and several ideas and concepts of how to fundamentally solve them. Hence, it aims at introducing ideas to create a generator for embedded systems electronics, reconfigurable logic and software.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

1 INTRODUCTION

There is currently a common trend in embedded systems software engineering: code generation. Examples of this include code generation for Application Software (ASW) and Basic Software (BSW) from MATLAB models within automotive Electronic Control Units (ECUs) (AUTOSAR, 2017), and even Real-Time Operating Systems (RTOSs) can be ported automatically to other platforms by generating code from formal models (Gomes and Baunach, 2019). In this work, we propose a concept going a step further and generating an entire embedded system with its hardware from software, as illustrated in Fig. 1. Hence, we formulate the overall research question as *"How can we enable automatic embedded systems generation from Application Software?"*. Hardware in this case subsumes not only reconfigurable logic in Field Programmable Gate Arrays (FPGAs), but also electronics on Printed Circuit Boards (PCBs).

Our newly introduced process, which is primarily designed for prototyping, will also help to evaluate the intended ASW changes after deployment to quantify the consequences on the lower layers and thus assess their feasibility and cost. This is especially important to find out if hardware changes in the system are necessary or if the existing system is still sufficient to execute the ASW. In addition, the process is generally designed to drive the use of reconfigurable logic by seamlessly integrating it into the development and maintenance process.

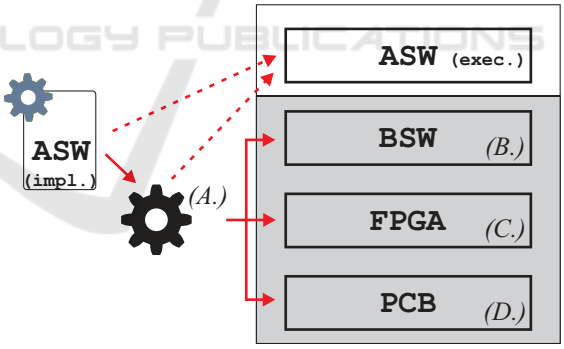


Figure 1: ***papageno***X divided into the main parts, shown in the stack of an embedded system (impl./exec. = implementation and executable of the ASW).

[a] https://orcid.org/0000-0003-0691-6119
[b] https://orcid.org/0000-0002-3716-2682

The present paper presents the work in progress and is organized as follows: Section 2 motivates the topic of embedded systems generation from application software. In Section 3, the main idea and the scientific approach for ***papageno*X** is outlined. In Section 4 all the envisioned parts of the concept are discussed in detail and the overall concept is sketched with an exemplary use case in Section 5. The paper concludes with Section 6, in which the current state, the progress made and the future work are explained.

2 MOTIVATION

The common process in embedded systems engineering today usually starts with designing the hardware according to requirements defined in natural language. Subsequently, based on educated guesses or best practices, a prototype is assembled, simulated and tested. After this step, software development on the prototype can start. As during software development, new requirements can arise, this process is prone to require multiple iterative redesign cycles in order to result in a suitable hardware solution. In this context, software development is much more dynamic and flexible than the new development of hardware prototypes. As a result, the prototypes are used for as long as possible in order to avoid the time and effort involved in hardware redesign. In some cases, however, the redesign becomes inevitable, resulting in many different prototype variants with diverse properties. AS a result the state of the art is what we call software development following hardware design ("software follows or adjusts to hardware").

With our contribution, we intend to invert the process of embedded systems design towards "hardware follows software", introduced in detail in the following Section 3.

3 SCIENTIFIC APPROACH WITHIN *papageno*X

In order to tackle the problems that arise as described above, we intend to derive the requirements of a system under development directly from previously implemented application code in order to automatically generate suitable and optimized lower layers. As application code does not explicitly specify discrete components but only functional requirements (FRs), the analysis shall also consider additional non-functional requirement (NFR) (e.g., communication load, data retention time).

We have given our concept the name ***papageno*X**, an abbreviation for **P**rototyping **AP**plication-based with **A**utomatic **GEN**eration **O**f **X**, where **X** is for the name of the corresponding layer (cf. Fig. 1).

The following structure is proposed to achieve the envisioned goal: The process starts with ASW development (implementation), followed by a thorough analysis of all its FRs and NFRs. Based on this analysis, and a library of available hardware and software modules, a configuration or selection space can be opened over all these components, modules, and possible interconnections. As this selection space can still contain system configurations not suitable for the objective (e.g., due to the unavailability of components), further filtering towards special design decisions must be applied. This yields several potential configurations composed from different components that can be optimized again in order to achieve the best system that is suitable for all requirements. Of course, optimization and selection metrics are also based on FRs and NFRs (e.g., cost, size, energy consumption, Electromagnetic Compatibility (EMC) characteristics). This process is applied to all three layers depicted in Fig. 1 in different extent and is explained in the following sections.

In this work, we additionally envision the use of FPGAs or System on Programmable Chips (SoPCs) to build hardware acceleration units by synthesizing complex algorithms in logic (proper interfacing included) to add further adaptability.

The fundamental paradigm shift towards "hardware follows software" is mandatory to enable the utmost flexibility when designing future embedded systems. Since we understand established development methods as being there for a reason, we will also research methods to quantify the suitability of previously generated hardware configurations after software changes, including possible necessary system modification proposals.

4 ENVISIONED PARTS OF *papageno*X

The toolchain of the ***papageno*X** concept will contain a number of powerful features that facilitate automatic systems generation. This set of tools can be divided into four main parts, being *(A.)* ASW analysis, *(B.)* BSW generation, *(C.)* FPGA generation, and *(D.)* PCB generation. All those parts are depicted in Fig. 1 and are discussed in detail below. This work presents solely the ideas of from a work in progress concept and only few new findings at this stage.

4.1 ASW Analysis

Starting from the analysis of applications, the research question to deal with is *"Which information needs to be included in ASW including NFRs to allow the creation a embedded system?"*. This means that the requirements of the software to the system must be extracted and forwarded to the next parts. Since the ASW to be analyzed does not have to be a stand-alone executable, the executable part must also be extracted (cf. dotted arrows in Fig. 1).

The requirements analysis derives a selection space by using constraint solving and design space exploration strategies (Saxena and Karsai, 2011)(Nethercote et al., 2007) and logical expression matching. This selection space contains multiple possible overall structures of the entire embedded system:

- suitable computing platform module(s) and
- other hardware components/modules alongside their interconnections
- BSW modules and drivers for hardware modules

Furthermore, this step also performs a pre-selection of algorithms within the code, which are best suited for mapping to reconfigurable logic (FPGAs) in part *(C.)*.

The results of the requirements analysis is forwarded to the following parts.

4.2 BSW Generation

The next part deals with BSW generation and the question on *"How can we derive the needed BSW features from ASW?"*. To deal with this question, we base our approach on our own RTOS, *MCSmart*OS (Martins Gomes et al., 2017), as it can be used with different computing platforms (e.g., MSP430 (Texas Instruments,), RISC-V (Chen and Patterson, 2016)). This operating system must include a generic driver management concept enriched with non-functional properties to allow modular composition of hardware modules including resource sharing mechanisms. Furthermore, such a management concept enables optimization towards different functional and non-functional requirements (e.g., selectable resource management strategies or scheduling algorithms). The output of this part is a BSW tailored to the ASW's needs.

4.3 FPGA Generation

Parallel to the previous part, the FPGA generation starts based on the ASW analysis. Here, the question *"How can we extract application specific logic and automatically map it to FPGAs?"* is addressed. As software functions are commonly executed on Microcontroller Units (MCUs), our contribution aims at hosting application-specific logic components on FPGA platforms, exemplarily depicted as filled squares connected to a soft core MCU in the FPGA module in Fig. 2.

The ASW analysis must discover mapping candidate software functions (via, e.g., heuristics) in order to enable this functionality within our generator. Research is on deciding which functions should be transferred from software to hardware and how to generate or interface them with respect to FRs and NFRs of the ASW (e.g., by transforming a function call to on-chip I/O operations). We will also investigate suitable ASW design patterns and programming paradigms to facilitate this process. Therefore, the entire concept is supported by research on the creation of a flexible soft-core MCU architecture for hosting application-specific logic (based on e.g., *mosart*MCU (Mauroner and Baunach, 2018), RISC-V (Waterman et al., 2016)). The goal is to work on consistent development processes, where all system functions and algorithms are still developed in software, but then automatically mapped to on-chip extensions during compilation and synthesis and interfaced accordingly. We are also working on partial reconfiguration support at runtime to achieve more hardware flexibility and simplified adaptation. In the expectation of more dynamic software updates, we must also consider modifications to the soft-core architecture. This is also important because modifiable hardware will be available in the future, but there is hardly any runtime support for it.

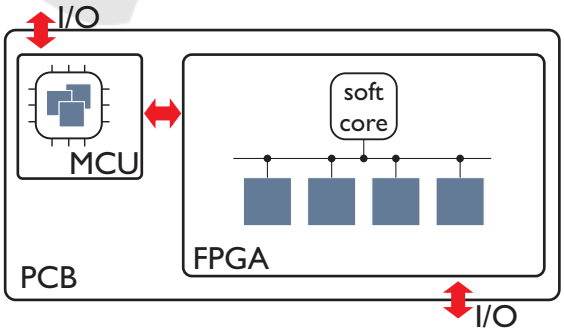


Figure 2: Hosting application-specific logic within an embedded system.

4.4 PCB Generation

The part dealing with the lowest layer in the embedded systems stack is sketched in this section. The

attempt is made to answer the questions *"What information is needed to automatically generate PCBs from ASW?"* and *"How can this information be used to generate a PCB prototype matching all ASW requirements?"*. These questions can be answered in two sub-parts by utilizing dedicated hardware modules as abstractions. The generated board is either a motherboard, where these modules can be plugged in, or a single PCB in which all the necessary electrical components are integrated.

The first of the two sub-parts deals with the translation of constraints to a selection space of possible system configurations. This is an abstract process and does not deal with any electrical properties yet. To do so, enriched module definitions must be created. Our definitions are based on generic JSON (ECMA International, 2017) descriptions including not only their functional, but also their non-functional properties. This approach allows for possible systems configurations to be established and a set of suitable configurations can be filtered out by the toolchain. The result is an intermediate system definition. This is because the electrical feasibility of the connections between the modules has not yet been considered, and no dedicated PCB has been generated yet.

In this second sub-part, the interconnection on electrical level is being dealt with. As stated in (Scheipel and Baunach, 2019), this sub-part uses an interconnected intermediate system definition to convert it into a PCB. Several different inputs are necessary for this:

- hardware module definition files and their corresponding design block schematics and board layouts,
- all necessary interface definitions to adequatly interconnect the previously defined hardware modules, and
- one dedicated system definition file which represents the overall structure of the embedded system (cf. first sub-part).

With these input files, schematics and board plans can be generated. In the first step, the output of our approach is based on the EAGLE (Autodesk, Inc.,) XML format.

5 USE CASE EXAMPLE

An idea to create an embedded system shall be derived from the need of observing or controlling some specific (physical) process. In our case, it starts with the need of "measuring distances for further processing in a car". This requirements definition in natural language spans a selection space of a great number of components and software possibilities for implementing our system. As the engineer knows that there is a need to "measure distance" (I.) and send the distance values to be "further processed in a car" (II.), application software development for the use case can start by using generic Application Programming Interfaces (APIs):

```
1: [...]
2: read_distance(&distance,
                 SensorType.Automotive);
3: send_data(&rec, &distance,
                 CommType.Automotive);
4: [...]
```

This piece of pseudo-code written by the engineer to meet the requirements in language specifies these requirements in more detail, narrowing down the list of components to be considered. In line 2, distance data is read from an automotive-grade sensor (requirement I.; technology need not be specified in detail). Subsequently, it is sent to the recipient `rec` over some automotive in-car communication technology (requirement II.; again, not specified in detail) in line 3. These abstract requirements could also be lists of requirements for further specification.

The next few lines try to give an impression on how ***papageno***X tackles the automatic system generation. Part *(A.)* has already started with the use of certain APIs and is carried out first (cf. Fig. 1). It selects a suitable BSW with drivers and services for part *(B.)*, and selects hardware components for part *(D.)*.

In the analysis part *(A.)*, the requirements are examined more closely in order to obtain a better overview of the actual needs of the ASW. The analysis goes hand in hand with the other parts of the concept.

Subsequently in *(B.)*, the BSW containing an Operating System (OS) is assembled, including all APIs and necessary drivers to use the hardware components selected in part *(D.)*. A tailored version of *MCSmart*OS with a generic driver platform is utilized for this reason. The set of features to enable this functionality is currently under development.

In part *(D.)*, from the set of all possible hardware components and modules several possible system configurations can arise. To simplify the explanation of our use case, the PCB generation part then selects a system containing: a ultra-sonic sensor of type HC-SR04 (Cytron Technologies, 2013) for requirement (I.), and a CAN (International Organization for Standardization, 2015) transceiver with a controller for requirement (II)., both connected to a MSP430 LaunchPad™ (Texas Instruments, 2017) as a computing platform.

Finally, the requirement "measuring distances for

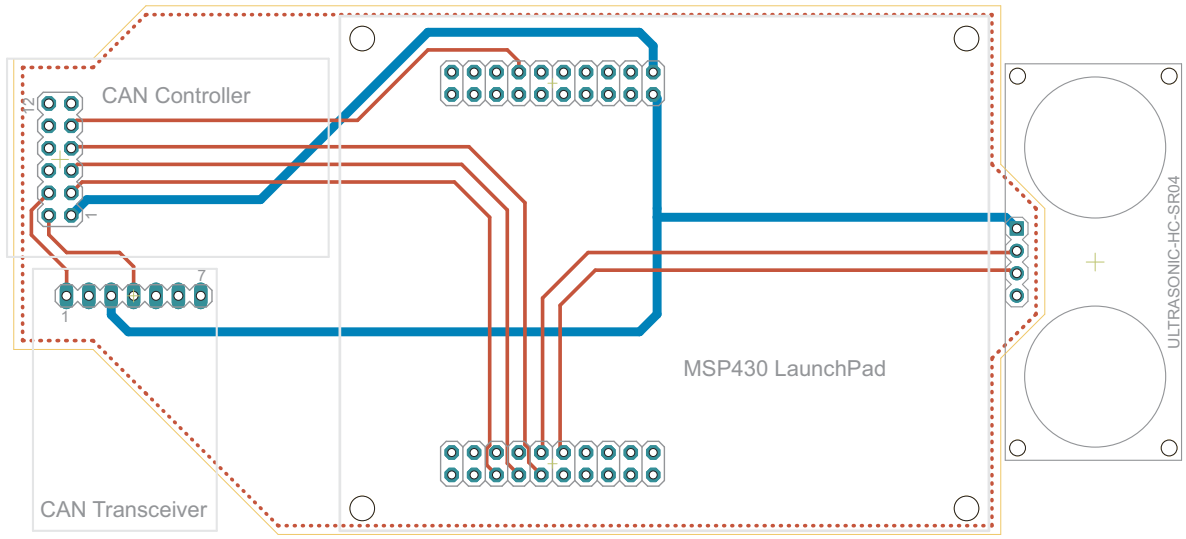


Figure 3: The generated PCB with module placeholders of the explained use case for measuring distances in a car.

further processing in a car" and two lines of code create the embedded system depicted in Fig. 3.

6 CONCLUSION AND FUTURE WORK

In the course of this work we presented a novel approach on how to generate an entire embedded system when only having application code at hand. All the ideas presented are from our current work in progress. As our approach aims at inverting the state-of-the-art process of embedded systems design from "software follows hardware" to "hardware follows software", this opens up new questions in the context of software analysis, and hardware, logic and software generation.

Since this is a work in progress, not all of its parts have yet been implemented. To date, hardware generation on PCB-level is already implemented and partially published, but there is room for improvement throughout the concept. Future steps include the improvement of constraint matching mechanisms for hardware module selection, the improvement of the requirements analysis and the entirety of the reconfigurable logic generation part. Also, the BSW layer must be constantly improved in order to meet the future demands on the concept. As ***papageno***X is generic and module based, it can be easily adopted to different domains in the future. Lowering of the module granularity to, e.g., electrical components level (resistors, capacitors, integrated circuits, and so on) is possible. The next steps in this work are to finish the PCB generation alongside the ASW analysis for FRs and NFRs.

REFERENCES

Autodesk, Inc. EAGLE. [retrieved: Nov, 2019].

AUTOSAR (2017). Classic platform release 4.3.1.

Chen, T. and Patterson, D. A. (2016). RISC-V Geneology. Technical Report UCB/EECS-2016-6, EECS Department, University of California, Berkeley.

Cytron Technologies (2013). *HC-SR04 User's Manual*.

ECMA International (2017). *ECMA-404: The JSON Data Interchange Syntax*, 2 edition.

Gomes, R. M. and Baunach, M. (2019). Code generation from formal models for automatic rtos portability. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 271–272.

International Organization for Standardization (2015). *ISO 11898: Road vehicles – Controller area network (CAN)* , 2 edition.

Martins Gomes, R., Baunach, M., Malenko, M., Batista Ribeiro, L., and Mauroner, F. (2017). A Co-Designed RTOS and MCU Concept for Dynamically Composed Embedded Systems. In *Proc. of the 13th Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pages 41–46.

Mauroner, F. and Baunach, M. (2018). mosartMCU: Multi-Core Operating-System-Aware Real-Time Microcontroller. In *Proc. of the 7th Mediterranean Conference on Embedded Computing*, pages 1–4.

Nethercote, N., Stuckey, P. J., Becket, R., Brand, S., Duck, G. J., and Tack, G. (2007). MiniZinc: Towards a Standard CP Modelling Language. In Bessière, C., editor, *Principles and Practice of Constraint Program-*

ming – CP 2007, pages 529–543, Berlin, Heidelberg. Springer Berlin Heidelberg.

Saxena, T. and Karsai, G. (2011). A meta-framework for design space exploration. In *2011 18th IEEE International Conference and Workshops on Engineering of Computer-Based Systems*, pages 71–80.

Scheipel, T. and Baunach, M. (2019). papagenoPCB: An Automated Printed Circuit Board Generation Approach for Embedded Systems Prototyping. In *ICONS 2019 - The Fourteenth International Conference on Systems*, pages 20–25.

Texas Instruments. MSP430 ultra-low-power sensing and measurement MCUs.

Texas Instruments (2017). *MSP430F5529 LaunchPad™ Development Kit (MSP--EXP430F5529LP)*.

Waterman, A., Lee, Y., Patterson, D. A., and Asanovic, K. (2016). The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.1. Technical Report UCB/EECS-2016-118, EECS Department, University of California, Berkeley.

SCITEPRESS
SCIENCE AND TECHNOLOGY PUBLICATIONS

7.4 E-IV: *papageno*ReQ: Generation of Embedded Systems from Application Code Requirements

◇ Authors: Tobias Scheipel, and Marcel Baunach
◇ Date: June, 2021
◇ DOI: 10.1109/ICECCE52056.2021.9514257
◇ Reference in bibliography: [152]
◇ Presented by myself at the 3rd International Conference on Electrical, Communication and Computer Engineering (ICECCE'21), Kuala Lumpur, Malaysia.
◇ Published in the proceedings of ICECCE'21 (IEEE).

Summary ***papageno***ReQ introduces the intermediate system definition generation from Application Software (ASW) requirements. The concept uses a query language to match these requirements to the properties of modules in a database. System configurations can be derived out of the matched modules, acting as input to ***papageno***PCB to generate Printed Circuit Boards (PCBs) for the final embedded system.

Author's Contribution I am the main author of this paper, and I have entirely envisioned, designed, and implemented the concept of the paper myself. My supervisor Marcel Baunach supported me through discussions and in the composition of the paper.

Copyright ©2021 by IEEE. The version included in this thesis is reprinted, with permission, from the published version stated above, which can be accessed in the IEEE Xplore Digital Library:
https://ieeexplore.ieee.org

papagenoReQ: Generation of Embedded Systems from Application Code Requirements

Tobias Scheipel
Institute of Technical Informatics
Graz University of Technology
Graz, Austria
tobias.scheipel@tugraz.at

Marcel Baunach
Institute of Technical Informatics
Graz University of Technology
Graz, Austria
baunach@tugraz.at

***Abstract*—At the beginning of every embedded system is a set of requirements to its software and hardware. In the common case, these requirements change a lot throughout the development cycle and are highly dependent on the software being coded. This already hints at a major problem of the state-of-the-art system development process, in which the hardware platform is established before the software engineering starts: changes to the hardware platform are very costly not only by means of money, but also time. To counteract this problem, we propose a concept where the hardware platform can be directly generated out of a set of requirements, making an inversion of this system development process possible. Using a query language to map requirements to the system to properties of modules that are composed into system configurations, we highly improve the flexibility within embedded systems design and development. The present work uses a set of requirements as an input and generates printed circuit boards of an embedded system matching those requirements based on a developer's module library.**

***Index Terms*—embedded systems, printed circuit board, requirements, properties, query languages, systems engineering**

I. Introduction and Motivation

In our modern world, embedded systems get ubiquitous more and more. This trend is caused by several developments like, e.g., the advance of the Internet of Things (IoT) [1] or an increased number of Electronic Control Units (ECUs) [2] in modern vehicles. Usually, the development of an embedded system is a very time-consuming issue, carried out by professionals from different fields of expertise (cf. the V-5model in the automotive industry [3]). This inevitably leads to numerous different error sources in the design process, for which usually a bottom-up approach is selected. This means that the computing platform as well as all the surrounding modules are combined before or in parallel to the software development process, based on some requirements to the system. If, however, these requirements change during the development, a redesign of hardware must take place. This can lead to severe problems, as, e.g., newly introduced functionality to the software cannot be executed on the already designed device due to hardware limitations.

To counteract the mentioned problems, we propose a novel, top-down method of designing the electronics hardware in embedded systems by making use of functional and non-functional requirements and properties of a system. By only defining requirements, our approach automatically selects suitable modules and creates a Printed Circuit Board (PCB) out of those modules. Based on our already proposed concepts called ***papageno***X [4] and ***papageno***PCB [5], [6], we introduce a concept that makes use of query languages in order to match properties of electronic modules to requirements of the system design process, and automatically creates feasible PCBs. Hence, the concept in this work takes requirements to a system as an input and creates electronic devices with properties fulfilling these requirements.

The remainder of this paper is organized as follows: Section II discusses already established concepts and related works in this field of research. It is followed by Section III, where the approach to our novel concept is outlined in detail and shown in application in Section IV. The paper is concluded in Section V, where also an outlook to future developments is given.

II. Related Work

This Section outlines related and existing work with respect to embedded system modeling and generation, as well as requirements matching. As functional and non-functional requirements are essential to embedded systems design, extraction and processing of these requirements has been researched thoroughly [7]–[9]. Also, design space exploration [10], [11] is of interest here. A nice way to do so is by modeling the embedded system and its constraints in [12]. This approach focuses on the combination of model driven engineering alongside electronic system level and design exploration technologies. Object-oriented modeling is already shown for hybrid systems in [13]. It can also be used to explore requirements patterns for embedded systems [14], [15]. There, structural and behavioral information is captured within their patterns, primarily applied to the automotive industry. When it comes to composition of embedded systems, [16] shows a component-based toolkit to speed up the design process, primarily applied within the avionics industry.

By means of constraint solving and matching, MiniZinc [17] or Picat [18] show interesting ways. These systems can also be applied to embedded systems design [19] for optimization purposes.

978-1-6654-3897-1/21/$31.00 ©2021 IEEE

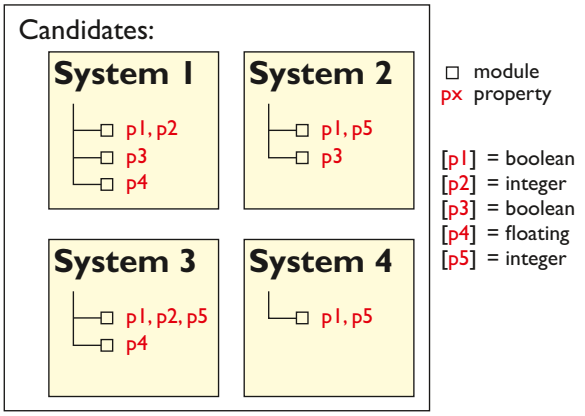


Fig. 1. Example system configuration candidates.

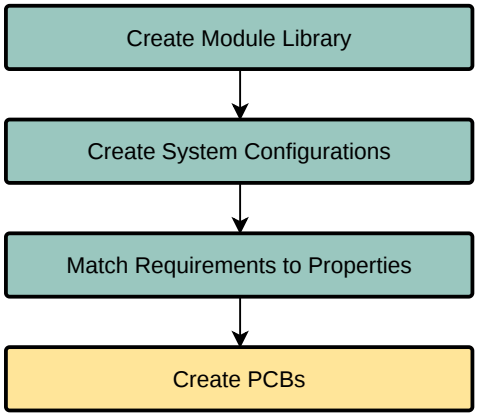


Fig. 2. The program flow within our implementation.

All these mentioned concepts and approaches influenced and motivated the present work, as no holistic framework to generate PCBs only from requirements input is proposed yet.

III. Approach

Within this Section, the approach to our concept is explained in detail. Its name ***papageno***ReQ stands for **P**rototyping **AP**plication-based with **A**utomatic **GEN**eration **O**f systems from **Re**quirement **Q**ueries. It is separated in Subsections III-A, where the fundamental mathematical definitions are shown and III-B, where the actual implementation is outlined in detail.

A. Mathematical Definitions

In this Section we define all the necessary terms needed for our approach. The main terms within this work are ”module”, ”system”, ”requirement” and ”property”, and will be explained as follows:

$$M = \{M_c, M_p\} = \{c_1, c_2, ..., c_n, p_1, p_2, ..., p_m\} \quad (1)$$

M denotes the set of all feasible modules, containing the two sets of the computing modules M_c (containing n modules c_i) and the peripheral modules M_p (containing m modules p_i).

From the set of all modules M, a sub-set $M_{library} \subseteq M$ is formed. This is since not all conceivable components are available in the library of the user. After this first filtering, the hardware components matching the requirements of the Application Software (ASW), $M_{matching} \subseteq M_{library}$, are selected and put together in a number of o feasible, not yet interconnected system configurations $S_{nc} = \{s_{nc,1}, s_{nc,2}, ..., s_{nc,o}\}$.

For every not-yet-interconnected system configuration $s_{nc,i}$, the following equation must hold:

$$s_{nc,i} = \{m \subseteq M_{matching} \mid m \text{ matches } \geq 1 \text{ requirements}\} \quad (2)$$

It is important to note that each of these system configurations $s_{nc,i}$ must contain at least one computing module c_i and an arbitrary number of peripheral modules p_i. If the system contains more than one module, these modules are tried to be interconnected in feasible system configuration candidates $S_c \subseteq S_{nc}$. An example is depicted in Fig. 1. As one can see, a system contains of modules with certain properties. All these properties combined form the properties of the system candidate, which can be matched against requirements.

The interconnectable final system configurations $S_{matching} \subseteq S_c$, which are matching all the initial requirements to our wanted system, form the output of this part. These configurations are selected due to their properties.

B. Implementation

To explain the implementation of our approach, we start with the necessary steps and the program flow needed in order to obtain a PCB from a set of requirements in Section III-B1. Subsequently, we continue with explaining the class structure within our approach in Section III-B2 and we finish with explaining the algorithm of how we match requirements to the final system to properties of modules within system candidates in Section III-B3. The implementation is carried out as a Java application.

1) Program Flow: The input for our program flow is a constraint string consisting of all requirements the final system must fulfil with its properties. At the beginning, a library of modules is being created. In this step, already a pre-filtering takes place, as modules that do not meet a single requirement are skipped beforehand.

Subsequently, out of all these modules feasible system configurations are established according to our mathematical definitions. In this step, a power set of all modules is created. Out of all those not-yet-interconnected modules (cf. Section III-A), the interconnectable systems are created. This step uses the class structure of our implementation, as explained in Section III-B2 and is discussed in detail there.

After all the feasible system configurations are determined, the constraint matching algorithm (explained in Section III-B3) can be applied in order to obtain the best-suitable set of systems. Out of these systems, PCBs are generated

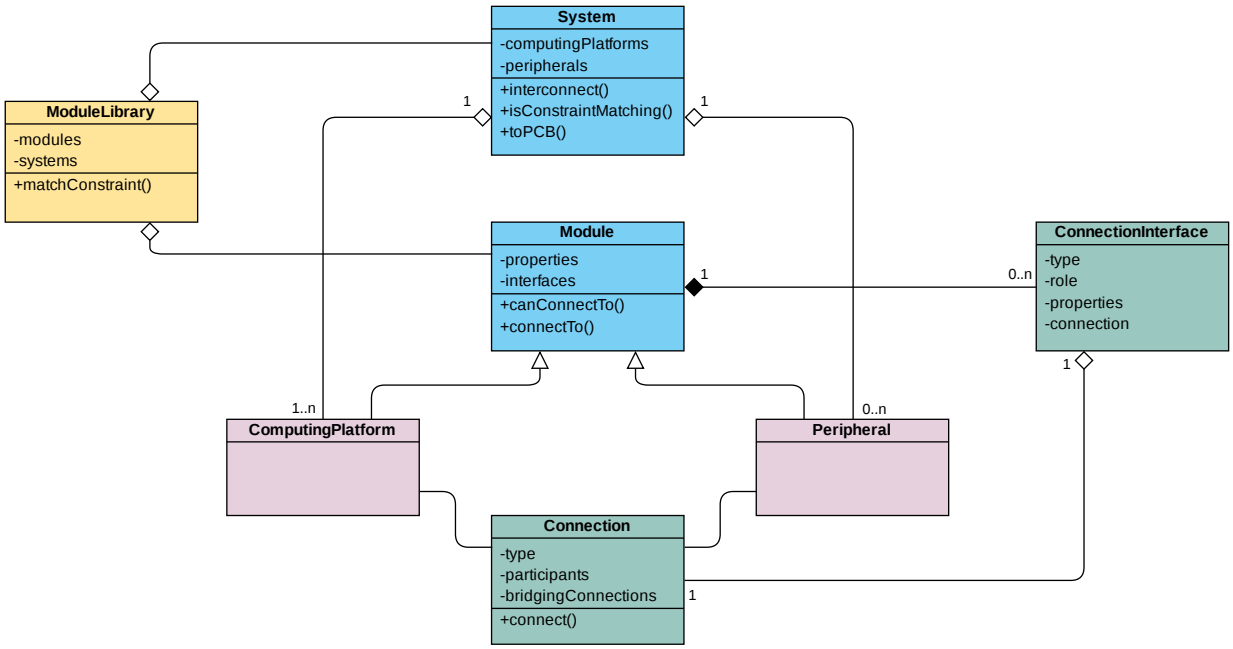


Fig. 3. Class diagram to obtain system configurations.

following the concept proposed in [6]. To do so, system descriptions based on JSON [20] must be generated as inputs for ***papageno***PCB in order for it to generate EAGLE-based [21] PCB schematics and layouts.

2) Class Diagrams: In order to implement the algorithm of our approach, the class diagram shown in Fig. 3 is realized and explained in the following. The library in our approach bases on a instance of the class `ModuleLibrary`, containing modules (cf. M in Section III-A) and final systems. System configurations (cf. S in Section III-A), as explained above, are represented by the class `System`, whereas a module is represented by the class `Module`.

A module consists of certain properties and interfaces; the latter being represented by the class `ConnectionInterface`. There are two specializations of `Module`, being `ComputingPlatform` (cf. M_c in Section III-A) and `Peripheral` (cf. M_p in Section III-A). A module can try to or directly connect to another module by using the methods `canConnectTo()` and `connectTo()`, respectively. Each `ConnectionInterface` has a certain type (e.g., SPI [22], I^2C [23]), a role (e.g., master, slave), some properties and a connection, being represented by an instance of the class `Connection`. It connects the interfaces of specializations of modules together, forming a connection. To do so, it has a certain type to be matched with a `ConnectionInterface`, participants and, if needed, links to other connections (bridging connections). With the method `connect()`, a `ConnectionInterface` of a module can be tried to be added to the connection.

