

## Targeting static and dynamic workloads with a reconfigurable VLIW processor

Hoozemans, Joost

**DOI**

[10.4233/uuid:c3be6373-f4f2-4865-b3f0-750bfb17871e](https://doi.org/10.4233/uuid:c3be6373-f4f2-4865-b3f0-750bfb17871e)

**Publication date**

2018

**Document Version**

Final published version

**Citation (APA)**

Hoozemans, J. (2018). *Targeting static and dynamic workloads with a reconfigurable VLIW processor*. [Dissertation (TU Delft), Delft University of Technology]. <https://doi.org/10.4233/uuid:c3be6373-f4f2-4865-b3f0-750bfb17871e>

**Important note**

To cite this publication, please use the final published version (if applicable).  
Please check the document version above.

**Copyright**

Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

**Takedown policy**

Please contact us and provide details if you believe this document breaches copyrights.  
We will remove access to the work immediately and investigate your claim.

# **Targeting static and dynamic workloads with a reconfigurable VLIW processor**



# **Targeting static and dynamic workloads with a reconfigurable VLIW processor**

## **Proefschrift**

ter verkrijging van de graad van doctor  
aan de Technische Universiteit Delft,  
op gezag van de Rector Magnificus prof. dr. ir. T.H.J.J. van der Hagen,  
voorzitter van het College voor Promoties,  
in het openbaar te verdedigen op donderdag 21 juni 2018 om 10:00 uur

door

**Joost Johannes HOOZEMANS**

Master of Science in Computer Engineering,  
Technische Universiteit Delft,  
geboren te Delft, Nederland.

Dit proefschrift is goedgekeurd door de

promotor: prof. dr. K.L.M. Bertels

copromotor: dr. ir. J.S.S.M. Wong

Samenstelling promotiecommissie:

Rector Magnificus,  
Prof. dr. K.L.M. Bertels  
Dr. ir. J.S.S.M. Wong

voorzitter  
Technische Universiteit Delft, promotor  
Technische Universiteit Delft, copromotor

*Onafhankelijke leden:*

Prof. dr. H.P. Hofstee  
Prof. dr. H. Corporaal  
Prof. dr. -ing. M. Hübner  
Prof. dr. ir. C. Vuik  
Prof. dr. ir. S. Hamdioui

Technische Universiteit Delft  
Technische Universiteit Eindhoven  
Ruhr-Universität Bochum, Duitsland  
Technische Universiteit Delft  
Technische Universiteit Delft, reservelid

*Overige leden:*

Dr. ir. Z. Al-Ars

Technische Universiteit Delft



This work has been supported by the ALMARVI European Artemis project nr. 621439.

**Keywords:** Computer architecture, VLIW processor, dynamically reconfigurable, polymorphic, embedded computing, FPGA, streaming

**Printed by:** Ipskamp printing

**Front & Back:** Conceptual diagram created by Bart Kallenbach for the Dig-it! research exhibition.  $\rho$ -VEX logo by Thijs van As, adapted by Jeroen van Straten.

Copyright © 2018 by J.J. Hoozemans

ISBN 978-94-6366-049-5

An electronic version of this dissertation is available at

<http://repository.tudelft.nl/>.

# Contents

<b>Summary</b>	<b>ix</b>
<b>Samenvatting</b>	<b>xi</b>
<b>Acknowledgements</b>	<b>xiii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Dynamic workloads call for dynamic processors . . . . .	2
1.2 Leveraging design-time customization for highly static workloads . . . . .	3
1.3 Embedded execution platforms . . . . .	4
1.3.1 General-purpose . . . . .	4
1.3.2 Application-Specific . . . . .	4
1.3.3 Reconfigurable . . . . .	5
1.4 Workload analysis . . . . .	6
1.4.1 Requirements . . . . .	7
1.4.2 Parallelism . . . . .	8
1.4.3 Code characteristics . . . . .	11
1.5 Problem formulation and scope . . . . .	12
1.5.1 Software: Workloads . . . . .	12
1.5.2 Hardware: Reconfigurable processors . . . . .	15
1.5.3 Scheduling: Tasks and processor configurations . . . . .	16
1.6 Proposed platform: a design-time configurable, run-time parametrizable VLIW processor . . . . .	17
1.6.1 Static (design-time) reconfigurability . . . . .	17
1.6.2 Dynamic (run-time) parameterization . . . . .	18
1.6.3 Environment . . . . .	21
1.6.4 Platform overview . . . . .	23
1.7 Approach . . . . .	23
1.7.1 Modeling & Simulation . . . . .	26
1.7.2 Using FPGA technology . . . . .	26
1.7.3 Code characterization method . . . . .	27
1.7.4 A runtime for scheduling tasks and configurations . . . . .	27
1.7.5 Workload generation . . . . .	28
1.7.6 Benchmarks . . . . .	28
1.8 Contributions and thesis outline . . . . .	29

<b>Part 1 - Static workloads, statically reconfigurable platform</b>	<b>33</b>
<b>2 Using VLIW softcore processors for image processing</b>	<b>35</b>
2.1 Introduction . . . . .	36
2.2 Related work . . . . .	37
2.3 The $\rho$ -VEX platform . . . . .	38
2.3.1 The VEX system: ISA and toolchain. . . . .	38
2.3.2 The $\rho$ -VEX VLIW processor . . . . .	38
2.4 Image processing applications . . . . .	39
2.5 Results and discussion . . . . .	40
2.6 Conclusions . . . . .	42
<b>3 A streaming FPGA computation fabric</b>	<b>43</b>
3.1 Introduction . . . . .	44
3.2 Related work . . . . .	45
3.3 Implementation . . . . .	45
3.3.1 Processing elements . . . . .	45
3.3.2 Memory hierarchy . . . . .	46
3.3.3 Platform . . . . .	47
3.4 Experimental setup . . . . .	47
3.5 Evaluation results . . . . .	48
3.5.1 Resource utilization . . . . .	48
3.5.2 Image processing performance . . . . .	49
3.6 Conclusions . . . . .	50
<b>4 Frame-Based Programming, Stream-Based Processing</b>	<b>53</b>
4.1 Introduction . . . . .	54
4.2 Related work . . . . .	55
4.2.1 Optimizing/accelerating image processing workloads . . . . .	55
4.2.2 FPGA acceleration . . . . .	55
4.2.3 FPGA overlays . . . . .	57
4.2.4 FPGA image processing overlays . . . . .	57
4.2.5 Integration frameworks . . . . .	57
4.3 Approach . . . . .	58
4.3.1 OpenCL's view on parallel computing . . . . .	58
4.3.2 OpenCL memory model . . . . .	58
4.3.3 Streaming data and OpenCL . . . . .	59
4.3.4 OpenCL data architecture . . . . .	60
4.4 Implementation - Hardware . . . . .	61
4.4.1 Processing element . . . . .	62
4.4.2 Memory structure . . . . .	62
4.4.3 Interfaces . . . . .	64
4.5 Implementation - Software . . . . .	65
4.5.1 Compilation and operation . . . . .	65
4.5.2 Buffer management . . . . .	66
4.5.3 Synchronization and communication . . . . .	67
4.5.4 Application development . . . . .	68

4.6	Experiments/Evaluation	69
4.7	Conclusions	70
<b>Part 2 - Dynamic workloads, dynamically reconfigurable platform</b>		<b>73</b>
<b>5</b>	<b>Evaluating auto-adaptation methods</b>	<b>75</b>
5.1	Introduction	76
5.2	Approach	77
5.2.1	Target processor	77
5.2.2	Proposed auto-adapting method	78
5.3	Implementation	79
5.3.1	Common	80
5.3.2	Window-based monitoring	81
5.3.3	BTCB	81
5.3.4	Phase change annotations	82
5.4	Evaluation	82
5.4.1	Experimental setup	82
5.4.2	Results	82
5.5	Related work	85
5.6	Conclusions	86
<b>6</b>	<b>Adapting to dynamic workloads</b>	<b>87</b>
6.1	Introduction	88
6.2	Background	89
6.3	The $\rho$ -VEX polymorphic VLIW processor	90
6.4	Approach	91
6.4.1	On-line profiling	91
6.4.2	Compiler annotations	92
6.4.3	Datapath assignment	92
6.5	Experiments/Evaluation	93
6.5.1	Annotation overhead and coverage	94
6.5.2	Throughput & Performance	94
6.6	Related work	98
6.6.1	Phase detection and workload characterization	98
6.6.2	Polymorphic processors	98
6.7	Conclusions	99
<b>Part 3 - Real-time and mixed-criticality systems</b>		<b>101</b>
<b>7</b>	<b>Evaluating real-time properties</b>	<b>103</b>
7.1	Introduction	104
7.2	Background	106
7.3	Related work	107
7.4	Implementation	108
7.5	Experimental Setup	110
7.6	Results	112
7.7	Conclusions	113



<b>8</b>	<b>A platform for mixed-criticality systems</b>	<b>115</b>
8.1	Introduction . . . . .	116
8.2	Background . . . . .	118
8.2.1	Processing platform . . . . .	118
8.2.2	Scheduling methodology for dynamic processors . . . . .	120
8.3	System architecture for Mixed-criticality systems . . . . .	122
8.3.1	Spatial isolation . . . . .	122
8.3.2	Temporal isolation . . . . .	124
8.3.3	Assigning unallocated cycles to non-critical tasks . . . . .	124
8.4	Scheduling approach . . . . .	125
8.4.1	Worst-case schedule creation . . . . .	125
8.4.2	Improving average-case performance . . . . .	128
8.5	Experimental setup . . . . .	129
8.6	Results . . . . .	131
8.6.1	Schedulability . . . . .	132
8.6.2	Performance & area utilization . . . . .	133
8.6.3	Resource utilization and throughput . . . . .	134
8.7	Related work . . . . .	138
8.8	Conclusions . . . . .	140
<b>9</b>	<b>Conclusion</b>	<b>141</b>
9.1	Conclusions . . . . .	141
9.2	Future research directions . . . . .	142
	<b>List of Publications</b>	<b>145</b>
	<b>References</b>	<b>147</b>
	<b>Curriculum Vitæ</b>	<b>161</b>

# Summary

Embedded systems range from very simple devices, such as a digital watch, to highly complex systems such as smartphones. In these complex devices, an increasing number of applications need to be executed on a computing platform. Moreover, the number of applications (or programs) usually exceeds the number of processors found on such platforms. This creates the need for scheduling. Furthermore, each program exhibits different characteristics and their interaction with the (real-life) environment leads to real-time requirements. Consequently, the set of programs, called workload, exhibits highly *dynamic* behavior. Workloads can be dynamic in intensity (i.e., the number of concurrent tasks), characteristics (amount and type of parallelism), and requirements (real-time constraints, power budgets, performance). We argue that dynamic workloads require a dynamic computing platform and propose to use one that comprises the  $\rho$ -VEX reconfigurable VLIW processor. It can dynamically adapt to the workload while it is running. Adaptations can be triggered by a user, programmer, compiler, or an operating system. The latter two methods can operate fully automatic and exploring these is one of the goals of this work.

Besides dynamic workloads, a number of new classes of embedded devices are running application programs that are very static, but require very high throughput. Examples are the latest generations mobile telecommunications hardware and vision-based applications (automation, surveillance, automated driving). In this case, adapting to the workload at run-time is not advantageous because there are no changes to adapt to. Optimizing for these applications is possible, but must be done before the hardware platform is manufactured (during the design phase) or by making use of Field-Programmable Gate Arrays (FPGAs).

This thesis explores the use of the proposed reconfigurable processor to target the full spectrum of embedded workloads. First, design-time reconfigurability is employed to optimize a hardware platform for a static, streaming image processing workload. Second, we explore the run-time reconfigurable processor for dynamic workloads. This is achieved by adapting to a single program to optimize energy efficiency, followed by adapting to a generated set of programs optimizing for throughput. Third, the real-time characteristics of the processor are evaluated and it is shown to have better schedulability compared to static processors. The VLIW architecture results in good timing-predictability, which allows finding tight bounds on the worst-case execution time. Last, we show that the processor is able to assign more parallel execution resources to a static program that is added into the workload, while still guaranteeing time-safety for critical tasks.



# Samenvatting

Er bestaan vele vormen van embedded (geïntegreerde) systemen, van simpele apparaten, zoals digitale horloges, tot veel complexere, zoals een smartphone. De complexere systemen worden geacht steeds meer en in toenemende mate diverse applicaties te kunnen uitvoeren op hun embedded processor. Het draaien van al deze verschillende programma's, allen met hun eigen karakteristieken, gecombineerd met de interactie met de omgeving (waardoor veel embedded systemen real-time restricties hebben), resulteert in een werklast die zeer *dynamisch* is. Een werklast kan dynamisch zijn qua intensiteit (i.e., het aantal gelijktijdig actieve taken), karakteristieken (hoeveelheid en type parallelisme), en restricties (real-time, vermogen budgets, prestatie-eisen). Om dit type werklast efficiënt uit te voeren, stellen we voor om gebruik te maken van een dynamisch computerplatform in de vorm van de  $\rho$ -VEX herconfigureerbare VLIW processor. Deze processor kan zich aanpassen aan de werklast terwijl hij draait. Aanpassingen kunnen handmatig in gang gezet worden door de gebruiker, de programmeur, de ontwerper, of automatisch door een compiler of besturingssysteem. Een van de doelstellingen van dit werk is om de automatische methodes te onderzoeken.

Behalve dynamische werklasten ontstaan er ook nieuwe elektronische systemen die programma's uitvoeren die zeer statisch zijn, maar wel zeer hoge rekensnelheid vereisen. Voorbeelden hiervan zijn de laatste generaties mobiele telecomapparatuur en applicaties die werken met beeldherkenning (zoals diverse automatiseringstoepassingen, (camera)toezicht, zelfrijdende auto's). In dit geval heeft het geen zin om tijdens het uitvoeren van de applicatie de processor aan te passen, omdat er geen veranderingen zijn. Een hardwareplatform kan nog steeds voor deze applicaties worden geoptimaliseerd, maar het moet gebeuren tijdens de ontwerpfase of door gebruik te maken van Field-Programmable Gate Arrays (FPGA).

Dit proefschrift onderzoekt het gebruik van een herconfigureerbare processor voor het uitvoeren van het gehele spectrum van embedded software. Eerst maken we gebruik van configureerbaarheid tijdens de ontwerpfase om een hardwareplatform te optimaliseren voor een statische, streaming beeldverwerkingsapplicatie. Vervolgens onderzoeken we de run-time herconfigureerbare processor voor dynamische werklasten. Dit doen we door eerst de processor aan te passen aan een enkel programma om de energie-efficiëntie te optimaliseren, gevolgd door het onderzoeken van aanpassingen aan een gegenereerde verzameling van programma's om de totale rekensnelheid te optimaliseren. Hierna evalueren we de real-time eigenschappen van de processor, en laten we zien dat we er real-time schedules voor kunnen maken die niet mogelijk zijn op vergelijkbare statische processoren. Ten slotte demonstreren we dat de processor in staat is om meer parallele reken-eenheden toe te wijzen aan een statische taak die aan de werklast is toegevoegd, terwijl de tijd-kritische taken nog steeds aan hun real-time restricties voldoen.



# Acknowledgements

The PhD. project has been an incredible journey and I have been lucky to have been surrounded with so many wonderful people throughout the whole process. I will attempt to thank most of them here.

Stephan, under your guidance I have enjoyed the freedom in choosing a path that I felt was truly my own. Your topic is most definitely not the easiest in our field and you have always countered adversity and skepticism with optimism, creativity and perseverance (qualities that I quickly recognized as being essential in bringing a PhD. to a successful conclusion). It has been an amazing adventure and I want to thank you for everything. Zaid, traveling abroad to project meetings and conferences with you felt more like city trips with friends. Your energy and drive never ceased to amaze me, and your faith in me has been deeply appreciated. Koen, I am grateful this is not an in memoriam and that you have made such an incredible recovery. I have been a part of your group for half a decade now and I felt at home right from the start.

I would like to thank my PhD. committee, Peter, Henk, Michael, Kees and Said for taking the effort to review my work and provide valuable suggestions and critique.

Sorin, you have given me the opportunity to teach some things about computer architecture to hundreds of students from all over the world for four years. It is something I am very proud of. Thank you for all the confidence you placed in me, and thanks to Thomas, Nicoleta and Jeroen for all their help.

A big thanks to the wonderful staff; Lidwina, Joyce, Arjan, and sysadmin Erik (I hope you didn't mind me occasionally making sure you weren't bored).

I have had delightful times with many colleagues from overseas, including Qi, Jian, Jintao, Shanshan, Cuong (whom I've had the pleasure of visiting in Vietnam!), Innocent, Hoang Anh, and the profoundly talented football players Lei and Joey. Our weekly matches were something I always looked forward to! Thanks to Imran, Daniel, Razvan, Motta, and the others (also from other groups, including Accel and Long). Thanks to my Dutch PhD. buddies Johan and Erik; I have enjoyed the lunches, borrels, events and discussions with you!

I have the warmest memories of our friends from UFRGS, Brazil, who I have had the pleasure of working together with (many even as office mates). Luigi, it was difficult to even picture my defense without you. But I am glad that I never have to address you as "opponent", as I feel like you have been one of my greatest supporters. Caco, I will not forget your powerful presence on the football field! Arthur and Anderson, it was really great to have you guys as office mates on more than one occasion. Thank you all, also the others (Tiago, Sebastian, Jeckson), for the wonderful times we shared in Delft and at conferences elsewhere!

Participating in the ALMARVI project has taught me a lot, and I would like to thank the partners, especially the ones that we occasionally collaborated closely with, such as the guys from TUT Finland (Pekka, Timo and friends), Jiri from UTIA Czechia, and of course Rob and Steven at Philips.

Many thanks to the folks in the MEST board; Sven and Jordi, Nauman, Jan, Boyao, Fei, Chock, Hui, Augusto, Andres, and my successor Haji.

Koen & Carmina's quantum effort has rocketeered the group into a leading position in this upcoming field and sparked the creation of a whole department. Carmina, Leon, Lingling, Savvas, Xiang, Nader, Hans and the others; I feel very proud to have been your colleague and wish you all the best of luck!

There has been a large number of students working on the  $\rho$ -VEX for their Master's thesis, each contributing something to the project, the team, and even to this thesis; a big thanks to Muez, Maurice, Klaas, Hugo, Jens, Koray, Panos, Bas, Rolf, Muneeb, Angelos, Lennart, Jonathan, Jacko, Saevar, Anurag, Uttam, and Federico. In addition, I have thoroughly enjoyed having Bachelor's honour track students Tom, Piet, Luc and Matti participate in our lab!

Anthony, things have never been the same after you left and I have missed you at the lab ever since. Jeroen, you have been like a whirlwind going through our lab from the start, fixing and improving everything you saw. I cannot imagine how things would have gone without you, and I have often had to shake my head in awe about how talented and hard-working you are. The three of us felt like an unstoppable team, and I want you to know how much I loved working with you.

Carsten, our paths have never strayed far from each other from school all the way up to the PhD., and I have always considered our friendship something very special and valuable. I will miss having you so close.

To all my friends; thanks for all the fun and I hope to enjoy more of your company at bars, boats, barbecues and beaches!

Steven, Prem and Suzanne; thank you for making me feel part of your family. To my own family; Coen and Suus, I'm so proud of you and always love being around you. Dearest Mom and Dad, I am thankful for everything you have given me and for having you in my life. Over the years, I have become deeply aware of how special you are.

And at the end of the day, Olga, none of it would be worth much if you weren't here to share with. You are the sunshine of my life.

# 1

## Introduction



Ever since electronic devices started to use programmable embedded processors instead of dedicated circuits, more and more software that increases their functionality is being added each generation. The embedded domain, that has historically been concerned with very low-power designs running simple programs, is expanding its reach with ever increasing numbers of devices executing an ever widening spectrum of software: from washing machines to self-driving cars. This creates the need for embedded systems to be able to target applications in the entire spectrum in an efficient manner. As the applications and their requirements are so diverse, there exists no single execution platform that can execute all workloads efficiently.

This chapter introduces two sides of the spectrum of embedded workloads in Section 1.1 and 1.2, briefly introduces embedded execution platforms that are currently used to execute these workloads in Section 1.3, and discusses general workload properties in Section 1.4. Then, the problem formulation and scope of this work are presented in Section 1.5, the proposed solution is discussed in Section 1.6, and the evaluation approach is presented in Section 1.7.

## 1.1. Dynamic workloads call for dynamic processors

We are continuously surrounded by electronic devices. Some of these just make our lives easier, while others perform crucial tasks such as operating the wings of an airplane or the brakes in our cars. In these devices, there has been a steady increase in usage of embedded processors, because of their relatively low cost compared to designing a dedicated electronic circuit for each task or device. For example, high-end cars currently have over 100 processors on board [1], and the demand for processing power may start to increase at a significantly higher pace now all competing brands have started to incorporate Advanced Driver Assistance Systems (ADAS) into their models.

The workloads that are executed on embedded processors are increasing in complexity [2]. Early embedded systems typically had only a single task to execute. To decrease costs, power consumption, and other factors such as the wiring in cars, the industry is pushing towards consolidating multiple tasks onto a single embedded processor [3]. As a result, systems have workloads consisting of multiple tasks that each have their own requirements and characteristics. Some of them will have very strict real-time requirements, others may have a performance requirement. Tasks can be multi- or single-threaded, and their execution time can be dominated by computation, control (branches), or memory accesses.

These complex workloads, combined with close interaction with the environment (e.g., responding to sensor input), lead to highly dynamic behavior. Workloads can be dynamic in intensity (i.e., the number of concurrent tasks), characteristics (amount and type of parallelism), and requirements (real-time constraints, power budgets, performance). A single program that has a high degree of Instruction-Level Parallelism (ILP) can be executed efficiently on a high-performance single-core processor. A program consisting of a large number of parallel threads should be executed on a processor with a large number of cores, in which case individual core performance is less important. Even within a program, some of these properties can change rapidly as execution passes through different phases [4]. On a workload

level, some programs more strict real-time requirements than others, leading to mixed-criticality systems [5]. However, contemporary embedded platforms, that are used to execute these highly dynamic workloads, are based on homogeneous or heterogeneous multi-core processors that are *static*.

Homogeneous multi-core processors have multiple instances of a processor core on a single chip, supporting concurrent execution of multiple tasks or threads. Heterogeneous multi-core processors usually have a number of high-performance and a number of energy-efficient (but lower performance) cores. This provides some potential to match a program to a core type, depending on its requirements (how much performance does it need?) and code characteristics (some code will not run significantly faster on a high-performance processor core, because for example it is bound by the performance of the memory system). This approach limits the accuracy of this match, because the number of core types is fixed and finite (current heterogeneous processors such as ARM big.LITTLE provide only two options). Additionally, it limits resource utilization, because if a program is running on one core, all the others are idle. A final drawback is that there is a penalty involved when migrating a task from one core to another. These observations have inspired architects from industry and academia to propose dynamically reconfigurable processors.

Dynamic (or ‘polymorphic’) processors adapt to the workload. Typically, they can split cores to run multiple programs at the same time (providing high total throughput), or merge them to provide high single-thread performance. Several designs have been described in literature [6] [7] [8], and a theoretical foundation for their potential performance advantage has been given by Hill and Marty [9]. One such design is the  $\rho$ -VEX polymorphic VLIW (Very Long Instruction Word) processor, which is the focus of this dissertation. It is discussed in more detail in Section 1.6.

Having a processor that is able to adapt to the workload is not enough. It is still necessary to analyze the currently executing workload and evaluate what the best processor configuration is. Without such functionality, it is up to the user to trigger processor adaptations manually during run-time, up to the designer to trigger them according to a certain rule set devised at design-time (this can for example be based on environmental observations such as battery level), or to the programmer to profile and annotate his code during compile-time. Our goal is to perform these steps fully automatically in hardware, during run-time.

## 1.2. Leveraging design-time customization for highly static workloads

On the other side of the spectrum of embedded workloads, applications exist that continuously performs a fixed set of operations of a stream of input data. For example, a currently rising class of applications uses some form of computer vision. Most notably, this includes automation (e.g., vision-guided robotics), surveillance, automotive and medical industries. A necessary step for these systems is to retrieve input from a sensor and to perform some form of signal processing. For some applications this step can be very simple, others need to perform a large number

of complex steps on the input data under strict real-time constraints.

These applications can be highly complex and computationally demanding, but their behavior is static. In this case, adaptations during run-time are not advantageous, because there is no dynamic behavior to adapt to. A processor can still be optimized for these workloads, but the optimization must be performed at design-time as opposed to run-time. This is possible because we know that the application will not change. Optimization may be necessary because the application can have very high performance requirements at limited power budgets, preventing the usage of Commercial Off-The-Shelf (COTS) processing systems.

### 1.3. Embedded execution platforms

This section briefly presents some background knowledge about processing platforms in embedded systems past and present.

#### 1.3.1. General-purpose

Classic embedded devices used dedicated electronic circuits to perform a certain function. At some point, these circuits were replaced by general-purpose processors, that could be used in any embedded systems and programmed to perform any function. However, the complexity of embedded systems grew from a digital watch or a washing machine (that only need to perform a single, simple function) into complex devices such as smartphones and tablets. These require more powerful embedded processors to provide the performance for, for example, high-quality video en/decoding and baseband communications.

To provide more performance at acceptable power utilization, manufacturers started creating multicore platforms. These processing systems can be homogeneous, meaning that each core is identical, or heterogeneous, where there are different types of cores. These differences can be transparent to a program, such as clock frequency, cache sizes and other factors that impact performance. These systems are called *single-ISA* (Instruction Set Architecture) heterogeneous multicore processors. They are utilized by smartphone manufacturers to provide both high performance when the users requests it, and long battery life when the device is not in use. The different core types can execute the same code, but it will require more power and less time on one core compared to the next.

In addition to single-ISA heterogeneous cores, cores can be completely different and highly specialized for a certain task. For example, a processor could have a general-purpose core, a DSP (Digital Signal Processor), and an AES (Advanced Encryption Standard) accelerator. These processors are designed specifically for their application, and need to be programmed separately.

#### 1.3.2. Application-Specific

Specialized Application-Specific Integrated circuits (ASICs) can provide orders of magnitude better performance or energy efficiency. However, specifically designing a circuit for a certain (family of) tasks requires significant investments and effort. Another approach is to start designing from a generic processor design,

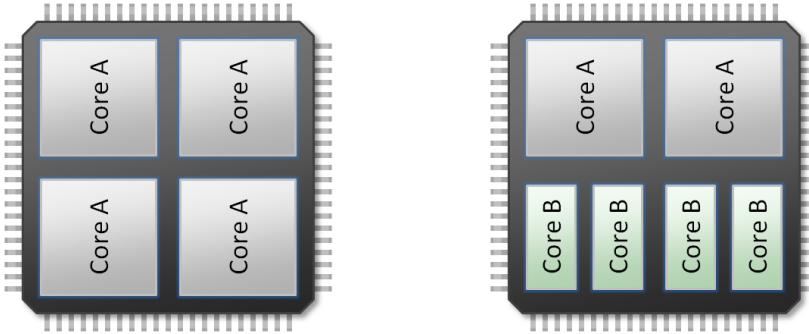


Figure 1.1: Conceptual depiction of homogeneous versus heterogeneous multicore processors.

and optimize it to the application. These processors are called Application-Specific Instruction-set Processors (ASIPs). They represent a middle ground between full-custom circuitry and general-purpose processors. By including specialized instructions for the application domain, they are still able to achieve good performance and energy efficiency. Additionally, they can still be programmed for different applications within their domain. For example, an ASIP for a baseband modem will include instructions for operations that are common for various error correction codes. This is advantageous in, for example, telecommunications hardware where multiple highly complex communication protocols (2G, 3G, 4G) must be supported [10].

### 1.3.3. Reconfigurable

Reconfigurable processors take the concept of optimizing for an application a step further. They are able to change their behavior after they have been manufactured. The most narrow view on this type of system is the Field-Programmable Gate Array (FPGA), which is a chip that consists of vast numbers of Configurable Logic Blocks (CLBs) and interconnects. CLBs can be configured to perform any logic function with a small number of in and outputs. For example, the Xilinx Virtex6 family uses Look-up-Tables (LUTs) with six inputs. Combined with the reconfigurable interconnects, FPGAs can implement electronic circuits including complex designs such as a complete processor. A circuit configuration can be loaded into the FPGA before starting a new application. FPGAs have been used to accelerate certain segments of a program that are very time-consuming in software but can be performed efficiently by an FPGA circuit. By using partial reconfiguration, parts of the FPGA can be reconfigured while other circuits continue to function on other parts of the FPGA. This way, the FPGA can adapt not only at the start of a program but also when a program changes to another phase in its execution.

In addition to FPGAs, there exist processing platforms that are not organized in CLBs and interconnects, but using larger (coarse-grained) blocks of logic. As the notion of what can be denoted as 'reconfigurable' is debatable, even a normal processor can be viewed as reconfiguring itself to perform a different operation

every clock cycle (depending on the instruction it is executing). In contrast to an FPGA, this type of reconfiguration can occur every clock cycle (as opposed to once when starting the application), and is concerned with an entire ALU (Arithmetic Logic Unit) that consists of several thousands of gates and can perform a multitude of operations (as opposed to individual logic blocks that are tiny in comparison).

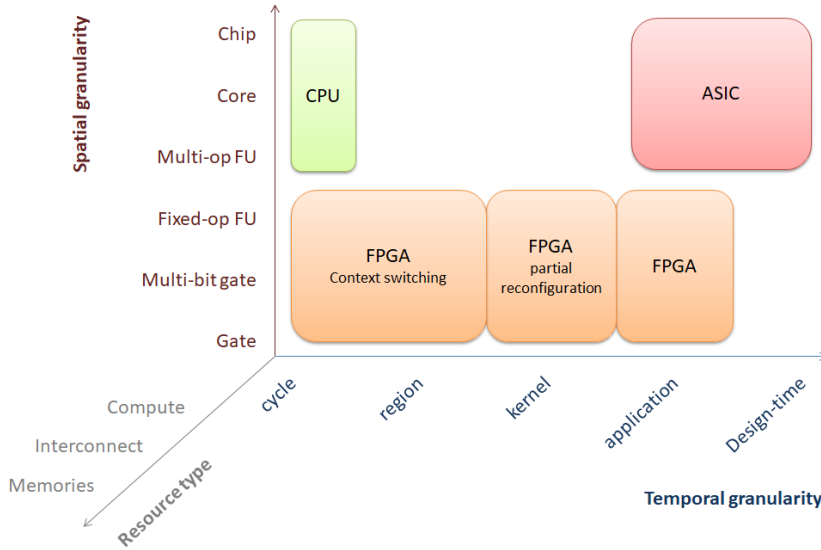


Figure 1.2: Classification of reconfigurable processors based on [11]. A processor can be reconfigurable regarding different components such as memory, compute units or connections between processors or components. Depending on the reconfiguration penalty involved, processors can be reconfigured at high frequencies (fine-grained) or, for example, per application. Similarly, the components that are reconfigurable can be very large (such as a whole core) or in case of FPGAs almost on a per-gate level.

This broad view creates a range of different reconfigurable processors, similar to the spectrum of different workload types. Figure 1.2 shows a classification, introduced by [11]. Different parts of a processor's organization can be reconfigured, most notably the memory resources (such as caches), compute resources (such as the ALU or other functional units), and the connections between them (not only between the functional units and the memory, but possibly also between functional units themselves). The spatial granularity denotes the size of the reconfigurable parts and the temporal granularity determines at which frequency these reconfigurations can be performed. Every type of reconfigurable processor has one property in common; each of them aims to adapt the hardware to the workload.

## 1.4. Workload analysis

This section discusses several requirements and characteristics that a workload can have, and the how they can be dealt with by an execution platform.

### 1.4.1. Requirements

An application can have several requirements that are imposed by the designer or the environment:

#### Time

In the embedded domain, timing requirements are very common. A real-time system needs to guarantee that a result is available within a specific amount of time (the deadline), otherwise the system will fail. Often, a distinction is made between tasks where the result is useless and may result in a catastrophic failure if it is not available before the deadline (hard real-time tasks - HRTT), and tasks where the requirement is not as strict and a late result may even be of some value (soft real-time tasks - SRTT). Embedded real-time systems can have tasks that are periodic and aperiodic in nature. New instances of a periodic task will be triggered (*released*) at its specific interval (the task *period*). Aperiodic tasks are triggered when a certain event happens, such as a button being pressed. These tasks also have a deadline before which the system must have handled the event.

#### Performance

As performance is the inverse of delay, most performance requirements can also be expressed as a timing requirement. For example, a performance requirement for encoding video at a frame rate of at least 25 frames per second (FPS) seems to be a performance requirement but can also be expressed as a periodic task (encoding a video frame received from the image sensor) that should always finish within 40 ms. For applications that have less easily identifiable blocks, a performance requirement can be expressed in terms of throughput, such as for example compressing data with a minimum throughput of 1 Gbit per second.

#### Power/Energy

Embedded systems are often battery-powered, and modern high-performance embedded systems such as mobile phone Systems-on-Chip (SoC) have thermal budgets to prevent devices from overheating. Also in the server domain, energy consumption is a key component of total costs and has established itself as a critical performance metric. For datacenters, it is becoming increasingly difficult to match power provisioning and cooling capacity to the demand, as workloads can be highly variable [12]. Consequently, energy or power requirements are common in both the embedded and server domains. Methods to meet these requirements include:

- **Multicore processors** can be used to divide a workload over multiple processors. Subsequently, the clock frequency and voltage of the processors can be reduced to achieve the same performance. As power utilization scales linear with frequency and superlinear with voltage, this should result in lower total power utilization.
- **Heterogeneous Computing** is applied in two separate fashions: **Single-ISA Heterogeneous Multicore Processors (HMP)** [13] such as ARM big.LITTLE processors [14] can provide high performance using big processor

cores and high energy-efficiency using little cores. This concept aims to meet the contradictory requirements of mobile phone SoCs to provide high performance for cutting edge media and 3D graphics processing but also very long standby times. **Hardware accelerators** provide optimized ASIC implementations of common intensive tasks such as video coding and encryption. These can be offloaded to the accelerators, instead of executing them inefficiently on the central processor.