As mentioned before, the composition of at least one computing platform and an arbitrary number of peripherals is called a system configuration (`System`). This class has a method called `interconnect()`, where the modules of a system configuration are tried to be interconnected (every instance's `connect()` is used). To match a system configuration against requirements, the method `isConstraintMatching()` can be used. If a system configuration ends up as a result of our approach, the method `toPCB()` is used to generate its PCB.

Coming back to the class `ModuleLibrary`, the method `matchConstraints()` is used to trigger the approach. It filters and combines all modules into system configurations and tries to interconnect them. The resulting system configurations are then returned and their PCBs can be generated. The algorithm on how to do this is explained in III-B3.

3) Matching Requirements to Properties: The final step within our approach marks the matching of the initial requirements to the properties of each and every system configuration, composed of several modules. The algorithm is based on decision tables, much like the one presented in Table I generated out of Fig. 1. Values $NULL$ in this regard mean that the property is not present in any kind within the system. The module library class (cf. Section III-B2) is able to retrieve this table for all system configuration candidates. Within this table, only relevant properties with respect to the initial requirements are present. Examples for constraints containing initial requirements (already with their solution systems) can be seen in Equations (3), (4) and (5).

TABLE I
DECISION TABLE FOR DIFFERENT SYSTEMS

	p1	p2	p3	p4	p5
Sys 1	*true*	100	*true*	3.2	*NULL*
Sys 2	*true*	*NULL*	*true*	*NULL*	250
Sys 3	*true*	50	*NULL*	7.1	700
Sys 4	*true*	*NULL*	*NULL*	*NULL*	100

$$(r_1 == true)\ \&\&\ (r_2 >= 100)\ \rightarrow\ \{\text{Sys 1}\} \quad (3)$$

$$(r_1 == true)\ \&\&\ (r_5 > 100)\ \rightarrow\ \{\text{Sys 2, Sys 3}\} \quad (4)$$

$$(r_2 > 75)\ ||\ (r_4 >= 7.0)\ \rightarrow\ \{\text{Sys 1, Sys 3}\} \quad (5)$$

The process of obtaining solution systems for a constraint is explained as follows. The decision table acts as a relational database, upon which queries can be executed. The technology behind this is an Embedded Derby database. After establishing this database, the initial requirements constraint is transformed into a relational database query in SQL [24]. An example for Equation (3) in this regard is

```
SELECT * FROM SYSTEMS WHERE (r1 = TRUE
    AND r2 >= 100);
```

This requirements query yields all interconnectable system configurations that match all the requirements mentioned in the input constraint string of our concept. If we have a look at Table I and Equations (3), (4) and (5), one can clearly see that the constraints directly map onto the decision table. The requirements matching delivers **all** the systems out of the generated filtered library that match the given constraint. Further non-theoretical examples will be given in the following Section IV.

Out of these systems now the intermediate system description can be formed. This description is needed for the PCB generation step that follows after this step and is explained in detail in Section III-B1.

IV. USE CASE

Let us consider an embedded system in the application field of the IoT, where several measurements within an environment must be made, processed and stored in a detachable way (e.g., by some memory card). A constraint for such a use case might be

$$(can_read_voltage_values = true)\ \&\&\ (can_store_data_detachable = true)\ \&\&\ (cpu_frequency >= 2MHz).$$

When creating and filtering our IoT system library w.r.t. the input constraint, the output of the decision table can look like displayed in Table II. The properties p1 to p6 have the following meaning and subsume all properties the pre-filtered modules contain:

```
{
  modules: [
    { name: "MSP2", type: "MSP430_device" },
    { name: "MIC3", type: "MicroSD_Board" },
    { name: "16B1", type: "16Bit_I2C_ADC" }
  ],
  connections: [
    {
      name: "I2C_CONNECTION_0",
      type: "I2C",
      participants: [
        { name: "MSP2", role: "master" },
        { name: "16B1", role: "slave" }
      ]
    },
    {
      name: "SPI_CONNECTION_1",
      type: "SPI",
      participants: [
        { name: "MSP2", role: "master" },
        { name: "MIC3", role: "slave" }
      ]
    }
  ]
}
```

Fig. 4. System description of the final system generated for the use case.

p1: number of modules
p2: can store data detachable
p3: can store data non-volatile
p4: can write voltage values
p5: can read voltage values
p6: Central Processing Unit (CPU) frequency

TABLE II
DECISION TABLE FOR THE USE CASE

	p1	p2	p3	p4	p5	p6
Sys 0	2	*true*	*true*	*NULL*	*NULL*	$25MHz$
Sys 1	3	*true*	*true*	*NULL*	*true*	$25MHz$
Sys 2	1	*NULL*	*NULL*	*NULL*	*NULL*	$25MHz$
Sys 3	2	*NULL*	*NULL*	*NULL*	*true*	$25MHz$

The solution in this case is Sys 1, fulfilling all requirements of the constraint. The output to the intermediate system description is depicted in Fig. 4 and contains three hardware modules interconnected by two hardware bus systems (I^2C and SPI). The module MSP2 is the computing platform, fulfilling the requirement "cpu frequency". It is connected to a 16-bit-Analog-to-digital Converter (ADC) 16B1 via I^2C, fulfilling the requirement "can read voltage values", as well as to a micro SD board MIC3 via SPI, fulfilling the requirement "can store data detachable". The generated PCB layout in EAGLE format can be seen in Fig. 5.

If one now wants to add the requirement of also writing voltage values to an output pin of the system, the constraint changes to

$$(can_read_voltage_values = true)\ \&\&\ (can_write_voltage_values = true)\ \&\&\ (can_store_data_detachable = true)\ \&\&\ (cpu_frequency >= 2MHz).$$

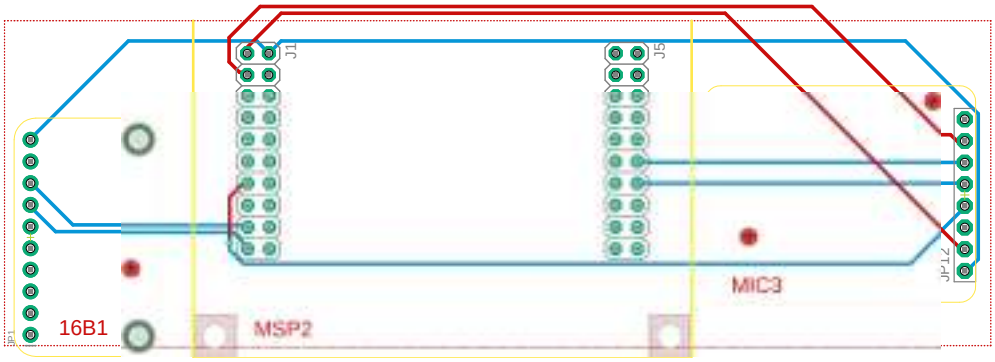


Fig. 5. Generated board layout of the use case.

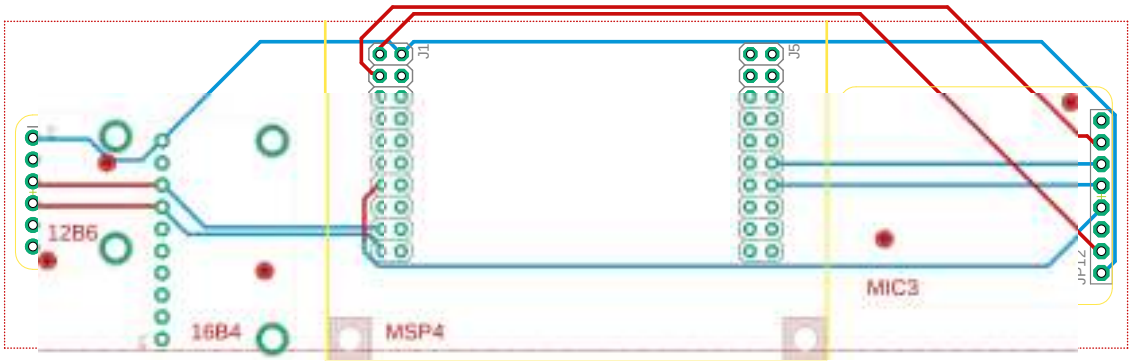


Fig. 6. Slightly adapted generated board layout of the extended use case.

This leads to another decision table, depicted in Table III. The solution in this extended use case is Sys 3, as it is fulfilling even the extended requirement. In the system description in Fig. 7, one can see that a 12-bit-Digital-to-analog Converter (DAC) labelled "12B6" was added to the modules list as well as to the I^2C connection. This is also visible in the board layout in Fig. 6. The names of the modules are slightly different, as can also be seen in Fig. 7; yet this is only for naming conventions. The modules behind these names stay the same and can be identified by their types.

TABLE III
DECISION TABLE FOR THE EXTENDED USE CASE

	p1	p2	p3	p4	p5	p6
Sys 0	2	$NULL$	$NULL$	$true$	$NULL$	$25MHz$
Sys 1	3	$true$	$true$	$true$	$NULL$	$25MHz$
Sys 2	3	$NULL$	$NULL$	$true$	$true$	$25MHz$
Sys 3	4	$true$	$true$	$true$	$true$	$25MHz$
Sys 4	2	$true$	$true$	$NULL$	$NULL$	$25MHz$
Sys 5	1	$NULL$	$NULL$	$NULL$	$NULL$	$25MHz$
Sys 6	3	$true$	$true$	$NULL$	$true$	$25MHz$
Sys 7	2	$NULL$	$NULL$	$NULL$	$true$	$25MHz$

These examples show that our proposed concept yields the expected results not only for theoretical but also for practical use cases. It also shows that by changing the requirements to

```
{
  modules: [
    { name: "MSP4", type: "MSP430_device" },
    { name: "16B4", type: "16Bit_I2C_ADC" },
    { name: "MIC3", type: "MicroSD_Board" },
    { name: "12B6", type: "12Bit_I2C_DAC" }
  ],
  connections: [
    {
      name: "SPI_CONNECTION_1",
      type: "SPI",
      participants: [
        { name: "MSP4", role: "master" },
        { name: "MIC3", role: "slave" }
      ]
    },
    {
      name: "I2C_CONNECTION_3",
      type: "I2C",
      participants: [
        { name: "MSP4", role: "master" },
        { name: "16B4", role: "slave" },
        { name: "12B6", role: "slave" }
      ]
    }
  ]
}
```

Fig. 7. System description of the final system generated for the extended use case.

the system, often only small adaptions must be made to the final PCB design. It can also result in no changes at all, if, e.g., the final system has more properties than required (cf. p3: can store data non-volatile) or the new requirements are only changes in values the system already fulfils.

V. Conclusion and Future Work

To sum up, the present work named ***papageno***ReQ shows a novel concept on how to create PCBs out of a set of requirements to the final system. To do so, a module-based approach is chosen, meaning that systems are composed out of a set of modules with certain properties. These properties are subsequently matched to the initial requirements to the system. The generation process consists of a requirements matching algorithm based on SQL, a system composition functionality based on an object-oriented data structure, and a PCB generation portion based on ***papageno***PCB. The proposed concept increases the flexibility in embedded systems design, as the hardware of the system must not be designed up front, but can be created after designing the software and knowing its requirements. Hence, it inverts the established system design process and introduces a "hardware follows software" mentality, meaning that software is no longer needed to work around hardware deficiencies, but the hardware is actively derived out of software's requirements. It also enables the possibility to check whether a redesign of the hardware is necessary, in case requirements change over the course of the system lifetime.

Our concept is evaluated by showing a use case with certain requirements and an extended version of this very use case. It is shown that the generated PCB is composed out of modules with properties that match the initial requirements.

In the future it is feasible to automatically export the requirements from the software instead of the present, more manual approach. At the moment, the driver structure of the used Operating System (OS) [25] within our framework yields those requirements. Also, optimizations like minimizing the size of the modules and algorithm improvements are envisioned.

References

[1] J. Manyika, M. Chui, P. Bisson, J. Woetzel, R. Dobbs, J. Bughin, and D. Aharon, *Unlocking the potential of the Internet of Things*, McKinsey Global Institute, Jun. 2015.

[2] "This car runs on code," IEEE Spectrum, March 2018. [Online]. Available: https://spectrum.ieee.org/transportation/systems/this-car-runs-on-code

[3] J. Schäuffele and T. Zurawka, *Automotive Software Engineering*, ser. ATZ/MTZ-Fachbuch. Springer Fachmedien Wiesbaden, 2016.

[4] T. Scheipel and M. Baunach, "Papagenox: Generation of electronics and logic for embedded systems from application software," 2020, pp. 136–141, cited By 0. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85082989951&partnerID=40&md5=5ed0661cf76f7d22c8a8689589c2ceff

[5] ——, "papagenoPCB: An Automated Printed Circuit Board Generation Approach for Embedded Systems Prototyping," in *ICONS 2019 - The Fourteenth International Conference on Systems*, 3 2019, pp. 20–25.

[6] ——, "Towards an Automated Printed Circuit Board Generation Concept for Embedded Systems," *International Journal on Advances in Systems and Measurements*, vol. 12, no. 3&4, 2019, invited, submitted. [Online]. Available: http://www.iariajournals.org/systems_and_measurements/

[7] M. Broy and T. Stauner, "Requirements engineering for embedded systems," *Informationstechnik und technische Informatik*, vol. 41, pp. 7–11, 1999.

[8] E. Sikora, B. Tenbergen, and K. Pohl, "Requirements engineering for embedded systems: An investigation of industry needs," in *International Working Conference on Requirements Engineering: Foundation for Software Quality*. Springer, 2011, pp. 151–165.

[9] T. Pereira, D. Albuquerque, A. Sousa, F. Alencar, and J. Castro, "Towards a metamodel for a requirements engineering process of embedded systems," in *2016 VI Brazilian Symposium on Computing Systems Engineering (SBESC)*. IEEE, 2016, pp. 93–100.

[10] S. Künzli, *Efficient design space exploration for embedded systems*. ETH Zurich, 2006, vol. 81.

[11] A. D. Pimentel, "Exploring Exploration: A Tutorial Introduction to Embedded Systems Design Space Exploration," *IEEE Design Test*, vol. 34, no. 1, pp. 77–90, Feb 2017.

[12] F. Herrera, H. Posadas, P. Peñil, E. Villar, F. Ferrero, R. Valencia, and G. Palermo, "The COMPLEX methodology for UML/MARTE Modeling and design space exploration of embedded systems," *Journal of Systems Architecture*, vol. 60, no. 1, pp. 55 – 78, 2014, [retrieved: Nov, 2019]. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S138376211300194X

[13] H. Elmqvist, F. E. Cellier, and M. Otter, "Object-oriented modeling of hybrid systems," -, 1993.

[14] S. Konrad and B. H. C. Cheng, "Requirements patterns for embedded systems," in *Proceedings IEEE Joint International Conference on Requirements Engineering*, 2002, pp. 127–136.

[15] S. Konrad, B. H. C. Cheng, and L. A. Campbell, "Object analysis patterns for embedded systems," *IEEE Transactions on Software Engineering*, vol. 30, no. 12, pp. 970–992, 2004.

[16] J. A. Stankovic, Ruiqing Zhu, R. Poornalingam, Chenyang Lu, Zhendong Yu, M. Humphrey, and B. Ellis, "VEST: an aspect-based composition tool for real-time systems," in *The 9th IEEE Real-Time and Embedded Technology and Applications Symposium, 2003. Proceedings.*, 2003, pp. 58–69.

[17] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack, "MiniZinc: Towards a Standard CP Modelling Language," in *Principles and Practice of Constraint Programming – CP 2007*, C. Bessière, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 529–543.

[18] n.-f. Zhou, H. Kjellerstrand, and J. Fruman, *Constraint Solving and Planning with Picat*. Springer International Publishing, 11 2015.

[19] K. Kuchcinski, "Constraint programming in embedded systems design: Considered helpful," *Microprocessors and Microsystems*, vol. 69, pp. 24–34, 2019.

[20] ECMA International, *ECMA-404: The JSON Data Interchange Syntax*, 2nd ed., Dec. 2017.

[21] Autodesk, Inc., "EAGLE," [retrieved: Nov, 2019]. [Online]. Available: https://www.autodesk.com/products/eagle/

[22] S. C. Hill, J. Jelemensky, M. R. Heene, S. E. Groves, and D. N. Debrito, "Queued serial peripheral interface for use in a data processing system," US Patent US4 816 996, 1989.

[23] NXP Semiconductors, Inc., *UM10204: I2C-bus specification and user manual*, Apr. 2014, rev. 6.

[24] *ISO/IEC 9075-1:2016 Information technology – Database languages – SQL – Part 1: Framework (SQL/Framework)*, International Organisation for Standardization Std.

[25] R. Martins Gomes, M. Baunach, M. Malenko, L. Batista Ribeiro, and F. Mauroner, "A Co-Designed RTOS and MCU Concept for Dynamically Composed Embedded Systems," in *13th Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, 2017, pp. 41–46.

7.5 R-I: System-Aware Performance Monitoring Unit for RISC-V Architectures

◇ Authors: Tobias Scheipel, Fabian Mauroner, and Marcel Baunach
◇ Date: August, 2017
◇ DOI: 10.1109/DSD.2017.28
◇ Reference in bibliography: [153]
◇ Presented by myself at the 20th Euromicro Conference on Digital Systems Design (DSD'17), Vienna, Austria.
◇ Published in the proceedings of DSD'17 (IEEE).

Summary This work proposes a novel Performance Monitoring Unit (PMU) for the RISC-V architecture that can measure execution times and events in tasks or the Operating System (OS) without interfering with the rest of the system. It is designed to be hardware and software platform independent and easily extendable, and it integrates a method to efficiently reuse hardware counter resources at runtime.

Author's Contribution I am the main author of this paper, and I have completely designed and implemented the concept of the paper myself. Fabian Mauroner supported me in the concept development, as the work is also used in his project. He and Marcel Baunach supported me through discussions and in the composition of the paper.

Copyright ©2017 by IEEE. The version included in this thesis is reprinted, with permission, from the published version stated above, which can be accessed in the IEEE Xplore Digital Library:
https://ieeexplore.ieee.org

System-Aware Performance Monitoring Unit for RISC-V Architectures

Tobias Scheipel, Fabian Mauroner, and Marcel Baunach
Institute of Technical Informatics
Graz University of Technology
Graz, Austria
tobias.scheipel@student.tugraz.at, {mauroner, baunach}@tugraz.at

***Abstract*—Due to increasing complexity of software in embedded systems, performance aspects become much more important this days. This should happen early in the development process. Often execution times and events are not easily countable or measurable due to a lack of functionality in these systems. Execution time monitoring is also relevant in terms of reacting to internal and external events dynamically.**

Especially for systems using multiple tasks with internal or external resource dependencies, this is a major discipline. Another problem is that measurements during the development process are often done by interfering the system as a whole. This method leads to biases in the measurement results, because the finalized system gets deployed without these interfering functionalities and can therefore work more efficiently than the development system.

The scope of the present work is to develop a module in a hardware description language (HDL) which is able to measure execution times and events task-aware and unaware without interfering the system. The measurements of this module must be handed to the programmer through an easy accessible interface. The main focuses of the project are the scalability, platform independency concerning processor and operating system (OS), as well as easy extendibility. Also, reusability of counters during runtime is included in this work.

***Keywords*-Embedded systems, field programmable gate array, performance monitoring unit, hardware/software codesign**

I. Introduction

Determination and monitoring of execution times and events in an embedded system is getting more important nowadays. Most modern processors include a so-called Performance Monitoring Unit (PMU) to measure the performance of a system and be able to react accordingly. Intel [1] and ARM [2] define their own functionalities in their developer's manuals. The present work introduces these possibilities into the RISC-V [3] architecture, as used by the *mosartMCU* project [4]. This Instruction Set Architecture (ISA) was defined by the University of California, Berkeley, and was implemented by several different organizations and companies (e.g., [5], [6], [7] and [8]) in different hardware description languages (HDLs) as well as discrete processors. The ISA already defines a minimal set of performance monitoring functions, but none of them are included in any implementation yet. Furthermore, the specification [9] only contains simple counters for events and has no awareness of the system itself, as they are only triggerable by events within the processor.

The scope of this paper is to implement a PMU within a RISC-V processor that is aware of the software and hardware system and can therefore also access and measure tasks, events and other structures. This means that it has to know the software it is addressed and used by as well as the hardware system it is surrounded by. The main focus hereby lays on profiling tasks within a multitask software system in multiple ways by hardware.

Furthermore, the developed unit has to be scalable as far as number of hardware counters and configuration is concerned, easily extensible when new functionality like counter types are added as well as platform independent. It works as an external observer and does not interfere the system at all. Additionally, the PMU is capable of assigning hardware counters to more than one purpose without losing measure values. Due to this fact, buffering of counter values is important as well.

The paper is organized as follows. Section II shows existing works on performance monitoring counters. Section III illustrates the present structure of the processor. The following Section IV introduces the hardware structure of the newly developed PMU whereas Section V deals with the corresponding software interface. Section VI shows the experimental setup and provides measurements and investigations. Finally, Section VII discusses the developed PMU and Section VIII concludes the paper.

II. Related Work

Current high-end processors like Intel- or ARM-based include a proprietary PMU. Mostly, these units are simple counters which increment when a certain event occurs. They are configurable through specific registers. Intel calls them Model Specific Registers (MSRs) [1] and ARM reserves a certain block in its memory mapped area [2]. Both of them have in common that they only measure events, not execution times of tasks or other software constructs and they are only usable with a certain processor [10] [11].

PMUs for embedded system processors are not so wide spread. There are a few implementations for soft-cores that

measure energy consumption or the overall performance of the system. Examples are the scalable Event Monitoring Unit for a multi-processor system [12] or the PMU for a LEON3 multi-core system [13].

All these systems are mainly built to measure events [14]. There are only a few methods to monitor tasks in hardware and those systems need a lot of hardware resources. Another disadvantage is that hardware counters are reserved once for a certain purpose and cannot be reused for other measurements without stopping the current assignment.

III. Present Processor Hardware and Architecture

The processor used in this work is a V-scale[1] RISC-V soft-core implemented in Verilog. It has a three-staged pipeline and follows the RV32IM specification [3] [15]. Hence, it supports a basic 32-bit integer unit alongside a multiplier/divider unit for integer values.

The main module in the hardware description is the pipeline, where all other modules like the control unit or the register file are connected, shown in Fig. 1. After analyzing the code of the core, the position determined best for the new PMU module is within the pipeline's hardware structure. This is due to the fact that all possible information can be gathered here without interfering the system itself. It is also possible to access the system memory from within this structure.

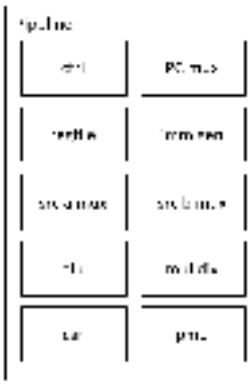


Figure 1. The pipeline of the V-scale with its modules.

To provide access to the counters of the unit, the Control and Status Registers (CSRs) are chosen. Thus, the user can read and write configuration or counter values directly with software structures provided by the RISC-V ISA. As far as platform independency is concerned, this interface can be easily exchanged for every architecture wanted. The further implementation as well as all the requirements of the unit itself will be illustrated in the following sections.

[1]https://github.com/ucb-bar/vscale

IV. PMU Hardware Structure

This hardware module contains features like several configurable counter registers, triggerable by events like change of the task pointer or the program counter, interrupts or external events. Hence, there must be different kinds of counters for global usage, task-aware global counters as well as counters belonging to a certain task. The PMU also must be configurable in terms of the number of counters itself and the number of configuration registers per counter. As far as task-aware counters are concerned, it must be possible to buffer counter values to the RAM at a task switch without delay. As the system clock of a RISC-V processor uses a 64-bit register, the same approach is taken within this work. This is due to the fact that overflows cannot be expected. A permanently incrementing 64-bit counter at 50 MHz clock reaches its maximum after approximately 12,000 years. Configuration registers however use the integer bit length of the processor itself, as the native integer length is the easiest to handle. In the present work, this bit length is 32 bit.

As stated before, the PMU is integrated directly into the pipeline module for several reasons. Due to the defined requirements, it has to have access to a certain amount of information from the system as well as a connection to the RAM. The memory link is needed to buffer values directly to the RAM and be able to reuse values later on. Furthermore, communication with the CSR file must be granted. The architecture of the PMU within the present hardware structure is shown in Fig. 2.

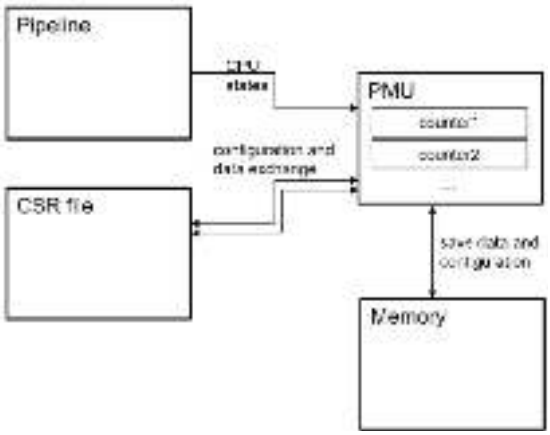


Figure 2. A rough scheme of the PMU and its connected modules in the hardware system. Connections not related with the PMU are not shown and arrows show data flow directions.

The following sections deals with a PMU of four counters with two configuration registers per counter. Regardless of this fact, the module stays scalable for whatever configuration required. At first, the configuration of the PMU is pointed out, followed by the combinatorial logic of a single

counter unit. Lastly, the buffering and reloading of values to and from the RAM is explained.

A. Configuration and registers of the PMU

The module itself consists of a main configuration register of 32-bit, in which every counter's type is defined. Those counter types are represented as 8-bit value and define how and when a counter has to measure. Additionally, every counter has its own two 32-bit configuration registers, in which further configurations to the certain type of counter can be applied. This means an overall number of nine configuration registers. Fig. 3 shows the configuration register's architecture.

A single counter value however consists of two discrete 32-bit registers, which leads to eight registers for this hardware constellation.

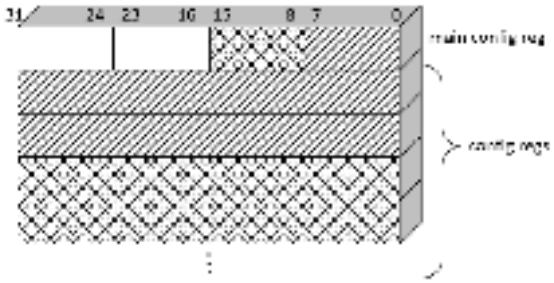


Figure 3. The main configuration register and the corresponding configuration registers of two counters.

Currently, there are three different groups of counter types:

- **Global counters**, measuring execution times and events independent of the current task. This configuration is globally valid and does not need to be buffered. An example in this case is a counter which records the time, a single task is running.
- **Task-aware counters**, measuring execution times and events for every running task. Therefore, the counter values are different from task to task. Hence, the counter values must be buffered at every task switch to guarantee valid values for each task. However, the configuration is persistent (e.g., a single counter that measures every task's individual execution time and buffers the values on task switches). Its configuration can be set once and is globally valid.
- **Counter belonging to a task**, measuring differently for every task. In this case, counter values as well as configuration values have to be buffered due to the fact that configuration is different from task to task. Here, a counter that measures if the program counter of a task is within a certain area is an example. The difference is that configuration has to be set within the task itself for every individual task and therefore must be saved and restored on each task switch.

The configuration registers for the counters itself are used to apply more configuration for the according type of counter if needed. For example, if a counter is configured to measure between two program counter values, these two addresses must be written into the counter's two configuration registers.

To access the registers mentioned above as well as the counter values, they all are mapped to CSR addresses. Hence, they are accessible via CSR operations defined in the ISA [3]. The read and write operations are passed from the CSR file to the PMU as soon as PMU-reserved addresses are recognized by the CSR file. Due to the fact that there are different address ranges defined for standard/non-standard read-only and read/write access within the CSRs [9], proper ranges must be chosen. To set configuration values, read/write addresses of the non-standard range, precisely these starting with 0x8F0, are defined. The counter values however must not be changed by the user. Furthermore, there is a certain address range predefined for hardware performance monitoring counters, starting with address 0xC03 for low-bytes and 0xC83 for high-bytes, respectively.

The module's access address ranges and logic scale with the configuration of the PMU. To ensure this functionality, the whole addressing is calculated by means of the hardware configuration's number of counters and registers and is handled by the module itself with separate address logic blocks for read and write rather than by the present CSR file.

B. Combinatorial logic of the counters

After proper configuration of a counter, its actual counting logic can be explained. Therefore, a single counter unit consists of a byte within the main configuration register, two dedicated configuration registers, a counter register as well as a combinatorial logic block to enable the current incrementation of the counter register. This circumstance is illustrated in Fig. 4.

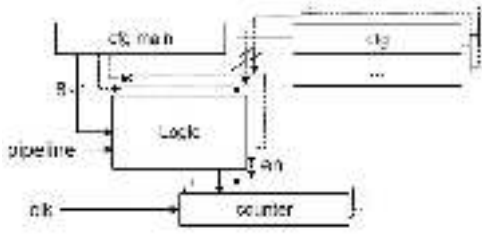


Figure 4. A simplified schematic of the combinatorial logic to enable a counter.

The logic block generates an enable flag to enable the sequential clock-triggered increment of the counter value.

Inputs for the logic block are the configuration registers as well as the current processor information gathered by inputs directly from the pipeline module. As this happens only by reading from certain wires, the system is not interfered by this approach.

The logic block itself contains sub-blocks for every different counter type and its own constraints. To implement an extensible structure, a logic of combinatorial blocks with a final or-gate of multiple inputs is chosen, as shown in Fig. 5.

Figure 5. Generalized logic to determine a counter's enable flag.

It illustrates the part of logic for generating a counter's enable flag (*cnt_en[i]*) with *PMU_CNT_TYPE_0* as counter type in its main configuration (*cfg[0]*). Therefore, this counter measures if the current task pointer of the pipeline (*tp*) has the configured value in *pmu_cfg[i][0]*, which is the first configuration register of the current counter *i*. Further counter types must be added at the inputs of the final or-gate at the end.

In this work, different usable counter types divided by the three groups are:

- For **global** counters, there are: a counter which measures all the time (i.e. system clock), one for the processor user mode execution time, a single task execution time counter, an overall task execution time counter, a counter for all but a specific task, an overall interrupt counter, a counter for missed interrupts by a specific task and an overall external port pattern match counter.
- The group of **task-aware** counters, which save all counter values separately for every task in memory, contains: an overall task execution time counter, a task interrupt counter and an external port pattern match counter.
- There is only one counter of which counter and configuration values must be buffered. The task part counter that measures if the program counter is between two values, has another special feature. In order to start the counter at a certain program counter value and end it at another one, the PMU uses the least significant bit of the configuration registers to mark whether or not the program counter already reached the configured value. This sequential logic must be executed due to the fact that it can be possible that a task is preempted while the counter is incrementing. To carry on measuring when the task is continued, this flag must be set and reset accordingly.

C. Buffering of counter and configuration values

To ensure the explained functionality, buffering of counter and configuration values is compulsory. Hence, there has to be a memory interface within the logic and the hardware has to be aware of the operating system's (OS) task concept. In particular, the current task pointer must be known in order to integrate task awareness into the unit. On a task switch, all the register's values must be saved somewhere belonging to the suspended task in the RAM and the buffered values of the new task must be loaded into the registers, respectively.