- **Dynamic Voltage and Frequency Scaling (DVFS)** is a common technique that changes the voltage and frequency of a processor during run-time, according to the demand for processing power.
- **Power/Clock gating** can be used to switch off parts of the processor that is not being used. In processors, this is commonly implemented as different sleep modes. Lower sleep modes conserve more power but require more time to restore to full functional state.

### 1.4.2. Parallelism

Exploiting parallelism is one of the key mechanisms to increase the performance of a computer system [15]. This can be done by overlapping (parts of) calculations that can be performed concurrently, by assigning them to different functional units or even different processors. Depending on the application, parallelism can be present on multiple levels.

#### Task-level parallelism

The most straightforward form of parallelism is when there are multiple unrelated *tasks* assigned to a system. These tasks can be assigned to different processors available on a multicore or multiprocessor system. Increasing the number of processors in a computer system increases the number of parallel tasks it can perform, until shared resources such as memory bandwidth will become the bottleneck.

#### Thread-level parallelism

A related form of task-level parallelism is thread-level parallelism, where a single program is divided into parts, each running on a different execution *thread*. In this case, the tasks are related, which means they need to perform some form of synchronization and possibly communication of intermediate results. This overhead limits the speedup that can be achieved by distributing a task over multiple threads and assigning each thread to a separate processor. For some types of applications, the overhead is relatively small and performance can be increased by using a large number of small processors. Otherwise, a limited number of threads executing on high-performance processors is a more suitable approach.

#### Data-level parallelism

Some applications can be divided in such a way that the exact same operations are performed on subsets of the *data*. This form of parallelism can be exploited by using SIMD (Single Instruction, Multiple Data) architectures. These architectures

execute a single stream of instructions on multiple different sets of data. If the bit depth of each individual data item is relatively small, multiple data items can be calculated using a single datapath. For example, a 32-bit datapath, in SIMD mode, can execute 2 16-bit operations or 4 8-bit operations simply by decoupling the carry chain [16, Fig. 5.6]. Additionally, specialized vector datapaths can be added to a processor that are much wider than the processor's general scalar datapath. Using SIMD greatly reduces the control overhead<sup>1</sup>, communication and synchronization compared to computing the data sets on separate processors. SIMD datapaths can be very wide, supporting several parallel operations, but need to access memory in such a way that the full width can be exploited. This means that either the register file must support these wide accesses, there must be a separate vector register file, or the SIMD datapaths need to be able to directly access a cache line (which, in turn, needs to be wide enough). For example, the Hexagon VLIW [17] provides four 1024-bit wide vector datapaths, and has a dedicated vector register file that directly connects to the L2 cache. On the software side, SIMD requires compiler support in the form of automatic vectorization.

### Instruction-level parallelism

Instead of splitting up tasks or datasets into multiple parts, a processor can also exploit parallelism on the level of individual *instructions*. It is the most transparent approach for the programmer, as it requires no manual program transformations or vectorizing compilers. Historically, exploiting ILP has been one of the key architectural driving forces behind processor performance increase. Classical approaches include pipelining and multi-issue processors, both of which are employed heavily in processor designs.

Pipelined processors divide individual instructions into several steps and overlap their execution. Pipelining has been one of the main enablers of the continuous increase in processor clock frequencies until the power density (that steadily increased with scaling down technology feature sizes) resulted in heat dissipation problems. This forced architects to search for other means to increase processor performance without increasing the frequency.<sup>2</sup> As the performance of a program is dictated by execution time =  $\frac{\text{cycles}}{\text{frequency}}$  and cycles =  $\frac{\text{instruction count}}{\text{instructions per cycle}}$ , increasing the number of instructions per cycle (IPC) has an equal effect on performance as the processor clock frequency. A natural way to do this is to allow the processor to start multiple independent instructions per clock cycle: multiple-issue processors.

There are two main paradigms for designing multi-issue processors: superscalar and VLIW processors. Superscalar processors use complex circuitry to detect whether operations can be executed in parallel or not. VLIW processors move this complexity to the compiler, thereby allowing simpler processors with decreased

<sup>1</sup>This includes all the circuitry needed to run the program that is not the datapaths themselves (such as branch units and pipeline control logic), but also for example the memory bandwidth required for the instruction stream.

<sup>2</sup>This is not to say there were no previous efforts in this direction: indeed, the first superscalar processors have been designed in the 1960s by Cray and IBM. However, frequency scaling provided a steady performance increase with every new technology generation up to this point, after which architectural improvements were the only way forward.



power consumption to achieve similar performance *as long as the compiler can find enough parallelism in the code*. There are some fundamental limits to exploiting ILP:

- There are limits on the available ILP in code. In the general-purpose domain, between 15 and 25 % of all executed instructions are branches, which means that there will be on average between three and six instructions available before a branch [15, p. 67]. This means that a processor must be able to cross branches to find a decent amount of ILP.
- When finding ILP across branches, the processor must speculate whether these branches are taken or not. Even with highly accurate predictors, mis-predictions will continue to occur occasionally. These require the processor to flush any speculatively issued instructions from the pipeline and restart execution at the resolved branch address.
- Speculation (when assumed to be imperfect) inherently reduces energy efficiency [15, p. 182].
- Some instructions have true dependencies that cannot be resolved by the processor (e.g., by using register renaming).

Even with perfect branch prediction, an infinite number of parallel datapaths and an infinite window for finding independent instructions, the number of instructions that can be executed in parallel is limited [15, Ch. 3]. In addition to the limits of finding enough ILP to efficiently utilize multiple parallel datapaths, there are some practical limits to increasing the issue width of a processor:

- The number of parallel datapaths cannot be increased indefinitely because of increasing circuit complexity<sup>3</sup>.
- Compiler techniques to increase ILP such as loop unrolling will increase binary sizes and subsequently increase the required instruction cache capacity and memory bandwidth.
- Processing multiple data items in parallel will also require more accesses to the data memory, creating the need for larger and multi-ported caches.

The limitations to both the hardware and software quickly result in diminished returns: a small increase in performance requires a large investment in circuit complexity and consequently energy consumption. Modern computing systems combine the techniques discussed in this section to exploit as much parallelism as possible: multicore superscalar processors with very deep pipelines and SIMD extensions are common even in the embedded domain.

<sup>3</sup>Fully connected VLIW processors will see the circuit area of both their register file and forwarding logic increase superlinearly with their issue width. There are approaches to mitigate this effect, such as clustering [18] and using exposed datapath architectures with limited connectivity [19] [20].

### 1.4.3. Code characteristics

In addition to the type and amount of parallelism available in an application, an important characteristic is the instruction mix and how this influences the performance of a processor. In a very general sense, instructions from any RISC (Reduced Instruction Set Computing) processor can be divided into three families:

- **Control flow operations** change the flow of the processor through the program by jumping to a certain address. These include function calls but also conditional branches, that for example implement if/else constructs.
- **Memory operations** load or store values to and from the processor's internal registers (the register file).
- **Arithmetic operations** perform calculations such as additions or multiplications on values stored in the register file.

Each of the instruction type uses a different processor subsystem that can cause the processor to stall, degrading performance.

#### Control-bound

As discussed in Section 1.4.2, processors with deep pipelines rely heavily on speculation to achieve high performance. Branch prediction is arguably the most important form of speculation, and mispredictions are very expensive. Programs with a large amount of branches require more complex prediction logic to keep the misprediction rate low. VLIW processors often rely on the compiler to perform static branch prediction. The compiler analyzes the most likely control flow and restructures the code so that the common case will execute fastest. In addition, techniques exist to expose ILP across branches by combining multiple blocks of code (trace, superblock and hyperblock scheduling). However, there are limitations to these techniques and exploiting ILP in branch-heavy programs using VLIW processors remains challenging. Control-bound applications are currently most suitable to be executed on general-purpose processors with a high operating frequency and sophisticated branch prediction.

#### Memory-bound

If a program performs a relatively large number of memory operations, and few operations on the data in between accesses, the memory accesses can create a performance bottleneck. There is a large discrepancy between latency of processors and memory devices, and in modern server-grade computer systems, a main memory access costs hundreds of processor cycles to complete. As this difference has been growing over the past decades, an increasing number of cache levels have been added to the memory hierarchy (three levels is currently very common). While caches solve part of the problem for applications with enough locality, a processor should still be able to overlap the time it is waiting for outstanding main memory requests with computations on already available data. For highly regular memory access patterns, this can be accomplished by prefetching data <sup>4</sup>. This

<sup>4</sup>Where superscalar processors have hardware circuitry that tries to predict future memory accesses to fetch these values from memory, VLIW processors commonly employ explicit prefetch instructions

can allow programs with limited cache locality to still achieve high performance. For example, some streaming workloads operate on a small set of input data only once, before continuing with the next input. Afterwards, a data set is never used anymore, rendering the caches to be of little use. However, the memory access pattern of these workload types is usually very regular and can achieve high sustained memory bandwidth.

Irregular memory access patterns are more challenging. In this case, an approach to overlap memory access latency with computation is to switch to another thread when the currently executing thread encounters a long latency cache miss (Switch-on-Event or coarse-grained Multi-Threading). This requires the processor to be able to store multiple program states in internal registers to allow it to quickly switch between them.

### Compute-bound

In contrast to memory-bound programs, some programs perform large amounts of operations per memory access. If there is enough ILP in the code, the situation can arise where there are more operations available for executions than there are functional units available to perform them. In this case, the processor does not fully utilize the available memory bandwidth. The program can execute faster if there are more execution units, for example by using a wider issue superscalar or VLIW processor.

## 1.5. Problem formulation and scope

The industry is continuously pushing for increasingly fast time-to-market, driven by a decreasing window of competitive advantage for new technologies. At the same time, maintaining multiple processor families, along with their complete ecosystem of toolchain, libraries, operating system and application support, is becoming increasingly expensive. This raises the question whether it is possible to use a single processor family to target both sides of the spectrum – allowing run-time adaptations for dynamic workloads, design-time optimization for static workloads, and short time-to-market. This leads to the following research question:

*How to target both highly static and dynamic workloads using a single processor architecture?*

As discussed, there is a large spectrum of both software (workloads) and hardware (embedded reconfigurable processors). In addition, having a run-time reconfigurable processor adds the question of how to schedule these configurations. The scope of this work concerning these aspects is defined in the following sections.

### 1.5.1. Software: Workloads

In this thesis, we will discuss three general types of workloads, each in their own embedded application domain. A workload is considered as the set of application

---

that can be inserted into the code by the compiler or programmer (once more reflecting the VLIW philosophy of moving complexity from hardware to the compiler).

programs that is executed on the system under consideration. Individual programs can have multiple instances (they can be restarted, for example, with a new set of input data). Not all programs are necessarily active at the same time (some can be triggered by an event, or have a dependency on the termination of another program). In increasing order of complexity, the workloads studied in this work are:

- **Static workloads** continuously perform the same (set of) operations on an input stream. In particular, this work focuses on a medical imaging workload consisting of convolution kernel filters that are applied to input from an image sensor. The input passes through a chain<sup>5</sup> of image processing filters that aim to improve image quality by means of, for example, noise reduction and contrast enhancement. This workload has a high degree of data, thread, and instruction-level parallelism.

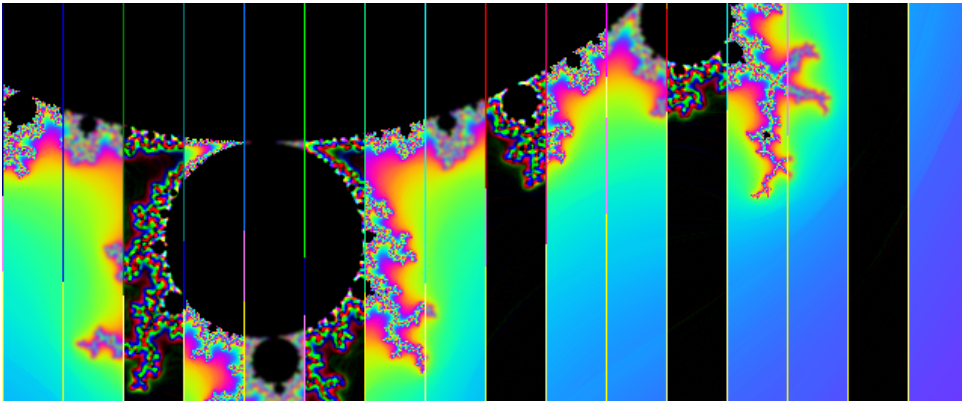


Figure 1.3: Image divided into slices, each processed on a different (set of) processors. In this still image, slices are alternating between no filters, a blurring filter, and blurring combined with an edge detection filter.

- **Dynamic workloads** consist of multiple different tasks, having different amounts of ILP. The number of concurrently executing tasks varies during the execution of the workload. Because of this, both Instruction-level and Task-Level Parallelism (TLP) are varying during the course of execution. In the time domain, this behavior occurs on both a very coarse-grained level and a fine-grained level, because 1) individual tasks typically go through numerous phases during their execution [4] and 2) some programs alternate between phases very rapidly.

Figure 1.4 shows the phases of JPEG with different window sizes, as detected by our modified compiler. It can be seen that the program shows changing behavior on a very fine-grained level (with a phase of only 50 cycles in the top

<sup>5</sup>These chains of filters are commonly referred to as a 'pipeline'. This term is also widely used to denote a processor datapath.

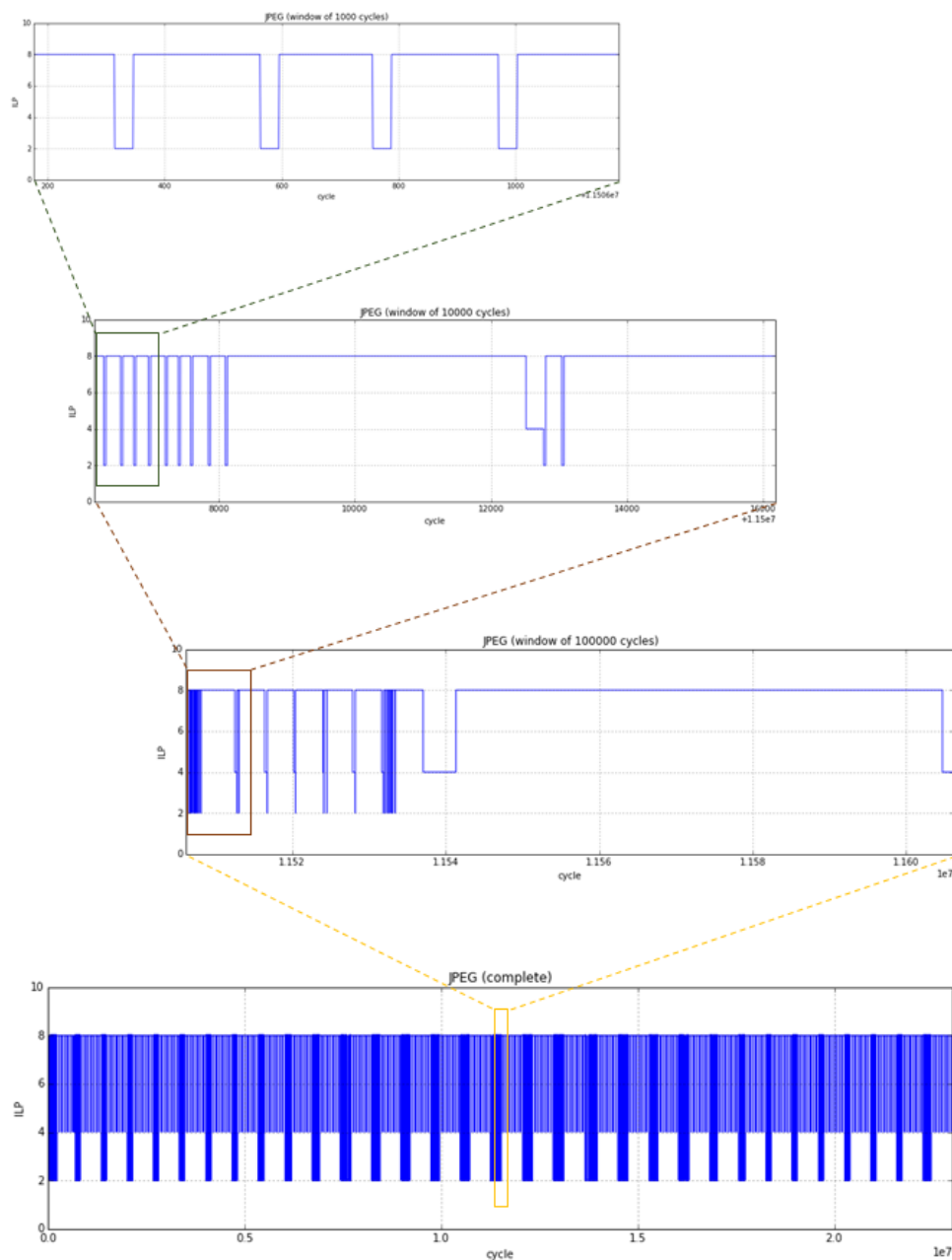


Figure 1.4: Graphic representation of ILP phases in JPEG as detected by a VLIW compiler. Depicted window sizes range from the full execution in the bottom figure, to 1000 cycles in the top figure. The subwindows are left-aligned at the same point exactly halfway through the full execution. These figures show changes in ILP requirements on different levels of temporal granularity.