One of the main problems concerning buffering is that memory access collisions must be avoided. As the present hardware structure uses a dual port RAM [4], one of the two memory ports is reserved for PMU purposes. As the memory allows access to a memory cell from the two channels simultaneously, consistent values are guaranteed. A memory like this is also placed within the used Artix-7 Field Programmable Gate Array (FPGA) [16].

The current task pointer of the OS must be stored in the *tp* register [3] of the processor. This allows the PMU to detect task switches and initiate the memory swap process on its own by a state machine. As long as the unit's memory access cycle is faster than the task context switch of the software system, valid values in the registers can be ensured. The data structure of a task's control block (TCB) within the OS is shown in Fig. 6.

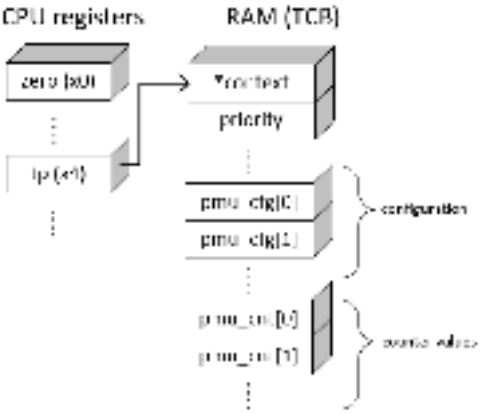


Figure 6. TCB in the software system.

With this knowledge, the module is able to calculate the memory addresses by adding certain offsets to the task pointer stored in the CPU register *tp*. To ensure scalability, these calculations take the hardware configuration into account. The state machine handles the memory swap by swapping out all the current register values into the memory

cells of the to be suspended task and reading data from the new task's TCB into the PMU registers. Task-aware and task belonging measurements must be stopped during this process. In order not to lose measuring cycles during this work, every counter type must have its own compensation. It is also important to differentiate between counter types, as far as overwriting of configuration and counter values are concerned. A task-aware counter's configuration must not be overwritten during this activity, for example. However, the swap from the registers into the TCB is made, no matter which counter type is configured. This is to ensure consistent data in the memory, in order for the OS to use valid data of a suspended task from within another task to profile it.

V. Software Interface

To use the functionality mentioned above in an embedded system, a simple OS is needed. For this purpose, the *mosartMCU*-OS is used. This OS is a minimalistic system consisting of a simple task and resource management system for a RISC-V architecture. Furthermore, there are several syscalls to start the OS itself, create tasks and use the system's resources and events.

A. Hardware/Software interface

The implemented scheduler manages each task as far as starting and preempting is concerned. Therefore, the task pointer of the instantaneous running task is stored directly in the register *tp* of the CPU. By this approach, as mentioned, the simple interface between hardware and software is created. This register must be maintained by the scheduler at all time to ensure proper performance measurements. In order to signal a task switch as soon as possible for the hardware, the new value must be set into the register at the very beginning of all actions related to it. Hence, the PMU starts its swapping process at the same time the OS executes its register context switch.

Additionally, the system's data structure must be equivalent to the one used in the buffering process of the PMU to ensure consistent and valid data storage. Due to this, the OS must be well aware of the hardware configuration of the unit, as far as number of counters and configuration registers is concerned.

B. Register access

The present system knows two types of register access:

- read counter words
- read/write configuration values

As the registers are represented by certain addresses within the CSR file, they can also be accessed via simple instructions defined by the RISC-V ISA [3]. Due to the fact that the addresses of the counter values are aligned with the RISC-V definition for hardware performance counters [9], the keywords `hpmcounter`N and `hpmcounter`N`h` can be used where N is defined within the interval $[3, 31]$.

The user must be aware that the PMU starts measuring at the time instant the main configuration register is set to a valid value. To stop and reset the unit, 0x0000 must be set. Additionally, configuration of counters belonging to a task must be done within the context of the corresponding task itself to ensure proper measurements.

C. Task profiling

All the accesses to registers provide the current values in the counter registers of the instantaneous executed task. As these values get buffered into the RAM at every task switch, they are also stored within the TCB. This on the contrary provides value snapshots of every past task execution. These can be used to easily profile all tasks from within another dedicated task or the OS itself in order to gain information concerning execution times or events. Based on these results, new schedules or the schedulability can be calculated for instance [17].

VI. Measurements and Investigations

This section deals with measurements and investigations concerning the resource usage in the Artix-7 FPGA with different hardware configurations on the one hand and validations of simulations and measurements of different test cases on a certain system on the other hand. All the illustrated studies use one setup explained as follows.

A. Experimental Setup

The setup for experimental purposes consists of a Nexys-4[2] development board of Digilent based on a Artix 7[3] FPGA of Xilinx. To measure data and validate the results, a digital oscilloscope – the PicoScope 2205 MSO [18] – is used. It has two analog alongside 16 digital channels, which can all be used simultaneously. To display results, it must be connected to a PC via USB. It has a maximum input frequency of 100MHz.

For measuring purposes, the PMU is extended by a special 16-bit measurement port that is wired directly to an output port of the development board. On this port, the oscilloscope's digital inputs are connected. It can be used to display signals like enable flags of certain counters as well as values currently stored in registers. Hence, each test case can implement its own measurement port configuration to achieve exact results.

The hardware system therefore consists of a *mosartMCU* based on a V-scale RISC-V core, a RAM, a ROM, several peripherals like GPIO ports and UART as well as the designed PMU module. It operates at a clock of 50MHz.

[2]https://reference.digilentinc.com/reference/programmable-logic/nexys-4/start

[3]https://www.xilinx.com/products/silicon-devices/fpga/artix-7.html

B. Different hardware configurations

Our reference model is a PMU with four counters and two configuration registers per counter (4x2). Taking the main configuration register into account, this leads to 17 32-bit registers overall. To synthesize a full RISC-V core with peripherals and a PMU, Xilinx Vivado 2016.2[4] is used. The first column of Table I shows the primitive statistics, net boundary statistics and clock report of the reference model.

By varying the hardware configurations by means of increasing the number of counters and/or the number of configuration registers per counter, the results shown in Table I are achieved. As values for MULT and OTHERS remain unchanged, they are not shown or discussed in the following section.

Table I
RESOURCE USAGE IN DIFFERENT HARDWARE CONFIGURATIONS.

	4x2	*4x4*	*4x8*	*8x2*	*16x2*
FLOP_LATCH	682	941	1453	1235	3141
LUT	1955	2380	2735	3994	9865
MUXFX	40	163	328	317	1095
CARRY	198	197	208	371	840
NETS	513	515	502	661	1473
CLK Inst	684	943	1455	1237	3113

When firstly looking at the change of values for increasing the number of configuration registers at a constant number of counters, Fig. 7 shows the trend. For this, the configurations 4x2, 4x4 and 4x8 are considered. The first digit in these configurations marks the number of counting registers, the second the number of configuration registers per counter (e.g., 4x2 is a PMU with four counter units and two configuration registers per counter). It shows a linear growth for all values except NETS and CARRY. This can be explained by the fact that configuration registers are not wired that complicatedly within the module and are optimized by the synthesis tool. The linear growth of the other values is due to the increase of registers itself. More Lookup tables (LUTs) and multiplexers (MUXFXs) are required to address the new added configuration registers. The number of flip flops (FLOP_LATCH) approximately follows the amount of clock instances (CLK Inst) for the reason that every latch needs its own clock input.

Secondly, the number of counters is increased by constant number of configuration registers per counter. This circumstance is shown in Fig. 8. In this case, the number of LUTs grows by a square function. A similar growth can be seen with flip flops and clock instances. This can be explained due to the fact that by increasing the number of counters, the overall number of configuration registers rises. Further, the addressing of the registers gets more complicated which can be seen at the increase of multiplexers.

[4]https://www.xilinx.com/products/design-tools/vivado.html

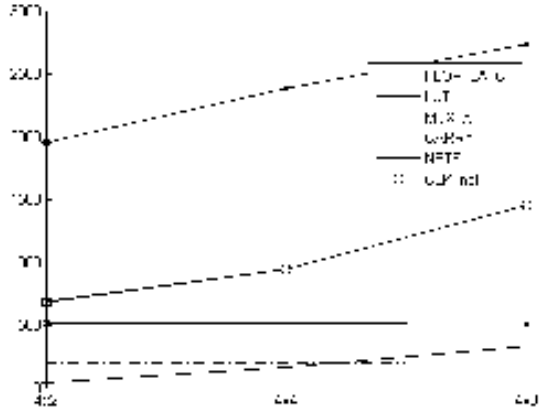


Figure 7. Growth of certain parameters when varying the number of configuration registers.

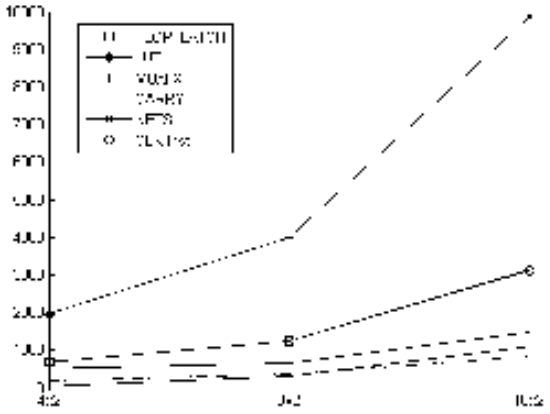


Figure 8. Growth of certain parameters when varying the number of counters.

C. Test cases

All the following test cases build on a 4x2 configured PMU and a simple *mosartMCU*-OS setup. As the system measures clock cycles in most times, the actual time can be calculated by

$$t = Cnt \cdot \frac{1}{f_{CPU}} = Cnt \cdot \frac{1}{50MHz}, \tag{1}$$

where Cnt is the value within a counter register and f_{CPU} is the clock frequency of 50MHz. The resulting time t presents time in seconds. This means, that the expected maximum deviation ε is

$$\varepsilon = \frac{1}{50MHz} = 20ns. \tag{2}$$

The test cases are divided into a simulation part showing the expected result followed by a measurement with the oscilloscope. All simulations are carried out with Vivado's built in simulation tool.

Table II
COUNTER SIMULATION AND MEASUREMENT RESULTS.

Counter type	*Sim result*	*Time*	*Measurement*
(1)	6736	$134.72\mu s$	$134.7\mu s$
(2)	14	$280ns$	$280.4ns$
(3)	8183	$163.66\mu s$	$163.8\mu s$
(4)	8815	$176.30\mu s$	$176.38\mu s$
(5)	709	$14.18\mu s$	$14.13\mu s$
(6)	8218	$164.36\mu s$	$164.4\mu s$
(7)	15	$300ns$	$309.7ns$
(8)	25	$500ns$	$509.7ns$
(9)	9411	$188.22\mu s$	$188.6\mu s$
(10)	946	$18.92\mu s$	$19.099\mu s$
(11)	10830	$216.6\mu s$	$216.8\mu s$
(12)	9411	$188.22\mu s$	$188.6\mu s$

1) Global counters: The first test case illustrates the usage of global counters. In fact, the first counter (1) of the setup measures all the time, whereas the second (2) measures all the user mode time of the processor, the third (3) is set to measure the overall task time and the fourth (4) the overall task time except for the idle task. Table II lists the simulation and measurement results of this test case at different time instances.

2) Simple task counters: The next test cases deal with a global counter for a single task (5), one for the overall task-aware task time (6) as well as a counter belonging to a task, which measures areas between two program counter values. Two results at different time instances are shown as (7) and (8) in the results table II. The results in (7) and (8) show the usage of one hardware counter in two different tasks. The values in the registers are only valid for the instantaneous running task.

3) Task-aware test case: At last, this test case shows a global overall task time counter, two global task time counters for single tasks, configured to two different tasks and a task-aware task time counter.

At the first time instant at which the global overall task time counter results in the value (9), the single task counter for the first task as well as the task-aware counter yield in value (10), which proofs correctness.

At the second time instant, the other task is investigated. The overall task time counter has the value (11) and the corresponding values for the single task counter for the second task and the task-aware counter again deliver the same value (12).

All these results show that all the global counters work properly within certain measurement uncertainties and result in the same values within ε as expected through simulation. Also, the final counting values in the registers match the expected values.

VII. DISCUSSION

The developed PMU has some main advantages to other present works. Compared to approaches which measure task constructs within the OS, this hardware solution is able to measure execution times without overhead or lost cycles. Software approaches slow down the whole system and can yield falsified results due to interferences. Additionally, the developed module can stay within the hardware structure rather than be removed after optimizing software structures.

The module is also able to measure a lot of different things with very few hardware counters. This is due to the fact, that a single counter can run a certain configuration for each task.

As far as other hardware solutions [12] are concerned, this module's scalability leads to minimal hardware increase, as hardware counters can be reused rather than having to add new ones to the system. It can also be used in multi core systems, as every core has its own PMU. In this case, only measurements for the certain core can be made, as the module is connected directly to it. Future aims towards an additional overall unit to include more information concerning multi core operations [13].

VIII. CONCLUSION

The present paper shows a new approach on how to measure execution times and events of an embedded system by integrating a PMU by means of hardware/software co-design approaches. It is constructed to be aware of the hardware and software system it is embedded in. Furthermore, the unit is able to reuse counter elements at runtime to minimize hardware complexity. It is also reconfigurable, scalable and easily portable to other hardware systems, as it is implemented in a HDL and has a simple and easily adjusted interface to the CPU itself.

Moreover, it could be shown that re-used task-aware counters yield the same results as multiple dedicated counters each reserved for a single task. At the same time hardware usage is minimized through this approach. Due to this fact, a more detailed performance profile can be created, as task awareness is built in. It is not only possible to consider the system as a whole, but also profile certain parts of it.

As it is non-intrusive, it can be used for worst case execution time analysis at an early development stage. Moreover, if deployed to an FPGA-based embedded system, even its software works alongside the system and therefore no altering in any execution state is expected.

REFERENCES

[1] Intel, "Intel Microarchitecture Codename Nehalem Performance Monitoring Unit Programming Guide," Intel, Tech. Rep. 253669-061US, Dec. 2010.

[2] ARM, "Cortex-A5 Technical Reference Manual," ARM, Tech. Rep., 2016. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0433c/DDI0433C_cortex_a5_trm.pdf

[3] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic, "The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.1," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-118, May 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-118.html

[4] F. Mauroner and M. Baunach, "Event based and Priority aware IRQ handling for Multi-Tasking Environments," 2017, Euromicro DSD (accepted).

[5] VectorBlox, "ORCA FPGA-Optimized RISC-V," 2016, http://riscv.org/wp-content/uploads/2016/01/Wed1200-2016-01-05-VectorBlox-ORCA-RISC-V-DEMO.pdf.

[6] A. Traber, F. Zaruba, S. Stucki, A. Pullini, G. Haugou, E. Flamand, F. K. Grkayank, and L. Benini, "PULPino: A small single-core RISC-V SoC," 2015, http://iis-projects.ee.ethz.ch/images/d/d0/Pulpino_poster_riscv2015.pdf.

[7] A. Traber, "RI5CY Core: Datasheet," ETH Zrich, Tech. Rep., Feb. 2016. [Online]. Available: http://www.pulp-platform.org/wp-content/uploads/2016/02/datasheet_RI5CY.pdf

[8] Y. Lee, A. Ou, and A. Magyar, "Z-scale: Tiny 32-bit RISC-V Systems," 2015, https://riscv.org/wp-content/uploads/2015/06/riscv-zscale-workshop-june2015.pdf.

[9] A. Waterman, Y. Lee, R. Avizienis, D. A. Patterson, and K. Asanovic, "The RISC-V Instruction Set Manual Volume II: Privileged Architecture Version 1.9.1," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-161, Nov 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-161.html

[10] D. Patil, P. Kharat, and A. K. Gupta, "Study of Performance Counters and Profiling Tools to Monitor Performance of Application," *21st IRF International Conference*, 2015.

[11] A. Singh, A. Buchke, and Y.-H. Lee, "A Study of Performance Monitoring Unit, perf and perf_events subsystem," -, 2012.

[12] J. A. Ambrose, V. Cassisi, D. Murphy, T. Li, D. Jayasinghe, and S. Parameswaran, "Scalable Performance Monitoring of Application Specific Multiprocessor Systems-on-Chip," *2013 IEEE 8th International Conference on Industrial and Information Systems*, Aug. 2013.

[13] N. Ho, P. Kaufmann, and M. Platzner, "A hardware/software infrastructure for performance monitoring on LEON3 multicore platforms," *24th International Conference on Field Programmable Logic and Applications*, 2014.

[14] B. Sprunt, "The Basics of Performance-Monitoring Hardware," *IEEE Micro (Volume: 22, Issue: 4, Jul/Aug 2002)*, 2002.

[15] T. Chen and D. A. Patterson, "RISC-V Geneology," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-6, Jan 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-6.html

[16] Xilinx, *7 Series FPGAs Memory Resources*, ug473 (v1.12) ed., Xilinx, Sep. 2016. [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf

[17] F. Dittman and S. Frank, "Hard real-time reconfiguration port scheduling," *Design, Automation and Test in Europe Conference and Exhibition*, Apr. 2007.

[18] *PicoScope 2205 MSO Mixed Signal Oscilloscope*, Pico Technology, 2016. [Online]. Available: https://www.picotech.com/download/datasheets/PicoScope2205MSODatasheet-en.pdf

7.6 R-II: A Hardware/Software Concept for Partial Logic Updates of Embedded Soft Processors at Runtime

◇ Authors: Tobias Scheipel, Peter Brungs, and Marcel Baunach
◇ Date: September, 2021
◇ DOI: 10.1109/DSD53832.2021.00040
◇ Reference in bibliography: [154]
◇ Presented by myself at the 24th Euromicro Conference on Digital Systems Design (DSD'21), Palermo, Italy.
◇ Published in the proceedings of DSD'21 (IEEE).

Summary The proposed concept of this paper makes it possible to update the Instruction Set Architecture (ISA) of an Microcontroller Unit (MCU) at runtime. It includes a pipeline design that allows to exploit partial reconfiguration and an Operating System (OS) design to deal with the changing ISA. As long as an instruction is not natively supported, the OS will automatically emulate it, but it can also add it to the pipeline for native support.

Author's Contribution I am the main author of this paper, and I have entirely envisioned, designed, and implemented the concept of the paper myself. Peter Brungs supported the implementation of the concept. He and Marcel Baunach supported me through discussions and in the composition of the paper.

Copyright ©2021 by IEEE. The version included in this thesis is reprinted, with permission, from the published version stated above, which can be accessed in the IEEE Xplore Digital Library:
https://ieeexplore.ieee.org

A Hardware/Software Concept for Partial Logic Updates of Embedded Soft Processors at Runtime

Tobias Scheipel
Institute of Technical Informatics
Graz University of Technology
Graz, Austria
tobias.scheipel@tugraz.at

Peter Brungs
Chair of Computer Science V
University of Würzburg
Würzburg, Germany
mail@peter-brungs.eu

Marcel Baunach
Institute of Technical Informatics
Graz University of Technology
Graz, Austria
baunach@tugraz.at

***Abstract*—Embedded systems are built from various hardware components and execute software on one or more microcontroller units (MCU). These MCUs usually contain a fixed integrated circuit, thus disallowing modifications to their logic at runtime. While this keeps the instruction set architecture (ISA) fixed as well, it leaves the software as the only flexible part in the system. But what if the MCU logic could be easily changed at runtime in order to fix bugs or if the ISA could be extended on-the-fly in order to introduce application-specific instructions and features on demand?**

This work demonstrates a concept for introducing more hardware flexibility through application-specific MCU modifications. Therefore, the MCU is implemented as a soft core on a field-programmable gate array (FPGA) and we reconfigure its logic with support of the operating system (OS) running on it. The reconfiguration happens on-the-fly, so no interruption of the application code or even a system restart is required. Therefore, (i) the MCU pipeline is specially designed for extensibility by new instructions, and (ii) the FPGA is selected to support partial self-reconfiguration of its logic cells at runtime. As long as an instruction is not yet part of the ISA, the OS supports its emulation to provide a consistent interface for applications. Apart, no special compiler support is required, but the application must provide either the emulation code or a hardware description for adding the required logic. For a proof of concept, we use a RISC-V based MCU on a Xilinx Artix-7 FPGA and for evaluating the general benefit of our approach we use an algorithm that is costly when executed with the original ISA but fast with application-specific instructions added at runtime. The experimental evaluation also shows that the on-the-fly hardware update does not disrupt or compromise the software execution flow.

***Index Terms*—partial reconfiguration, embedded systems, reconfigurable logic, field-programmable gate array, processor architecture, operating system, hardware software co-design**

I. Introduction and Motivation

Embedded systems are commonly constructed from two fundamentally different parts: a flexible and easily changeable software layer on top of a static and fixed hardware layer. Even though the hardware logic can be changed at early design stages of the next revision of the Microcontroller Unit (MCU), this is a costly and time-consuming process and most processor cores cannot be changed after production. In case of hardware bugs or changing requirements, workarounds and new features must then be implemented in software. In contrast, new software can easily be flashed into the MCU memory and thus be altered at will. However, in the era of Internet of Things (IoT) where embedded devices need to guarantee continuous dependability [6], we expect the traditional system design with *fixed* computing hardware to become insufficient – this is because we can no longer afford to ignore revealed hardware problems [11], [12]. When SHA-1 [7] became deprecated in 2011, many hardware accelerations in, e.g., Intel processors became obsolete [9] and needed replacement since updates were not possible. Similar situations are to be expected in the automotive domain, where embedded devices must stay operational for years and decades. Therefore, future processor cores will need to support logic updates after production, e.g., to extend the system lifetime by adapting to new software requirements, changing legal regulations [18] , or simply by fixing bugs. If hardware can be updated similar to software, this would also add to sustainable system design, since devices can be maintained for much longer, reducing environmental pollution.

Field-programmable Gate Arrays (FPGAs) give embedded engineers the freedom to introduce their own logic and even custom processors as soft cores into their systems. Even though this option has been around for years, it is rarely used due to higher costs, energy and space demands, etc. compared to Application-specific Integrated Circuits (ASICs) or commercial off-the-shelf (COTS) MCUs. However, the option can significantly increase the flexibility and performance of the overall system through careful hardware/software co-design. But even for most FPGA-based systems, the logic is still designed once during development and stays unaltered at runtime. Similarly, embedded software is often also designed and compiled statically for the specific use case, since dynamic software composition is a common source of errors. To achieve the intended long-term operation and maintenance, both processors and software and must be designed to add and use new features at runtime. The software must be prepared to handle changing processor architectures, where parts of the application can be optimized by replacing slow, software-implemented algorithms by fast, hardware-accelerated substitutes in the updated processor core. Consequently, we propose a concept for partial updates of embedded soft MCUs at runtime, so that software-like flexibility can be reached in hardware as well. Our approach allows to add custom instructions directly to the pipeline of the processor core on which

the software itself is running. The required update of the logic cells may come with the application binary (or a driver, etc.) and happens on-the-fly without any system reset. The software can still be compiled with normal, unaltered compilers and the Operating System (OS) takes care of the extensions introduced to the system.

The paper is structured as follows: Section II summarizes related work in the field of dynamic MCU architectures, OS support for dynamic Instruction Set Architectures (ISAs), and partial reconfiguration of FPGAs at runtime. In Section III, our system architecture is sketched, whereas its usage is shown and evaluated by utilizing a RISC-V based MCU in Section IV. Section V concludes the paper with a summary and an outlook to future research.

II. Related Work

This Section summarizes existing work that contributed, influenced, and motivated the proposed concept.

Regarding **flexible MCU architectures**, we refer to the RISC-V ISA [25] as this architecture is designed to be highly extensible. In our case, these extensions shall not only allow to change on-chip peripherals, but also the MCU core(s). Some RISC-V implementations include custom functionality like vector instructions or hardware loops [21], [22], as some use cases demand for such accelerators. While the accelerators were tailored for specific systems, the core does not support a generic way of adding new functionality. Other RISC-V implementations have specific extensions for cryptographic applications [1]. While being highly modular at design time, runtime adaptability is not supported. Further research projects aim at hardware description language to support modular cores [24] in general. Again, only design time changes are considered here. The scope of our work, however, aims at a generic way to extend the instruction set *at runtime* and with minimal modifications to the pipeline, in order to simplify updates and keep the management overhead low.

OS support for flexible hardware and ISAs can be found in multiple projects. An example is the use of special instructions to reduce the energy consumption of certain code patterns in the application at runtime [20]. There is also research towards flexible ISA extensions for embedded FPGAs [14], where a general-purpose processor was coupled with an FPGA to add application-specific instructions. A computational model for reconfigurable computing was introduced for multi-threaded programming in such hybrid processor/FPGA systems [2]. This allows threads to be compiled either for the processor or synthesized for the FPGA with an Application Programming Interface (API) similar to pthreads [15]. An approach to develop hardware coprocessors in a software-like environment based on a Linux OS is presented in [5]. Unlike the related work, our approach aims at systems that can cope with changing ISAs during operation. To do so, our very own embedded OS was extended as explained in the course of this paper.

Reconfigurable computing platforms, combining both topics mentioned before, have also received great attention. A very important work in this regard is the MOLEN polymorphic processor [23], consisting of a general-purpose processor and some reconfigurable hardware that can provide new instructions like a co-processor. Its logic is created at compile time by extracting short code sequences from the application binary and mapping them to synthesized instructions in the co-processor. While the software on the general-purpose processor can use them like regular instructions, the interconnect to the co-processor introduces some overhead. In addition, the concept requires adaptations to the toolchain and prohibits modifications at runtime. As a major difference to this approach, we do not aim at introducing static co-processing units, but dynamically modify the processor core while it is executing other instructions. Apart, our concept uses standard compilers, also simplifying code portability and binary reuse.

PRISM [3] is another, very early concept to develop reconfigurable processors. There, application code is also transformed into a hardware description to generate a co-processor running on an FPGA. Other static approaches like [29] use bus systems and programmable functional units which are attached to the hosting processors in different ways to achieve a flexible instruction set. It is stated, however, that this approach leads to performance decrease compared to a highly-customized functional unit directly in the processor. As our approach directly attaches to the pipeline, we overcome this drawback.

More similar to our envisioned work is the eMIPS [19] processor for workstations, where co-processing modules can be loaded into the pipeline while the system is running: Basic blocks within application code can be implemented as a single instruction and the original code is skipped over in case the corresponding module is loaded. While the eMIPS processor just notifies the OS when a new instruction is loaded, our approach lets the OS decide between loading or emulating an instruction depending on current software requirements. Apart, the eMIPS concept targets workstations for on-demand performance improvements while our concept targets embedded systems for long-term operation and maintenance.

Regarding the **partial reconfiguration of FPGAs at runtime**, the literature is manifold. There are simple partial reconfiguration frameworks [4], where certain algorithms can be accelerated by using a bitstream repository to change logic modules. Further, runtime reconfiguration of certain parts on the FPGA allow multi-processor systems to enable new single instruction/multiple data (SIMD) operations [16]. Also, surveys concerning the performance of partial reconfiguration were carried out [17]. Our work concentrates mainly on partial reconfiguration of the MCU pipeline, where it is important to carefully reconfigure signals or areas in order to avoid undefined system states or behavior.

To briefly recapitulate, our proposed concept differs from the literature in a number of important aspects. Our processor reconfiguration is managed by the OS and happens on-the-fly without stopping the software or resetting the system. No modifications to toolchains or compilers are necessary for our concept to work, and the new instructions are integrated into

```
100     [...]
101     addi  t4, zero, 12
102     cinsi t3, t4,   10
103     [...]
```

Listing 1. Example assembly code for execution on systems with and without `cinsi` instruction.

the processor pipeline instead of building a co-processor or on-chip peripherals. Finally, our proposed work finds its main application in embedded devices, where it aims to significantly improve long-term operation and maintenance.

III. System Architecture and Reconfiguration

The proposed concept consists of four parts explained next: *(A.)* application code being executed on top of *(B.)* an embedded OS with integrated features to support changing ISAs in *(C.)* a soft core processor with pipeline adaptations for on-the-fly modifications, and *(D.)* the partial reconfiguration of an FPGA at runtime. Even though our implementation and evaluation is based on an existing system framework (see Section IV-A), the general concept can be applied to any system composed of a reconfigurable computing platform and an OS managing the modifications.

A. Application Software

Algorithms in many use cases can benefit from application-specific hardware acceleration that may not be available in all target processors. While compilers will still generate binaries (from, e.g., C code) for the target architecture, we take a different approach by simply calling an assumed processor instruction that shall provide the required acceleration. An example is shown in Listing 1, line 102: The `cinsi` instruction might or might not be natively supported by the target architecture. The different ways of executing the instruction are shown in Fig. 1: If not yet supported, the MCU will raise an *illegal instruction exception*. Then, the functionality is (a) emulated by a code sequence (e.g., a C function) which is injected into the execution flow by the OS. Once available (b), the instruction will execute like any other instruction of the original ISA. In any case, both the emulating function or the required logic come with the application code, and the code execution is transparent to the application. However, while it yields the same result, the timing behavior might differ.

In this paper, we execute Listing 1 on a RISC-V architecture. Within its original ISA [25], [26], the `cinsi` instruction (meaning `c`ustom `ins`truction `i`mmediate) in line 102 is unknown, so we can explain our concept as well as the behavior of hardware and software.

In order to provide a consistent API to application developers, the OS offers a functionality to register emulation code for a custom instruction. Example C code is shown in Listing 2. Line 13 registers the emulated instruction. Parameter `cinsi` is used for naming an internal OS management data structure, `0x1B` is the opcode of the new instruction, and `cinsi_emu` is a pointer to the emulating function in line 10. After

```
10  void cinsi_emu( ... ) {
11   [... code for emulating cinsi ...]
12  }
13  OS_DECLARE_EMULATED_INSTRUCTION(
        cinsi, 0x1B, cinsi_emu);
14
15  OS_TASKENTRY(task1) {
16   [...]
17   asm(".insn i 0x1B, 0, r1, r2, 0xF");
18   [...]
19  }
```

Listing 2. Example C code using the custom `cinsi` instruction within our OS framework.

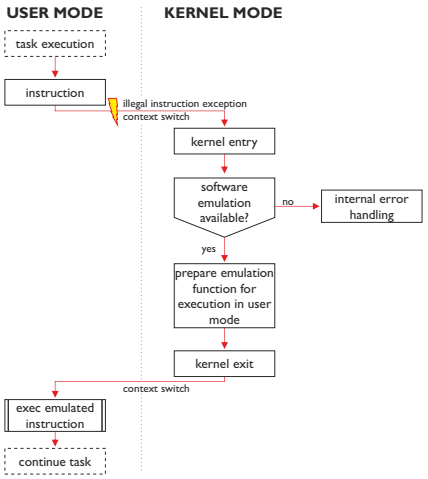
registering the instruction, a task can use the instruction as sketched in line 17 by generating the required encoding for the binary. This line in the C code corresponds to line 102 in the assembly code in Listing 1, and the syntax represents a custom instruction call within a RISC-V compiler. With this approach, standard RISC-V compilers can be used without modification. The new functionality is only introduced within the system's application code. Therefore, code execution on non-altered standard-compliant RISC-V cores remains possible, as illegal instruction exceptions must be supported by every type of core [26], and these are handled within the OS.