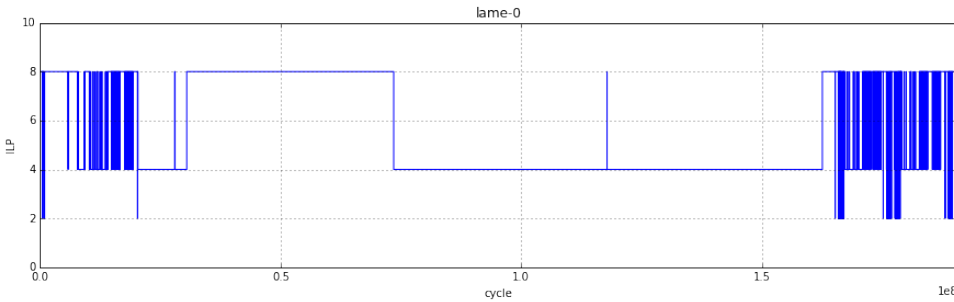


Figure 1.5: Graphic representation of ILP phases in LAME as detected by a VLIW compiler. It has a much larger portion of long, stable phases compared to JPEG.

figure) as well as on a larger scale (with a stable phase of 60,000 cycles in the second lowest figure). In comparison, Figure 1.5 plots the full execution of MP3 encoder LAME, which shows long stable phases of tens or hundreds of millions of cycles. These two examples show that there are widely different programs regarding phase behavior.

- **Real-time workloads** add strict timing requirements to dynamic workloads. This necessitates the execution platform to provide a guarantee to each real-time task that, every time it is activated, it will receive a number of execution cycles no less than its Worst-Case Execution Time (WCET). We will consider task sets with multiple tasks that can each have a distinct WCET and timing requirement. In this work, we use real-time workloads that contain multiple *periodic* tasks. Each task has a period after which a new instance of the task is triggered. If the previous instance has not finished execution at that time, it has missed its deadline (deadlines are *implicit*).
- **Mixed-criticality systems** combine real-time workloads with a static workload by adding a static program to a real-time workload. The static program does not have a timing requirement such as the real-time tasks, but should be optimized for throughput. This workload type represents the mixed-criticality systems field of research. This field originated from the need to certify avionics systems [5] that has tasks with different levels of criticality (requiring different certifications), but evolved to also include systems with critical and non-critical tasks.

The method in which we generate workloads for these different types, for use in our evaluations, is discussed in Section 1.7.5.

### 1.5.2. Hardware: Reconfigurable processors

This thesis focuses on the  $\rho$ -VEX VLIW processor that is reconfigurable on two extremes on the temporal granularity axis: static (design-time) reconfigurability (also application-level by using FPGA technology) [21], and dynamic (run-time) parameterization [22] on a region level. The use of the term parameterization

serves to distinguish the two types of reconfigurability, but also signifies that this type of reconfigurability is limited to changing a set of parameters that is fixed at design-time.

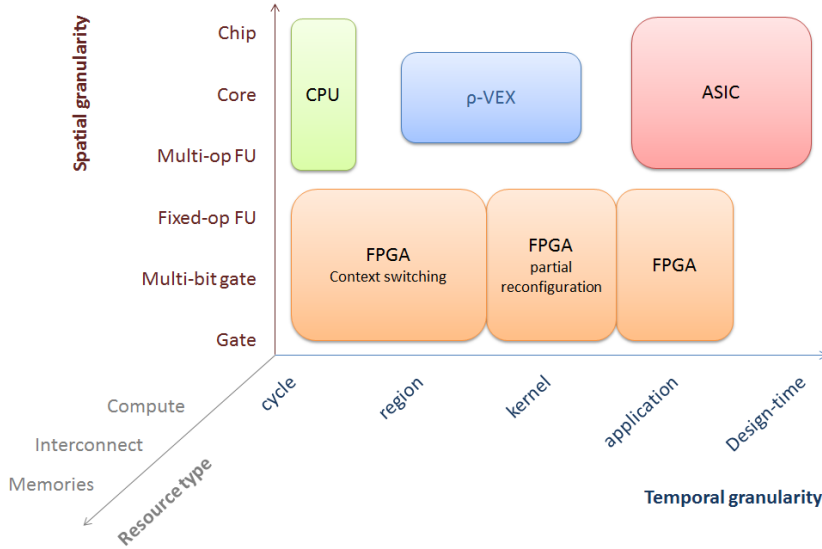


Figure 1.6: The position of the  $p$ -VEX dynamically reconfigurable processor in the classification of Figure 1.2 (based on [11]).

The design-time reconfigurability of the processor is used to target highly static workloads. The run-time parameterization is used for the dynamic, real-time and mixed critical workloads. The component that is parameterized is the interconnects between memories (caches and register files) and compute units (datapaths). This is discussed in more detail in Section 1.6.

### 1.5.3. Scheduling: Tasks and processor configurations

When given a compute problem consisting of a number of tasks, and a computing platform consisting of a set of resources (e.g., processors, memory, and interconnections), the problem of creating a schedule emerges. In classical single-core computing, creating a schedule was a single-dimensional problem that could be solved by assigning processing cycles (time) to tasks. Some important properties of a solution is the ordering (which task to execute when) and time partitioning (what fraction of the available time does each task receive). Multi-core processors slightly complicated schedule creation, especially when considering Heterogeneous Multi-Processors (HMPs). Now, a spatial dimension is added to the problem, concerned with which tasks to assign to which core (task mapping). Similarly, making a processor reconfigurable adds a dimension to the scheduling problem, as the processor can execute in a number of different configurations at any point in time.

There are two general approaches to scheduling - static and dynamic. Here,

static indicates that the schedule is created at compile-time and remains fixed during the execution of the workload (similar to static reconfigurability and static workloads that do not change during run-time). Dynamic means that the scheduling is performed during run-time (corresponding to dynamic parameterization and dynamic workloads).

## 1.6. Proposed platform: a design-time configurable, run-time parametrizable VLIW processor

In this work, we will use the  $\rho$ -VEX design-time configurable, run-time parametrizable VLIW processor. It is a proof-of-concept for VLIW-based polymorphism, implemented in the context of the Liquid Computing research theme at the Computer Engineering Laboratory of Delft University of Technology, the Netherlands.

### 1.6.1. Static (design-time) reconfigurability

The core is implemented using synthesizable VHDL (VLSI Hardware Description Language) code, that can be modified by the designer. To aid designers in their DSE, design-time reconfigurability is provided for the parameters listed in Table 1.1. A

Subsystem	Configuration options
Platform	Number of cores Clock frequency Connectivity (Bus, NoC) Peripherals (GRLIB, Xilinx IP libraries) Trace unit (enable/disable) DRAM controller (enable/disable)
Core	Number of pipelines per core Number of pipeline groups (dynamic reconfigurability) Number of execution contexts (Virtual cores) Code compression (using stop bits [23]) Pipeline organization (Nr. of pipeline stages, amount and locations of functional units) Instruction-set extensions Various debug and trace options
Memory	Caches (enable/disable, capacity) Scratchpad memories (enable/disable, capacity, sharing, address mappings)

Table 1.1: Design-time configuration options of the  $\rho$ -VEX processor

number of different platforms is provided, including standalone cores without main memory, SoC designs with an AMBA (Advanced Microcontroller Bus Architecture) bus and peripherals (by using Xilinx or GRLIB IP libraries), and a streaming platform with a simple NoC (Network-on-Chip) and local scratchpad memories for each core (one of the topics of this thesis). Discussing the details of all configuration options



is beyond the scope of this thesis. They are documented in the  $\rho$ -VEX user manual [24].

Two of the configuration options are of particular interest for this work, as they are concerned with dynamic reconfigurability. These are the core-level options for setting the number of contexts and the number of pipeline groups, which will be discussed in the following section.

### 1.6.2. Dynamic (run-time) parameterization

The polymorphic nature of the  $\rho$ -VEX processor allows it to adapt to the workload by assigning computational resources to threads in a flexible manner. In case of the  $\rho$ -VEX, this is implemented by being able to split and merge together groups of parallel datapaths (pipelines) of the VLIW, and connecting them to one or more execution *contexts*<sup>6</sup>. This way, pipelines can work together to provide high single-thread performance or work separately, each executing a different application, to provide high total throughput.

#### A classification of dynamically reconfigurable processors

There are numerous dynamically reconfigurable processors described in literature. In addition to the more general classification for reconfigurable processors proposed by [11] (displayed in Figure 1.6), a specific classification concerning *dynamically* reconfigurable processors is proposed by [25]. A slightly modified version that illustrates the position of the  $\rho$ -VEX is presented in Figure 1.7. The first focus is on the number of contexts that a processor supports (in contrast to the original paper, we have used the number of *hardware* contexts only). Secondly, it classifies the size of the individual processing elements (PEs). The last factor is the number of PEs in the system. An FPGA has a large number of very small PEs (LUTs) and a single context (although this can depend on the design). A superscalar processor is essentially a very large PE that has a single context (although processors can have multiple contexts by means of hardware multi-threading). VLIWs are typically somewhat smaller in size. A multicore consists of multiple PEs that can have various sizes (each PE can be as large as a full superscalar processor). A  $\rho$ -VEX can be configured to have between 1 and 8 contexts and between 1 and 8 PEs.

The design space for dynamically reconfigurable processors is considerably larger than that of the  $\rho$ -VEX. Individual PEs can be larger, for example when using full superscalar cores as proposed by [6] [7] [26] (defined as “Tile processors” by [25]), or smaller as is the case when combining simple functional units in a Coarse-Grained Reconfigurable Arrays (CGRAs) [11], in which case the PEs themselves are not individually programmable (the configuration determines the functionality). The number of contexts can be much larger than what the  $\rho$ -VEX supports, but it must be noted that there are practical limits to the number of PEs that can be combined to speed up a single context (e.g., wire delays or inter-cluster communication).

<sup>6</sup>A context represents the full register state of a thread or process, including all general-purpose registers (GPRs) and Control Registers such as the program counter.

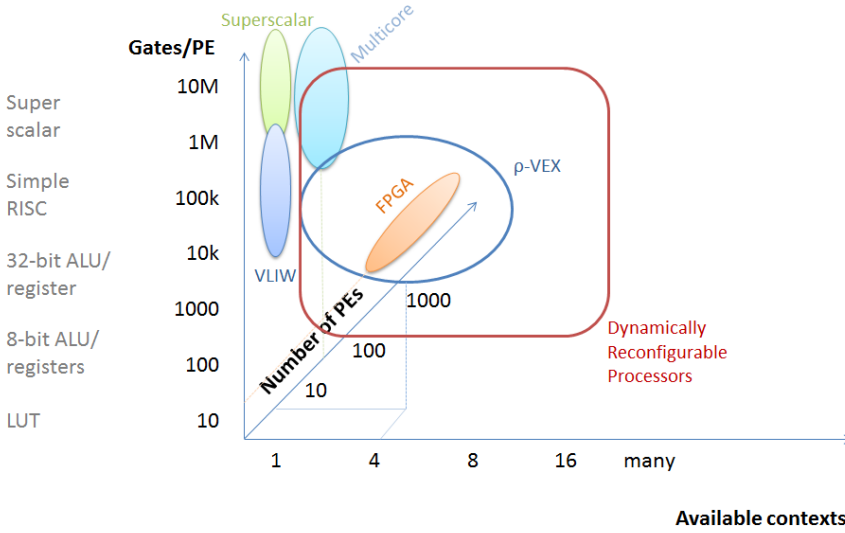


Figure 1.7: Classification of dynamically reconfigurable processors based on [25]. The horizontal axis shows the number of threads (contexts) a processing element is able to keep on-chip. The vertical axis shows the approximate granularity of individual processing elements. The third axis shows the number of processing elements in a processor. An FPGA has enormous amounts of tiny processing elements (CLBs). A (single-core) superscalar has one large processing element.

### Bringing Simultaneous Multi-Threading to the embedded domain

The goal to distribute computational resources among multiple threads is similar to SMT (Simultaneous Multi-Threading) [27], a common design technique used in the high-performance domain (for example, the IBM POWER8 processor [28]). The  $\rho$ -VEX brings this functionality to the embedded domain, by means of the differences discussed in this section. The first difference with SMT is that the assignment is performed in pipeline *groups*, where each group of pipelines is able to execute the full instruction set. This has two advantages in the embedded domain: Firstly, it allows unused pipelines to be fully disabled to reduce power consumption. Secondly, it removes the possibility for pipeline resource contention between concurrent threads.<sup>7</sup> The organization of pipelines into groups is a design-time parameter as listed in Table 1.1. The default  $\rho$ -VEX configuration organizes pipelines into pairs, where both pipelines have an ALU and Multiplier, one pipeline has a memory unit and the other a branch unit.

Each pipeline that is assigned to a context, can potentially execute one operation from the program during each cycle, so assigning more pipelines to a context

<sup>7</sup>Consider the case where an SMT processor has eight parallel pipelines, two of which are able to execute a memory operation. When running in SMT-2 mode, two threads can both dispatch up to four operations in the same clock cycle. However, when both threads want to execute two memory operations, one of them will need to wait for the memory lanes to become available, leading to a well-known drawback of SMT systems: performance interference.

may increase its performance. However, the availability of an operation depends on a number of factors (including branch penalty and memory stall cycles, not further discussed here), most notably the instruction-level parallelism in the code. High-performance superscalar processors use complex circuitry to detect whether operations can be executed in parallel or not. Some processors, such as the POWER8, perform register renaming and instruction re-ordering to increase the available parallelism, at the cost of additional power consumption and circuit area.

In contrast, the  $\rho$ -VEX uses a VLIW-style architecture [16] that relies on the compiler to find this parallelism and encode it explicitly in the binary. Apart from decreasing circuit complexity and power consumption, this provides a way to measure the amount of ILP in the code using simple hardware mechanisms. The binaries are encoded in such a way that they can be executed in all supported processor configurations [29], and allow horizontal NOP removal [23].

Using a VLIW architecture provides another advantage for the embedded domain, in the form of a high degree of time-predictability. Contemporary systems are focused on delivering very high performance for the common case (by using techniques such as caches and branch prediction), but it is very difficult on these platforms to reason about the worst case, which is necessary to rule out deadline misses for critical tasks. The  $\rho$ -VEX compiler generates code that is completely statically scheduled, which means that all pipeline latencies are explicitly dealt with (not relying on pipeline interlocking<sup>8</sup>). Branches are optimized by reordering basic blocks at compile-time, using heuristics or code profiling to find the most probable control flow. The result is that code is executed exactly as scheduled by the compiler, assuming a perfect memory system (in Chapter 8, we propose to use single-cycle local scratchpad memories for critical tasks).

The last differences with SMT architectures are that the pipeline to context assignment is explicit, and the temporal granularity is slightly reduced. In SMT machines, the issue slots are distributed among the threads by the hardware each cycle. The distribution can be performed using a simple round robin method (such as a barrel processor), using heuristics such as which threads have the least outstanding cache misses, or using sophisticated priority-based methods (as used by the POWER8). The  $\rho$ -VEX uses a control register that can be written to change the pipeline to context assignment. Such a *reconfiguration request* requires approximately 4 cycles latency to decode (depending on the current and new configuration), and results in 5 cycles penalty to flush the pipeline and start the new configuration. Context-pipeline couplings that are not changed by the reconfiguration request continue to execute unaffected. Decreasing the temporal granularity is a design choice that considerably reduces design complexity (and, as a result, circuit complexity), but is not a fundamental limitation of the  $\rho$ -VEX's concept.

In summary, the  $\rho$ -VEX design has the following advantages in the embedded

<sup>8</sup>Pipeline interlocking means that hardware circuitry detects hazards between instructions, and stalls the pipeline if necessary. Using a VLIW architecture does not prevent the usage of pipeline interlocking: a notable example is the st231 VLIW processors from STMicroelectronics. The increase in circuit complexity may be compensated by the reduction in binary size (resulting in improved instruction cache hit rates and reduced instruction fetch bandwidth requirements) because it allows the removal of vertical NOPs [16, Section 3.5.2].

application domain:

- **Reduced circuit complexity** because of the slightly reduced temporal granularity and VLIW-style architecture.
- **Performance isolation** between contexts because of a fixed resource assignment, statically scheduled VLIW code, and a private memory access port per lanepair.
- **Performance predictability** because of statically scheduled VLIW code, explicit pipeline latencies, and the use of scratchpad memories.
- Potential **power savings** by allowing datapaths to be disabled as each individual lanepair can execute the full ISA.