B. Operating System

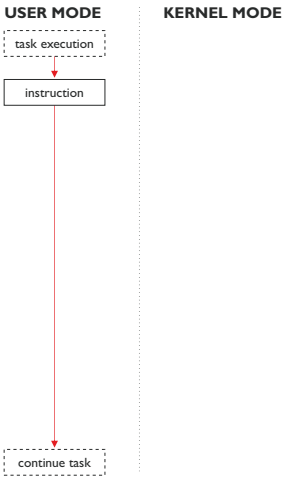
In order to support custom instructions of the application code, the OS must first be able to execute the emulation in case a called instruction is not yet available in the ISA. Second, the OS must be able to modify the underlying hardware by adding the logic for new instructions.[1] Both cases lead to different effects. In case an instruction is not available, many processors throw a so-called illegal instruction exception [26]. In our case, our very own *MCSmart*OS [13] is modified to handle this exception, as it most commonly indicates corrupted software binaries. In this concept, we further extend the exception handler to execute the registered emulation code as provided by the application. For the application code in Listing 1, we thus follow two different software execution flows:

(a) In order to **emulate** `cinsi`, *MCSmart*OS follows the execution flow depicted in Fig. 1a: Right after fetching and decoding the instruction, the processor throws an illegal instruction exception, as it is unaware of the `cinsi` opcode. The OS catches this exception for consistent handling across all applications. Therefore, the application is interrupted and kernel mode is entered. Then, the OS processes the binary encoding of the current instruction and checks its internal data structures for a corresponding emulated instruction that was registered before. In case the look-up fails, *MCSmart*OS treats the instruction as illegal and internal error handling steps in (e.g., terminate the application). In case an emulation for the

[1] Removing the logic if no longer required (e.g., when terminating an application), or replacing it (e.g., for "more important" applications) if no more space is available in the FPGA, is part of our ongoing work about related optimization algorithms in the OS.



(a) Software execution flow in case an unknown or illegal instruction must be emulated by *MCSmart*OS kernel.



(b) Software execution flow in case the instruction is natively supported by the ISA (scaled according to Subfigure 1a). No kernel code involved.

Fig. 1. Comparison of software execution flows in case of (a) unknown or illegal instructions and (b) normal instructions.

instruction is found, the corresponding function is prepared for execution in the calling task's context. After leaving kernel mode and restoring the task context, the emulation is executed as a regular function in user mode and at the task's privilege level. Finally, the task directly continues further code execution.

(b) **Natively** supported instruction in the ISA follow the software execution flow from Fig. 1b: Whenever a previously emulated instruction was added to the pipeline of the processor, the application doesn't need to enter the kernel any longer, but the MCU simply executes the new instruction like any other instruction. The difference to the emulated case is that no kernel overhead is introduced, as the instruction is plainly executed in the processor core of the MCU (cf. Section III-C).

C. Microcontroller Unit

Right below the OS layer of the system, the MCU is located. For now, all software instructions are executed on a single processor core. The soft core features a pipeline with at least Instruction Decode (ID) and Execute (EX) stages, as new instructions are introduced by changing or extending their logic. The structure of our MCU is depicted in Fig. 2. Goal is to separate the static part of the core (containing its original implementation) from the dynamically reconfigurable part (for custom extensions at runtime). Within the area "Custom Logic", new instruction logic can be placed and connected to different stages of the pipeline, as long as the interface between the static and the dynamic part is fixed. Therefore, the pipeline is implemented in a way to avoid undefined states and behavior when reconfigured while working. Starting with a modularization of the pipeline, attachable connections between all the integral modules must be created. Within these connections, relevant signals form the interface to the dynamic

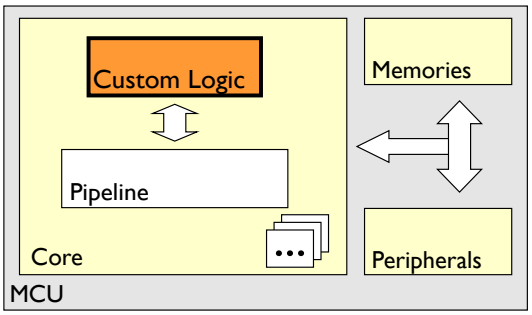


Fig. 2. MCU concept with custom instruction logic attached to the pipeline.

area where custom extensions and instructions can be added. The signals include, e.g., operands, instruction words, function description flags, and internal clock wires. Also, memory access channels can be added. Since these signals provide a docking point for new instructions at runtime, they must be accessible in a well-defined way. Thus, we keep them fixed on the FPGA, as explained in Section III-D. Technical details are given in Section IV-B, where example modules for different use cases are shown as proof-of-concept and evaluation.

D. Partial Reconfiguration at Runtime

Once MCU, OS, and application code operate as explained above, the partial reconfiguration of the FPGA is the final step to provide a new instruction to the software at runtime. Therefore, we program so-called "partial bitstreams" [28] into the FPGA on-the-fly. While the clocks of the FPGA remain active and the MCU keeps running, this ensures that the configuration of the static cells is retained.

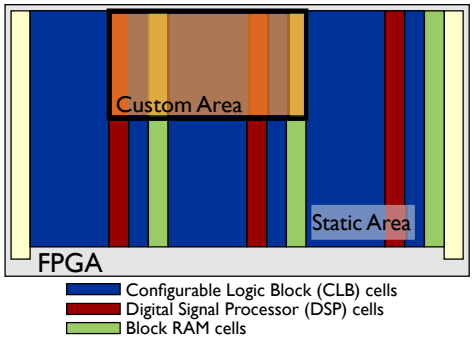


Fig. 3. Floorplan structure of an example FPGA. The Custom Area pre-allocates cells for partial reconfiguration, other areas contain static MCU parts.

In our implementation, *MCSmart*OS keeps track of how often an application uses an emulated instruction and triggers the ISA update in case of expected benefits. For now, the partial bitstreams are then provided externally by a regular PC. It is envisioned, however, to enable self-adaptation of the MCU in the future, as pointed out in Section V. Still, the process does not introduce any delays or interruptions to the running system. While the process happens in parallel to regular system execution, two aspects are relevant to not disturb the operation of the original MCU:

(1.) Placement and interfacing of the custom area: Initially, it is important to mention that the partial reconfiguration process changes all cells in affected areas of the FPGA. Thus, all connections between the cells in the *custom area* are reset and newly routed according to the updated logic description they shall contain. Connections in other areas, especially in the *static area*, remain unchanged. While there is no general recipe on where to place the different areas, this section gives a few hints on how to succeed. As a reference, the floorplan of an FPGA in Fig. 3 distinguishes between different cell types in an FPGA:

- Configurable Logic Blocks (CLBs) containing Lookup Tables (LUTs) and Flip-flops (FFs),
- Digital Signal Processor (DSP) cells, and
- Block RAM (BRAM) cells.

Cells like FFs and BRAMs can store data (sequential logic), while other cells like LUTs and DSPs can process data (combinatorial logic). During reconfiguration, all logic cells in the dynamic area will be put to a new defined state while the ones in the static area must retain their data/configuration under all circumstances. Apart, the interaction between the static and the custom area will only continue if there is a fixed interface between them. This interface is provided by adding fixed LUTs at the edge of the static area. The information about their exact locations is known to the update mechanism to so they can be used as a docking points for new logic in the dynamic area. A resulting limitation to our concept is that – despite the runtime flexibility within the dynamic area – the sizes and locations of both the static and the dynamic area in the FPGA cannot be changed at runtime.

(2.) Reconfiguration order: Apart from the area placement and interfacing, it is also important to reconfigure the contained cells in the correct order: The most crucial signal in this regard is the *illegal instruction* signal of the pipeline, as it indicates whether or not an instruction is supported by the processor core. This signal is generated in the ID stage, where instruction words are decoded and their execution in the EX stage (as well as processing in other stages) is prepared. In case an undefined instruction is found, an illegal instruction exception triggers its potential software emulation as explained in Section III-B. After adding the instruction to the core, the signal must indicate a valid instruction. Hence, the logic for this signal is the last one to be updated during the reconfiguration. In fact, updating this signal logic prior to providing the instruction logic itself would lead to undefined states and unpredictable behavior. Therefore, the cells containing the instruction logic must be set first. Second, the LUTs for the new opcode must be updated in the instruction decoder (ID). Third and last, the illegal instruction signal is updated.

By updating the relevant logic and signals in the correct order, the new instructions become integral part of the pipeline and are natively supported in the ISA. Thus, no interfacing through, e.g., peripheral buses for co-processors, is required and no corresponding execution time overhead is introduced. This commonly results in performance gains (native instead of emulated instructions) and also allows to replace or refine functions in case of detected bugs or new requirements. Corresponding use cases and evaluations are shown in Section IV.

IV. Use Cases and Evaluation

This Section shows experimental setups of use cases as well as a general evaluation of our concept. First, Section IV-A introduces the software and hardware for our target system. Second, Section IV-B shows a generic use case for a proof of concept and basic evaluations, which are carried out in Section IV-C. Third, Section IV-D provides a use case which is more related to practical applications. The measurements and evaluation are presented in Section IV-E.

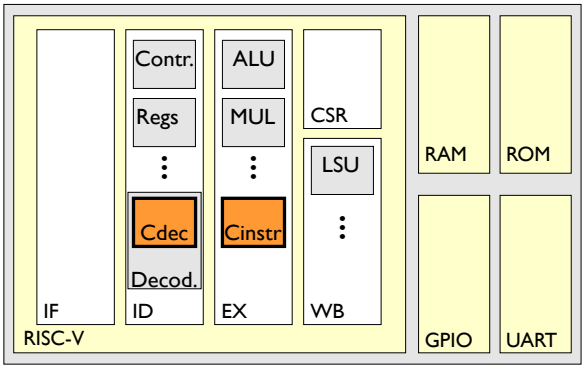


Fig. 4. Structure of the MCU containing a RISC-V core, RAM, ROM, GPIO and UART. The core features a 4-stage pipeline. Custom logic is highlighted.

A. Experimental Setup

For our evaluation, we chose a RI5CY-based [21] 32-bit RISC-V MCU (RV32IM [25], [26]) with 4 pipeline stages, running at 50 MHz clock speed. The stages are identified as Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), and Write Back (WB), whereas the latter stage contains a Load/Store Unit (LSU). For I/O, the MCU contains a UART interface and a GPIO module. A general overview of the hardware structure can be seen in Fig. 4 and we have chosen an XC7A35T FPGA [27] on a Basys3 board [8] for synthesis. Within our use case, we have connected the UART of the MCU to a USB socket, and some GPIO pins to on-board LEDs – both used for debugging and status messages.

The MCU runs our embedded OS [13] with an extension for emulating invalid instructions at runtime, as explained in Section III-B. The test application contains one task, and OS features are used to monitor different runtime aspects (e.g., execution times for code sequences or counters for calls to application-specific instructions).

B. Use Case 1: A general case

As we commonly deal with resource-constrained embedded systems for the IoT or Cyber-Physical System (CPS), our use case is a simple but common algorithm for such devices: we want to calculate sums of products of an arbitrary, bounded sensor value, following an equation like

$$y = \sum_{i=1}^{n} x \cdot i, \tag{1}$$

where x and n are parameters of the function y that will later be computed by the `cinsi` instruction. For our further evaluation, the values $x = 12$ and $n = 15$ are assumed to achieve reproducible and comparable measurements. The calculation itself is computationally expensive when implemented as a regular loop, resulting in a complexity of $\mathcal{O}(n)$. As the final execution time depends on n, this might raise predictability issues for real-time systems which rely a lot on such functions.

Predictability and general performance improvements are addressed by this evaluation. Therefore, the computationally expensive algorithm follows different implementation options:

A) Implementation in software in the traditional way, i.e., without our hardware modifications;
B) Implementation as emulated instruction as explained in Section III-B and in Listing 1;
C) Implementation in hardware by adding the instruction to the ISA at runtime as explained in Sections III-C and III-D. The logic of the new instruction is shown in Fig. 5.

For implementation options B and C, the same application code can be used (cf. Listing 1), while implementation option A needs separate coding.

Implementation option B requires the application software to provide the emulation as a function (to be executed within the illegal instruction exception), and implementation option C requires the application to also provide the corresponding logic for partial reconfiguration. Once it is observed that the emulated instruction is called often, or if explicitly requested by the application, the partial update mechanism is used to add the new instruction to the ISA. Therefore, two steps are needed (cf. Fig. 4):

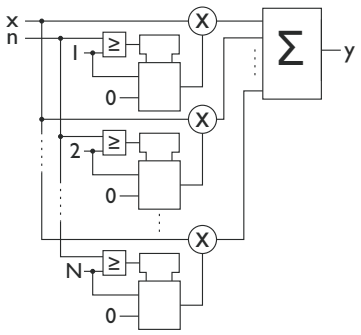


Fig. 5. Logic for calculating a sum of products in hardware.

S1 The new logic is placed into the Custom Area "Cinstr" of the EX stage of the pipeline. The logic is added as a new function slice comparable to other ALU instructions.
S2 The new opcode is introduced into the decoder's Custom Area "Cdec", located in the ID stage of the pipeline. This is done by adding a LUT that marks the new opcode as "legal", and therefore does not throw an illegal instruction exception any more.

Both steps are carried out on-the-fly (in parallel to ongoing MCU operation) and therefore do not introduce interruptions or delays to the system. By the time step S2 is finished, the application software automatically uses the newly introduced instruction. Consequences on the execution time are shown in Section IV-C. The major advantage of the implementation of Eq. 1 in hardware is in the possible parallelism. Depending on the available logic cells in the FPGA, m multiplications can be parallelized and (together with the adder) the entire function might require just one clock cycle in the EX stage. As long as $n \leq m$, the execution time reduces to $\mathcal{O}(1)$. Thus, the developer needs to trade execution time against logic consumption on the FPGA; and can still revise the decision at runtime. However, data type restrictions may apply, depending on which logic elements are available on the used FPGA.

The proof of concept evaluated in the following sections takes the following progression:

(i) Implementation option A is used as base line of our evaluation.
(ii) Implementation option B adds flexibility but is expected to slightly increase the overall execution time due to OS overhead for emulation.
(iii) Implementation option C adds the instruction logic to the pipeline and is expected remove the OS overhead while significantly reducing the execution time compared to B and even A.

C. Use Case 1: Measurements and Evaluation

This Section discusses the runtime effects of our approach, followed by an analysis of the resource consumption of both software and hardware. Both parts refer to the scenario sketched in Sections IV-A and IV-B.

1) Execution Time: Table I and Fig. 6 show the runtimes of the implementation options A-C for Eq. 1. The measurements were carried out in three different ways: by pure simulation of the processor logic, by system self-observation though OS profiling in the real hardware, and by attaching an oscilloscope to the debug interface of the same hardware.

TABLE I
RUNTIME EVALUATION

#	Implementation Option	t_{exec}	cycle #
OS profiling			
0	A – traditional software	$\approx$5.46 μs[a]	273
1	B – instruction emulation	$\approx$7.16 μs[a]	358
2	C – instruction in hardware	$\approx$20 ns[a]	1
Simulation[b]			
3	A	5.46 μs	273
4	B	7.16 μs	358
5	C	20 ns	1
Oscilloscope measurements[c] (using debug ports/pins)			
6	A	5.461 μs	$\approx$273[a]
7	B	7.163 μs	$\approx$358[a]
8	C	21.667 ns	$\approx$1[a]

[a] calculated values using $f_{clk} = 50MHz$ and $T_{clk} = 20ns$.
[b] Xilinx Vivado 2020.1 WebPACK
[c] Keysight MSOS254A 2.5 GHz oscilloscope [10].

The values show different effects – not only in simulation, but also in the running system: First, the emulation of the instruction (option B) adds an overhead of 85 cycles to the baseline implementation (option A). This is due to code injection overhead in the OS (cf. Section III-B and Fig. 1b). The increased ISA flexibility is thus payed with performance loss, but OS optimizations might still improve the overhead. Second, a significant improvement in execution time can be reached when using option C. This option removes the OS overhead entirely and at the same time ensures a fixed runtime of 1 cycle. We also want to point our explicitly, that the simulation results are equal for the self-measurements with internal OS profiling features and the scope measurements that confirm our simulation.

2) Resource Consumption: The resource consumption of different hardware variants of the MCU on our target FPGA is shown in Table II. Certain features like Input/Outputs (I/Os), BRAMs and clocks are not considered, as they stay the same throughout the variants.

Variant I is the RISC-V MCU without further modification, thus not supporting our concept. It executes implementation option A only. **Variant II** directly adds the `cinsi` instruction to the ISA of Variant I, but without our concept for adding further ISA extensions. This variant is only used to show the resource consumption in a static system with hardware support for `cinsi`. **Variant IIIa** introduces our concept of partial reconfiguration from Fig. 3, but initially without the logic for

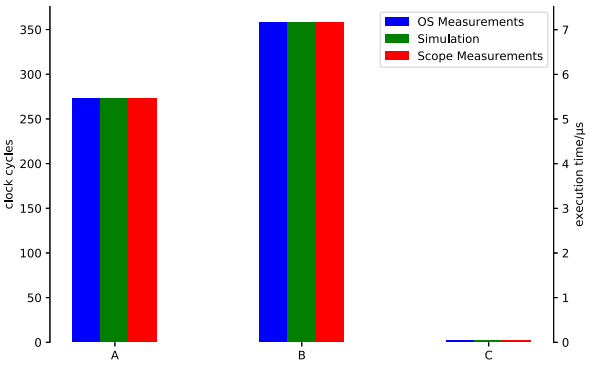


Fig. 6. Runtimes of the implementation options A-C for Eq. 1.

the `cinsi` instruction. **Variant IIIb** basically equals Variant IIIa but after the `cinsi` instruction was added at runtime into the Custom Area (cf. Fig. 4).

Thus, variants I and II come with a static ISA while variants IIIa and IIIb support a dynamic ISA, and refer to implementation options B and C, respectively.

TABLE II
RESOURCE CONSUMPTION ON FPGA

	static ISA		dynamic ISA	
	Variant I	**Variant II**	**Variant IIIa**	**Variant IIIb**
Latches	2628	2628	2496	2496
LUTs	7597	8685	6929	7913
MUX	430	430	486	486
Carries	234	378	222	366

As one can see, Latches and Multiplexers are the same for I and II (static ISA) and for IIIa and IIIb (dynamic ISA). This can be explained, as the added logic is purely combinatorial and only creates signal lines driven by one source. However, LUTs and Carries increase from I to II and from IIIa to IIIb, as they implement the added `cinsi` logic. Apart, it is interesting to see that the introduction the dynamic ISA through partitions on the FPGA initially leads to less resource usage (apart from the multiplexers). A possible explanation is that a more efficient way of logic cell placement can be found during synthesis when partitioning the system in different parts. The subsequent routing of these separate parts can then be achieved with fewer resources. All values were obtained by using Xilinx Vivado 2020.1 WebPACK[2].

D. Use Case 2: An application-oriented case

A very common algorithm to process sensor values on embedded systems is the moving average filter. It calculates the arithmetic average over the last N values and is defined as

$$y = \frac{1}{N}\left(x + \sum_{i=1}^{N-1} x_i\right) = \frac{1}{N} \cdot x + \sum_{i=1}^{N-1} \frac{1}{N} \cdot x_i, \qquad (2)$$

[2] www.xilinx.com/products/design-tools/vivado/vivado-webpack.html

where again x is the only variable parameter of the function y that will later be computed by the `cinsi` instruction. In order to give certain values in time a certain weight, the weighted moving average filter is used. It is defined in an equation like

$$y = w_0 \cdot x + \sum_{i=1}^{N-1} w_i \cdot x_i, \tag{3}$$

where the same parameters apply. A common case is a "linear weighted moving average", where the most recent value is weighted highest and the weight distribution is linearly decreasing.

The hardware structures of both filters can be seen in Fig. 7 and Fig. 8. The main difference compared to the use case before is that previous values must be saved. In this case, a shift register with the length $N-1$ is used to implement the algorithms. Of course, dealing with memory components (e.g., flip-flops, registers) introduces the importance of data consistency when updating the logic, meaning that contents of the memories must either be retained or dealt with in a retaining manner.

This use case shows another major feature of our concept: maintenance/update of a previously added instruction in the field. Imagine a progression in the lifetime of the system, where an algorithm was initially introduced in software. After some time, a hardware implementation of the functionality was added as a new instruction to the ISA. Even later, the implementation shows to be incorrect or needs an update due to changed application requirements. We therefore update the instruction within our ISA, so we can switch from the moving average to the *weighted* moving average filter to overcome the deficiencies in our already deployed system.

E. Use Case 2: Measurements and Evaluation

Similar to Section IV-C, this Section shows an analysis for Use Case 2 with regard to execution time improvements and resource consumption.

1) Execution Time: Fig. 9 shows the runtimes of the different implementations (normal, emulated and hardware execution) of the two algorithms implementing Eq. 2 and 3. For the evaluation, we assume $N = 10$.

Compared to the use case before, emulating the instruction introduces a similar overhead, but the implementation in hardware once more yields a massive performance improvement. The first two implementations have a complexity of $\mathcal{O}(n)$ and scale with N, while the implementation in hardware again is in $\mathcal{O}(1)$. Again, OS measurements, simulation, and scope measurements show the same results.

2) Resource Consumption: The resource consumption of three different hardware implementations is outlined in Table III. The corresponding logic modules can be put into the Custom Area "Cinstr" of our system (see Fig. 4):

- **Normal MA**: The logic for the implementation of a standard moving average filter.
- **Weighted MA**: The extended logic for the implementation of a linear weighted moving average filter.

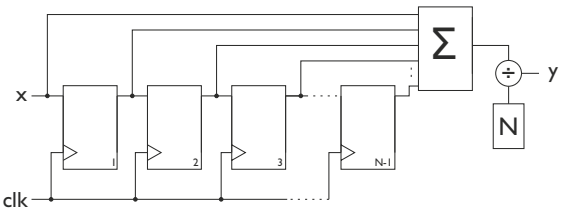


Fig. 7. Hardware structure of a moving average filter.

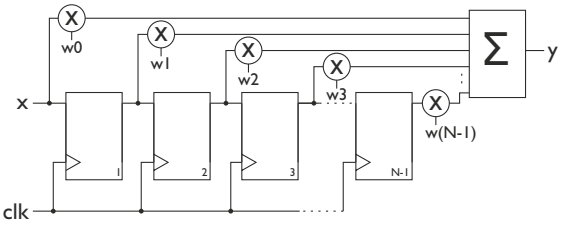


Fig. 8. Hardware structure of a weighted moving average filter.

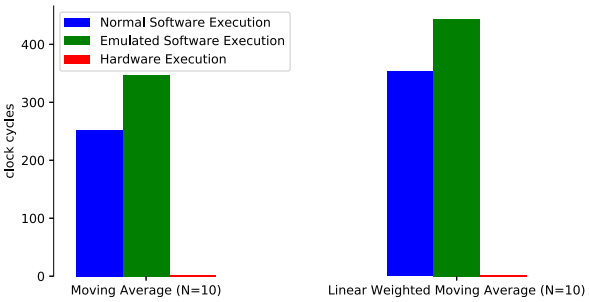


Fig. 9. Runtimes of the different implementation options for both moving average (cf. Eq. 2) and linear weighted moving average filter (cf. Eq. 3).

- **Combined**: Both modules mentioned before combined into one module, making two different instructions available for systems where both algorithms are required.

TABLE III
MODULE RESOURCE CONSUMPTION ON FPGA

	Normal MA	Weighted MA	Combined
Latches	288	288	576
LUTs	947	1066	1937
MUX	0	0	0
Carries	128	161	289

As shown in Table III, the number of required latches stays the same for both normal and weighted moving average, as both modules need to save the same amount of data ($(N-1) \cdot 32bit = 9 \cdot 32bit = 288bit$). The combined module requires exactly the sum of both individual variants. LUT requirements, however, increase dramatically from the normal to the weighted module. This can be explained with all the additional logic needed to calculate the final result value y. The combined module in this case can optimize and reuse

some LUTs for either algorithm, as their count is slightly less than the sum of both modules. Multiplexers are not used in any of the modules. Carries increase a bit from one module to the other; the combined module, however, requires the sum of both modules again. The values for the resource consumption were obtained in the same way as in Section IV-C.

V. Conclusion and Future Work

This paper presents a concept for designing a soft-core processor to support a dynamic Instruction Set Architecture (ISA) by means of (a) partial logic reconfiguration at runtime or (b) instruction emulation. New instructions can simply be compiled into the software binary with standard compilers. Managed by the OS running on the core itself, the approach then allows to switch between (a) native support or (b) software emulation for such instructions, depending on the expected runtime benefits. Switching happens on-the-fly and does not interrupt the execution flow or even reset the device.

In case of native support, the pipeline in the processor core is directly modified and extended, thus making new instructions an integral part of the ISA – no on-chip peripherals, like co-processors, are required and no corresponding overhead for interconnects is introduced. In case of emulation, an exception triggers the execution of some emulating code once the instruction enters the pipeline. The logic (partial bitstreams) or code (using original instructions only) comes with the application using the new instruction.

Regarding the need for dependable long-term operation of embedded devices (e.g., running for decades in the IoT or in vehicles) our concept improves their maintainability by adding flexibility to the hardware. Future FPGA-based embedded systems code can be accelerated by application-specific instructions, or bug/security fixes and functional updates of processor architectures can be applied in the field.

The proposed concept is evaluated with two use cases in which computationally expensive functions are shifted from software to hardware at runtime.

The major contribution of this work is in improving the flexibility and maintainability of dependable embedded systems for long-term operation through continuous hardware/software updates. To the best of our knowledge, the combination of features in our approach advances the state of the art since existing work only addresses individual aspects.

However, there is still room for improvement: For example, the decision-making process within *MCSmart*OS as well as improved concepts for reusing the custom logic area with regard to defragmentation and displacement strategies are part of our ongoing and future research.

References

[1] E. Alkim *et al.*, "ISA Extensions for Finite Field Arithmetic - Accelerating Kyber and NewHope on RISC-V," Cryptology ePrint Arch. 2020.

[2] D. Andrews, R. Sass, E. Anderson, J. Agron, W. Peck, J. Stevens, F. Baijot, and E. Komp, "The Case for High Level Programming Models for Reconfigurable Computers." 01 2006, pp. 21–32.

[3] P. M. Athanas and H. F. Silverman, "Processor reconfiguration through instruction-set metamorphosis," *Computer*, vol. 26, no. 3, 1993.

[4] C. Beckhoff, D. Koch, and J. Torresen, "Go Ahead: A Partial Reconfiguration Framework," in *20th Int'l Symposium on Field-Programmable Custom Computing Machines*, 04 2012, pp. 37–44.

[5] N. Bergmann, J. Williams, J. Han, and Y. Chen, "A Process Model for Hardware Modules in Reconfigurable System-on-Chip." 2006, pp. 205–214.

[6] C. Boano, K. Römer, R. Bloem *et al.*, "Dependability for the Internet of Things: From dependable networking in harsh environments to a holistic view on dependability," *e&i*, vol. 133, no. 7, 2016.

[7] Q. H. Dang, "Secure Hash Standard," Tech. Rep., jul 2015.

[8] *Basys 3 Reference Manual*, Digilent, Inc. [Online]. Available: https://reference.digilentinc.com/reference/programmable-logic/basys-3/reference-manual

[9] S. Gully, V. Gopal, K. Yap, W. Feghali, J. Guilford, and G. Wolrich, "New Instructions Supporting the Secure Hash Algorithm on Intel Architecture Processors," Intel Corporation, Tech. Rep., 2013.

[10] *Infiniium S-Series*, Keysight Technologies, 2019. [Online]. Available: https://literature.cdn.keysight.com/litweb/pdf/5991-3904EN.pdf

[11] P. Kocher, J. Horn, A. Fogh *et al.*, "Spectre attacks: Exploiting speculative execution," in *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[12] M. Lipp, M. Schwarz, D. Gruss *et al.*, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium*, 2018.

[13] R. Martins Gomes, M. Baunach, M. Malenko, L. Batista Ribeiro, and F. Mauroner, "A Co-Designed RTOS and MCU Concept for Dynamically Composed Embedded Systems," in *13th Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, 2017, pp. 41–46.

[14] B. Neumann, T. Sydow, H. Blume, and T. Noll, "Application domain specific embedded FPGAs for flexible ISA-extension of ASIPs," *Journal of Signal Processing Systems*, vol. 53, pp. 129–143, 01 2008.

[15] B. Nichols, D. Buttlar, and J. P. Farrell, *Pthreads programming*. O'Reilly & Associates, Inc., 1996.

[16] J. R. G. Ordaz and D. Koch, "A Soft Dual-Processor System with a Partially Run-Time Reconfigurable Shared 128-Bit SIMD Engine," in *29th Int'l Conf. on Application-specific Systems, Architectures and Processors*, 2018.

[17] K. Papadimitriou, A. Dollas, and S. Hauck, "Performance of Partial Reconfiguration in FPGA Systems: A Survey and a Cost Model," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 4, no. 4, 2011.

[18] B. Penzenstadler and J. Leuser, "Complying with Law for RE in the Automotive Domain," 09 2008.

[19] R. Pittman, N. Lynch, A. Forin, and N. Pittman, "eMIPS, A Dynamically Extensible Processor," 2006.

[20] D. She, Y. He, and H. Corporaal, "An Energy-Efficient Method of Supporting Flexible Special Instructions in an Embedded Processor with Compact ISA," *ACM Trans. Archit. Code Optim.*, vol. 10, no. 3, a 2013.

[21] A. Traber, "RI5CY Core: Datasheet," ETH Zurich, Tech. Rep., Feb. 2019. [Online]. Available: https://www.pulp-platform.org/docs/ri5cy_user_manual.pdf

[22] A. Traber *et al.*, "PULPino: A small single-core RISC-V SoC," 2015, http://iis-projects.ee.ethz.ch/images/d/d0/Pulpino_poster_riscv2015.pdf.

[23] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte, "The MOLEN polymorphic processor," *IEEE Transactions on Computers*, vol. 53, no. 11, pp. 1363–1375, 2004.

[24] T. Verbeure, "The VexRiscV CPU – A New Way to Design," Dec. 2018. [Online]. Available: https://tomverbeure.github.io/rtl/2018/12/06/The-VexRiscV-CPU-A-New-Way-To-Design.html

[25] A. Waterman and K. Asanovic, "The RISC-V Instruction Set Manual, Volume I: Unprivileged-Level ISA, Version 20191214-draft," SiFive Inc., UC Berkeley, Tech. Rep., Jul. 2020.

[26] ——, "The RISC-V Instruction Set Manual Volume II: Privileged Architecture Version 1.12-draft," SiFive Inc., UC Berkeley, Tech. Rep., Jul. 2020.

[27] *7 Series FPGAs Data Sheet: Overview*, DS180 (v2.6) ed., Xilinx, Inc., Feb. 2018.

[28] *Dynamic Function eXchange (UG909, v2019.2)*, Xilinx, Inc., Jan. 2020. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_2/ug909-vivado-partial-reconfiguration.pdf

[29] Z. A. Ye, A. Moshovos, S. Hauck, and P. Banerjee, "CHIMAERA: a high-performance architecture with a tightly-coupled reconfigurable functional unit," in *27th Int'l Symp. on Computer Architecture*, 2000.

7.7 R-III: *Smart*OS: An OS Architecture for Sustainable Embedded Systems

◇ Authors: Tobias Scheipel, Leandro Batista Ribeiro, Tim Sagaster, and Marcel Baunach
◇ Date: March, 2022
◇ DOI: 10.18420/fgbs2022f-01
◇ Reference in bibliography: [19]
◇ Presented by myself at the 2022 Spring Meeting of the Expert's Group Operating Systems (FGBS'22), Hamburg, Germany.
◇ Published in the proceedings of FGBS'22 (GI).

Summary *Smart*OS is an embedded Operating System (OS) architecture that is designed to improve sustainability of embedded systems. Several concepts extend the set of basic features to achieve this goal. The shown approach is based on hardware/software co-design of the entire system and the idea of enabling modular updates for both hardware and software at runtime. Hence, dynamic software composition and integration, logic reconfiguration, and formal methods for verification and portability play an essential role.