Making the configuration explicit and providing an interface to request adaptations, allows the following methods to perform configuration optimization:

- The **user** can manually request a configuration.
- The **programmer** can manually request a configuration at any point in his code (for example, at the beginning of a loop).
- The **Operating System** (OS) can change configuration during scheduling based on system policy (high performance, energy efficient) or task priority.
- The **designer** can create a circuit that changes the configuration based on sensor input such as temperature/battery level, or based on performance monitors.
- The **compiler** can insert configuration change requests at certain points in the code.

The first two require manual intervention. This thesis focuses on exploiting the last three automatic methods to adapt the dynamic processor to the workload. A more detailed rationale of the run-time adaptable version of the  $\rho$ -VEX and discussion of related architectures is presented in [30].

### 1.6.3. Environment

The HDL implementation of a processor is only a small part of a processing environment. A large set of tools is required to be able to execute programs on it, including compilers, operating systems and libraries. This section gives an overview of these tools and the efforts that have been made in this area to be able to use the  $\rho$ -VEX as intended. Where possible, we have ported the tools that were available for the st200 series of VLIW processors by STMicroelectronics. The st200 VLIW family is the architectural sibling of VEX, the ISA of the  $\rho$ -VEX. VEX and st200 are both descendants from the Lx architecture created by HP (Hewlett-Packard) laboratories in collaboration with STMicroelectronics in the 1990s [31] and share many similarities.

### Toolchain

The  $\rho$ -VEX is supported by an elaborate toolchain that contains several compilers, a port of the GNU binutils toolcollection, a debugger, and several libraries. Within the context of this work, the open source st200 compiler, that is based on Open64, was ported to the  $\rho$ -VEX and integrated into the existing toolchain. As this compiler was maintained by STMicro for their industrial st200 VLIW family, the performance, reliability, and compatibility is higher compared to the available alternatives including HP VEX that is based on the Multiflow compiler and provided to the academic community by [16].

The  $\rho$ -VEX toolchain currently provides two C standard runtime libraries: newlib and uClibc. Newlib can be linked together with programs that are intended to run *bare-metal* – without any operating system directly on the processor. It provides a system call interface that, in its turn, relies on a low-level library that implements functionality such as memory allocation. A simple filesystem library can be generated that implements the `open`, `close`, `read`, `write`, and `lseek` POSIX system calls. uClibc is based on the st200 port from the STLinux distribution, and can be linked together with programs that are intended to run on Linux.

In addition to the C standard library, a math library (libm) and a runtime support library that contains floating point emulation and various division routines. Lastly, experimental support is available for pthread and OpenMP (by means of the  $\rho$ -VEX port of the LLVM 3.9 compiler and the GNU OpenMP library, libgomp).

### Operating Systems

In an earlier project, uCLinux (micro-controller Linux) was ported to the  $\rho$ -VEX [32]. It is highly experimental and uses a limited version of an early Linux kernel (2.0 nommu). It does not support virtual memory as the  $\rho$ -VEX does not have a Memory Management Unit (MMU). This means that there is only a single address space in which all active applications must execute alongside each other and the kernel itself. To this end, we have ported the bFLT flat file format, that writes all the relocation information generated by the linker into the application binary. This information is then parsed by the bFLT loader in the Linux kernel, that was modified to fill in all the relocations based on the address where the kernel wants to place the program in memory.

### Simulator: `sim-rvex`

An architectural simulator for the  $\rho$ -VEX is available named `sim-rvex`. It was implemented in C and based on a very early (open source) version of the `sim-200` simulator. It is binary compatible with the HDL implementation and provides the following features:

- Control registers including the full set of performance counters
- Dynamic reconfigurability and multiple contexts
- Tracing execution of multiple concurrently executing contexts
- Trace output shows disassembly and symbols loaded from the program's ELF file

- Optional system call emulation on host machine
- Optional cache simulation
- Simulates several peripherals including UART, framebuffer, timers, interrupt controller

With cache simulation disabled, the simulator is cycle accurate with respect to the HDL implementation of the processor connected to single-cycle memories. With cache simulation enabled, the accuracy decreases significantly as the cache model simply uses miss penalties instead of fully modeling a DRAM controller connected via a bus. The simulator achieves up to approximately 100 MIPS<sup>9</sup> on an Intel i5-4690 CPU @ 3.50GHz.

#### 1.6.4. Platform overview

Figure 1.8 displays an overview of the full system stack of the  $\rho$ -VEX environment, and Figure 1.9 shows the toolflow of running applications on a  $\rho$ -VEX hardware prototype or simulator.

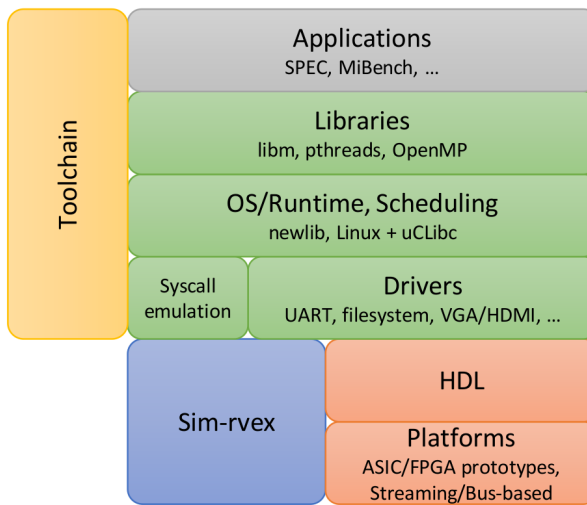


Figure 1.8: Overview of the full system stack.

## 1.7. Approach

This section discusses the tools and methods that are used to evaluate the proposed platform.

<sup>9</sup>Here, MIPS denotes millions of simulated instructions per second, with an instruction being a single operation within a VLIW bundle. The performance of the simulator is highly dependent on the simulated program.

1

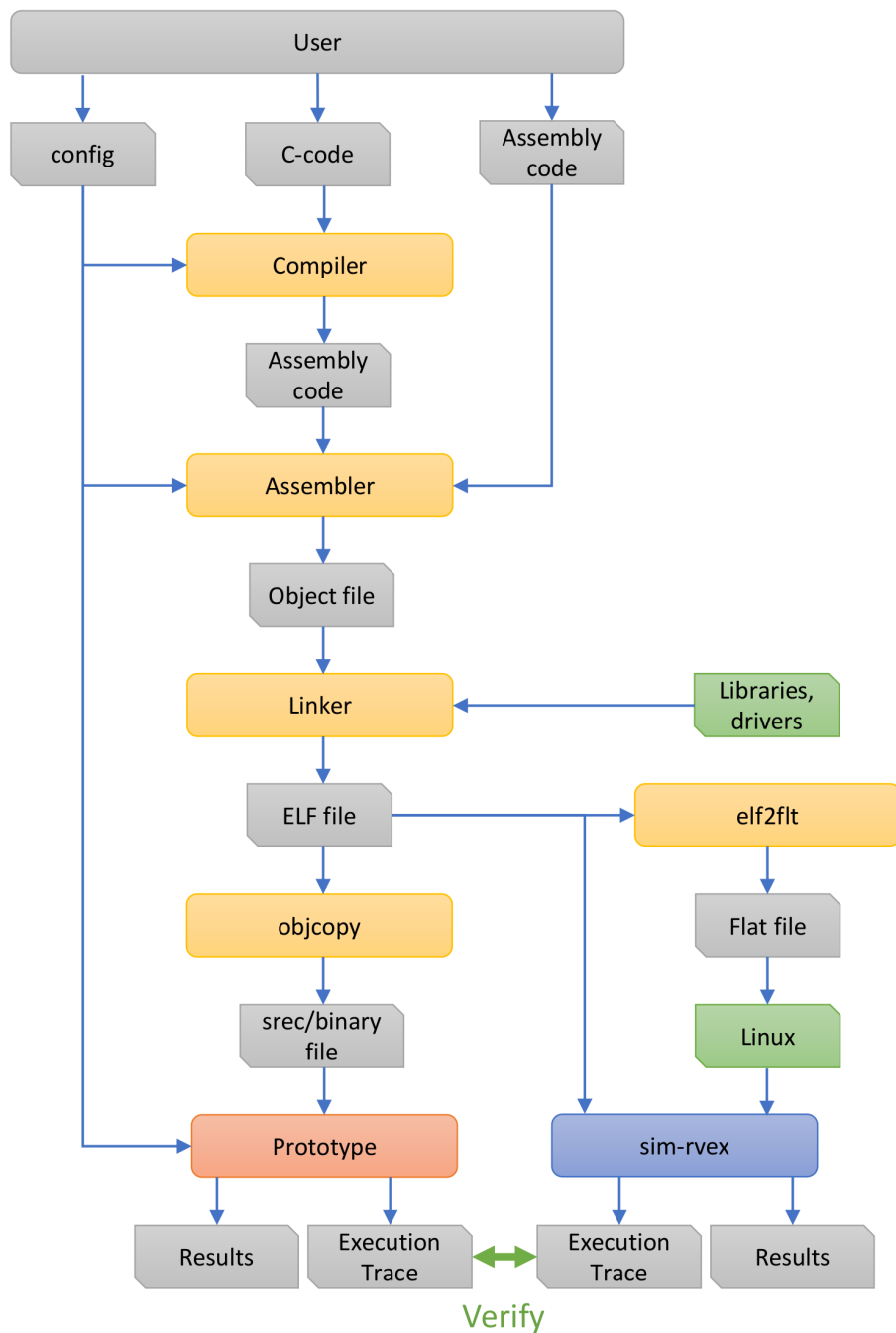


Figure 1.9: Overview of the toolchain.

Code repository name	Description
almaif	ALMARVI interface for the Portable OpenCL framework (pocl)
asic-pcb	Printed Circuit Board design for the ASIC prototype
asic-synthesis	Scripted toolflow for the ASIC prototype
binutils-gdb <sup>*</sup>	Utilities concerning $\rho$ -VEX binaries such as assembler and linker. Also contains the GNU Debugger (GDB)
confsched <sup>†</sup>	Configuration scheduler
elf2flt <sup>†</sup>	Utility to convert ELF (Executable and Linkable Format) files to relocatable Flat format
FreeRTOS	Real-time OS
GCC <sup>*</sup>	Compiler
LAO <sup>*</sup>	Linear Assembly Optimizer
libgcc-rvex <sup>*</sup>	Run-time library for software emulation of division and floating-point operations
LLVM-OMP <sup>*</sup>	Compiler with OpenMP support
streaming-rvex <sup>*</sup>	Manycore streaming platform
Mälardalen <sup>*</sup>	Benchmark suite
MiBench <sup>*</sup>	Benchmark suite
newlib <sup>*</sup>	Runtime library implementing the C standard library functions (bare-metal)
Open64 <sup>†</sup>	Compiler
PolyBench <sup>*</sup>	Benchmark suite
Powerstone <sup>*</sup>	Benchmark suite
rvex-debug	Interface to the hardware debug unit (core revision 3)
rvex-drivers	Device drivers for various peripherals
rvex-hdl	Hardware implementation (core revision 3)
rvex-linux-2.0 <sup>†</sup>	Linux OS kernel (nommu)
rvex-vhdl <sup>*</sup>	Hardware implementation (core revision 2)
sim-rvex <sup>†</sup>	Architectural simulator
SPEC <sup>*</sup>	Benchmark suite
testfloat <sup>*</sup>	Floating-point test suite
toolbuild	Script to build a design and all required tools for a given configuration
uCLibc <sup>*</sup>	Runtime library implementing the C standard library functions (Linux)
USB-grlib-JTAG-bridge <sup>*</sup>	Interface to the hardware debug unit (core revision 2)
vexparse	Assembly-level rescheduler for generic binaries

<sup>\*</sup> Some effort or a simple port was required for use in this work

<sup>†</sup> Considerable effort or full implementation was required for use in this work



### 1.7.1. Modeling & Simulation

To rapidly evaluate architectural concepts, a common approach is to implement them in multiple steps of increasing accuracy. First, to evaluate a general concept, a model can give a rough estimation of how the idea might perform. In the case of polymorphic processors, this model is presented by [9].

A next step is to implement an architectural simulator that is able to execute (interpret) actual code targeting the envisioned processor. An architectural simulator can have different degrees of simulation speed and accuracy (see [16, p254]). In this work, most of the evaluations were performed using an architectural simulator called `sim-rvex` that is discussed in more detail in Section 1.6.3.

Finally, the actual implementation can be performed using a Hardware Description Language such as VHDL and simulated using a circuit-level or gate-level simulation. These simulations are mostly used for design verification and are not suitable to evaluate architectural concepts due to their long execution times. Instead, they can also be synthesized to reconfigurable hardware (FPGA) which is discussed in the following section.

### 1.7.2. Using FPGA technology

FPGA technology is used for two distinct purposes that are discussed in the following sections; to prototype designs targeting ASIC technology during development, and for designs that are not expected to be sold in large volumes.

#### FPGA-based implementations

Manufacturing a design as an ASIC (Application-Specific Integrated Circuit), for example using CMOS (Complementary Metal Oxide Semiconductor) technology, is a very time-consuming and expensive process. For designs that are not expected to be sold in enormous volumes (millions), an FPGA implementation is a possible alternative. FPGA chips can be programmed with a circuit design that will be mimicked using large quantities of configurable logic blocks and interconnections. The resulting product will operate at reduced clock frequencies and energy efficiency compared to an ASIC. Developing an ASIC will take at least several months and requires an enormous investment for the mask set that is needed for manufacturing (depending on the technology used, this can cost anywhere from 100.000 to millions of Euros). In contrast, an FPGA is an off-the-shelf component, and a design can be synthesized in a matter of hours. This makes it a suitable platform for designs that are highly optimized for a particular application, as is the case for our design-time reconfigurable processor targeting static workloads. When implemented on an FPGA, these processors are referred to as *softcore* processors. The dynamically reconfigurable processor, as it is meant to adapt itself to various workloads, is aimed more towards ASIC implementations.

#### FPGA Prototyping for ASIC implementations

FPGA technology is also used to aid the development of ASICs. FPGA prototyping is a commonly used method to verify a processor design [33], not only regarding functionality (i.e., is the implementation correct?) but also to explore the feasibility

(e.g., estimating how much circuit area a certain component will require) and various performance metrics. These can include for example the number of logic levels to identify critical paths, the expected clock frequency, and fully cycle-accurate execution times of certain benchmarks. As FPGA implementations can achieve operating frequencies in the range of several MHz, benchmarks that would require weeks or months to execute on a circuit simulator can be executed on an FPGA prototype in a matter of hours [34].

The  $\rho$ -VEX is implemented in VHDL and its default configuration achieves an operating frequency of 80MHz on a Xilinx Virtex7 FPGA. As discussed, the FPGA prototype is used mostly for verification and feasibility checking. Our architectural simulator performs relatively fast, and can be executed on a high performance computer with many instances running concurrently. This approach was used for most of the measurements presented in this work.

### 1.7.3. Code characterization method

To adapt a processor to the workload, an important step in finding the most suitable configuration is to measure the characteristics of the currently running code. The first question here is *what* characteristics to measure, as there are numerous. One of the key properties of a VLIW architecture is that parallelism is encoded explicitly in the code, which allows it to be measured using a simple counter mechanism. This information gives a strong indication of the performance using different core configurations.

The second question is *how* to perform the measurements, which is one of the topics of part 2. There are two general approaches that are introduced here.

#### Compiler-based

As VLIW processors are statically scheduled (the assignment of operations to parallel datapaths is performed by the compiler), it is possible to analyze the number of cycles that is required to execute any code section using a certain core configuration. We modified the  $\rho$ -VEX Open64 compiler so that it annotates each relevant section of code (particularly loops and straight-line functions of a certain minimum length) with the number of execution cycles for every supported core configuration of the  $\rho$ -VEX.

#### Monitoring-based

In addition to analyzing code at compile-time, it is possible to use performance counters (monitors), embedded in the processor, to measure the amount of parallelism. The  $\rho$ -VEX already includes a counter that keep track of the number of committed operations, and a counter for the number of committed VLIW bundles. The ratio between these metrics represents the average datapath utilization, which gives an indication of the expected performance using different core configurations.

### 1.7.4. A runtime for scheduling tasks and configurations

Now that information about code performance is available, a runtime system is required that uses this information to optimize the processor configuration to the

workload. We implemented a run-time scheduler for the  $\rho$ -VEX, written in C, that not only assigns tasks to cores based on a taskgraph with inter-task dependencies, but also samples the code characteristics periodically and requests processor adaptations accordingly.

### 1.7.5. Workload generation

The input for the scheduler is a taskgraph with dependencies. The purpose of the taskgraph is to create a workload with varying number of concurrently executing tasks (dynamic in intensity). Combined with randomly selected tasks with different characteristics, this causes both the amount and type of parallelism (Instruction-level versus Thread or Task-level) to vary over time. The taskgraph generator is a script with the following input parameters:

- A list of tasks that will be randomly sampled (simple random sampling without replacement),
- The maximum number of tasks to sample (the generator will randomly choose a number of tasks between 1 and this number),
- The maximum number of dependencies to create per task.

The output is a C file that includes arrays with task IDs and dependencies. It can be included in full by (or linked together with) the scheduler code. The script will consequently rebuild the scheduler with the generated file, and call the simulator with all the required binaries for each benchmark in the taskgraph. As the  $\rho$ -VEX by default does not include a memory management unit (MMU), it has only a single address space in which all programs must be able to execute concurrently. To support this, each benchmark binary has been linked with a distinct start address, taking care that no two programs will every access overlapping memory regions. Another option is to encode every binary using the bFLT (binary flat) format as is used by the  $\rho$ -VEX Linux port, and subsequently have the scheduler allocate sufficient memory and relocate the program at start time. We did not choose this method as it is more complex.

### 1.7.6. Benchmarks

We have used a number of different benchmark suites for our evaluations. They were ported to run bare-metal on the  $\rho$ -VEX using the newlib C standard library to minimize any overhead. The benchmark suites each represent a certain application domain.

- **SPEC**, the Standard Performance Evaluation Corporation, provides the industry-standard SPEC benchmark suite, primarily targeting workstations (high-performance desktop machines). As the  $\rho$ -VEX does not target floating point applications, SPEC CINT2006 was used. Not all programs are supported by the  $\rho$ -VEX toolchain.
- **MiBench** [35] is a benchmark suite that targets the embedded domain. It contains the most relevant programs for our dynamic embedded workloads.

The suite includes programs from different embedded application domains including media, communications, and automotive.

- **PolyBench** contains applications with heavy numerical computations that are common in the high-performance computing domain.
- The **Mälardalen WCET Benchmark suite** was used for real-time applications.

## 1.8. Contributions and thesis outline

Reflecting back to the scope of this work discussed in Section 1.5, we have identified two general categories – static and dynamic – regarding software (our targeted workload types), hardware (processor reconfigurability), and scheduling. This creates a taxonomy as depicted in Table 1.2. Classical approaches use fixed (static, non-reconfigurable) hardware to execute both static and dynamic workloads. The scheduling methodology depends on the application domain. Typically, it will be dynamic, but in some cases a static schedule may be desirable (for example in some real-time systems). In the first part of this dissertation (Chapters 2 - 4), we will use static hardware (design-time optimized) to execute static workloads. The scheduling method used is also static, as we will only perform task mapping at design-time. Each core continuously performs the same calculation on new input data, therefore there is no ordering or partitioning required in the time domain. In Part 2, Chapters 5 and 6 will use the run-time parameterized (dynamic) hardware to execute dynamic workloads. The scheduling method used is also dynamic, as both the processor configuration, task mapping and ordering are determined during run-time. In addition to dynamic workloads and dynamic scheduling, Chapters 7 - 8 in Part 3 add real-time requirements to the workloads which necessitates a static schedule. Lastly, it will add a static component to the workload and adds a dynamic scheduler.

	Hardware	Software	Scheduling
<b>Classical</b>	Static	Static & Dynamic	Static & Dynamic
<b>Part 1</b>	Static (fixed, design-time)	Static	Static (compile-time)
<b>Part 2</b>	Dynamic (adaptable, run-time)	Dynamic	Dynamic (run-time)
<b>Part 3</b>	Dynamic (adaptable, run-time)	Static & Real-time	Static (compile-time) & Dynamic (run-time)

Table 1.2: Overview of the approaches in the different overall parts of this thesis

In more detail, the individual parts provide the following contributions:

### Part 1 - Static workloads, statically reconfigurable platform

Part 1 introduces a design-time customizable computation fabric based on VLIW

softcore processors and a streaming memory hierarchy:

- First, the suitability of a VLIW-based architecture for this application domain is evaluated in Chapter 2. We show up to a factor of 3.2× better performance with similar resource utilization compared to the industry-standard MicroBlaze processor, as published in [36].
- Second, in Chapter 3 we introduce the computational fabric with design-time optimizable memory hierarchy, and show that streaming data directly between cores results in considerably better performance compared to a bus-based topology [37].
- Lastly, in Chapter 4 we show how this stream-based platform can be programmed using OpenCL in a frame-based fashion, abstracting away the hardware complexity from software programmers. This platform can be used for rapid prototyping, debugging and optimization to bridge the gap to High-Level Synthesis (HLS). It features a wide number of configuration parameters that can be explored by the designer without needing to program or modify HDL code. Additionally, it has improved scalability compared to a similar platform targeting the biomedical imaging domain, allowing it to increase the core count and operating frequency on an FPGA [38].

## Part 2 - Dynamic workloads, dynamically reconfigurable platform

In Part 2, we introduce mechanisms that allow a polymorphic processor to automatically evaluate code characteristics of a highly dynamic workload and adapt accordingly:

- We evaluate the ability of fine-grained reconfigurable processors to closely match dynamic program characteristics using high frequency adaptations in Chapter 5 [39].
- Chapter 6 studies if automatic reconfigurations enable reconfigurable processors to achieve better performance on dynamic workloads compared to static heterogeneous processors [40].

## Part 3 - Real-time and mixed-criticality systems

In Part 3, we add real-time requirements to the workload, and explore mixed static and dynamic workloads and scheduling:

- In Chapter 7, the architectural properties of the  $\rho$ -VEX that are advantageous for real-time systems (e.g., low interrupt latency) are examined [41].
- A method to increase static real-time schedulability by leveraging the dynamic properties of the run-time parameterizable processor are evaluated in Chapter 8, as published in [42].
- Lastly, the platform architecture targeting Mixed-Criticality systems proposed in Chapter 8 is evaluated in terms of increasing throughput for non-critical

static tasks. Time-safety for critical tasks is still guaranteed by dynamically (i.e., at run-time) re-assigning cycles that are left unused in the static schedule [43].

In Chapter 9, we summarize and conclude the work and mention several possible future research directions.



# Part 1 - Static workloads, statically reconfigurable platform

*The first chapter of this thesis introduced a number of embedded domains with each their own set of requirements and characteristics. Part 1 focuses on the static domain, where a continuous stream of data must be processed in a highly structured and repetitive fashion (e.g., image processing). The usage of vision-based applications is increasing in multiple mobile domains (smartphones, action cameras, surveillance, automotive). In these domains, computational power is required but power budgets are limited.*

*The aim of Part 1 is to demonstrate how a design-time configurable processor, based on a VLIW-style architecture, is able to effectively target these types of workloads. As explained in Section 1.7, we are using FPGA prototyping to evaluate the platform and the proposed concepts. Chapter 2 examines the suitability of using our proposed VLIW architecture in comparison with the industry-standard MicroBlaze softcore processor. Chapter 3 proposes a design-time optimizable FPGA computation fabric using VLIW processors and a custom memory hierarchy to efficiently stream data through the processors. Chapter 4 adds support for programming this framework in OpenCL, allowing the programmer to keep a frame-oriented view. Additionally, it shows how it can be used to perform rapid design-space exploration and debugging for FPGA-based image processing frameworks.*

	Hardware	Software	Scheduling
Classical	Static	Static & Dynamic	Static & Dynamic
<b>Part 1</b>	<b>Static (fixed, design-time)</b>	<b>Static</b>	<b>Static (compile-time)</b>
Part 2	Dynamic (adaptable, run-time)	Dynamic	Dynamic (run-time)
Part 3	Dynamic (adaptable, run-time)	Static & Real-time	Static (compile-time) & Dynamic (run-time)





# 2

## Using VLIW softcore processors for image processing

*In Section 1.6, we have proposed to use a VLIW-based reconfigurable processor to target a spectrum of workloads, including image processing. This chapter studies the suitability of the VLIW architecture for executing image processing filters that are common in this application domain. As we are focusing on FPGA-based devices, we will compare our proposed VLIW processor with an industry-standard softcore processor, the Xilinx Microblaze.*

## Abstract

The ever-increasing complexity of advanced high-resolution image processing applications requires innovative solutions to ensure addressing this issue efficiently and cost effectively. This chapter discusses the utilization of reconfigurable general-purpose softcore processors in image processing applications such that hardware resources are efficiently utilized and at the same time ensure high image processing performance for the targeted application. Results show that the  $\rho$ -VEX softcore processor can achieve remarkably better performance compared to the industry-standard Xilinx MicroBlaze (up to a factor of 3.2 times faster) on image processing applications.

### 2.1. Introduction

Whenever image/video processing is an integral function of the end user product, the stringent performance and power consumption requirements (that are hard to meet in software) are often fulfilled by embedding the imaging/video functionality as hardware accelerators implemented in FPGAs or as ASICs. The main advantages of embedded accelerator implementations of image/video processing functions lie in computation speed, high energy and area efficiency, etc. However, the transition from pure software prototypes towards production-grade FPGA or ASIC-based systems is associated with high engineering and manufacturing cost. Moreover, hardware development requires expensive toolsets and dedicated know-how, which usually results in a relatively high per-unit cost due to smaller production quantities and higher customization overhead. As a result, the development cycles of hardware approaches increasingly lag behind the demands of the fast-paced markets.

In recent years, however, software-based approaches on commodity hardware, notably on embedded graphics processors (GPUs) and multi-core CPUs, have increasingly gained attention. Although GPUs seem like the most logical choice to accelerate imaging applications [44], they have a number of characteristics that cause them to fail requirements in some cases. In these cases, FPGAs may be preferred. First, they draw considerable amounts of power. Second, some product areas such as medical imaging systems (e.g., X-ray) require the availability of system components over an extended period of time (up to 15 years). However, in these areas there are conflicting requirements such as a high degree of maintainability, that are normally not compatible with FPGA acceleration (changes to the software could lead to required changes in the acceleration fabric which is tedious). Therefore, it is highly desirable to have an acceleration fabric that is more easily programmable than the reconfigurable logic itself. Putnam et al. [45] shows the viability of using application-tailored softcores to speed up datacenter applications. [46] presents a softcore specifically designed to perform fast fourier transforms at efficiency levels comparable to that of dedicated FPGA circuits. A softcore-based environment benefits from standardized hardware design and a pre-existing development platform and toolchain that allows it to be programmed using common programming languages. A single softcore will not be able to achieve performance levels similar to dedicated accelerators written in VHDL (or synthesized using high-level synthe-

sis). However, the total system performance for data-intensive applications (such as image processing) will often be bound by the available memory bandwidth (as is true for any multicore system [47]). Moreover, the parallel nature of image processing provides a high level of scalability for multicore systems. This means that if the number of softcores that can be placed on the FPGA is sufficiently high as to achieve “wirespeed performance” (i.e., fully utilize the available bandwidth to the FPGA), the application speedup of the softcore-based system will be equal to an implementation that uses dedicated FPGA accelerators but with reduced development effort and increased maintainability.

In this chapter, we propose to use the  $\rho$ -VEX softcore based on the VEX ISA for image processing applications (which is one of the main application domains for this architecture) considering the aforementioned scenario. Our  $\rho$ -VEX softcore implementation is design-time reconfigurable and run-time parametrizable which allows it to adapt to varying requirements of applications. This has the promise of providing low development cost and good maintainability as well as efficient resource utilization to achieve efficient image processing power. In this chapter, we will show that the  $\rho$ -VEX exhibits good performance in the application domain compared to an industry standard softcore (the Xilinx MicroBlaze).

The chapter is organized as follows. Section 2.2 discusses related work. Section 2.3 presents the  $\rho$ -VEX platform including the ISA, the toolchain and the softcore design. Section 2.4 discusses the applications used in the evaluation. Section 2.5 presents the test setup and the measurement results, and Section 2.6 concludes the chapter and discussed possible future directions for research.

## 2.2. Related work

Spyder [48] appeared as the first softcore VLIW processor. The provided toolchain was not complete and the processor was not run-time reconfigurable. An FPGA-based design of a softcore VLIW processor based on the ISA of the Altera NIOS-II soft processor is presented in [49]. The compilation scheme consists of a Trimaran [50] as the frontend and the extended NIOS-II as the back-end. Due to the licensed Altera NIOS-II, this VLIW design is not very flexible and not open-source. Additionally, the design is not run-time reconfigurable. In [51], a modular design of a VLIW processor is reported. Certain parameters of the processor architecture could be altered in a modular fashion. In [52], the architecture and micro-architecture of a customizable softcore VLIW processor are presented. Additionally, tools are discussed to customize, generate, and program this processor. The limitation is the absence of a compiler. A VLIW processor with reconfigurable instruction set is presented in [53]. In this case, a reconfigurable unit is coupled to a VLIW processor. The co-processor can be configurable for any custom instruction. The  $\rho$ -VEX is different from this design in the sense that it does not couple a reconfigurable co-processor. We can add a custom unit to the data paths of our processor at design time and reconfigure the issue slots at run-time. In [54], we present the rationale and the design and implementation of an open-source softcore VLIW processor. This processor is design-time parametrized and can be configured to make its issue-width adjustable during run-time [22][55].

## 2.3. The $\rho$ -VEX platform

### 2.3.1. The VEX system: ISA and toolchain

The VEX stands for VLIW Example [16]. The VEX is developed by Hewlett-Packard (HP) and STMicroelectronics. The VEX instruction set architecture (ISA) is a 32-bit clustered VLIW ISA that is scalable and customizable to individual application domains. The VEX ISA is loosely modeled on the ISA of HP/ST Lx (ST200) family of VLIW embedded cores [16]. Based on trace scheduling, the VEX C compiler is a parameterized ISO/C89 compiler. A flexible programmable machine model determines the target architecture, which is provided as input to the compiler. A VEX software toolchain including the VEX C compiler and the VEX simulator is made freely available by the Hewlett-Packard Laboratories [56].

### 2.3.2. The $\rho$ -VEX VLIW processor

The  $\rho$ -VEX is a configurable (design-time) open-source VLIW software processor [21]. The ISA is based on the VEX ISA [56]. Different parameters of the  $\rho$ -VEX processor, such as the number and type of functional units (FUs), number of multiported registers (size of register file), number and type of accessible FUs per syllable, width of memory buses, and different latencies can be changed at design time.

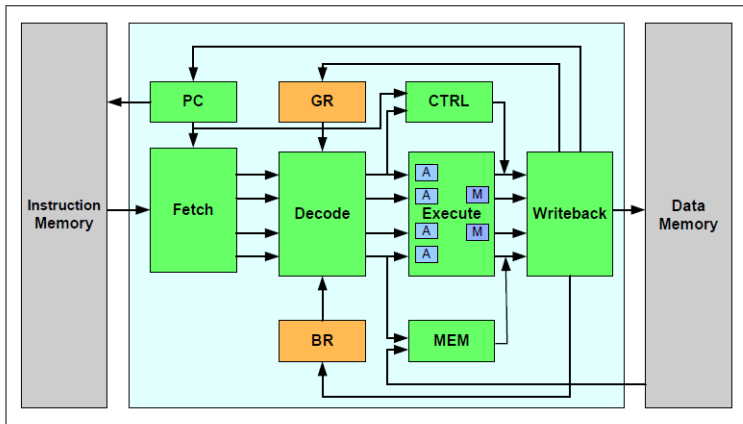


Figure 2.1: Design overview of a 4-issue instance of the  $\rho$ -VEX VLIW software processor.

Figure 1 depicts the organization of a 32-bit, 4-issue  $\rho$ -VEX VLIW processor. The  $\rho$ -VEX processor consists of fetch, decode, execute, and writeback stages/units. Operations take place in either the parallel Arithmetic logic unit (A) and multiplier (M) units, or the branch (CTRL) or load/store (MEM) units. All jump and branch operations are handled by the CTRL unit, and all data memory load and store operations are handled by the MEM unit. The different write targets could be the general register (GR) file, branch register (BR) file, or data memory. All operations normally have a delay of one cycle, except for MEM and MUL operations

which need an extra cycle. The core contains forwarding logic to minimize pipeline stalls. Additionally, the  $\rho$ -VEX processor supports reconfigurable operations, as the VEX compiler supports the use of custom instructions via pragmas within an application code. The instruction and data caches for the processor are implemented with BRAMs (Block RAM resources on the FPGA). The GRLIB SoC library [57] is used to connect the processor core to off-chip DDR memory and peripherals via an AMBA bus system. This setup allows us to use any IP that is compatible with this bus, and to use existing tools to connect to the board in order to load applications, start/stop the core, etc. The GRLIB library also contains the framebuffer used to visually inspect the result images.