Author's Contribution I am the main author of this paper alongside Leandro Batista Ribeiro, who equally contributed to this work. We have jointly envisioned, designed, and implemented the concept of the paper with the support of Tim Sagaster and Marcel Baunach.

Copyright ©2022 by the Authors. The version included in this thesis is reprinted, with permission, from the published version stated above, which can be accessed in the GI Digital Library:
https://dl.gi.de

*Smart*OS: An OS Architecture for Sustainable Embedded Systems

Tobias Scheipel
tobias.scheipel@tugraz.at
Graz University of Technology
Graz, Austria

Leandro Batista Ribeiro
lbatistaribeiro@tugraz.at
Graz University of Technology
Graz, Austria

Tim Sagaster
tim.sagaster@student.tugraz.at
Graz University of Technology
Graz, Austria

Marcel Baunach
baunach@tugraz.at
Graz University of Technology
Graz, Austria

ABSTRACT

The number of embedded devices is growing, and so are the concerns about dependability and sustainability. However, the life-span of modern devices is commonly very short, due to their lack of long-term maintainability in both hardware and software. This yields an increased amount of e-waste, as the individual devices are commonly very cheap and can therefore easily be replaced in case of (partial) obsolescence.

In this work, we show an operating system architecture which is designed to make embedded systems more sustainable and prepared for long-term use. To do so, we implement a general basic architecture alongside extended concepts and special features within the operating system. Our approach is based on hardware/software co-design and the opportunity to update software as well as hardware in a modular way at runtime. Therefore, logic reconfiguration of the host platform, dynamic software composition and integration, as well as formal methods for verification and portability are supported.

KEYWORDS

operating system, embedded systems, sustainability, partial updates, partial reconfiguration, formal methods

1 INTRODUCTION

This paper addresses the question how future Operating System (OS) architectures can support the development of *sustainable* software and hardware. Focus is on the aspects *long-term maintenance* and *dependability* of individual HW/SW components as well as the composed overall systems. In particular, new or currently underrepresented technological and methodological possibilities will be included in the architecture of the system software.

The impulse for the considerations is given by the overarching goal of keeping future embedded systems operational for decades with continuously guaranteed dependability [10, 12, 36, 58]. During development and operation, (limited) resources shall be used responsibly. Therefore, e-waste shall be avoided and existing systems shall be more easily reusable, i.e., efficiently adaptable to new functions and be able to operate in mixed networks of old and new systems in the long-run.

This overarching goal leads to more extensive or entirely new *demands* for the architecture of such systems and their components: Over the entire lifetime, (D1) partial software updates shall be facilitated and there shall be (D2) support for updating or modifying even the hardware. For each update, (D3) the integration of new or modified HW/SW shall be done automatically, and (D4) hard correctness guarantees for the new composition shall be provided. In order to be successful in the long run, (D5) the operating system itself shall be efficiently portable for new and changing target architectures as well as their derivatives in order to permanently guarantee a dependable basis for changing software compositions.

In this respect, technological and methodological progress in the areas of hardware design and formal methods opens up new *Possibilities*: Future embedded systems will (P1) increasingly support partial reconfiguration of logic at runtime [11, 67] and (P2) increasingly use formal methods for software development and maintenance [22, 25].

Today's operating systems are based on architectures which have virtually not changed for decades and do not or not sufficiently support the demands D1-D5. In particular,

Except as otherwise noted, this paper is licensed under the Creative Commons Attribution-Share Alike 4.0 International License.

FGBS '22, March 17–18, 2022, Hamburg, Germany
© 2022 Copyright held by the authors.
https://doi.org/10.18420/fgbs2022f-01

they do not yet take advantage of the available possibilities P1-P2 for ensuring continuous sustainability of systems.

In the following, we discuss some potential approaches for future operating system architectures to enable maintenance (in the sense of portability, support for reconfigurable hardware, and partial software updates) and guarantee dependability (in the sense of runtime updates and verification capability) of future embedded systems. Initial insights on first implementations and applicability are shown by the example of *Smart*OS.

The remainder of this paper is structured as follows: Section 2 summarizes current and previous work in related research areas and motivates the presented concepts. Subsequently, Section 3 shows the architecture, the main concepts as well as the utilized methodologies of our operating system *Smart*OS. Section 4 details the implemented features and resulting characteristics of *Smart*OS. In Section 5, an application scenario shows, how the features of *Smart*OS can help to overcome system shortcomings in practice, whereas the final Section 6 concludes the paper.

2 MOTIVATION AND RELATED WORK

Being able to update software in a deployed embedded system is not a new concept. There are multiple extensions available on top of widely used OSs like FreeRTOS [23], Device OS [55], QNX [13], VxWorks [71], or TinyOS [66] that provide some form of updates in the field. While most of those extensions will replace the firmware as a monolithic binary, almost all require a reboot of the system to finish the update [24]. There has been some work regarding the partial update of application code at runtime as well as of parts of the OS code itself [29, 43]. Some OSs have the built-in ability to update some modules and programs at runtime like Contiki [21]. Regarding the OS design, there is also some work using Domain-Specific Language (DSL) and formal methods [57] to model an OS that can be updated at runtime.

Since update-induced reboots or down times can lead to undefined conditions like timing violations or consequences on other systems [52], *Smart*OS supports runtime modularity in the upper layers.

While reconfigurable hardware (in, e.g., FPGAs) is becoming more common, it is still rare compared to fixed devices (ASICs). However, we see an increasing demand for hardware updates in future embedded devices and propose to implement the idea of "updating hardware like software" directly into the OS. Such an option would allow to fix hardware bugs [33, 38], improve algorithms or add new ones (e.g., for artificial intelligence or machine learning), or adapt to changing requirements and legislation [56]. Today, the hardware is either replaced or software workarounds are applied in such situations. Even if Field-programmable Gate Arrays (FPGAs) are used, we observe the same measures as for software: Logic updates are either monolithic or not offered at all. When it comes to OS support, only few kernels can exploit the flexibility of embedded FPGAs to enable dynamic Instruction Set Architectures (ISAs) [18] or extensible on-chip peripherals of soft core processors.

To support a seamless integration of hardware adaptation already in the OS, *Smart*OS can benefit from OS-awareness and partial reconfiguration in the host platform. This enables rebootless hardware updates at runtime [62].

To successfully integrate software updates, all involved processes should be done automatically. This includes dependency resolution, resource availability analysis (e.g., memory and OS data structures), and module management. While such features are standard in non-embedded devices, only few embedded OSs allow modular application updates at runtime due to the extra resource overhead. Still, some approaches allow updates to the bytecode of VM-based applications [37] or to dynamically link and load native code [20, 61].

In *Smart*OS, the so-called pluggability check is performed upon updates. New modules/versions can only be integrated if the target system has enough memory and free slots on the OS management data structures. Missing dependencies are automatically installed in the scope of an update protocol [8].

A reliable software integration also demands guarantees that the resulting software composition meets all system requirements. There are some approaches regarding schedulability [51, 64] as well as on guarantees about mutual exclusion properties and freedom from deadlocks [74] after updates. The demand for provable correctness as well as the occurrence of increasingly efficient algorithms to do so has led to a rise in popularity of formal methods in the area of embedded systems. An example for a formally verified kernel is sel4 [32], but it does not allow for runtime updates. Reconfiguring components of an embedded system at runtime in a safe manner is discussed in [57].

In *Smart*OS, we have proposed the interoperability check, a term that refers to operations performed to evaluate whether all system requirements will still be met in case an update was applied [8].

Portability of OSs is critical w.r.t. both coding effort and (non-) functional correctness for all supported target platforms. As detailed by [44], this goal can be achieved by applying formal methods for model-based design, verification, and code generation. However, even more recent OSs that declare portability as an inherent design feature, like Zephyr [73] or Atomthreads [4], only provide more or less detailed developer guides [5, 72] on how to port the OS to new processors or hardware platforms. In the automotive industry,

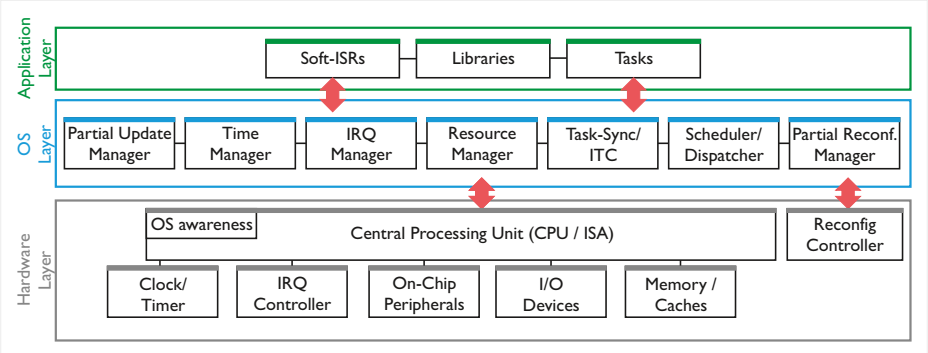


Figure 1: *Smart*OS layering with basic and extended concepts.

the problem of often error-prone and time-consuming re-implementation work [15] has been overcome by the AUTOSAR standard [6].

For *Smart*OS we seek to facilitate the porting process by formally modelling hardware-dependent kernel code and relevant parts of the target architectures. Code can then be generated from composed HW/SW models, allowing formal verification and avoiding error-prone re-implementation work.

For few existing embedded OS, the mentioned requirements or features do already exist independently from each other - often, however, only as extensions or workarounds. This motivates the creation of a sustainable OS that inherently incorporates features for all the introduced demands D1-D5 by design. To do so, we design *Smart*OS as an embedded OS, not only supporting rebootless updates for both software and hardware, but also compositional software aspects alongside portability and verification techniques. This way, we align to the paradigm proposed by European Union Agency for Network and Information Security (ENISA), which says that "the research in the area of patching and updating equipment without disruption of service and tools" [36] is of importance to future devices.

3 OS ARCHITECTURE AND BASIC CONCEPTS

*Smart*OS is a Real-Time Operating System (RTOS) for resource-constrained embedded devices, consisting of a microkernel that supports some basic features as explained next. The source code is supported by an increasingly complete set of formal models, representing all those features as well its target architectures for both formal verification and code generation (cf. Section 4.3). Furthermore, there exists an extensive build environment [45] that supports code separation for the selected target architecture. Currently, the main supported Microcontroller Unit (MCU) architectures are MSP430 [65] and RISC-V [60], but there are also ports for ARMv7E-M [3] (in MSP432 and STM32H7), SuperH [59], and Aurix [31]. Apart from research, *Smart*OS is also used in teaching.

*Smart*OS-based embedded systems feature a strict **layering** across the entire system stack that is depicted in Figure 1. Hardware components like the CPUs (which comes with optional OS-awareness, cf. Section 4.1.3) and its peripherals, as well as the (optional) Reconfiguration Controller (cf. Section 4.1.1) are featured in the Hardware Layer. In the OS Layer the *Smart*OS kernel with all its different Managers is situated. Application tasks, libraries and soft-ISRs belong to the Application Layer.

The *Smart*OS microkernel implements six central concepts, on which the extended concepts explained in Section 4 build. Table 1 summarizes the corresponding syscalls and their behaviour.

The **Internal Timeline** of *Smart*OS is directly driven by a hardware clock. On any supported architecture, it is 64 bits wide and has a resolution of 1 μs. The notion of time is provided to higher software layers through a temporal semantics of many kernel functions, that allow tasks to e.g., sleep or wait on events and resources with absolute or relative deadlines.

Tasks and their entry functions are the main building blocks of any *Smart*OS application. Tasks execute in task mode, are preemptive at any time, and can synchronize on each other (through events and resources) as well as on the hardware (through interrupts). Thus, they can be in running, ready, or waiting state (cf. Figure 2). To support proper interleaving by the scheduler, tasks have individual stacks of fixed size and variable priorities (both defined or pre-selected at compile time). *Smart*OS always provides an idle task that

Table 1: *Smart*OS syscalls. T is the invoking task.

Syscall	Behavior
`waitEventUntil(ev, deadline)`	T goes to waiting state until `ev` is set or the absolute `deadline` is reached. If `ev` is already set, T consumes the event and continues executing. If `deadline` is reached, T goes to ready state.
`waitEventFor(ev, timeout)`	Like `waitEventUntil`, but with a relative `timeout` instead of the absolute `deadline`.
`waitEvent(ev)`	Like `waitEventUntil`, but with an infinite `deadline`.
`setEvent(ev)`	T triggers `ev`: If `ev` was already set, nothing happens. If other tasks are waiting for `ev`, the one with highest priority immediately consumes it and goes to ready state. If no tasks are waiting, `ev` is set for later consumption.
`notifyEvent(ev)`	T triggers `ev`: If `ev` was already set, nothing happens. If other tasks are waiting for `ev`, they all immediately consume it and go to the ready state. If no tasks are waiting, `ev` is set for later consumption.
`getResourceUntil(res, deadline)`	T goes to waiting state until `res` is free or the absolute `deadline` is reached. If it is already free T takes the ownership of `res` and continues executing. If it is already owned by T, T increases the ownership of `res` by one and continues executing. If `deadline` is reached, T goes to ready state.
`getResourceFor(res, timeout)`	Like `getResourceUntil`, but with a relative `timeout` instead of the absolute `deadline`.
`getResource(res)`	Like `getResourceUntil`, but with an infinite `deadline`.
`releaseResource(res)`	T decreases its ownership of `res` by one. If it is already free or if T does not own it, nothing happens. If other tasks are waiting for it, the one with highest priority takes ownership of `res` and goes to ready state.
`getCurrentTime()`	Returns the current system time.
`sleepUntil(deadline)`	T goes to waiting state, and after absolute `deadline` is reached it goes to ready state.
`sleep(timeout)`	Like `sleepUntil`, but with a relative `timeout` instead of the absolute `deadline`.
`yield()`	T is removed from and immediately reinserted into the ready queue, handing over to another task with the same priority in ready state.

runs at lowest priority and is responsible for, e.g., power management. A hypothetical task can be seen in Listing 1.

System Calls are functions to request OS services from the higher software layers. When called by a task, they switch to kernel mode first and then execute the contained code in an atomic fashion. If supported by the CPU, *Smart*OS makes use of hardware acceleration or applies special protection mechanisms to enforce the isolation of the OS and the higher software layers [70].

Events allow Inter-Task-Communication (ITC) between two or more tasks and signalling between ISRs and tasks. While tasks and ISRs can trigger events, only tasks can wait for events (with or without deadline). This allows the simple mapping of IRQs to responsible tasks – including the consistent reflection of IRQ priorities (HW) in relation to the tasks priorities (SW) [46, 48]. In any case, triggering an event can resume either the highest prioritized task or all tasks waiting for it.

Resources allow tasks to exclusively allocate shared physical or virtual objects, like on-chip peripherals, data structures, etc. Resources can be created at any software layer,

Listing 1: A hypothetical *Smart*OS task.

```
1  OS_TASKENTRY (task1) {
2    [...]
3    while (1) {
4      waitEvent (ev1);
5
6      getResource (res1);
7      [...]
8      releaseResource (res1);
9
10     setEvent (ev2);
11   }
12 }
```

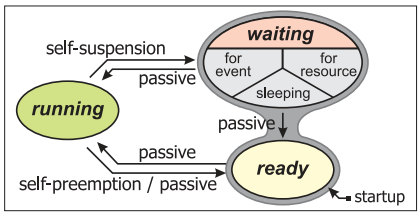


Figure 2: Task states and transitions in *Smart*OS.

e.g., by drivers for their specific hardware component or by tasks just to synchronize with others. In any case, each resource can only be held by one task at a time, but a task may hold multiple resources at once or the same resource several times. Requesting a resource from the kernel can be done with or without deadline: If a resource is currently not available, the requesting task transits to waiting state until the resource becomes free or the deadline is reached. *Smart*OS implements various resource management protocols (PIP, PCP, HLP; cf. [16]) that take task priorities into account and feature different deadlock avoidance/resolution strategies. This includes a special extension that signals tasks that prevent higher priority tasks from running due to an allocated resource to release the resource – which reduces priority inversion and resolves deadlocks [9, 48]. While managing resources is different from protecting them, *Smart*OS can use related hardware security mechanisms as described in [40].

Interrupts are always handled in a unified way: The kernel provides a unique ISR for all IRQ sources, captures the timestamp on occurrence, and then calls a soft-ISR that can be provided by any software layer at runtime. This gives the opportunity to relate to the timestamp (e.g., for real-time systems) or to demultiplex IRQ sources (e.g., to introduce one soft-ISR per pin of a multi-pin GPIO port with just one shared IRQ vector). In any case, ISRs and soft-ISRs are executed atomically in kernel context and on the kernel stack. While this obviates the prophylactic over-provisioning of task stacks, ISRs can neither be nested nor are they allowed to wait for events or allocate resources. If such synchronization is needed, they should trigger a task for the actual IRQ processing.

4 EXTENDED CONCEPTS AND SPECIAL FEATURES

Apart from the basic concepts described in Section 3, our research in various directions has resulted in several extended concepts and special *Smart*OS features that tackle the demands D1-D5 from Section 1. These concepts can be divided into three different overarching topics: OS-specific hardware support and reconfiguration (Section 4.1), compositional software design and partial updates (Section 4.2), and formal methods for verification and portability (Section 4.3). *Smart*OS runs on all aforementioned platforms without restrictions. However, if it is used with our extended concepts and features in both hardware and software, there are several advantages.

4.1 MCU/OS Co-Design

The interaction between hardware and software requires mutual support on *both* sides. While "software follows hardware" is the prevalent credo in most domains, many embedded systems can benefit significantly from applications-specific hardware extensions. Thus, we do not only tailor our OS concepts to common hardware concepts, but we also investigate how OS kernels can benefit from specific MCU extensions, namely:

- *partial reconfiguration* of the host computing platform (i.e., processors, peripherals) at runtime (cf. Section 4.1.1)
- *hardware security* features (e.g., memory protection) within the host MCU (cf. Section 4.1.2), and
- *OS-aware* logic (cf. Section 4.1.3) of the host processor.

4.1.1 Partial Hardware Reconfiguration at Runtime. If a processor's logic changes during code execution, we talk about partial reconfiguration at runtime. With our goal to support dynamic composition in software *and* hardware, our concept of partial logic updates [62] allows to update and extend the ISA of processors as well as their on-chip peripherals on-the-fly without resynthesis of the entire logic and even without halting or resetting the system. As this significantly influences the software execution flow up to the application layer, the OS must support and manage such modifications, and support throughout all software layers must be introduced. In this section, we describe the underlying OS/MCU concepts at the example of the RISC-V-based *more*MCU.

To understand the concept, assume an application that calls an assumed instruction `cinsi` that might or might not be implemented by the processor. This is shown in Line 12 in Listing 2. Within the original RISC-V ISA [68, 69], this instruction is unknown and leads to an illegal instruction exception within the processor. In this case, the OS is responsible for emulating the unknown instruction's functionality in software. To do so, the currently running task is interrupted and kernel mode is entered. Then, the OS processes the binary encoding of the instruction causing the exception and checks its internal data structures for a corresponding emulation function that was registered by an application. If the emulation was registered before, this functionality is injected into the task's execution flow by the OS, and executed after the kernel returns to task mode. In case this look-up fails, the instruction is illegal and internal error handling steps in, and the task is most likely killed.

If, however, the instruction was natively supported by the processor, no further steps must be taken. Therefore, *Smart*OS is able to manipulate the ISA of the underlying computing platform actively. This is supported by our specifically tailored *more*MCU architecture, designed as a soft core for FPGA. As it features a runtime partial reconfiguration controller, parts of the CPU pipeline and the peripherals can

```
10    [...]
11    addi  t1, zero, 6
12    cinsi t0, t1,   2    ; unknown
          instr.
13    [...]
```

Listing 2: Unknown instruction execution code.

be hot-swapped between hardware realization and emulation on the fly. The OS can therefore decide, which instructions shall be added to the hardware, and which functionality shall be emulated in software. While the applications have to provide and register the emulation code or FPGA bitstreams to benefit from this feature, they can introduce new and application-specific hardware acceleration even during updates after hardware deployment.

4.1.2 Security Feature Support. A common design approach, especially in resource-constrained embedded devices is to have applications, OS components, and device drivers or libraries reside in a single non-isolated address space, which represents one vast attack surface. In order to reduce complexity, cost, and energy usage, embedded devices often lack memory protection mechanisms. Apart, programming flaws can easily lead to malfunctions or crash the whole system. As embedded devices are increasingly integrated in critical infrastructures, cyber-security attacks can have devastating consequences, including impacts on human lives or the environment [2, 30, 34].

Efficient memory protection in embedded devices can be achieved only if hardware and software components are co-designed to cooperate. We have designed and implemented lightweight hardware extensions for RISC-V-based MCUs which work in synergy with the *Smart*OS kernel extensions to provide several memory protection concepts. First, we enabled a memory protection mechanism which confines tasks to their own code and data memory regions, while still enabling full access to peripherals and protected access to shared memory for communication. Then, we enforced the kernel's resource management protocol in hardware by locking memory-mapped peripherals and protecting them from unauthorized task access [41]. Furthermore, we extended this concept to enable fine-grained protection of peripherals with multiple channels [40]. Finally, in order to protect the system from malfunctioning device drivers, we provided a device driver isolation mechanism, which limits the memory regions that drivers are allowed to access [39].

We show that these protection mechanisms have small hardware and software footprints. They do not impose significant run-time overhead and are suitable for maintaining the existing real-time properties of the system.

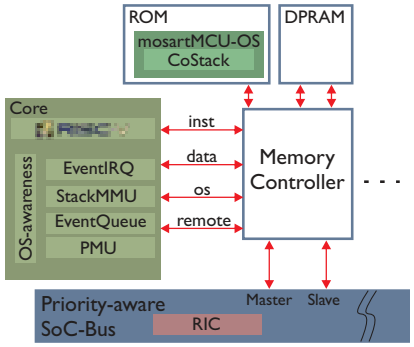


Figure 3: Overview of the *mosart*MCU.

4.1.3 OS-aware Processor Extensions. To support OS features in hardware, we introduced the mosartMCU [49, 50], being a flexible multi-core MCU architecture. Based on the RISC-V ISA [60], its implementation as soft core for FPGAs allows us to conduct hardware/software co-design for application-specific computing platforms. By introducing OS-awareness for improved dependability and composition aspects (cf. Figure 3), the *mosart*MCU and *Smart*OS support each other, but can also be used independently.

When used in combination, the MCU can concurrently access and modify internal OS data structures to temporally bound or entirely avoid cross-core priority inversions [48, 63]. Similarly, the co-design of OS/MCU memory management concepts and data structures improves shared stack memory usage to facilitate predictable software execution times [47]. In general, the OS-awareness enables system properties that could not be achieved through pure software solutions, and opens entirely new research opportunities for OS/MCU co-design.

4.2 Compositional Software and Automatic Integration

At build time, software composition of embedded systems is usually modular, i.e., different pieces of software (applications, libraries, OS, etc.) are combined to generate a binary image with the desired features and behavior. However, this image is mostly monolithic, and updates require the creation of another full image, be it deployed through complete replacement or by differential approaches. Our research efforts aim to provide concepts to support dynamic software (re-)composition at runtime [8]. This includes two steps: (S1) partially update the running software in a modular way while preserving code dependencies, and (S2) check if all functional and non-functional requirements would still be satisfied in case a given update was applied. Updates will

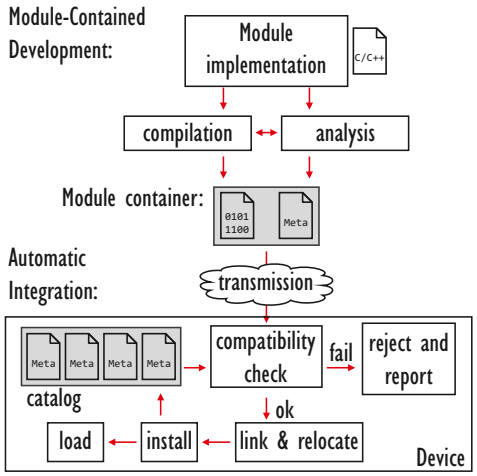


Figure 4: Module development and automatic integration.

only be applied in case all compatibility checks are successful, as shown in Figure 4.

Partial updates demand extra OS functionality. Already for step S1, it is necessary to keep track of all installed modules, handle dependencies, manage memories, and offer an accessible update service. For S2, complex algorithms – often based on formal methods – need to be implemented and demand additional computational resources.

From an implementation perspective of S1, *Smart*OS supports the update of application modules on conventional MCUs. However, the update of shared modules, such as drivers and libraries, is not supported, due to potentially inconsistent references. For each dependency update, dependent modules must adapt their references accordingly. To support this rather complex operation, we have worked on a hw/sw solution that supports loose coupling based on runtime relocation [42]. With our approach, all dependents remain unchanged when their dependencies are updated. It is also possible to freely move modules within the physical memory, allowing memory defragmentation. This feature is essential for long-term maintenance, since repeated updates will lead to memory fragmentation and the under-utilization of this commonly scarce resource.

When it comes to requirements checking for step S2, there are a variety of research challenges, especially regarding the mutual impacts of old and new software components w.r.t. (non-)functional requirements. For example, how to guarantee that already existing and newly installed real-time tasks will continue to meet their deadlines after a re-composition. Similar questions arise for other requirements, e.g., energy consumption, safety, security, etc. To enable the analysis of such aspects, it is necessary to generate or extract metadata from the developed modules. Our efforts in this regard include (i) a compact notation to describe the control-flow and interaction of tasks [7]; and (ii) comprehensive Uppaal models of *Smart*OS and application modules. Goal is to combine the metadata of individual modules in order to reliably conclude whether the resulting software composition meets the specified requirements. The approach that provides the strongest guarantees is formal methods, and that is the direction of our research efforts.

4.3 Formal Methods for Verification and Portability

Our approach uses formal languages to initially create independent models of application software, operating systems, and processor logic. Models of different layers of a concrete system can then be merged into an overall model to (1) verify different aspects of (non-)functional properties and (2) generate hardware-specific code for different target architectures.

For *Smart*OS, this approach currently allows us to give liveness and real-time guarantees for compositional application software, and to automatically port low-level kernel code to different target architectures.

Regarding (1), we use Uppaal [35] to model application tasks and the OS itself as a network of timed automata. Figure 5 shows how the execution block of a task is modeled (a task is composed of interactions with its execution block and with the OS). An execution block finishes when the task has executed for an interval within [BCET, WCET]. The execution time is not increased (`et' == 0`) before the execution block starts, or when the task is preempted.

One focus of the models is on the intended interaction (e.g., though explicit ITC) and implicit interference (e.g., through emerging conflicts) between tasks. Since the OS provides the central mechanisms for coordinating the tasks, we have modeled the internal functionality, timing parameters, and interfaces of the *Smart*OS syscalls (see Table 1) and interrupt concept. On top, application task models can interact with the OS model and also incorporate functional as well as non-functional aspects. This way, we are able to prove liveness (e.g., freedom from deadlocks), timing behavior, etc. throughout the entire system stack. A particular benefit of this approach is the possibility to (i) include even low-level OS details into such analysis (which is often neglected in many works on e.g., schedulability analysis), and to (ii) easily replace the OS and application models independently from each other (which allows us to verify the same application against various different mechanisms and implementation options in the kernel).

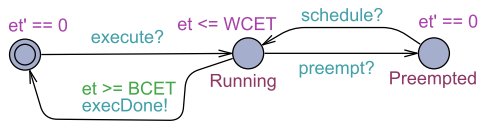


Figure 5: UPPAAL timed automaton of a task execution block.

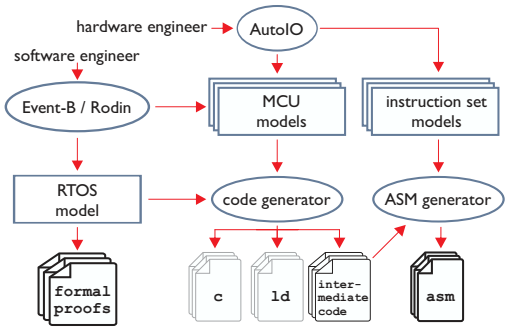


Figure 6: Model-based porting of hardware-specific code.

Regarding (2), we work on automatic generation of code for OS low-level functionalities, such as system initialization or context switches [26, 27]. The framework overview is depicted in Figure 6.

We use Event-B [1] to formally model and verify abstract versions of low-level OS operations. We also developed AutoIO [54], which is used to specify MCU architectures, including, e.g., on-chip components, buses, configuration registers, and the instruction set. We then feed the code generator with the verified OS model and the MCU model for the target architecture in order to generate architecture-independent code (in LLVM-IR). Finally, we feed the ASM generator with the generated intermediate code and the architecture instruction set model in order to create specific assembly code, proven correct. The automatic code generation guarantees that the assembly code is correctly translated from the models, and the formal verification guarantees that the models are correct. These features avoid implementation mistakes, which are very common when manually porting OSes.

5 APPLICATION SCENARIO

Having motivated requirements for future embedded systems and shown possibilities for designing suitable operating systems, we want to describe a concrete application scenario, that certainly demands sustainability, continuous dependability, and long-term maintenance.

The control systems of power plants are composed of multiple interconnected computing devices. For safety and security reasons, they are organized in levels that are mostly only accessible internally. However, external access to these devices is still required by the so-called emergency off-site facilities. Also for safety reasons, such facilities are located many miles away from the power plant [17]. Since the embedded systems of the power plant can thus be externally accessed, they could be hijacked and controlled remotely – demanding for continued and guaranteed dependability in every respect. In fact, the vast majority of computer systems rely on hardware acceleration for better performance on cryptography operations/security enforcement. However, cryptographic algorithms might become obsolete during the long lifespan of power plants. For example, in 2011 SHA-1 [19] became deprecated, thus making hardware accelerators in processors useless (cf. Intel processors [28]). Similar examples can be found in other areas which depend on the IoT as a critical infrastructure, that connects billions of devices [14].

To overcome this issue using immutable hardware, either (i) the new algorithm is implemented purely in software, and the obsolete silicon remains as useless logic, never again used; or (ii) the whole device or CPU is exchanged, which is not a sustainable choice. With reconfigurable hardware, the obsolete hardware logic could be replaced, resulting in faster operation and a more sustainable solution.

However, regardless of which changes are applied, the system's functional (e.g., new cryptographic algorithm) and non-functional properties (e.g., timing and memory consumption) will be modified, and it might be necessary to test, verify, and re-certify the updated system or submit so-called license amendment requests, which add cost and regulatory risk [53]. In this regard, the use of formal methods can be extremely beneficial, since it provides stronger guarantees than testing, and can facilitate regular updates [25].

6 CONCLUSION

In this paper, we present *Smart*OS, an OS architecture aiming to improve the sustainability of embedded systems, which is achieved through extended features on top of its basic functionality. These extended features include (i) a tightly coupled design of the OS and its underlying MCU, (ii) support for compositional software, and (iii) the use of formal methods to support software development and maintenance.

So far, approaches using the three mentioned features have not yet been fully adopted in research or industry. However, the demand for increased sustainability will create a need for these kind of features in the future. Therefore, the goal of this paper is to raise and intensify the awareness for the necessity of such concepts and simplify their use by providing an OS architecture that inherently supports them by design.

REFERENCES

[1] Jean-Raymond Abrial. 2010. *Modeling in Event-B: System and Software Engineering* (1st ed.). Cambridge University Press, New York, NY, USA.

[2] Riham Altawy and Amr Youssef. 2016. Security Tradeoffs in Cyber Physical Systems: A Case Study Survey on Implantable Medical Devices. *IEEE Access* (2016).

[3] ARM. 2021. *ARMv7-M Architecture Reference Manual.* Technical Report. ARM.

[4] Atomthreads. 2022. Atomthreads: Open Source RTOS | Free Lightweight Portable Scheduler. [Online] http://atomthreads.com/.

[5] Atomthreads. 2022. Porting Guide | Atomthreads: Open Source RTOS. http://atomthreads.com/index.php?q=node/3 [Online] http://atomthreads.com/index.php?q=node/3.

[6] AUTOSAR. 2017. Classic Platform Release 4.3.1.

[7] Leandro Batista Ribeiro and Marcel Carsten Baunach. 2019. COFIE: a regex-like interaction and control flow description. In *2019 IEEE Industrial Cyber-Physical Systems.*

[8] Leandro Batista Ribeiro, Fabian Schlager, and Marcel Baunach. 2020. Towards Automatic SW Integration in Dependable Embedded Systems.. In *Int'l Conf. on Embedded Wireless Systems and Networks (EWSN'20).* 85–96.

[9] Marcel Baunach. 2011. Dynamic hinting: Collaborative real-time resource management for reactive embedded systems. *Journal of Systems Architecture* 57, 9 (Oct 2011), 799–814. https://doi.org/10.1016/j.sysarc.2011.07.001

[10] Marcel Baunach, Renata Martins Gomes, Maja Malenko, Fabian Mauroner, Leandro Batista Ribeiro, and Tobias Scheipel. 2018. Smart mobility of the future – a challenge for embedded automotive systems. *e & i Elektrotechnik und Informationstechnik* (27 Jun 2018), 304–308. https://doi.org/10.1007/s00502-018-0623-6

[11] Christian Beckhoff, Dirk Koch, and Jim Torresen. 2012. Go Ahead: A Partial Reconfiguration Framework. In *20th Int'l Symposium on Field-Programmable Custom Computing Machines.* 37–44. https://doi.org/10.1109/FCCM.2012.17

[12] Ron Bell. 2006. Introduction to IEC 61508. In *Acm Int'l Conf. proceeding series*, Vol. 162. 3–12.

[13] BlackBerry. 2022. QNX. [Online] https://blackberry.qnx.com/en/.

[14] Carlo Boano, Kai Römer, Roderick Bloem, et al. 2016. Dependability for the Internet of Things: From dependable networking in harsh environments to a holistic view on dependability. *e&i* 133, 7 (2016), 6 pages.

[15] Manfred Broy. 2006. Challenges in Automotive Software Engineering. In *Proc. of the 28th Int'l Conf. on Software Engineering* (Shanghai, China) *(ICSE '06).* ACM, New York, NY, USA, 33–42. https://doi.org/10.1145/1134285.1134292

[16] Albert M. K. Cheng and James Ras. 2007. The Implementation of the Priority Ceiling Protocol in Ada-2005. *Ada Lett.* XXVII, 1 (apr 2007), 24–39.

[17] Chi-Shiang Cho, Wei-Ho Chung, and Sy-Yen Kuo. 2016. Cyberphysical Security and Dependability Analysis of Digital Control Systems in Nuclear Power Plants. *IEEE Trans. on Systems, Man, and Cybernetics: Systems* 46, 3 (2016), 356–369.

[18] Marvin Damschen, Martin Rapp, Lars Bauer, and Jörg Henkel. 2020. *i-Core: A Runtime-Reconfigurable Processor Platform for Cyber-Physical Systems.* 1–36.

[19] Quynh H. Dang. 2015. *Secure Hash Standard.* Technical Report.

[20] Adam Dunkels, Niclas Finne, Joakim Eriksson, and Thiemo Voigt. 2006. Run-time dynamic linking for reprogramming wireless sensor networks. In *Proc. of the 4th Int'l Conf. on Embedded networked sensor systems.* ACM, 15–28.

[21] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. 2004. Contiki-a lightweight and flexible operating system for tiny networked sensors. In *Local Computer Networks, 2004. 29th Annual IEEE Int'l Conf. on.* IEEE, 455–462.

[22] Marie Farrell, Matt Luckcuck, and Michael Fisher. 2018. Robotics and Integrated Formal Methods: Necessity Meets Opportunity. In *Integrated Formal Methods.*

[23] FreeRTOS. 2018. The FreeRTOS Kernel. [Online] https://freertos.org.

[24] FreeRTOS. 2022. AWS IoT Over the Air (OTA) Library. https://freertos.org/ota/.

[25] Mario Gleirscher, Simon Foster, and Jim Woodcock. 2019. New Opportunities for Integrated Formal Methods. *ACM Comput. Surv.* 52, 6, Article 117 (oct 2019).

[26] Renata Martins Gomes, Bernhard Aichernig, and Marcel Baunach. 2020. A Formal Modeling Approach for Portable Low-Level OS Functionality. In *Software Engineering and Formal Methods*, Frank de Boer and Antonio Cerone (Eds.). Springer International Publishing, Cham, 155–174.

[27] R. M. Gomes and M. Baunach. 2019. Code Generation from Formal Models for Automatic RTOS Portability. In *2019 IEEE/ACM Int'l Symp. on Code Generation and Optimization (CGO).* 271–272. https://doi.org/10.1109/CGO.2019.8661170

[28] Sean Gully, Vinodh Gopal, Kirk Yap, Wajdi Feghali, Jim Guilford, and Gil Wolrich. 2013. *New Instructions Supporting the Secure Hash Algorithm on Intel Architecture Processors.* Technical Report. Intel Corporation.

[29] Simon Holmbacka, Wictor Lund, Sébastien Lafond, and Johan Lilius. 2013. Lightweight Framework for Runtime Updating of C-Based Software in Embedded Systems. In *5th Workshop on Hot Topics in Software Upgrades.*

[30] Abdulmalik Humayed, Jingqiang Lin, Fengjun Li, and Bo Luo. 2017. Cyber-Physical Systems Security - A Survey. *IEEE Internet Things J.* 4, 6 (2017).

[31] Infineon Techn. AG. 2018. *AURIX™ 32-bit microcontrollers for automotive and industrial applications – Highly integrated and performance optimized.*

[32] Gerwin Klein et al. 2009. seL4: Formal Verification of an OS Kernel. In *Proc. of the ACM SIGOPS 22nd Symp. on Operating Systems Principles (SOSP '09).* 207–220.

[33] Paul Kocher, Jann Horn, Anders Fogh, et al. 2019. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P'19).*

[34] Charalambos Konstantinou, Michail Maniatakos, Fareena Saqib, Shiyan Hu, Jim Plusquellic, and Yier Jin. 2015. Cyber-physical systems: A security perspective. In *20th IEEE European Test Symposium.* 1–8.

[35] Kim G Larsen, Paul Pettersson, and Wang Yi. 1997. UPPAAL in a nutshell. *Int'l Journal on software tools for technology transfer* 1, 1-2 (1997), 134–152.

[36] Rafał Leszczyna et al. 2011. Protecting Industrial Control Systems. Recommendations for Europe and Member States. *The European Union Agency for Network and Information Security (ENISA)* (2011). https://www.enisa.europa.eu/publications/protecting-industrial-control-systems.-recommendations-for-europe-and-member-states

[37] Philip Levis and David Culler. 2002. MatÉ: A Tiny Virtual Machine for Sensor Networks. *SIGARCH Comput. Archit. News* 30, 5 (Oct. 2002), 85–95.

[38] Moritz Lipp, Michael Schwarz, Daniel Gruss, et al. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium.*

[39] Maja Malenko and Marcel Baunach. 2019. Device Driver and System Call Isolation in Embedded Devices. In *22nd Euromicro Conf. on Digital System Design.* IEEE, 283–290. https://doi.org/10.1109/DSD.2019.00049

[40] Maja Malenko and Marcel Baunach. 2019. Hardware/Software Co-designed Peripheral Protection in Embedded Devices. In *IEEE Int'l Conf. on Industrial Cyber Physical Systems*. 790–795.

[41] Maja Malenko and Marcel Baunach. 2019. Hardware/Software Co-designed Security Extensions for Embedded Devices. In *Proc. of the 32nd Int'l Conf. on Architecture of Computing Systems*. 3–14.

[42] Maja Malenko, Leandro Batista Ribeiro, and Marcel Baunach. 2021. Improving Security and Maintainability in Modular Embedded Systems with Hardware Support: Work-in-Progress. In *Proc. of the 2021 Int'l Conf. on Hardware/Software Codesign and System Synthesis*.

[43] Pedro Marrón, Matthias Gauger, Andreas Lachenmann, Daniel Minder, Olga Saukh, and Kurt Rothermel. 2006. FlexCup: A Flexible and Efficient Code Update Mechanism for Sensor Networks. *Europ. Workshop on Wireless Sensor Networks*.

[44] Renata Martins Gomes and Marcel Baunach. 2021. A Study on the Portability of IoT Operating Systems. In *Tagungsband des FG-BS Frühjahrstreffens 2021*. Gesellschaft für Informatik e.V., Bonn. https://doi.org/10.18420/fgbs2021f-01

[45] Renata Martins Gomes, Marcel Baunach, Maja Malenko, Leandro Batista Ribeiro, and Fabian Mauroner. 2017. A Co-Designed RTOS and MCU Concept for Dynamically Composed Embedded Systems. In *13th Workshop on Operating Systems Platforms for Embedded Real-Time Applications*. 41–46.

[46] Fabian Mauroner and Marcel Baunach. 2017. EventIRQ: An Event based and Priority aware IRQ handling for Multi-Tasking Environments. In *Proc. of the 20th Euromicro Conf. on Digital System Design (DSD)*. 102–110.

[47] Fabian Mauroner and Marcel Baunach. 2018. CoStack: Collaborative Stack Sharing for Embedded Real-Time Systems. In *Proc. of the 13th Int. Conference on Systems (ICONS)*.

[48] Fabian Mauroner and Marcel Baunach. 2018. EventQueue: An Event based and Priority aware Interprocess communication for Embedded Systems. In *Proc. of the 13th Int. Symposium on Industrial Embedded Systems*. IEEE.

[49] Fabian Mauroner and Marcel Baunach. 2018. mosartMCU: Multi-Core Operating-System-Aware Real-Time Microcontroller. In *Proc. of the 7th Mediterranean Conference on Embedded Computing (MECO)*.

[50] Fabian Mauroner and Marcel Baunach. 2019. OSARM: A Resource Management Approach for an Operating-System-Aware Microcontroller. *Under review – Journal of Microprocessors and Microsystems (MICPRO)* (2019).

[51] Jami Montgomery. 2004. A model for updating real-time applications. *Real-Time Systems* 27, 2 (2004), 169–189.

[52] Imanol Mugarza, Jorge Parra, and Eduardo Jacob. 2020. Cetratus: A framework for zero downtime secure software updates in safety-critical systems. *Software: Practice and Experience* 50, 8 (2020), 1399–1424.

[53] Michael Muhlheim, Peter A Sandborn, E Quinn, Paul Hunton, Richard Edward Hale, and R England. 2019. *Development of an Obsolescence Cost Model for Nuclear Power Plants*. Technical Report. Oak Ridge National Lab.(ORNL), USA.

[54] Paul Nagele. 2017. Design and Implementation of a I/O specication Tool for MCU Architectures. Bachelor's thesis, EAS group.

[55] Particle. 2022. Device OS. [Online] https://docs.particle.io/tutorials/device-os/device-os/.

[56] Birgit Penzenstadler and Joerg Leuser. 2008. Complying with Law for RE in the Automotive Domain. https://doi.org/10.1109/RELAW.2008.3

[57] Juraj Polakovic, Sebastien Mazare, Jean-Bernard Stefani, and Pierre-Charles David. 2007. Experience with Safe Dynamic Reconfigurations in Component-Based Embedded Systems. 242–257. https://doi.org/10.1007/978-3-540-73551-9_17

[58] Marvin Rausand and Jørn Vatn. 2008. Reliability centred maintenance. In *Complex system maintenance handbook*. Springer, 79–108.

[59] Renesas. 2006. *Renesas 32-Bit RISC Microcomputer SuperH RISC engine Family*. Technical Report. Renesas.

[60] RISC-V Foundation. [n.d.]. RISC-V. https://riscv.org/ [Online] https://riscv.org/.

[61] Peter Ruckebusch et al. 2016. GITAR: Generic extension for Internet-of-Things ARchitectures enabling dynamic updates of network and application modules. *Ad Hoc Networks* 36 (2016), 127 – 151.

[62] Tobias Scheipel, Peter Brungs, and Marcel Baunach. 2021. A Hardware/Software Concept for Partial Logic Updates of Embedded Soft Processors at Runtime. In *Proc. of the 24th Euromicro Conf. on Digital System Design*. 199–207.

[63] Tobias Scheipel, Fabian Mauroner, and Marcel Baunach. 2017. System-Aware Performance Monitoring Unit for RISC-V Architectures. In *Proc. of the 20th Euromicro Conf. on Digital System Design*. 86–93.

[64] Habib Seifzadeh, Ali Asghar Pourhaji Kazem, Mehdi Kargahi, and Ali Movaghar. 2009. A method for dynamic software updating in real-time systems. In *8th IEEE/ACIS Int'l Conf. on Computer and Information Science, 2009*. IEEE, 34–38.

[65] Texas Instruments. [n.d.]. MSP430 ultra-low-power sensing and measurement MCUs. http://www.ti.com/microcontrollers/msp430-ultra-low-power-mcus/overview/overview.html

[66] TinyOS. 2022. TinyOS. http://www.tinyos.net/ [Online] http://www.tinyos.net/.

[67] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte. 2004. The MOLEN polymorphic processor. *IEEE Trans. Comput.* 53, 11 (2004), 1363–1375. https://doi.org/10.1109/TC.2004.104

[68] Andrew Waterman and Krste Asanovic. 2020. *The RISC-V Instruction Set Manual, Volume I: Unprivileged-Level ISA, Version 20191214-draft*. Technical Report. SiFive Inc., UC Berkeley.

[69] Andrew Waterman and Krste Asanovic. 2020. *The RISC-V Instruction Set Manual Volume II: Privileged Architecture Version 1.12-draft*. Technical Report. SiFive Inc., UC Berkeley.

[70] Samuel Weiser, Mario Werner, Ferdinand Brasser, Maja Malenko, Stefan Mangard, and Ahmad-Reza Sadeghi. 2019. TIMBER-V: Tag-Isolated Memory Bringing Fine-grained Enclaves to RISC-V. https://doi.org/10.14722/ndss.2019.23068

[71] WindRiver. 2022. VxWorks. [Online] https://www.windriver.com/products/vxworks.

[72] Zephyr Project. 2022. Porting - Zephyr Project. [Online] https://docs.zephyrproject.org/latest/guides/porting/index.html.

[73] Zephyr Project. 2022. Zephyr Project. [Online] https://www.zephyrproject.org/.

[74] Min Zhang, Kazuhiro Ogata, and Kokichi Futatsugi. 2013. Formalization and Verification of Behavioral Correctness of Dynamic Software Updates. In *Proc. of the 2013 Validation Strategies for Software Evolution (VSSE) Workshop*. 12–23.

7.8 R-IV: *more*MCU: Runtime-reconfigurable RISC-V Platform for Sustainable Embedded Systems

◇ Authors: Tobias Scheipel, Florian Angermair, and Marcel Baunach
◇ Date: August, 2022
◇ DOI: 10.1109/DSD57027.2022.00013
◇ Reference in bibliography: [89]
◇ Presented by myself at the 25th Euromicro Conference on Digital Systems Design (DSD'22), Maspalomas, Spain.
◇ Published in the proceedings of DSD'22 (IEEE).

Summary The *more*MCU platform is designed to make embedded systems more sustainable by using and re-using the available chip area on an Field-Programmable Gate Array (FPGA) at runtime. This way, embedded systems are prepared for long-term hardware adaptations and maintenance. This is done by co-designing an Microcontroller Unit (MCU) based on the RISC-V architecture and an embedded Operating System (OS). To enable reconfiguration of the system, an Field-Programmable Gate Array (FPGA) is used. The MCU includes a reconfiguration controller that can manipulate parts of the pipeline and the peripherals at runtime. The whole hardware is fully supported by the OS. It can change its underlying computing platform without halting or rebooting the system.

Author's Contribution I am the main author of this paper, and I have entirely envisioned, designed, and implemented the concept of the paper myself. Florian Angermair supported the implementation of the concept. My supervisor Marcel Baunach supported me through discussions and in the composition of the paper.

Copyright ©2022 by IEEE. The version included in this thesis is reprinted, with permission, from the published version stated above, which can be accessed in the IEEE Xplore Digital Library:
https://ieeexplore.ieee.org

*more*MCU: A Runtime-reconfigurable RISC-V Platform for Sustainable Embedded Systems

Tobias Scheipel
Institute of Technical Informatics
Graz University of Technology
Graz, Austria
tobias.scheipel@tugraz.at

Florian Angermair
Institute of Technical Informatics
Graz University of Technology
Graz, Austria
florian.angermair@student.tugraz.at

Marcel Baunach
Institute of Technical Informatics
Graz University of Technology
Graz, Austria
baunach@tugraz.at

***Abstract*—As the number of embedded systems continues to grow, so does the amount of disposed electronic devices. This is mainly due to partially or fully outdated hardware, caused by new legal regulations in jurisdiction or cutting-edge features within a new generation of devices or hardware components. As most devices are designed without having long-term maintainability in mind and can be easily replaced without much monetary effort, it is often easier to dispose of them. This throw-away mentality, however, increases the carbon footprint enormously.**

Within this work, we propose a platform that can be used to design future embedded systems in a more sustainable way by preparing them for long-term hardware adaptations. To do so, we aim to make logic updatable and re-usable while the device stays operational. This is achieved by carefully co-designing an operating system and a microcontroller platform with reconfigurable logic. In this paper, we use a RISC-V-based microcontroller running on a field-programmable gate array. The said microcontroller is designed to feature a modular pipeline and replaceable on-chip peripherals alongside a partial reconfiguration controller that can hot-swap parts of the microcontroller while it is running. It is supported by an operating system that handles the reconfiguration as well as functionality emulation, in case it is not (yet) available in hardware. Both the hardware and the software are aware of each other and can manipulate shared data structures for the management of the reconfiguration concept. The experimental evaluation that was carried out on a Artix-7 device shows the proper operation alongside performance measurements and resource utilization of the on-the-fly reconfiguration of a proof-of-concept system without affecting the execution of the remainder of the system.

***Index Terms*—dynamic partial reconfiguration, embedded systems, processor architecture, operating system, hardware software co-design, RISC-V**

I. Introduction and Motivation

In a world where embedded devices become more and more ubiquitous and the number of consumer electronics skyrockets, reusability and long-term maintainability seem to become less and less important. As an example, the Internet of Things (IoT) being a huge global infrastructure is expected to feature a total number of 25 billion connected devices by 2030 [39]. Within this network, individual devices are simply replaced and disposed once an updated version shows up. Thus, the amount of e-waste is expected to rise proportionally to the frequency of new device generations – increasing the carbon footprint enormously. In some cases, hardware deficiencies can be resolved by software workarounds. However, in case this is no longer an option, the affected devices must be replaced by newer ones, even if the performance of these may not necessarily be better. A similar replacement approach also occurs when, e.g., only parts of the computing device become obsolete, since adaptations of the hardware after production or even deployment in the field are sometimes impossible nowadays. This happens regularly in practice, when hardware accelerators of, i.e., a cryptographic algorithm are exploited [16] or other hardware bugs [22] are found during the lifetime of a computing device. These occurrences yield situations where unused logic gates within silicon are still powered, while the affected chip area could be used in a more useful way. Reusability of logic gates can be achieved by using Field-Programmable Gate Arrays (FPGAs), but the concept of having these devices in deployment rather than in prototyping is very underrepresented compared to fixed hard-wired logic in Application-specific Integrated Circuits (ASICs). In such ASIC-based systems, hardware shortcomings are usually addressed by software workarounds rather than fixing the hardware in the field.

To overcome this unsustainable behavior that we see in the field, we derive the following research question for this paper: "*How can we design future embedded systems, so that they can make sustainable use of their underlying hardware?*"

In particular, our overarching goal is to create a platform that allows future embedded systems to stay operational for a long time, while using their limited resources in a more sustainable way. In contrast to currently established processes, where restarts are tolerated, we want to make sure that the operation of the system does not have to be interrupted during hardware update, as this can have safety-critical effects in various usage scenarios [21]. This rebootless reconfiguration gives the ability to retain system states while updating the underlying logic of the system. In order to tackle the research question, we make use of methodological and technological advances in the area of hardware design, as we aim at an increased support and usage of partial reconfiguration features of modern FPGAs (also called Dynamic Partial Reconfiguration – DPR or Dynamic Function eXchange – DFX [51]) at runtime. As we see an increasingly growing demand for

changing the logic of computing devices in future embedded systems during their lifetime [27], we want to raise an awareness for runtime-reconfigurability within these devices. This also allows to fix hardware deficiencies [18], introduce updated hardware accelerators or even add totally new features when adapting to, e.g., new user requirements or legal regulations [25]. And all of this shall be done automatically by the system whilst it is running and performing its intended tasks uninterruptedly.

Special **focus points** in this work are: (**F1**) the runtime-reconfigurable logic design of the entire Microcontroller Unit (MCU) computing platform, and (**F2**) the corresponding support by the Operating System (OS). Both focus points lead to the envisioned **properties** of the hardware and software platform: A computing platform that is designed to have a (**P1**) modular pipeline as well as (**P2**) replaceable on-chip peripherals, which allows (**P3**) partial reconfiguration of these parts at runtime without influencing the execution or rebooting the system. This reconfiguration is (**P4**) fully handled by the OS executed on top, which is designed to run on flexible Instruction Set Architectures (ISAs) in order to facilitate reconfigurable pipelines for its execution. Our design methodology is based on OS/MCU co-design, meaning both the underlying RISC-V-based MCU and the executed OS make use of their awareness of each other.

The remaining paper is organized as follows: In Section II, we present background information and related work that influenced the present approach significantly. The subsequent Section III illustrates the architecture of our envisioned architecture, whereas Section IV shows an evaluation and a proof of concept of our approach. We conclude this paper with Section V, where also an outlook to a possible future of the presented work is stated.

II. Related Work

Within this Section, an overview of related work is given, divided with respect to the four platform properties **P1-4** that were derived in the motivation:

There exists a multitude of processor cores with either a **modular pipeline**, or **replaceable on-chip peripherals**, as this design methodology is a core feature of many soft-core implementations of, e.g., the RISC-V [31] standard. Examples to be named are the PULP platform [38] with their three different RISC-V core implementations CV32E40P [35], CVA6 [53], and Ibex [23], which they compose into different MCU architectures. An implementation with a pure focus on modularity by targeting the underlying hardware description language is VexRiscv [41], whereas other implementations like QuantumRISC [3] or Shakti-T [24] go towards security support in hardware.

All of the mentioned works in the literature aim at a certain modularity, but only at design time. Once the core is deployed, no modifications of the logic are possible. *more*MCU, however, is designed to remain modifiable at runtime.

Regarding **partial reconfiguration at runtime** of logic in general and MCUs in particular, also some works can be found. A general overview, how DPR can be introduced into real-time systems in a beneficial way is provided in [27]. Technology-wise, AC_ICAP [6] introduced an improved way on how to use the Internal Configuration Access Port (ICAP) [47] of modern Xilinx FPGAs [50]. With ICAP, in-system reconfigurations can be achieved without being dependent on external configuration of the FPGA. While this port is used by most runtime-reconfigurable systems running on top those FPGAs, the authors accelerated the management of partial bitstreams and also provided IP cores for the MicroBlaze [48] processor. A reconfiguration controller for real-time systems is RT_ICAP [28], featuring a rich toolchain for the creation of bitstreams. PCAP [19] is specifically designed to allow processors access to reconfiguration capabilities and it acts as a reference design for the Zynq-7000 System on Chip (SoC). For RISC-V processors, a similar controller was proposed in RV-CAP [7]. There, partial reconfiguration management of hardware accelerators that are connected via an AXI-interface [4] can be achieved with a very high throughput and a low resource utilization overhead. The approach also features a driver Application Programming Interface (API) to trigger the reconfiguration from software.

As reconfigurable ISAs form a major part in this work, similar ideas are to be mentioned. The i-Core [8] features application-specific instructions that can be added by either microcode or reconfiguration based on the LEON3 [9] processor. It also features a similar emulation concept that was also shown in [33], where a RISC-V core was used to achieve similar flexibility. In PRISM [5], application code can be transformed into a hardware accelerated co-processor to speed up software execution. The MOLEN polymorphic processor [40] is a very early work, where a general-purpose processor is coupled with reconfigurable logic hardware to provide new instructions similar to a co-processor. The sophisticated toolchain extracts code from the application at compile time and synthesizes it into the reconfigurable fabric for acceleration. Runtime-reconfigurability is not targeted here, however. CHIMAERA [52] goes into a similar direction, where programmable functional units that are coupled to a bus system can act as flexible instruction set. The eMIPS [30] concept, whilst being used for workstations, can load co-processing modules into the pipeline at runtime without halting the system. It uses an approach, where basic blocks of the software are converted into an instruction. At execution, if the accelerator is present, the software implementation is simply skipped in favor to the faster version as a hardware instruction. A quite new approach is to use embedded FPGAs (eFPGAs) in order to create partial reconfigurable SoCs. Usually, those SoCs feature hard-wired ASIC MCUs with one or several embedded FPGA portions for reconfiguration. FlexBex [10] shows an open-source framework for adding eFPGAs to RISC-V processors, where both are directly coupled. This way, instructions can be

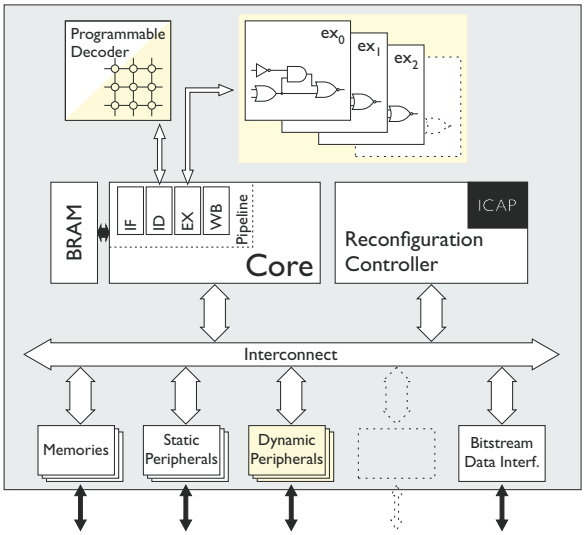


Fig. 1. Hardware block diagram of the runtime-reconfigurable MCU. Yellow fields mark dynamically changeable parts.

easily swapped at runtime. Another approach is illustrated in Arnold [36], where the eFPGA is connected to the processor core via a bus-interfaced memory controller. It is mainly used for sensor data preprocessing for IoT applications. While having advantages of being partial ASIC, both concepts lack flexibility of a normal FPGA when it comes to design-time adaptations. Additionally, it is quite costly to create an ASIC with those features.

The presented partial reconfiguration controllers as well as reconfigurable ISAs represent specific application scenarios, while the approach in *more*MCU is more aimed at general usability.

Lastly, we cover the aspect of **reconfiguration handling by an OS** as well as flexible ISA support. A review and survey of OS concepts for reconfigurable computing is given in [14]. There are some concepts that deal with task offloading and acceleration in hardware, where the functionality of a task or thread is fully implemented and scheduled in an FPGA: R3TOS [1] is a reconfigurable computing operating system for reliable real time applications. It supports hardware tasks that are organized in a 2D grid, named computation regions. Scheduling of the hardware tasks follows a so-called finishing-aware earliest deadline first, where reconfiguration times are included already. The underlying architecture of ReconOS [2] connects the processor and its reconfigurable areas with a bus system. The hardware threads are handled by a hardware interface which is accessed by a delegate software thread in the OS. However, it is more a workstation run-time environment than an operating system, as it runs on top of a host OS. The same applies for HybridOS [17], which is executed in a Linux environment. It provides accelerator virtualization through standardized APIs. The proposed CAP-OS [15], which is an OS for runtime reconfigurable multiprocessor systems deals with the synchronization of multiple ICAPs in a system, where the main focus is on the scheduling of hardware tasks over a Network-on-Chip (NoC). The microkernel of Genode OS [13] runs on a hard-wired core and uses DPR to reconfigure the AXI-connected FPGA for hardware acceleration purposes. This makes DPR available for high-performance computing and safety-critical applications.

All the presented OS concepts handle the reconfiguration for their very special field of application and can access the reconfigurable parts of the underlying system by a generalized connection (e.g., a bus or an interconnect). There is, however, a lack of support for flexible ISAs and peripherals alongside the reconfiguration functionality. Most of the works are also designed for high-performance computing applications, making them not suitable for embedded devices. The envisioned OS concept in *more*MCU overcomes these lacks and provides a very generic support for the handling of partial reconfiguration at runtime.

In contrast to the works presented, this work aims at a holistic co-design of both the self- and runtime-reconfigurable embedded MCU and the OS running on top of it. The OS running on the main processor core is able to reconfigure it directly, and is also able to directly exploit the changed hardware. We explicitly focus on embedded devices with inherent real-time features that can benefit from rebootless reconfiguration. This work is based on our previously published paper on partial logic updates [33].

III. System Architecture and Runtime Reconfiguration

The architecture of the proposed hardware/software co-designed platform overarches both the OS and MCU design as well as reconfigurable logic description. Therefore, our concept covers the hardware design, and great emphasis is placed on the design of a specially tailored embedded OS. Our goal is to provide a holistic platform for embedded system developers to exploit the runtime reconfiguration of an FPGA within an OS/MCU environment. For this purpose, we modify *Smart*OS [32] and a CV32E40P-based [35] MCU called *more*MCU (**m**odular **OS**-aware **r**untime-reconfigurable **e**mbedded MCU). Since the proposed platform is designed for modular use, it is left to the user to decide which parts are appropriate for each use case. Each part of the system architecture is therefore explained individually, divided into hardware and software topics.

The following Sections III-A, III-B, and III-D explain the hardware features of the proposed *more*MCU in detail (w.r.t. properties **P1**, **P2**, and **P3**), whereas Section III-C shows the data structures within the bitstream repository (adding to property **P3**). The subsequent Sections III-E and III-F deal with the implementation of the OS that is executed atop the MCU (w.r.t. platform property **P4**).

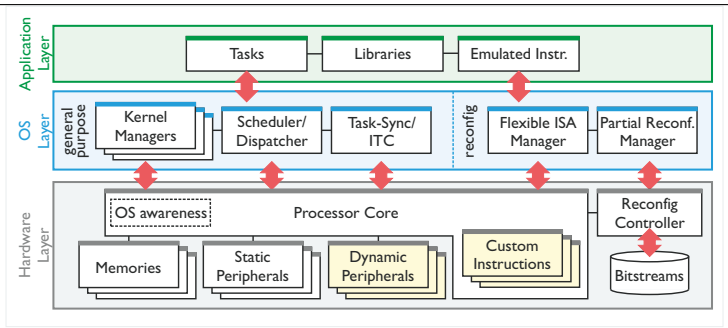


Fig. 2. Layering concept of the overall system architecture from an OS perspective.

A. Pipeline Design to enable flexible ISA

As depicted in Fig. 1, the pipeline of the MCU is designed to consist of a static pipeline portion within the Core. The two pipeline stages Instruction Decode (ID) and Execute (EX) are statically connected to their corresponding dynamic areas, being

- the programmable decoder part to introduce new opcodes to the ID stage, and
- the reconfigurable area for adding new logic divided into several partitions (marked as ex_0, ex_1, ex_2, ... in the upper yellow block) connected to the EX stage.