The  $\rho$ -VEX core can be configured at design time to be dynamically (run-time) reconfigurable or not. When configured to be dynamic, it can couple or decouple its datapaths to either run in a single-core mode with a large issue-width, or in a multi-core mode with smaller processors that can run separate tasks or threads. Dynamic reconfigurability will result in the flexibility to balance instruction-level parallelism (ILP) for high performance on a single thread with thread-level parallelism (TLP) for applications with low ILP but that can benefit from utilizing multiple threads. This comes at a cost of increased FPGA resource utilization. Partial reconfiguration is not needed for this concept to work; the principle is applicable also for ASIC implementation. This chapter focuses on the architecture and not the dynamic reconfigurability. The applications used in this chapter for evaluation do not exhibit dynamic behavior (see Section 2.4) and as such are not suitable to study these properties. Therefore, the processor core used in this chapter is a static (not run-time reconfigurable), 4-issue VLIW. Using a higher issue width and/or dynamic reconfigurability will result in increased resource utilization and possibly lower operating frequency.

## 2.4. Image processing applications

To evaluate the suitability of our architecture for imaging applications, we implemented two basic algorithms that are commonly found in the application domain: a greyscale converter and a convolution filter with adaptable filter size. Both applications can be used for varying image sizes and the convolution filter can be configured to use different kernels, each performing a different operation on the image. The execution time is independent of the values used in the kernel. The images are represented in memory as an array containing a 32-bit word per pixel. This representation allows it to be displayed directly using a framebuffer device on the FPGA to facilitate visual inspection of the resulting images.

The greyscale converter is essentially a single loop performing a single operation on every pixel. The convolution filter performs a number of operations per pixel, depending on the size of the kernel. Both applications are representative for image processing steps in a medical imaging system. The greyscale converter represents the step of assigning a color value to the output of the sensor, depending on the precision of its output (which is often higher than 8 bits). The convolution filter can be used for a range of operations such as edge detection and image sharpening.

Care has been taken to ensure that the measurements contain as little overhead

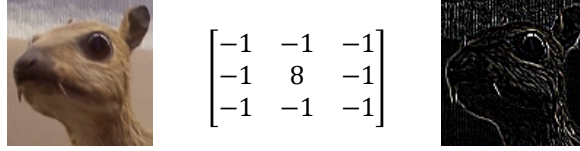


Figure 2.2: Example of a convolution kernel that performs edge detection [58].

	MicroBlaze		$\rho$ -VEX	
	#	%	#	%
Registers	17,477	5%	10,927	3%
LUTs	15,099	10%	18,900	12%
Slices	7376	19%	7506	19%

Table 2.1: Synthesis results

as possible. The programs do not contain input/output inside the measured parts of the program. The programs run without operating system and there are no interrupts enabled in the system.

## 2.5. Results and discussion

We use the Xilinx ML605 board as evaluation platform. The  $\rho$ -VEX is synthesized at 75 MHz and the MicroBlaze was synthesized using the platform studio base system wizard included in the Xilinx ISE 13.4 toolset. The MicroBlaze was created using the “maximum performance” setting of the wizard. The system contains a core that is clocked at 150 MHz and an AXI bus at 75 MHz (as are the default maximum settings). The MicroBlaze contains a multiplication unit but no division or floating point unit. Both cores contain 32 KiB of instruction memory and 1 KiB of data memory. In both cases, the `.text` section of the programs is small enough to fit in the instruction cache. Using data cache sizes of more than 1 KiB did not result in any performance increase for either processors. However, a larger cache prevents the MicroBlaze from meeting its timing at 150 MHz, which necessitates lower clock frequencies, further reducing the MicroBlaze performance. Since the  $\rho$ -VEX runs at a lower frequency of 75 MHz, cache sizes can be increased without further lowering the frequency. The 150 MHz frequency and 1 KiB data cache size were chosen to keep the comparison between the two processors as fair as possible. Synthesis results can be seen in Table 2.1. The resource utilization of both platforms is comparable, with the  $\rho$ -VEX utilizing slightly more lookup tables (LUTs) and the MicroBlaze utilizing more registers.

The same code is compiled for both cores, with pre-processor macros selecting the timer and print functions, and the memory locations of the image in/output according to the location of the DDR RAM in each platform’s memory map. For the  $\rho$ -VEX, our VEX port of the Open64 compiler is used with full optimization (`-O`). For the MicroBlaze, the default GCC-based compiler is used that is included

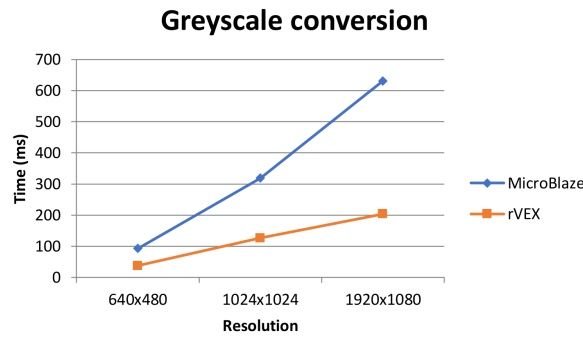


Figure 2.3: Execution time for the greyscale conversion algorithm.

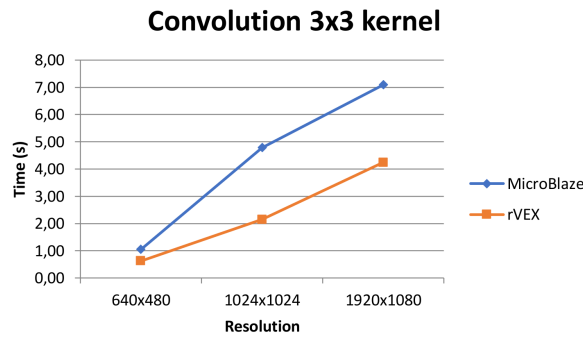


Figure 2.4: Execution time for convolution of a 3x3 kernel.

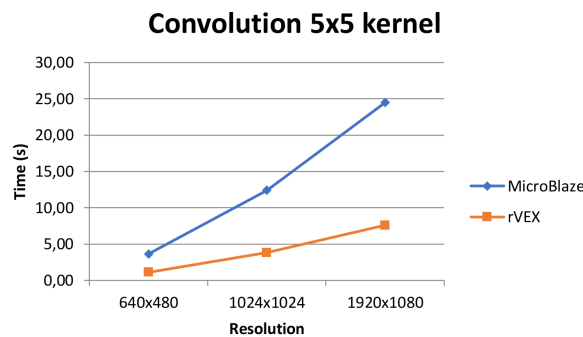


Figure 2.5: Execution time for convolution of a 5x5 kernel.



with the Xilinx toolset. Full optimization was also enabled, but for convolution it appeared that optimizing for size (`-Os`) resulted in significantly better performance. Therefore, size optimization was used when compiling the convolution code for the MicroBlaze. The resolutions used in these experiments include two industry standards (VGA 640x480 and HD 1080p) as well as 1024x1024, a resolution taken from the requirements of an actual medical imaging system.

The execution of the  $\rho$ -VEX is measured by resetting the execution cycle counter that is included in the platform before the program enters the calculation section (thereby removing the overhead of initialization code and printing startup messages to the UART) and reading the number of cycles again when execution has finished. The execution time of the MicroBlaze is measured by starting a timer unit attached to the AXI bus running at 75 MHz, reading its value before and after running the calculations and printing the difference to the UART.

The results can be seen in Figures 2.3 - 2.5. For convolution, the  $\rho$ -VEX is 80% faster compared to the MicroBlaze for the smallest input size and a factor 3.2 times faster for the largest input size. For greyscale conversion, the  $\rho$ -VEX is faster with factors of 2.3 to 3 times.

With the  $\rho$ -VEX being a 4-issue processor at 75 MHz and the MicroBlaze a single-issue processor at 150 MHz, the expected difference in performance between the cores is a factor of 2. However, inspection of the assembly code shows that the compiler uses loop unrolling to decrease branching delays and, more importantly, fill the issue slots so the VLIW can fully utilize all of its resources. As the processor is calculating values for multiple adjacent pixels at the same time, it is able to keep more input pixel values in registers and/or make use of better cache locality compared to the MicroBlaze that needs to reload input pixel values for every inner loop iteration. This effect will continue to have impact as long as the number of available registers is sufficient, as is shown by the growing performance difference between the cores as the problem size increases.

## 2.6. Conclusions

In this chapter, we have shown that the  $\rho$ -VEX softcore processor can achieve remarkably better performance compared to the industry-standard Xilinx MicroBlaze (up to a factor of 3.2 times faster) on image processing applications. In order to be able to use the  $\rho$ -VEX as a competitive platform capable of accelerating industrial grade image processing applications, a number of improvements can be implemented. These improvements are needed in the following of areas:

- How to efficiently stream data to and from the FPGA
- Designing a fast memory hierarchy on the FPGA or a means to efficiently stream data between different cores (each core might perform a certain step in the image processing pipeline, or each core will have a certain part of the image assigned and will perform all the steps)
- Investigating instruction-set extensions that can perform or speed up common image processing operations

# 3

## A streaming FPGA computation fabric

*In the previous chapter, we have seen that the  $\rho$ -VEX VLIW architecture performs up to a factor of  $3.2 \times$  better compared to the general-purpose MicroBlaze in the image processing domain. Still, several cores are needed to achieve the required performance. However, using a standard bus-based connection is not scalable to larger numbers of processors. This chapter discusses how to design a processing fabric using several processors, with a streaming memory structure that can be easily optimized for the application. This structure allows the workload to be spread over as many cores as can fit on an FPGA, without creating a memory access bottleneck.*

## Abstract

In this chapter, we present and evaluate an FPGA acceleration fabric that uses VLIW softcores as processing elements, combined with a memory hierarchy that is designed to stream data between intermediate stages of an image processing pipeline. These pipelines are commonplace in medical applications such as X-ray imagers. By using a streaming memory hierarchy, performance is increased by a factor that depends on the number of stages ( $7.5\times$  when using 4 consecutive filters). Using a Xilinx VC707 board, we are able to place up to 75 cores. A platform of 64 cores can be routed at 193MHz, achieving real-time performance, while keeping 20% resources available for off-board interfacing.

Our VHDL implementation and associated tools (compiler, simulator, etc.) are available for download for the academic community.

### 3.1. Introduction

In contemporary medical imaging platforms, complexity of image processing algorithms is steadily increasing (in order to improve the quality of the output while reducing the exposure of the patients to radiation). Manufacturers of medical imaging devices are starting to evaluate the possibility of using FPGA acceleration to provide the computational resources needed. FPGAs are known to be able to exploit the large amounts of parallelism that is available in image processing workloads. However, current workflows using High-Level Synthesis (HLS) are problematic for the medical application domain, as it impairs programmability (increasing time-to-market) and maintainability. Additionally, some of the image processing algorithms used are rather complex and can yield varying quality of results. Therefore, in this chapter, we propose a computation fabric on the FPGA that is optimized for the application domain, in order to provide acceleration without sacrificing programmability. By analyzing the structure of the image processing workload type (essentially a pipeline consisting of multiple filters operating on the input in consecutive steps), we have selected a suitable processing element and designed a streaming memory structure between the processors.

The image processing workload targeted in this chapter consists of a number of filters that are applied to the input data in sequence. Each filter is a stage in the image processing pipeline. The input stage of a filter is the output of the previous stage - the stages *stream* data to each other. Making sure these transfers are performed as efficiently as possible is crucial to provide high throughput.

The processing element used in this work is based on a VLIW architecture. These type of processors are ubiquitous in areas such as image and signal processing. They are known for their ability to exploit Instruction-Level Parallelism (ILP) while reducing circuit complexity (and subsequently power consumption) compared to their superscalar counterparts. In the medical imaging domain, power consumption is not a main concern, but as image processing workloads can be divided into multiple threads easily, a reduction in area utilization will likely result in an increase in total throughput.

The remainder of this chapter is structured as follows: Section 3.2 discusses

related work, Section 3.3 discusses the implementation details, Section 3.4 and 3.5 present the evaluation and results, and Section 3.6 provides conclusions and future work.

## 3.2. Related work

A prior study on using VLIW-based softcores for image processing applications is performed in [36], showing that a VLIW-based architecture has advantages over a scalar architecture such as the MicroBlaze in terms of performance versus resource utilization. In [59], an FPGA-based compute fabric is proposed using the LE-1 softcore (based on the same Instruction Set Architecture - VEX), targeting medical image processing applications. This work focuses solely on offering a highly multi-threaded platform without providing a memory hierarchy that can sustain the needed bandwidth through the pipeline. A related study on accelerating workloads without compromising programmability is [60], with one of the design points being a convolution engine as processing element. A well-known prior effort, and one of the inspirations of this work, uses softcores to provide adequate acceleration while staying targetable by a high level compiler is the Catapult project [45]. The target domain is ranking documents for the Bing search engine. A related effort that aims to accelerate Convolutional Neural Networks is [61]. However, this project did not aim to conserve programmability (only run-time reconfigurability), as the structure of this application does not change enough to require this. In the image processing application domain, [62] provides a comparison of convolution on GPU or FPGA using a Verilog accelerator, [46] and [63] present resource-efficient streaming processing elements, and [64] introduces a toolchain that targets customized softcores.

## 3.3. Implementation

The computation fabric developed in this work consists of two facets; the processing elements and the memory hierarchy, as shown in Figure 3.1. The implementation of both will be discussed in this section. Then, the process of designing a full platform using these components is discussed.

### 3.3.1. Processing elements

This section describes the design and implementation of our fabric. The processor cores in the fabric are derived from the  $\rho$ -VEX processor [21]. The  $\rho$ -VEX processor is an VLIW processor based on the VEX ISA introduced by Fisher et al [16]. The  $\rho$ -VEX processor has both run-time and design-time reconfigurable properties, giving it the flexibility to run a broad selection of applications in an efficient way.

Image processing tasks are highly parallelizable in multiple regards; 1) The code is usually computationally dense, resulting in high ILP, and 2) Every pixel can in theory be calculated individually and it is easy to assign pixels to threads (by dividing the image into blocks). In other words, there is an abundance of Thread-Level Parallelism (TLP). Exploiting TLP is usually more area efficient than exploiting ILP - increasing single-thread performance comes at a high price in power and area

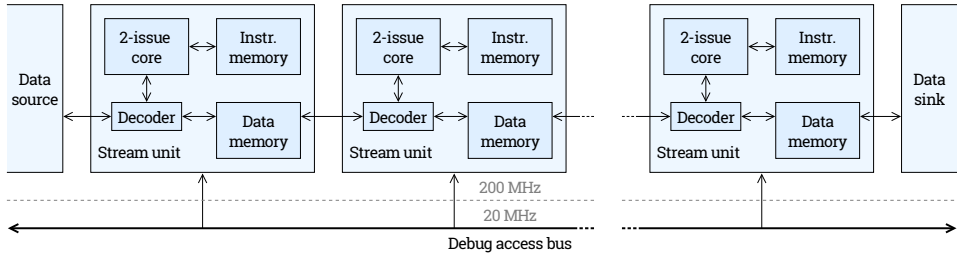


Figure 3.1: Organization of a single stream of processing elements (Stream unit) and the streaming connections that link the data memories. Each processor can access the memory of its predecessor. Each processor's memories and control registers can be accessed via a bus that runs on a low clock frequency to prevent it from becoming a timing-critical net.

utilization and will quickly show diminishing returns. This is why GPUs exploit TLP as much as possible by using many small cores. Therefore, the processing elements of our fabric will use the same approach and we will use the smallest 2-issue VLIW configuration as a basis. This will still allow it to exploit ILP by virtue of having multiple issue slots and a pipelined datapath.

By placing multiple instances of our fabric on an FPGA, TLP can be exploited in two dimensions; by processing multiple blocks, lines or pixels (depending on the filter) concurrently, and by assigning each step in the image processing pipeline to a dedicated core (pipelining on a task level in contrast to the micro-architectural level).

To explore the design space of the processor's pipeline organization, we have measured code size and performance of a 3x3 convolution filter implemented in C. This convolution code forms a basis with which many operators can be applied to an image depending on the kernel that is used (blurring, edge detection, sharpening) so it is suitable to represent the application domain. The main loop can be unrolled by the compiler using pragmas. Figure 3.2 lists the performance using different levels of loop unrolling for different organizations of a 2-issue  $\rho$ -VEX pipeline; the default pipeline with 5 stages and forwarding, one with 2 additional pipeline stages to improve timing, and one using the longer pipeline and with Forwarding (FW) disabled to further improve timing and decrease FPGA resource utilization. Loop unrolling will allow the compiler to fill the pipeline latency with instructions from other iterations. The performance loss introduced is reduced from 25% to less than 2% when unrolling 8 times. Additionally, disabling forwarding reduces the resources utilization of a core allowing more instances to be placed on the FPGA (see Figure 3.1).