The programmable decoder forms a part of the actual decoder within the ID stage that is responsible to indicate whether or not an instruction is valid. It consists of a set of registers that map reconfigurable partitions of the EX stage to opcodes. To manage these opcodes, a memory-mapped interface for manipulation can be used from the OS or the Reconfiguration Controller. The corresponding partitions ex_n within the partially reconfigurable EX stage are statically connected to the rest of the pipeline in order to make a dynamic ISA possible. This static interface consists of all those signals necessary and sufficient (e.g., instruction words, operands, internal clock signals, function description flags or even memory access channels) to exchange data between the static and dynamic parts of the pipeline. The signals must be well-defined, since they are responsible to provide a consistent interface. To avoid illegal states within the processor core, it is of utmost importance to keep coherence between the programmable decoder and the reconfigurable area. As the OS and the Reconfiguration Controller can manipulate these two parts of the MCU, they are also in charge of the housekeeping. Whenever a new functionality is added to the core, **first** the logic in the EX stage must be reconfigured; **second**, the reprogramming of the ID stage must happen; and vice versa when stripping functionality. This ensures that the illegal instruction signal of the pipeline can still be used correctly by the OS to emulate instructions (cf. Section III-A) in case they are not already or no longer present in hardware. After the updating procedure, a newly added instruction is directly added to the pipeline as a native functionality with no execution time overhead compared to originally available instructions, resulting in a performance gain over the software emulated versions.

B. MCU Design to enable flexible Peripherals

The design process for supporting flexible on-chip peripherals is very similar, although significantly less critical than changing parts of the pipeline. As these peripherals need an interface to the peripheral bus rather than to the pipeline, these interfaces can be put on the edge of the core. In our case, the access to on-chip peripherals is designed to be memory mapped via a Wishbone (WB) bus [26], also acting as a processor interconnect (cf. Fig. 1). To enable partial reconfiguration, it must only be ensured that the bus interfaces to the dynamic peripheral partitions are static within the FPGA. Further interface-related design decisions can be made by the development engineer. Not even the peripheral interface is fixed but can be designed according to the application requirements. However, possible invalid states within the peripheral access must be considered individually.

C. Data Structures and Bitstream Repository

As the Reconfiguration Controller (cf. Section III-D) makes use of the data structures that are located in the bitstream repository (cf. Fig. 2), the relevant design decisions are pointed out in the following. The bitstream repository contains partial bitstreams [51] for every runtime-reconfigurable

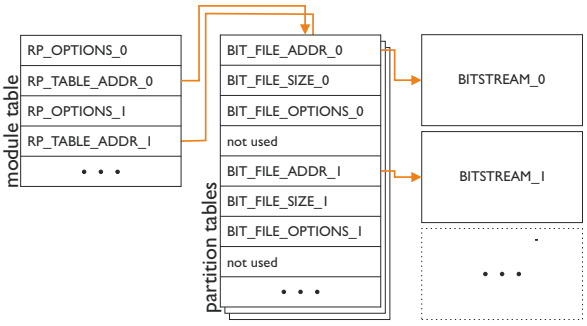


Fig. 3. Data structure within the bitstream repository.

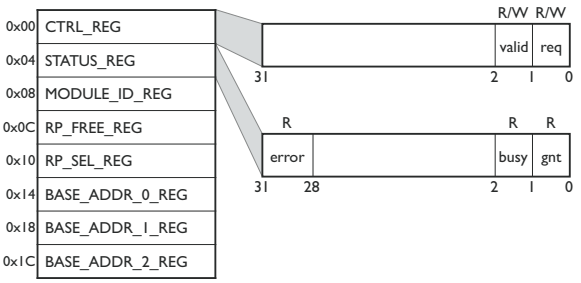


Fig. 4. Control and status registers of the Reconfiguration Controller.

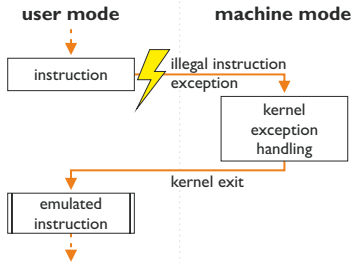


Fig. 5. Software emulation flow of an unknown instruction.

module (instruction or peripheral) and every reconfigurable partition (in the dynamic areas marked in yellow in Fig. 1) on the *more*MCU. To make the management easier, the well-defined structure depicted in Fig. 3 is used. It consists of a module table that holds reconfigurable partition (RP) placement options (`RP_OPTIONS_n`) and a table address (`RP_TABLE_ADDR_n`) for every RP. The table address points to the corresponding partition table that holds the relevant information about all the possible partial bitstreams that fit into this partition. The bitstream information holds a bitstream address (`BIT_FILE_ADDR_n`) pointing to the actual partial bitstream that can be loaded into a reconfigurable partition alongside its size (`BIT_FILE_SIZE_n`) and some options (`BIT_FILE_OPTIONS_n`) like compression state for every module applicable to a certain partition. To be prepared for future adaptations, a "not used" memory word is also included. As the Reconfiguration Controller supports decompression while programming partial bitstreams, a very simple compression algorithm is used in order to save space on the memory device. The memory device that contains the bitstream repository in *more*MCU is connected via a bitstream data interface (cf. Fig. 1). This way, different memory technologies (Static Random-Access Memory – SRAM, Flash-memory, etc.) can be used interchangeably.

D. Runtime Reconfiguration Controller

The Reconfiguration Controller is responsible for partially reconfiguring *more*MCU at runtime. Partial bitstreams are taken from the bitstream repository, are decompressed, and programmed to the corresponding partition of the configuration memory[1] via the ICAP of the FPGA on-the-fly. This means that all clocks remain active, the current logic on the FPGA remains operational, as well as the OS continues to be executed. The Reconfiguration Controller acts in parallel to the remaining MCU and thus does not influence its operation. It is aware of all the relevant data structures that are used by the OS as well as the current state of the FPGA and its reconfigurable partitions. It can actively decide whether a piece of logic can be placed in a dynamically reconfigurable partition or not (due to, e.g., current occupation) and is also responsible to take care of the programmable decoder (cf. Section III-A) in case of a pipeline reconfiguration. Whenever a partial reconfiguration happens at runtime, all cells in the corresponding dynamic area are reset and newly connected to follow the intended, updated hardware description. The static cells, however, stay untouched.

The control and status registers and their relative addresses depicted in Fig. 4 form the interface that the Reconfiguration Controller exposes on the MCU interconnect. It can be used in a memory-mapped environment to control the behavior of the controller by a piece of software. The required program flow used by the OS is explained in Section III-F. In addition to the interconnect bus interface, there is also a direct port that can be used to perform reconfiguration from another hardware component. When both interfaces are used, differentiation between the requested bitstream repository base addresses can be made with the registers `BASE_ADDR_n_REG`. For use in software, it is sufficient to set only `BASE_ADDR_0_REG` to the starting address of the bitstream repository explained in Section III-C.

E. Flexible ISA Handling

To enable support for flexible ISAs, *Smart*OS must be prepared for initially unknown instructions within the application code. To do so, the OS features a Flexible ISA Manager module (cf. Fig. 2). Whenever an instruction is not natively available in the ISA, an illegal instruction exception occurs (cf. Fig. 5) within the core. While usually this exception indicates corrupted software, the OS makes use of it and triggers the execution of the missing instruction's emulation functionality. If the application wants the OS to emulate an instruction using this technique, then it must be registered beforehand (see below). On the event of the exception, the kernel is entered, and the binary encoding of the "illegal" instruction is used to extract important information like opcode, operands, etc. Having this information, the OS looks up a valid corresponding emulated function and prepares it for execution in the context of the calling task. Upon kernel exit and restoration of the calling task's context, the registered functionality is executed in user mode and the task can continue normally. If the lookup fails, however, the instruction is considered illegal indeed, and task termination is proceeded. For the sake of completeness, it needs to be noted

[1]The configuration memory – as its name indicates – holds the current configuration of all logic cells of the FPGA.

that if an instruction is already available, no further action of the operating system is required, and the said instruction will be executed as any other instruction in the ISA.

From an application perspective, the OS provides a set of API functions to register instructions for emulation in case they are not natively supported, as can also be seen in Fig. 2. There is also some profiling in place to capture information about those emulated instructions for the OS in order to facilitate the decision, whether the system would profit from hardware acceleration or not. The actual reconfiguration is then triggered by the Partial Reconfiguration Manager in the OS layer. Our approach also ensures that standard-compliant RISC-V compilers can still be used, since the emulation handling only uses well-defined behavior (cf. illegal instruction exceptions [44]) to which every RISC-V core must adhere anyway. Conversely, this also means that the entire software stack can be executed on any non-altered, standard-compliant RISC-V MCU without restrictions. In any case, access to the Reconfiguration Controller is obviously only possible if it is also available within the MCU.

F. Runtime Reconfiguration Handling from Software

After all information is gathered, the OS triggers the actual reconfiguration of the logic. For this purpose the Partial Reconfiguration Manager (cf. Fig. 2) of the OS indicates to the Reconfiguration Controller in hardware whenever a custom instruction or dynamic on-chip peripheral is to be introduced to or stripped from the system at runtime. This *Smart*OS decision making can be based on different metrics like instruction call count, duration, etc. and can be adapted according to the use case of the system. Once it decides to reconfigure *more*MCU, it uses the interface of the Reconfiguration Controller (cf. Fig. 4) as follows: At the beginning, the controller must be requested. This is done by setting the `req` bit in the control register `CTRL_REG`. Access is granted, once `gnt` in the status register `STATUS_REG` is set. Subsequently the index of the selected runtime-reconfigurable module must be set in `MODULE_ID_REG` alongside the preferred reconfigurable partition index in `RP_FREE_REG`. By setting the `valid` bit in the control register, reconfiguration is requested to start. Whenever `busy` is reset again, the process is completed or has failed, according to the `error` flags. `RP_SEL_REG` shows the actual selected reconfigurable partition – also in case reconfiguration was triggered by a hardware device. Since these are all memory-mapped registers, the typical protection mechanisms can be used to avoid unauthorized access.

IV. Proof of Concept and Evaluation

For the evaluation of our proposed OS/MCU platform, we run *more*MCU at 50 MHz clock speed on a Nexys4-DDR board [12] that features a Xilinx XC7A100T FPGA [50]. Within our use case, we connect the UART of the MCU to a USB socket, and General-Purpose Input/Output (GPIO) pins to on-board LEDs – both used for debugging and status messages. For timing measurement with a PicoScope 2205 MSO [29], further GPIOs come into play. We use Xilinx Vivado 2020.1 WebPACK running on Ubuntu 18.04.6 LTS for both hardware simulation and synthesis.

TABLE I
Resource Utilization of *more*MCU.

	Module	Resource Utilization LUT	FF	BRAM
full *more*MCU	top†	8310	3509	17
	Core	6626	2539	0
	EX stage	1132	118	0
	Decoder	165	21	0
	Programmable Decoder	20	21	0
	Peripheral Controller (inc. Mem.)	312	151	16
	Reconf. Controller + WB interface	513	494	1
	Bitstream Data Interface	577	289	0
w/o extens.	top†	6857	2641	16
	Core	6509	2418	0
	EX stage	1067	118	0
	Decoder	124	0	0
	Peripheral Controller (inc. Mem.)	279	151	16

†In top, the core is connected with all on-chip peripherals and memories.

*more*MCU executes *Smart*OS including the concept-specific extensions, with a test software showing the functionalities in a multitasking application. Features of the OS as well as the built-in Performance Monitoring Unit (PMU) [34] of the MCU are used to monitor and measure different runtime aspects.

As the proposed platform can be used in a modular way, the following configuration is chosen for the evaluation: We use *more*MCU with three RPs for custom instructions (each with 2400 Lookup Tables – LUTs and 4800 Flip-flops – FFs) as well as two RPs for dynamic peripherals (each with 400 LUTs and 800 FFs) – both connected to the main WB interface. The bitstream repository is located in a dedicated, already available on-board 16 MiB QSPI flash memory and is interfaced by the bitstream data interface on the FPGA side. Memory-wise, 32 KB RAM and ROM is allocated, respectively; located in Block RAM (BRAM) cells directly within the FPGA.

A. moreMCU Resource Utilization

Table I shows the resource utilization on the FPGA of the top modules of *more*MCU in the configuration explained above compared to a *more*MCU without the proposed extensions. In addition, some important submodules of the corresponding top module are explicitly described. Modules that are not shown are irrelevant for the comparison because they are equal in both cases (such as the interconnect and some peripherals).

It can be seen that the extensions within the RISC-V core only impose 1.7 % more LUTs and 5 % more FFs; expectedly within the EX stage (+6 % LUTs) and the decoder (+35 % LUTs and 21 total FFs) of the pipeline, whereas the latter adds a total of 20 LUTs and all of the 21 FFs within the newly-introduced programmable decoder. The remaining difference is attributed to its interfacing. The peripheral controller which is designed to connect the core and the memory-mapped components (e.g., on-chip peripherals, interconnects) also adds a total number 33 LUTs compared to the non-extended MCU. This is due to the fact that the interfacing to the dynamic peripherals must be done in here

TABLE II
COMPARISON OF RESOURCE UTILIZATION AND CORE FEATURES OF DIFFERENT RECONFIGURATION CONTROLLERS IN LITERATURE.

Reconfiguration Controller	Resource Utilization LUT	FF	BRAM	Interface	Software
ZyCAP [43]	620	806	0	AXI	drv
AC_ICAP [6]	1286	1193	22	PLB/direct	-
RT-ICAP [28]	190	88	0	direct	drv
RV-CAP [7]	420	909	0	AXI	drv
D²PR [11]	249	112	0	direct	-
Vipin et al. [42]	586	672	8	PLB/AXI	-
ICAP-I [20]	177	303	0	direct	-
DPRM [37]	109	77	0	PLB/XPS/dir.	-
AXI_HWICAP [46]	546	741	2	AXI	-
XPS_HWICAP [45]	741	745	3	XPS	-
PRC [49]	1171	1203	0	AXI	-
***more*MCU-RC**	513	494	1	WB/direct	OS

as well. Overall, it can be said that the resource overhead w.r.t. LUTs is $21\,\%$ and w.r.t. FFs is $33\,\%$. The partitioning of the MCU has little to no effect on the resource utilization.

B. Reconfiguration Controller Resource Utilization and Comparison to Related Work

As there are already several reconfiguration controllers for different systems available, Table II gives a comparison of how the Reconfiguration Controller of the proposed *more*MCU (*more*MCU-RC) performs. In addition to resource utilization, comparison criteria include the interface that is provided and the software support, indicating how easy it is to incorporate the solution into existing systems.

From the numbers, one can see that there are some implementations using less resources than the proposed one. However, most of them do not include the resources of the interface controller into their numbers, or they only feature a direct, register-based access interface without hardware overhead in this regard. From the bus-connected implementations, only RV-CAP [7] has similar numbers (but significantly more FFs), even though no OS-awareness is included in this solution. From a software point of view, only a few implementations feature simple driver concepts (drv), and not a single one is fully backed by an OS implementation – *more*MCU's Reconfiguration Controller is. Its Wishbone bus interface is also unique among the solutions, where AXI dominates, followed by Processor Local Bus (PLB).

C. Proof of Concept

The evaluation application software running ontop of *Smart*OS registers three instructions for emulation and hardware acceleration. Very simple instructions are used, but they are sufficient to show how the concept works. The OS measures both the number of calls as well as the duration of the execution of the emulated functions. For demonstration and evaluation purposes, introducing and stripping of custom instructions in hardware is triggered periodically. The same applies to the dynamic peripherals. The most interesting metrics for the evaluation are the reconfiguration time and the OS overhead, as well as the performance benefit from hardware acceleration by introducing instructions in hardware.

TABLE III
RECONFIGURATION TIME, OS OVERHEAD AND RUNTIME ADVANTAGES.

	Measurement	clock cycles[a]	time
reconf.	reconf. of dynamic peripherals	1897838	$37.957\,\mathrm{ms}$[b]
	reconf. of custom instructions	5621104	$112.422\,\mathrm{ms}$[b]
	reconf. OS overhead	16	$320\,\mathrm{ns}$[c]
emul.	instruction emulation OS overhead	83	$1.660\,\mu\mathrm{s}$[c]
	traditional software	659	$13.180\,\mu\mathrm{s}$[c]
	OS emulated instruction	692	$13.840\,\mu\mathrm{s}$[c]
	native instruction	1	$20\,\mathrm{ns}$[c]

[a] PMU measurement.
[b] PicoScope measurement.
[c] calculated values using $f_{clk} = 50MHz$ and $T_{clk} = 20ns$.

As Table III shows, the reconfiguration of a dynamic peripheral partition takes an average total time of $37.957\,\mathrm{ms}$ in average, whereas the reconfiguration of a custom instruction needs $112.422\,\mathrm{ms}$ in average. The bottleneck here is the QSPI flash that can only work at $25\,\mathrm{MHz}$ and must transfer the bitstream data serially. As the reconfiguration process happens in parallel to the normal operation of the MCU, no execution delay is introduced to regular code execution. The start of the reconfiguration introduces an OS management overhead of $320\,\mathrm{ns}$ or $16\,\mathrm{cycles}$. To avoid influencing other tasks, a reconfiguration is only started whenever the system is in idle mode. For instruction emulation handling, the OS overhead is $83\,\mathrm{cycles}$. We evaluated a use case with a weighted moving average algorithm, that can be executed in software in a "traditional" way within $659\,\mathrm{cycles}$. When choosing the emulated approach, it can thus be executed in $692\,\mathrm{cycles}$. The native execution as a custom instruction improves the cycle amount drastically down to $1\,\mathrm{cycle}$. OS overhead and traditional execution do not sum up exactly to the emulated version, as function calls also add a certain amount of overhead to the functionality.

V. CONCLUSION AND FUTURE WORK

In this paper, we showcase *more*MCU, a OS/MCU platform based on RISC-V, where runtime reconfiguration is used to make embedded systems more sustainable. This is achieved by providing certain properties necessary for the design of a sustainable system in addition to the basic functionalities of the platform. These properties deal with modular MCU design to support dynamic on-chip peripherals and custom instructions, partial reconfiguration at runtime, and OS handling of the whole process. While partial reconfiguration approaches are already introduced in various, mostly high-performance computing applications, they are not yet established in the field of embedded systems. Due to sustainability goals introduced in Section I, we see a shift in thinking in this regard for embedded systems as well.

The work presented enables system developers to make their embedded systems usable for longer, which is achieved by the co-design of an MCU and an OS. The platform thus enables the underlying hardware of an embedded system to be handled more sustainably, as the existing chip area can be used and re-used for various purposes at runtime. The

simple usability of an OS goes hand in hand with the runtime reconfigurability of the MCU, which is also shown in the evaluation. The resource overhead is within a reasonable range, and the runtime performance benefits outweigh the costs. The proposed platform can be used on all FPGAs that feature a ICAP component. In the future, we will perform further investigations on the replacement strategies within the OS and the Reconfiguration Controller.

References

[1] A. Adetomi *et al.*, "R3TOS-Based Integrated Modular Space Avionics for On-Board Real-Time Data Processing," in *NASA/ESA Conf. on Adaptive Hardware and Systems*, 2018, pp. 1–8.

[2] A. Agne, M. Happe, A. Keller, E. Lübbers, B. Plattner, M. Platzner, and C. Plessl, "ReconOS: An Operating System Approach for Reconfigurable Computing," *IEEE Micro*, vol. 34, no. 1, pp. 60–71, 2014.

[3] E. Alkim *et al.*, "ISA Extensions for Finite Field Arithmetic - Accelerating Kyber and NewHope on RISC-V," Cryptology ePrint Arch. 2020.

[4] ARM Ltd., *AMBA AXI and ACE Protocol Specification*, 2017.

[5] P. M. Athanas and H. F. Silverman, "Processor reconfiguration through instruction-set metamorphosis," *Computer*, vol. 26, no. 3, 1993.

[6] L. Cardona and C. Ferrer, "AC-ICAP: A flexible high speed ICAP controller," *Int'l Journal of Reconfigurable Computing*, pp. 1–15, 2015.

[7] N. Charaf, A. Kamaleldin, M. Thuemmler, and D. Göhringer, "RV-CAP: Enabling Dynamic Partial Reconfiguration for FPGA-Based RISC-V System-on-Chip," 05 2021.

[8] M. Damschen, M. Rapp, L. Bauer, and J. Henkel, *i-Core: A Runtime-Reconfigurable Processor Platform for Cyber-Physical Systems*, 01 2020, pp. 1–36.

[9] M. Daněk, L. Kafka, L. Kohout, J. Sýkora, and R. Bartosiński, *The LEON3 Processor*. New York, NY: Springer New York, 2013.

[10] N. Dao, A. Attwood, B. Healy, and D. Koch, "FlexBex: A RISC-V with a Reconfigurable Instruction Extension," in *2020 International Conference on Field-Programmable Technology (ICFPT)*, 2020, pp. 190–195.

[11] S. Di Carlo *et al.*, "A portable open-source controller for safe Dynamic Partial Reconfiguration on Xilinx FPGAs," in *Int'l Conf. on Field Programmable Logic and Applications*, 2015.

[12] *Nexys 4 DDR Reference Manual*, Digilent, Inc. [Online]. Available: https://digilent.com/reference/programmable-logic/nexys-4-ddr/reference-manual

[13] A. Dörflinger, M. Albers, B. Fiethe, and H. Michalik, "Hardware Acceleration in Genode OS Using Dynamic Partial Reconfiguration," in *Architecture of Computing Systems*. Springer, 2018, pp. 283–293.

[14] M. Eckert, D. Meyer, J. Haase, and B. Klauer, "Operating System Concepts for Reconfigurable Computing: Review and Survey," *Int'l Journal of Reconfigurable Computing*, vol. 2016, p. 2478907, Nov 2016.

[15] D. Göhringer, M. Hübner, E. Nguepi Zeutebouo, and J. Becker, "Operating System for Runtime Reconfigurable Multiprocessor Systems," *Int'l Journal of Reconfigurable Computing*, vol. 2011, p. 121353, Apr 2011.

[16] S. Gully, V. Gopal, K. Yap, W. Feghali, J. Guilford, and G. Wolrich, "New Instructions Supporting the Secure Hash Algorithm on Intel Architecture Processors," Tech. Rep., 2013.

[17] J. H. Kelm and S. S. Lumetta, "HybridOS: Runtime Support for Reconfigurable Accelerators," in *Proc. of the 16th Int'l ACM/SIGDA Symp. on Field Programmable Gate Arrays*, 2008, p. 212–221.

[18] P. Kocher, J. Horn, A. Fogh *et al.*, "Spectre attacks: Exploiting speculative execution," in *40th IEEE Symp. on Security and Privacy*, 2019.

[19] C. Kohn, "Partial reconfiguration of a hardware accelerator on zynq-7000 all programmable soc devices," *Xilinx, XAPP1159 (v1. 0)*, 2013.

[20] V. Lai and O. Diessel, "ICAP-I: A reusable interface for the internal reconfiguration of Xilinx FPGAs," in *Int'l Conf Conference on Field-Programmable Technology*, 2009, pp. 357–360.

[21] R. Leszczyna *et al.*, "Protecting Industrial Control Systems. Recommendations for Europe and Member States," *The European Union Agency for Network and Information Security (ENISA)*, 2011.

[22] M. Lipp, M. Schwarz, D. Gruss *et al.*, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium*, 2018.

[23] lowRISC, "Ibex: An embedded 32 bit RISC-V CPU core," lowRISC, Tech. Rep., 2022, [online] https://ibex-core.readthedocs.io/en/latest/.

[24] A. Menon, S. Murugan, C. Rebeiro, N. Gala, and K. Veezhinathan, "Shakti-T: A RISC-V Processor with Light Weight Security Extensions," in *Proc. of the Hardware and Architectural Support for Security and Privacy*, 2017.

[25] B. Penzenstadler and J. Leuser, "Complying with Law for RE in the Automotive Domain," 09 2008.

[26] W. D. Peterson, *Wishbone B4, WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores*, Silicore Corp., 2010. [Online]. Available: http://cdn.opencores.org/downloads/wbspec_b4.pdf

[27] L. Pezzarossa, A. T. Kristensen, M. Schoeberl, and J. Sparsø, "Can real-time systems benefit from dynamic partial reconfiguration?" in *IEEE Nordic Circuits and Systems Conference*, 2017, pp. 1–6.

[28] L. Pezzarossa, M. Schoeberl, and J. Sparsø, "A Controller for Dynamic Partial Reconfiguration in FPGA-Based Real-Time Systems," in *IEEE 20th Int'l Symp on Real-Time Distributed Computing*, 2017, pp. 92–100.

[29] *PicoScope 2205 MSO Mixed Signal Oscilloscope*, Pico Technology, 2016. [Online]. Available: https://www.picotech.com/download/datasheets/PicoScope2205MSODatasheet-en.pdf

[30] R. Pittman, N. Lynch, A. Forin, and N. Pittman, "eMIPS, A Dynamically Extensible Processor," 2006.

[31] RISC-V Foundation, "RISC-V," [Online] https://riscv.org/.

[32] T. Scheipel, L. Batista Ribeiro, T. Sagaster, and M. Baunach, "SmartOS: An OS Architecture for Sustainable Embedded Systems," in *Tagungsband des FG-BS Frühjahrstreffens*, Hamburg, 2022.

[33] T. Scheipel, P. Brungs, and M. Baunach, "A Hardware/Software Concept for Partial Logic Updates of Embedded Soft Processors at Runtime," in *Proc. of the 24th Euromicro Conf. on Digital System Design*, 2021.

[34] T. Scheipel, F. Mauroner, and M. Baunach, "System-Aware Performance Monitoring Unit for RISC-V Architectures," in *Proc. of the 20th Euromicro Conference on Digital System Design*, Aug 2017, pp. 86–93.

[35] D. Schiavone, "CV32E40P User Manual," 2022, [online] https://docs.openhwgroup.org/projects/cv32e40p-user-manual/.

[36] P. D. Schiavone *et al.*, "Arnold: an eFPGA-Augmented RISC-V SoC for Flexible and Low-Power IoT End-Nodes," *CoRR*, 2020. [Online]. Available: https://arxiv.org/abs/2006.14256

[37] J. Tarrillo, F. A. Escobar, F. L. Kastensmidt, and C. Valderrama, "Dynamic partial reconfiguration manager," in *IEEE 5th Latin American Symp. on Circuits and Systems*, 2014, pp. 1–4.

[38] A. Traber *et al.*, "PULPino: A small single-core RISC-V SoC," 2015, http://iis-projects.ee.ethz.ch/images/d/d0/Pulpino_poster_riscv2015.pdf.

[39] L. S. Vailshery, "Number of Internet of Things (IoT) connected devices worldwide from 2019 to 2030," Mar. 2022, [online]https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide.

[40] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte, "The MOLEN polymorphic processor," *IEEE Transactions on Computers*, vol. 53, no. 11, pp. 1363–1375, 2004.

[41] T. Verbeure, "The VexRiscV CPU – A New Way to Design," Dec. 2018. [Online]. Available: https://tomverbeure.github.io/rtl/2018/12/06/The-VexRiscV-CPU-A-New-Way-To-Design.html

[42] K. Vipin and S. A. Fahmy, "A high speed open source controller for FPGA Partial Reconfiguration," in *Int'l Conf. on Field-Programmable Technology*, 2012, pp. 61–66.

[43] ——, "ZyCAP: Efficient Partial Reconfiguration Management on the Xilinx Zynq," *IEEE Embedded Systems Letters*, vol. 6, no. 3, 2014.

[44] A. Waterman and K. Asanovic, "The RISC-V Instruction Set Manual Volume II: Privileged Architecture Version 1.12-draft," SiFive Inc., UC Berkeley, Tech. Rep., Jul. 2020.

[45] *LogiCORE IP XPS HWICAP (v5.01a) DS586*, Xilinx, Inc., Jun. 2011.

[46] *AXI HWICAP v3.0 PG134*, Xilinx, Inc., Oct. 2016.

[47] *7 Series FPGAs Configuration User Guide*, UG470 (v1.13.1) ed., Xilinx, Inc., Aug. 2018, [online] https://docs.xilinx.com/v/u/en-US/ug470_7Series_Config.

[48] *MicroBlaze Processor Reference Guide*, UG984 (v2018.2) ed., Xilinx, Inc., Jun. 2018.

[49] *Partial Reconfiguration Controller v1.3 PG193*, Xilinx, Inc., Apr. 2018.

[50] *7 Series FPGAs: Overview DS180 (v2.6.1)*, Xilinx, Inc., 2020, [online] https://docs.xilinx.com/v/u/en-US/ds180_7Series_Overview.

[51] *Dynamic Function eXchange (UG909, v2019.2)*, Xilinx, Inc., Jan. 2020. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_2/ug909-vivado-partial-reconfiguration.pdf

[52] Z. A. Ye, A. Moshovos, S. Hauck, and P. Banerjee, "CHIMAERA: a high-performance architecture with a tightly-coupled reconfigurable functional unit," in *27th Int'l Symp. on Computer Architecture*, 2000.

[53] F. Zaruba and L. Benini, "The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology," *IEEE Trans. on VLSI Systems*, vol. 27, no. 11, Nov. 2019.

Bibliography

[1] R. N. Charette, "This Car runs on Code," *IEEE Spectrum*, 2009. [Online]. Available: https://spectrum.ieee.org/transportation/systems/this-car-runs-on-code → [p2]

[2] J. Manyika, M. Chui, P. Bisson, J. Woetzel, R. Dobbs, J. Bughin, and D. Aharon, *Unlocking the potential of the Internet of Things*, McKinsey Global Institute, Jun. 2015. → [p2]

[3] J. Schäuffele and T. Zurawka, *Automotive Software Engineering*, ser. ATZ/MTZ-Fachbuch. Springer Fachmedien Wiesbaden, 2016. → [p2]

[4] L. S. Vailshery, "Number of Internet of Things (IoT) connected devices worldwide from 2019 to 2030," Mar. 2022, [online]https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide. → [p2]

[5] Q. H. Dang, "Secure Hash Standard," National Institute of Standards and Technology, Tech. Rep., Jul. 2015. → [p3]

[6] S. Gully, V. Gopal, K. Yap, W. Feghali, J. Guilford, and G. Wolrich, "New Instructions Supporting the Secure Hash Algorithm on Intel Architecture Processors," Intel Corporation, Tech. Rep., 2013. → [p3]

[7] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, M. Hamburg, and R. Strackx, "Meltdown: Reading Kernel Memory from User Space," in *Proceedings of the 27th USENIX Security Symposium*, 2018, pp. 973–990. → [p3]

[8] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre Attacks: Exploiting Speculative Execution," in *Proceedings of the 40th IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 1–19. → [p3]

[9] C. Boano, K. Römer, R. Bloem, K. Witrisal, M. Baunach, and M. Horn, "Dependability for the Internet of Things: From dependable networking in harsh environments to a holistic view on dependability," *e&i Elektrotechnik und Informationstechnik*, vol. 133, no. 7, pp. 304–309, 2016. → [p3]

[10] B. Penzenstadler and J. Leuser, "Complying with Law for RE in the Automotive Domain," in *Proceedings of the 1st International Workshop on Requirements Engineering and Law (RELAW)*, Sep. 2008, pp. 11–15. → [p3]

[11] R. Leszczyna *et al.*, "Protecting Industrial Control Systems. Recommendations for Europe and Member States," *The European Union Agency for Network and Information Security (ENISA)*, 2011. [Online]. Available: https://www.enisa.europa.eu/publications/protecting-industrial-control-systems.-recommendations-for-europe-and-member-states → [p3]

[12] L. Pezzarossa, A. T. Kristensen, M. Schoeberl, and J. Sparsø, "Can real-time systems benefit from dynamic partial reconfiguration?" in *Proceedings of the IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC)*, 2017, pp. 1–6. → [p3], [p26]

[13] P. Marwedel, *Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems, and the Internet of Things*, 3rd ed., N. D. Dutt, G. Martin, and P. Marwedel, Eds. Springer Nature Switzerland, 2021. → [p9], [p11], [p15]

[14] D. Giusto, A. Iera, G. Morabito, and L. Atzori, Eds., *The Internet of Things – 20th Tyrrhenian Workshop on Digital Communications*. Springer, 2010. → [p9]

[15] P. Barry and P. Crowley, *Modern Embedded Computing: Designing Connected, Pervasive, Media-Rich Systems*, T. Green and R. Day, Eds. Morgan Kaufmann, 2012. → [p9], [p12], [p14], [p15]

[16] T. Noergaard, *Embedded Systems Architecture: A Comprehensive Guide for Engineers and Programmers*, 2nd ed. Newnes, 2013. → [p10], [p11], [p13], [p15]

[17] M. Staron, *Automotive Software Architectures: An Introduction*, 2nd ed. Springer, 2021. → [p11]

[18] M. Baunach, "Advances in Distributed Real-Time Sensor/Actuator Systems Operation," Dissertation, University of Würzburg, Germany, 2012. [Online]. Available: http://opus.bibliothek.uni-wuerzburg.de/frontdoor/index/index/docId/6429 → [p11]

[19] T. Scheipel, L. Batista Ribeiro, T. Sagaster, and M. Baunach, "SmartOS: An OS Architecture for Sustainable Embedded Systems," in *Tagungsband des FG-BS Frühjahrstreffens*. Hamburg, Germany: Gesellschaft für Informatik e.V., Mar. 2022, pp. 1–10. → [p11], [p60], [p155]

[20] W. D. Peterson, *Wishbone B4, WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores*, Silicore Corp., 2010. [Online]. Available: http://cdn.opencores.org/downloads/wbspec_b4.pdf → [p13], [p63]

[21] *AMBA AXI and ACE Protocol Specification*, ARM Ltd., 2017. → [p13]

[22] *UM10204: I2C-bus specification and user manual*, NXP Semiconductors, Inc., Apr. 2014, rev. 6. → [p13]

[23] S. C. Hill, J. Jelemensky, M. R. Heene, S. E. Groves, and D. N. Debrito, "Queued Serial Peripheral Interface for Use in a Data Processing System," US Patent US4 816 996, 1989. → [p13]

[24] *IEEE Standard for Ethernet*, IEEE, 2018. → [p13]

[25] *Universal Serial Bus 3.2 Specification*, Apple Inc., Hewlett-Packard Inc., Intel Corporation, Microsoft Corporation, Renesas Corporation, STMicroelectronics, Texas Instruments, Jun. 2022, revision 1.1. → [p13]

[26] *Universal Asynchronous Receiver/Transmitter (UART)*, Philips Semiconductors, Aug. 2006. → [p13]

[27] Intel, "Intel Architecture Instruction Set Extensions and Future Features," Jun. 2022. → [p15]

[28] Arm Ltd., "Arm A64 Instruction set for A-profile Architecture," 2022. → [p15]

[29] A. Waterman and K. Asanovic, "The RISC-V Instruction Set Manual, Volume I: Unprivileged-Level ISA, Version 20191214-draft," SiFive Inc., UC Berkeley, Tech. Rep., Jul. 2022. → [p15], [p24], [p67], [p70], [p71], [p75]

[30] K. Diefendorff, R. Oehler, and R. Hochsprung, "Evolution of the PowerPC architecture," *IEEE Micro*, vol. 14, no. 2, pp. 34–49, 1994. → [p15]

[31] C. Price, "MIPS IV Instruction Set, Revision 3.2," MIPS Technologies, Inc., Tech. Rep., Sep. 1995. → [p15]

[32] *SH-4 CPU Core Architecture*, Hitachi, 2002. [Online]. Available: http://www.st.com/content/ccc/resource/technical/document/user_manual/69/23/ed/be/9b/ed/44/da/CD00147165.pdf/files/CD00147165.pdf/jcr:content/translations/en.CD00147165.pdf → [p15]

[33] *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*, Hewlett-Packard, 1994. [Online]. Available: https://parisc.wiki.kernel.org/images-parisc/6/68/Pa11_acd.pdf → [p15]

[34] *The SPARC Architecture Manual Version 8*, SPARC International, Inc, 1992. [Online]. Available: http://www.gaisler.com/doc/sparcv8.pdf → [p15]

[35] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 6th ed., S. Merken and N. McFadden, Eds. Morgan Kaufman, 2019. → [p15]

[36] A. Waterman, K. Asanovic, and J. Hauser, "The RISC-V Instruction Set Manual Volume II: Privileged Architecture, Version 20211203," SiFive Inc., UC Berkeley, Tech. Rep., Jul. 2022. → [p15], [p32], [p70], [p71], [p72], [p73]

[37] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz *et al.*, "The Rocket Chip Generator," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, vol. 4, 2016. → [p16]

[38] D. Schiavone, "CV32E40P User Manual," Open HW Group, Tech. Rep., 2022. [Online]. Available: https://docs.openhwgroup.org/projects/cv32e40p-user-manual/ → [p16], [p25], [p60], [p80]

[39] M. B. Gokhale and P. S. Graham, *Reconfigurable Computing: Accelerating Computation with Field-Programmable Gate Arrays*. Springer Publishing, 2005. → [p16]

[40] *7 Series FPGAs: Overview DS180 (v2.6.1)*, Xilinx, Inc., Sep. 2020. [Online]. Available: https://docs.xilinx.com/v/u/en-US/ds180_7Series_Overview → [p16], [p26], [p75], [p84], [p88], [p192]

[41] S. Hauck and A. DeHon, *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. Morgan Kaufmann Publishers Inc., 2008. → [p17]

[42] *Dynamic Function eXchange (UG909, v2019.2)*, Xilinx, Inc., Jan. 2020. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_2/ug909-vivado-partial-reconfiguration.pdf → [p17], [p63]

[43] L. Cardona, "Dynamic Partial Reconfiguration in FPGAs for the Design and Evaluation of Critical Systems," Ph.D. dissertation, University of Bologna, 2016. → [p17]

[44] G. M. Swinkels and L. Hafer, "Schematic generation with an expert system," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 9, no. 12, pp. 1289–1306, Dec. 1990. → [p19]

[45] B. Singh, D. O'Riordan, B. G. Arsintescu, A. Goel, and D. R. Deshpande, "System and method for circuit schematic generation," US Patent US7 917 877B2, 2011. → [p20]

[46] devicetree.org, *Devicetree Specification Release v0.4-rc1*, Nov. 2021. → [p20]

[47] Linux Kernel Organization, Inc., "The Linux Kernel Archives," Aug. 2022. [Online]. Available: https://www.kernel.org/ → [p20]

[48] A. Kouba, J. Navratil, and B. Hnilička, "Engine Control using a Real-Time 1D Engine Model," in *VPC – Simulation und Test 2015*, J. Liebl and C. Beidl, Eds. Wiesbaden: Springer Fachmedien Wiesbaden, 2018, pp. 295–309. → [p20]

[49] Infineon Technologies AG, "TC1797 – 32-Bit Single-Chip Microcontroller," 2014. → [p20]

[50] B. Eichberger, E. Unger, and M. Oswald, "Design of a Versatile Rapid Prototyping Engine Management System," in *Proceedings of the FISITA World Automotive Congress*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 135–142. → [p20]

[51] Infineon Technologies AG, "TC1796 – 32-Bit Single-Chip Microcontroller," 2007. → [p20]

[52] Infineon Technologies AG, "TC1798 – 32-Bit Single-Chip Microcontroller," 2014. → [p20]

[53] M. Broy, I. H. Kruger, A. Pretschner, and C. Salzmann, "Engineering Automotive Software," *Proceedings of the IEEE*, vol. 95, no. 2, pp. 356–373, 2007. → [p21]

[54] E. Sikora, B. Tenbergen, and K. Pohl, "Requirements Engineering for Embedded Systems: An Investigation of Industry Needs," in *Requirements Engineering: Foundation for Software Quality*, D. Berry and X. Franch, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 151–165. → [p21]

[55] T. Pereira, D. Albuquerque, A. Sousa, F. Alencar, and J. Castro, "Towards a metamodel for a requirements engineering process of embedded systems," in *Proceedings of the 6th Brazilian Symposium on Computing Systems Engineering (SBESC)*, 2016, pp. 93–100. → [p21]

[56] H. Elmqvist, F. E. Cellier, and M. Otter, "Object-oriented modeling of hybrid systems," -, 1993. → [p21]

[57] S. Konrad and B. H. C. Cheng, "Requirements patterns for embedded systems," in *Proceedings of the IEEE Joint International Conference on Requirements Engineering*, 2002, pp. 127–136. → [p21]

[58] S. Konrad, B. H. C. Cheng, and L. A. Campbell, "Object analysis patterns for embedded systems," *IEEE Transactions on Software Engineering*, vol. 30, no. 12, pp. 970–992, 2004. → [p21]

[59] J. A. Stankovic, Ruiqing Zhu, R. Poornalingam, Chenyang Lu, Zhendong Yu, M. Humphrey, and B. Ellis, "VEST: an aspect-based composition tool for real-time systems," in *Proceedings of the 9th Real-Time and Embedded Technology and Applications Symposium (RTTAS)*. IEEE, 2003, pp. 58–69. → [p22]

[60] J. Schnerr, O. Bringmann, A. Viehl, and W. Rosenstiel, "High-performance Timing Simulation of Embedded Software," in *Proceedings of the 45th Annual Design Automation Conference (DAC)*, Anaheim, California, Jun. 2008, pp. 290–295. → [p22]

[61] B. Schommer, C. Cullmann, G. Gebhard, X. Leroy, M. Schmidt, and S. Wegener, "Embedded Program Annotations for WCET Analysis," in *Proceedings of the 18th International Workshop on Worst-Case Execution Time Analysis (WCET)*, Barcelona, Spain, Jul. 2018. → [p22]

[62] S. Chakravarty, Z. Zhao, and A. Gerstlauer, "Automated, Retargetable Back-annotation for Host Compiled Performance and Power Modeling," in *Proceedings of the 9th International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Piscataway, NJ, USA, 2013, pp. 1–10. → [p22]

[63] A. D. Pimentel, "Exploring Exploration: A Tutorial Introduction to Embedded Systems Design Space Exploration," *IEEE Design & Test*, vol. 34, no. 1, pp. 77–90, Feb. 2017. → [p23]

[64] S. Künzli, *Efficient design space exploration for embedded systems*. ETH Zurich, 2006, vol. 81. → [p23]

[65] F. Herrera, H. Posadas, P. Peñil, E. Villar, F. Ferrero, R. Valencia, and G. Palermo, "The COMPLEX methodology for UML/MARTE Modeling and design space exploration of embedded systems," *Journal of Systems Architecture*, vol. 60, no. 1, pp. 55–78, 2014. → [p23]

[66] T. Saxena and G. Karsai, "A Meta-Framework for Design Space Exploration," in *Proceedings of the 18th IEEE International Conference and Workshops on Engineering of Computer-Based Systems*, Apr. 2011, pp. 71–80. → [p23]

[67] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack, "MiniZinc: Towards a Standard CP Modelling Language," in *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, C. Bessière, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 529–543. → [p24]

[68] N.-F. Zhou, H. Kjellerstrand, and J. Fruman, *Constraint Solving and Planning with Picat*. Springer, Nov. 2015. → [p24]

[69] K. Kuchcinski, "Constraint programming in embedded systems design: Considered helpful," *Microprocessors and Microsystems*, vol. 69, pp. 24–34, 2019. → [p24]

[70] A. Traber *et al.*, "PULPino: A small single-core RISC-V SoC," 2015. [Online]. Available: http://iis-projects.ee.ethz.ch/images/d/d0/Pulpino_poster_riscv2015.pdf → [p25]

[71] A. Traber, "RI5CY Core: Datasheet," ETH Zurich, Tech. Rep., Feb. 2019. [Online]. Available: https://www.pulp-platform.org/docs/ri5cy_user_manual.pdf → [p25]

[72] lowRISC, "Ibex: An embedded 32 bit RISC-V CPU core," lowRISC, Tech. Rep., 2022. [Online]. Available: https://ibex-core.readthedocs.io/en/latest/ → [p25]

[73] F. Zaruba and L. Benini, "The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 11, pp. 2629–2640, Nov. 2019. → [p25], [p27]

[74] T. Verbeure, "The VexRiscV CPU – A New Way to Design," Dec. 2018. [Online]. Available: https://tomverbeure.github.io/rtl/2018/12/06/The-VexRiscV-CPU-A-New-Way-To-Design.html → [p25]

[75] E. Alkim, H. Evkan, N. Lahr, R. Niederhagen, and R. Petri, "ISA Extensions for Finite Field Arithmetic - Accelerating Kyber and NewHope on RISC-V," *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2020. → [p25]

[76] A. Menon, S. Murugan, C. Rebeiro, N. Gala, and K. Veezhinathan, "Shakti-T: A RISC-V Processor with Light Weight Security Extensions," in *Proceedings of the Hardware and Architectural Support for Security and Privacy (HASP)*, Jun. 2017, pp. 1–8. → [p25]

[77] K. Papadimitriou, A. Dollas, and S. Hauck, "Performance of Partial Reconfiguration in FPGA Systems: A Survey and a Cost Model," *ACM Transactions on Reconfigurable Technology Systems*, vol. 4, no. 4, pp. 1–24, Dec. 2011. → [p26]

[78] K. Vipin and S. A. Fahmy, "ZyCAP: Efficient Partial Reconfiguration Management on the Xilinx Zynq," *IEEE Embedded Systems Letters*, vol. 6, no. 3, pp. 41–44, Sep. 2014. → [p26], [p85]

[79] L. Cardona and C. Ferrer, "AC-ICAP: A flexible high speed ICAP controller," *International Journal of Reconfigurable Computing*, pp. 1–15, 2015. → [p26], [p85]

[80] L. Pezzarossa, M. Schoeberl, and J. Sparsø, "A Controller for Dynamic Partial Reconfiguration in FPGA-Based Real-Time Systems," in *Proceedings of the 20th International Symposium on Real-Time Distributed Computing (ISORC)*, 2017, pp. 92–100. → [p26], [p27], [p85]

[81] N. Charaf, A. Kamaleldin, M. Thuemmler, and D. Göhringer, "RV-CAP: Enabling Dynamic Partial Reconfiguration for FPGA-Based RISC-V System-on-Chip," in *Proceedings of the International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2021, pp. 172–179. → [p26], [p27], [p85], [p86]

[82] S. Di Carlo, P. Prinetto, P. Trotta, and J. Andersson, "A portable open-source controller for safe Dynamic Partial Reconfiguration on Xilinx FPGAs," in *Proceedings of the 25th International Conference on Field Programmable Logic and Applications (FPL)*, Sep. 2015, pp. 1–4. → [p26], [p85]

[83] K. Vipin and S. A. Fahmy, "A high speed open source controller for FPGA Partial Reconfiguration," in *Proceedings of the International Conference on Field-Programmable Technology (FPT)*, 2012, pp. 61–66. → [p26], [p85]

[84] V. Lai and O. Diessel, "ICAP-I: A reusable interface for the internal reconfiguration of Xilinx FPGAs," in *Proceedings of the International Conference on Field-Programmable Technology (FPT)*, Dec. 2009, pp. 357–360. → [p26], [p85]

[85] J. Tarrillo, F. A. Escobar, F. L. Kastensmidt, and C. Valderrama, "Dynamic partial reconfiguration manager," in *Proceedings of the 5th Latin American Symposium on Circuits and Systems (LASCAS)*, 2014, pp. 1–4. → [p26], [p85]

[86] *AXI HWICAP v3.0 PG134*, Xilinx, Inc., Oct. 2016. → [p26], [p85]

[87] *LogiCORE IP XPS HWICAP (v5.01a) DS586*, Xilinx, Inc., Jun. 2011. → [p26], [p85]

[88] *Partial Reconfiguration Controller v1.3 PG193*, Xilinx, Inc., Apr. 2018. → [p26], [p85]

[89] T. Scheipel, F. Angermair, and M. Baunach, "moreMCU: A Runtime-reconfigurable RISC-V Platform for Sustainable Embedded Systems," in *Proceedings of the 25th Euromicro Conference on Digital System Design (DSD)*, Euromicro. Maspalomas, Spain: IEEE, Aug. 2022, pp. 24–31. → [p26], [p167]

[90] *MicroBlaze Processor Reference Guide UG984 (v2018.2)*, Xilinx, Inc., Jun. 2018. → [p26]

[91] *LogiCORE IP ProcessorLocal Bus (PLB) v4.6 (v1.05a)*, Xilinx, Inc., Sep. 2010. → [p26]

[92] *LogiCORE IP Fast Simplex Link (FSL) V20 Bus (v2.11c)*, Xilinx, Inc., Apr. 2010. → [p26]

[93] *AMBA AXI Protocol v1.0*, ARM Ltd., 2004. [Online]. Available: http://mazsola.iit.uni-miskolc.hu/~drdani/docs_arm/AMBAaxi.pdf → [p26]

[94] C. Kohn, "Partial Reconfiguration of a Hardware Accelerator on Zynq-7000 All Programmable SoC Devices," *Xilinx, XAPP1159 (v1. 0)*, 2013. → [p27]

[95] L. H. Crockett, R. A. Elliot, M. A. Enderwitz, and R. W. Stewart, *The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable SoC*. UK: Strathclyde Academic Media, 2014. → [p27], [p31]

[96] C. Beckhoff, D. Koch, and J. Torresen, "Go Ahead: A Partial Reconfiguration Framework," in *Proceedings of the 20th International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2012, pp. 37–44. → [p27]

[97] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte, "The MOLEN polymorphic processor," *IEEE Transactions on Computers*, vol. 53, no. 11, pp. 1363–1375, Nov. 2004. → [p28]

[98] P. M. Athanas and H. F. Silverman, "Processor reconfiguration through instruction-set metamorphosis," *Computer*, vol. 26, no. 3, pp. 11–18, Mar. 1993. → [p28]

[99] Z. A. Ye, A. Moshovos, S. Hauck, and P. Banerjee, "CHIMAERA: a high-performance architecture with a tightly-coupled reconfigurable functional unit," in *Proceedings of 27th International Symposium on Computer Architecture (ISCA)*, 2000, pp. 225–235. → [p28]

[100] R. Pittman, N. Lynch, A. Forin, and N. Pittman, "eMIPS, A Dynamically Extensible Processor," Microsoft Research, Tech. Rep., Oct. 2006. → [p28]

[101] M. Damschen, M. Rapp, L. Bauer, and J. Henkel, *i-Core: A Runtime-Reconfigurable Processor Platform for Cyber-Physical Systems*. Springer, Jan. 2020. → [p28]

[102] M. Daněk, L. Kafka, L. Kohout, J. Sýkora, and R. Bartosiński, *The LEON3 Processor*. Springer, 2013. → [p29], [p32]

[103] J. R. G. Ordaz and D. Koch, "A Soft Dual-Processor System with a Partially Run-Time Reconfigurable Shared 128-Bit SIMD Engine," in *Proceedings of the 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2018, pp. 1–8. → [p29]

[104] P. Schiavone, D. Rossi, A. D. Mauro, F. K. Gurkaynak, T. Saxe, M. Wang, K. Yap, and L. Benini, "Arnold: an eFPGA-Augmented RISC-V SoC for Flexible and Low-Power IoT End-Nodes," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2020. [Online]. Available: https://arxiv.org/abs/2006.14256 → [p29]

[105] N. Dao, A. Attwood, B. Healy, and D. Koch, "FlexBex: A RISC-V with a Reconfigurable Instruction Extension," in *Proceedings of the International Conference on Field-Programmable Technology (FPT)*, 2020, pp. 190–195. → [p29]

[106] B. Neumann, T. Sydow, H. Blume, and T. Noll, "Application domain specific embedded FPGAs for flexible ISA-extension of ASIPs," *Journal of Signal Processing Systems*, vol. 53, pp. 129–143, Nov. 2008. → [p29]

[107] M. Eckert, D. Meyer, J. Haase, and B. Klauer, "Operating System Concepts for Reconfigurable Computing: Review and Survey," *International Journal of Reconfigurable Computing*, vol. 2016, pp. 1–11, Nov. 2016. → [p30]

[108] A. Agne, M. Happe, A. Keller, E. Lübbers, B. Plattner, M. Platzner, and C. Plessl, "ReconOS: An Operating System Approach for Reconfigurable Computing," *IEEE Micro*, vol. 34, no. 1, pp. 60–71, Jan. 2014. → [p30]

[109] J. H. Kelm and S. S. Lumetta, "HybridOS: Runtime Support for Reconfigurable Accelerators," in *Proceedings of the ACM/SIGDA 16th International Symposium on Field Programmable Gate Arrays (FPGA)*, Feb. 2008, pp. 212–221. → [p30]

[110] A. Adetomi, G. Enemali, X. Iturbe, T. Arslan, and D. Keymeulen, "R3TOS-Based Integrated Modular Space Avionics for On-Board Real-Time Data Processing," in *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, Aug. 2018, pp. 1–8. → [p30]

[111] A. Dörflinger, M. Albers, B. Fiethe, and H. Michalik, "Hardware Acceleration in Genode OS Using Dynamic Partial Reconfiguration," in *Proceedings of the 31st International Conference on Architecture of Computing Systems (ARCS)*. Springer, 2018, pp. 283–293. → [p31]

[112] *Virtex-5 Family Overview DS100 (v5.1)*, Xilinx, Inc., Aug. 2015. → [p31]

[113] D. Göhringer, M. Hübner, E. Nguepi Zeutebouo, and J. Becker, "Operating System for Runtime Reconfigurable Multiprocessor Systems," *International Journal of Reconfigurable Computing*, vol. 2011, pp. 1–16, Mar. 2011. → [p31]

[114] *Using EDK to Run Xilkernel on aMicroBlaze Processor UG758 (v7.0)*, Xilinx, Inc., Aug. 2010. → [p31]

[115] D. Andrews, R. Sass, E. Anderson, J. Agron, W. Peck, J. Stevens, F. Baijot, and E. Komp, "The Case for High Level Programming Models for Reconfigurable Computers," in *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, Jan. 2006, pp. 21–32. → [p31]

[116] B. Nichols, D. Buttlar, and J. P. Farrell, *Pthreads programming*. O'Reilly & Associates, Inc., 1996. → [p31]

[117] D. She, Y. He, and H. Corporaal, "An Energy-Efficient Method of Supporting Flexible Special Instructions in an Embedded Processor with Compact ISA," *ACM Transactions on Architecture and Code Optimization*, vol. 10, no. 3, pp. 1–25, Sep. 2013. → [p31]

[118] N. Bergmann, J. Williams, J. Han, and Y. Chen, "A Process Model for Hardware Modules in Reconfigurable System-on-Chip," in *19th International Conference on Architecture of Computing Systems (ARCS)*, 2006, pp. 205–214. → [p31]

[119] Intel, "Intel Microarchitecture Codename Nehalem Performance Monitoring Unit Programming Guide," Intel, Tech. Rep. 253669-061US, Dec. 2010. → [p32]

[120] ARM, "Cortex-A5 Technical Reference Manual," ARM, Tech. Rep., 2016. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0433c/DDI0433C_cortex_a5_trm.pdf → [p32]

[121] B. Sprunt, "The basics of performance-monitoring hardware," *IEEE Micro*, vol. 22, no. 4, pp. 64–71, Jul. 2002. → [p32]

[122] D. Patil, P. Kharat, and A. K. Gupta, "Study of Performance Counters and Profiling Tools to Monitor Performance of Application," *Proceedings of the 21st IRF International Conference*, pp. 45–49, 2015. → [p32]

[123] A. Singh, A. Buchke, and Y.-H. Lee, "A Study of Performance Monitoring Unit, perf and perf_events subsystem," -, 2012. → [p32]

[124] J. A. Ambrose, V. Cassisi, D. Murphy, T. Li, D. Jayasinghe, and S. Parameswaran, "Scalable performance monitoring of application specific multiprocessor Systems-on-Chip," in *Proceedings of the 8th IEEE International Conference on Industrial and Information Systems (ICIInfS)*, Dec. 2013, pp. 315–320. → [p32]

[125] N. Ho, P. Kaufmann, and M. Platzner, "A hardware/software infrastructure for performance monitoring on LEON3 multicore platforms," in *Proceedings of the 24th International Conference on Field Programmable Logic and Applications (FPL)*, Sep. 2014, pp. 1–4. → [p32]

[126] AUTOSAR, "AUTOSAR Classic Platform Release 4.4.0," Oct. 2018. → [p36]

[127] R. Martins Gomes, M. Baunach, M. Malenko, L. Batista Ribeiro, and F. Mauroner, "A Co-Designed RTOS and MCU Concept for Dynamically Composed Embedded Systems," in *Proceedings of the 13th Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, 2017, pp. 41–46. → [p37]

[128] ECMA International, *ECMA-404: The JSON Data Interchange Syntax*, 2nd ed., Dec. 2017. → [p39]

[129] W3C, *Extensible Markup Language (XML) 1.0 (Fifth Edition)*, Nov. 2008. [Online]. Available: https://www.w3.org/TR/2008/REC-xml-20081126/ → [p40]

[130] Texas Instruments, *MSP430F5529 LaunchPad™ Development Kit (MSP--EXP430F5529LP)*, Apr. 2017. → [p39]

[131] Adafruit Industries, *Micro SD Card Breakout Board Tutorial*, Jan. 2019. [Online]. Available: https://cdn-learn.adafruit.com/downloads/pdf/adafruit-micro-sd-breakout-board-card-tutorial.pdf → [p39]

[132] Autodesk, Inc., "EAGLE." [Online]. Available: https://www.autodesk.com/products/eagle/ → [p40], [p47]

[133] *ISO/IEC 9075-1:2016 Information technology – Database languages – SQL – Part 1: Framework (SQL/Framework)*, International Organisation for Standardization Std. → [p46]

[134] Autodesk, Inc., *EAGLE XML Data Structure 9.1.0*, 2018. → [p47]

[135] Texas Instruments, "MSP430 ultra-low-power sensing and measurement MCUs." [Online]. Available: http://www.ti.com/microcontrollers/msp430-ultra-low-power-mcus/overview/overview.html → [p50]

[136] Texas Instruments, *Ultra-Small, Low-Power,16-Bit Analog-to-Digital Converter with Internal Reference*, Oct. 2009, [retrieved: Nov, 2019]. [Online]. Available: http://www.ti.com/lit/ds/symlink/ads1114.pdf → [p53]

[137] Microchip, *12-Bit Digital-to-Analog Converter with EEPROM Memory in SOT-23-6*, 2009, [retrieved: Nov, 2019]. [Online]. Available: https://cdn-shop.adafruit.com/datasheets/mcp4725.pdf → [p53]

[138] H. Ishiwara, M. Okuyama, and Y. Arimoto, *Ferroelectric Random Access Memories: Fundamentals and Applications* . Springer Science & Business Media, 2004, vol. 93. → [p53]

[139] Fujitsu Semiconductor, *64KBit SPIMB85RS64V*, 2013, [retrieved: Nov, 2019]. [Online]. Available: https://cdn-shop.adafruit.com/datasheets/MB85RS64V-DS501-00015-4v0-E.pdf → [p53]

[140] M. Malenko, L. Batista Ribeiro, and M. Baunach, "Protection and Relocation Extension for RISC-V," in *Sixth Workshop on Computer Architecture Research with RISC-V*, Jun. 2022. → [p71]

[141] *Basys 3 Reference Manual*, Digilent, Inc. [Online]. Available: https://reference.digilentinc.com/reference/programmable-logic/basys-3/reference-manual → [p75]

[142] *PicoScope 2205 MSO Mixed Signal Oscilloscope*, Pico Technology, 2016. [Online]. Available: https://www.picotech.com/download/datasheets/PicoScope2205MSODatasheet-en.pdf → [p75], [p86]

[143] *Infiniium S-Series*, Keysight Technologies, 2019. [Online]. Available: https://literature.cdn.keysight.com/litweb/pdf/5991-3904EN.pdf → [p75], [p79]

[144] *Nexys 4 DDR Reference Manual*, Digilent, Inc. [Online]. Available: https://digilent.com/reference/programmable-logic/nexys-4-ddr/reference-manual → [p84]

[145] V. Sivashanmugam, T. Scheipel, M. Baunach, and B. Adabala, "A Conversion Concept for a Legacy Software Model towards AUTOSAR Compliance," in *Proceedings of the 1st International Conference on Computing and Applied Engineering (ICCAE)*, vol. 9. Goa, India: IJCRT, Aug. 2021. → [p96]

[146] G. Fiala, T. Scheipel, W. Neuwirth, and M. Baunach, "FPGA-Based Debugging with Dynamic Signal Selection at Run-Time," in *Proceedings of the 17th Workshop on Automotive Software Engineering (ASE)*. Klagenfurt, Austria: RWTH Aachen, Jan. 2020. → [p96]

[147] M. Baunach, R. Martins Gomes, M. Malenko, F. Mauroner, L. Batista Ribeiro, and T. Scheipel, "Smart mobility of the future – a challenge for embedded automotive systems," *e & i Elektrotechnik und Informationstechnik*, pp. 304–308, Jun 2018. [Online]. Available: https://doi.org/10.1007/s00502-018-0623-6 → [p96]

[148] T. Scheipel, F. Mauroner, and M. Baunach, "Einheit zur anwendungsbezogenen Leistungsmessung für die RISC-V-Architektur," in *Logistik und Echtzeit*, W. A. Halang and H. Unger, Eds. Boppard, Germany: Springer Berlin Heidelberg, Nov. 2017, pp. 69–78. → [p96]

[149] T. Scheipel and M. Baunach, "papagenoPCB: An Automated Printed Circuit Board Generation Approach for Embedded Systems Prototyping," in *Proceedings of the 14th International Conference on Systems (ICONS)*. Valencia, Spain: IARIA, Mar. 2019, pp. 20–25. → [p99]

[150] T. Scheipel and M. Baunach, "Towards an Automated Printed Circuit Board Generation Concept for Embedded Systems," *International Journal on Advances in Systems and Measurements (SysMea)*, vol. 12, no. 3&4, pp. 236–246, Dec. 2019. [Online]. Available: http://www.iariajournals.org/systems_and_measurements/ → [p107]

[151] T. Scheipel and M. Baunach, "papagenoX: Generation of Electronics and Logic for Embedded Systems from Application Software," in *Proceedings of the 9th International Conference on Sensor Networks (SENSORNETS)*, INSTICC. Valetta, Malta: SciTePress, Feb. 2020, pp. 136–141. → [p119]

[152] T. Scheipel and M. Baunach, "papagenoReQ: Generation of Embedded Systems from Application Code Requirements," in *Proceedings of the 3rd International Conference on Electrical, Communication, and Computer Engineering (ICECCE)*. Kuala Lumpur, Malaysia: IEEE, Jun. 2021, pp. 1–6. → [p127]

[153] T. Scheipel, F. Mauroner, and M. Baunach, "System-Aware Performance Monitoring Unit for RISC-V Architectures," in *Proceedings of the 20th Euromicro Conference on Digital System Design (DSD)*, Euromicro. Vienna, Austria: IEEE, Aug 2017, pp. 86–93. → [p135]

[154] T. Scheipel, P. Brungs, and M. Baunach, "A Hardware/Software Concept for Partial Logic Updates of Embedded Soft Processors at Runtime," in *Proceedings of the 24th Euromicro Conference on Digital System Design (DSD)*, Euromicro. Palermo, Italy: IEEE, Sep. 2021, pp. 199–207. → [p145]