### 3.3.2. Memory hierarchy

In our fabric, processing elements are instantiated in 'streams' of configurable length. This length should ideally be equal to the number of stages in the image processing pipeline. Each stage will be executed by a processor using the output of the previous processor. A connection is made between each pair of  $\rho$ -VEX proces-

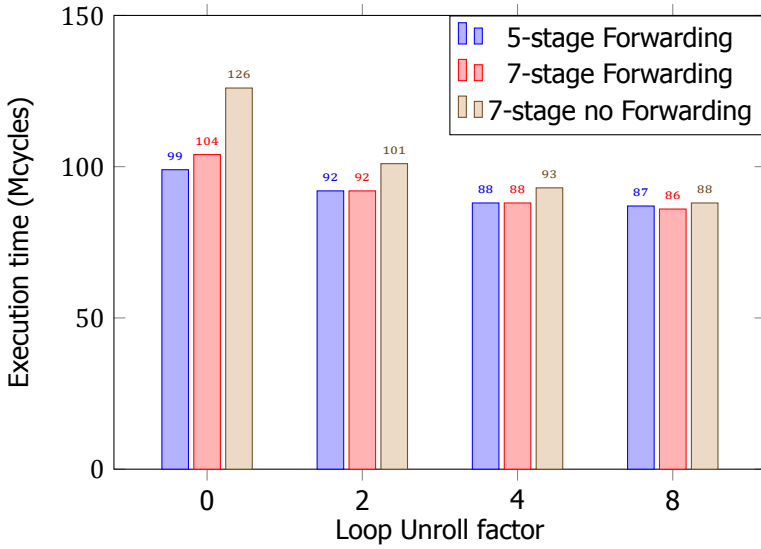


Figure 3.2: Execution times of a 3x3 convolution filter on a single processor using different loop unrolling factors.

sors in a stream, so that a core can read the output of the previous step (computed by the previous core in the stream) and write the output into its own data memory (making it available for reading by the next core in the stream). The memory blocks are implemented using dual-port RAM Blocks on the FPGA. Each port can sustain a bandwidth of one 32-bits word per cycle per port, so both processors connected to a block (current, next) can access a block without causing a stall. The blocks are connected to the processors by means of a simple address decoder between the memory unit and the data memories.

The first and last core should be connected to DMA (Direct Memory Access) units that move data to and from input and output frame buffers (eventually going off-board).

### 3.3.3. Platform

The VHDL code of the components is written in a very generic way and there are numerous parameters that can be chosen by the designer. First of all, the  $\rho$ -VEX processor can be configured in terms of issue width, pipeline configuration, forwarding, traps, trace unit, debug unit, performance counters, and caches. Secondly, there is an encompassing structure that instantiates processors in streams. The number of streams and length per stream are VHDL generics.

## 3.4. Experimental setup

Since the target application of the designed system is related to medical image processing, an X-ray sample image is used as input for the evaluation. Typical

medical imagers work with images that have a size of 1000 by 1000 pixels. The dimensions of our benchmark images are 2560 by 1920 pixels. The image is resized to other dimensions in order to determine the scalability of system performance. Each pixel is represented by a 32-bit value (RGBA). Using a technique described in the following section, the image may be scaled down to 1280 by 960 and 640 by 480 pixels.

A workload of algorithms based on a typical medical image processing pipeline is used. The first step in the image processing pipeline is an interpolation algorithm used to scale the size of the source image. The bi-linear and nearest neighbor interpolation algorithms both have the same computational complexity making them equally feasible. Because of its slightly higher flexibility, we select the bi-linear interpolation algorithm for the evaluation. Secondly, a gray scaling algorithm is applied. This algorithm is selected because it operates on single pixels in the input dataset. The third stage is a convolution filter that sharpens the image, followed by the final stage, an embossing convolution filter.

### 3.5. Evaluation results

Pipeline organization		Cores	Resource utilization			Freq. (MHz)
Forwarding	Stages		LUT	FF	BRAM	
Enabled	7	64	99%	29%	81%	149
Enabled	5	64	93%	26%	81%	103
Disabled	7	75	96%	33%	95%	162
Disabled	5	75	98%	30%	95%	143
Disabled	7	4	5%	2%	5%	200
Disabled	7	64	82%	28%	81%	193

Table 3.1: Resource utilization and clock frequency of different platform configurations on the Xilinx VC707 FPGA board.

#### 3.5.1. Resource utilization

We have synthesized the platform using various configurations targeting the Xilinx VC707 evaluation board. As stated, the pipeline organization of the processing elements has influence on the resource utilization and timing. In Table 3.1, 4 options have been evaluated using the standard synthesis flow (unconstrained). With forwarding enabled, the platform completely fills the FPGA using 64 cores. When forwarding is disabled, this can be increased to 75.

Additionally, we have performed a number of runs where we created simple placement constraints that steered the tool towards clustering the cores per stream so that they are aligned on the FPGA in accordance with their streaming organization. A single stream consisting of 4 cores achieves an operating frequency of 200MHz. Using 16 streams, timing becomes somewhat more difficult as the FPGA fabric is not homogeneous (some cores will need to traverse sections of the chip that are reserved for clocking, reconfiguration and I/O logic, and the distribution of

RAM Blocks is not completely uniform). Still, this configuration achieves an operating frequency of 193 MHz at 80% LUT utilization, leaving room for interfacing with off-board electronics.

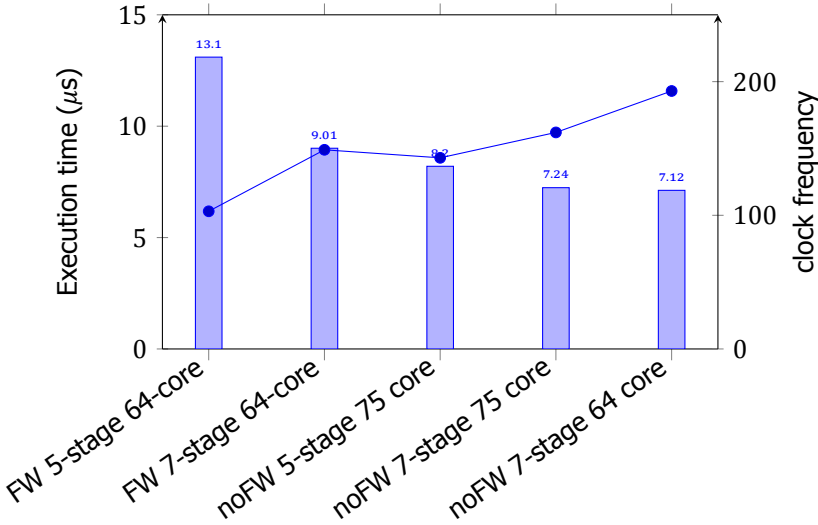


Figure 3.3: Execution times of a convolution 3x3 filter for the platforms in the design-space exploration as listed in Figure 3.1 using 8x loop unrolling (from Figure 3.2).

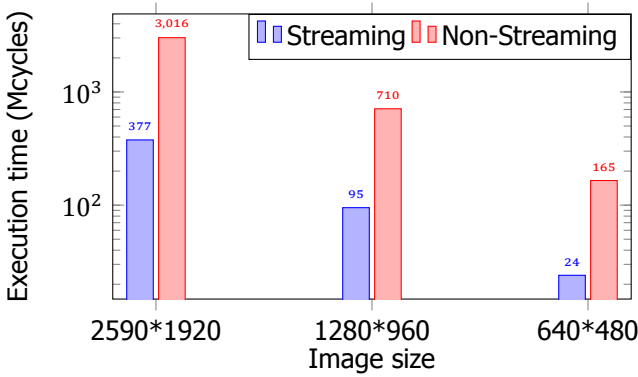


Figure 3.4: Execution times of a 4-stage image processing pipeline on a streaming versus non-streaming platform using different image sizes

### 3.5.2. Image processing performance

Figure 3.3 depicts the execution times of a 3x3 convolution filter on the various platforms, taking into account the number of cores, execution frequency, code performance on the pipeline organization (using 8x loop unrolling).



The results on using the streaming architecture for consecutive filters versus the same system with caches and a bus are depicted in Figure 3.4. Enabling streaming of data results in speedup of 7.5 times. Processing an image sized 1280 by 960 requires 94.72 million clock cycles (see Figure 3.4). Using 16 streams consisting of 4 cores (64 cores in total) at an operating frequency of 193 MHz, this would mean that our fabric can process approximately 34 frames per second.

Note that the difference will increase with the number of stages, so the fabric will perform better with increasingly complex image processing pipelines.

## 3

### 3.6. Conclusions

In this chapter, we have introduced and evaluated an implementation of a FPGA-based computation fabric that targets medical imaging applications by providing an image processing pipeline-oriented streaming memory hierarchy combined with high-performance VLIW processing elements. We have shown that the streaming memory hierarchy is able to reduce bandwidth requirements and increase performance by a factor of 7.5 times when using a single stream of only 4 processing stages. The platform stays fully targetable by a C-compiler and each core can be instructed to perform an individual task. The platform is highly configurable and designers can modify the organization to best match their application structure. For future work, there is room for further design-space exploration of the processing elements in terms of resource utilization versus performance, introducing design-time configurable instruction sets, increasing the clock frequency, and other architectural optimizations. The platform, simulator and toolchain are available for academic use at <http://www.rvex.ewi.tudelft.nl>.

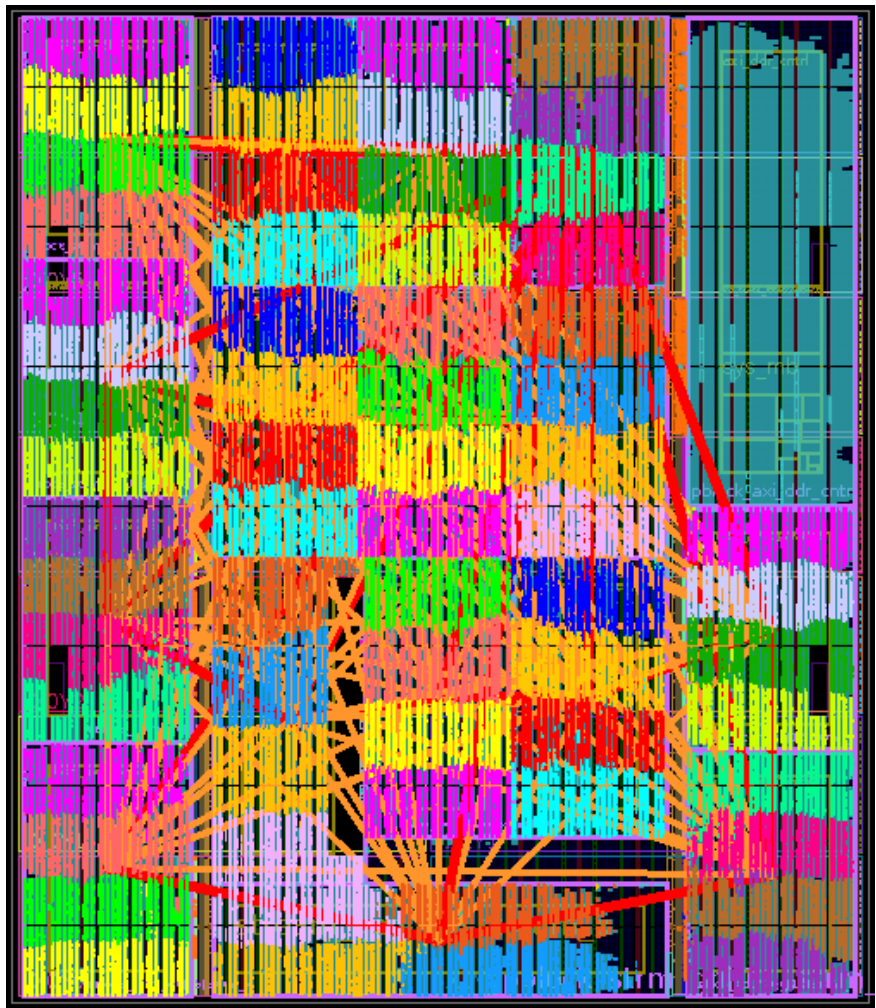


Figure 3.5: The layout of the 64-core, 193MHz platform on the Xilinx XC7VX485T-2FFG1761C FPGA on the Virtex 7 VC707 evaluation board. Manually created placement constraints were used to group each stream together. Each colored block represents an individual core.



# 4

## Frame-Based Programming, Stream-Based Processing

*Chapter 3 presented an image processing fabric that can be optimized for the image processing filter chain. However, there is still a large conceptual gap between the stream-based hardware platform and the programmer. Programmers typically have little experience with the details of custom-designed hardware. Normally, programming an image processing algorithm is done from the perspective of operating on a full frame. However, the streaming fabric splits each frame into segments that are processed on a separate pipeline of processing cores. In this chapter, we show how we use the OpenCL computing paradigm to provide a programming framework that facilitates mapping frame-based image processing code onto the streaming fabric.*

## Abstract

This chapter presents and evaluates an approach to deploy image processing pipelines that are developed frame-oriented on a hardware platform that is stream-oriented, such as an FPGA. First, this calls for a specialized streaming memory hierarchy and accompanying software framework that transparently moves image segments between stages in the image processing pipeline. Second, we use softcore VLIW processors, that are targetable by a C compiler and have hardware debugging capabilities, to evaluate and debug the software before moving to a time-consuming and specialistic High-Level Synthesis flow. This allows both software developers and hardware designers to test changes in a matter of seconds (compilation time) instead of hours (synthesis or circuit simulation time).

## 4

### 4.1. Introduction

The goal of an interventional X-Ray (iXR) system is to provide the physician with real-time images from the anatomy of the patient while performing a medical intervention. Typical interventions on the system include repairing blood vessel deformations such as aneurysms by positioning stents or replacing heart valves. During these procedures, blood vessels are filled with a contrast medium, which is visualized by X-rays and shown in real-time high resolution video images to the physician. As radiation is harmful to patients, doses need to be kept to a minimum. Using lower doses leads to more noise in the images, which can be reduced by using image processing filters.

The iXR is a complex system with strong real-time requirements. The system consists of many different compute architectures. The image processing algorithms are often closely tuned to the platform architecture. This makes it difficult to service the systems. In the context of the ALMARVI project we have worked on portability of image processing algorithms across platforms (CPU, GPU, embedded) in order to become less platform dependent. Mainly FPGA (SoC) platforms are interesting due to the long life time, strong performance and good real-time capabilities. However, FPGAs are often perceived as being difficult to design for. In order to address this, we have zoomed in on enablers for portability towards FPGAs exploiting novel tools and techniques such as High Level Synthesis (HLS) tools.

Image or video processing algorithm development is mainly done in a frame based manner which allows random access of the frame and parallelization techniques such as tiling. When moving to FPGA accelerators we cannot buffer a full frame before we start processing, due to amongst others memory bandwidth, power and latency requirements. Therefore, the algorithm has to be implemented in a stream based manner, where we wish to process pixels as soon as they come in and, as quickly as possible, pass the result on to the next processing step (accelerator). This involves intensive hand optimizations such as using line buffers and data re-ordering instead of random memory access. We wish to abstract from this implementation level in order to ease the implementation for the programmer.

Frameworks exist that facilitate mapping computations to FPGA (including frameworks specifically targeting image processing), but these do not solve the

frame versus stream problem. Mapping the frame-based software to a stream-based hardware platform on FPGA creates the following challenges; creating a framework that moves and buffers data (in the form of image segments) between stages, and the development/test cycle time increases tremendously because of synthesis. In this chapter, we propose an approach to solve these challenges by using an FPGA overlay fabric consisting of software processors that are targetable by OpenCL and a streaming memory framework.

## 4.2. Related work

A common aim in the development of support frameworks for specific application domains is to reduce Non-Recurring Engineering costs (NRE) by speeding up development time and facilitate component re-use. The performance may be slightly negatively affected (a specific full-custom design is hard to beat), but that is usually offset by lower development costs and time-to-market.

In this section, we will first discuss general approaches to speed up image processing workloads, then proceed to focus on FPGA acceleration including software support approaches, followed by a discussion of hardware support approaches using overlays and image processing fabrics and hardware integration frameworks.

### 4.2.1. Optimizing/accelerating image processing workloads

Currently, there are numerous ways to either optimize image processing code or mapping it to FPGA/GPU. On common ARM and x86-based systems, Single Instruction, Multiple Data (SIMD) instruction set extensions such as AVX can be exploited to gain considerable performance [65]. There exists efforts to be able to insert these instructions automatically and some compiler support exists. When a new generation of processors or SIMD extensions is introduced, however, code must be optimized, leading to large costs in testing and validation. A study about optimizing HEVC (High Efficiency Video Codec) using the AVX SIMD extension concludes that "The large speedup, however, could only be achieved with high programming complexity and effort." [66, p853]

GPUs suffer from the problem of performance portability [67]. This means that for a new GPU generation, the same issue arises where engineering effort may be required to optimize the code again. Additionally, GPU primarily target floating point calculations, that are usually algorithmically not necessary for image processing.

To facilitate the optimization process and to aid design space exploration for various target execution platforms, the Halide programming language and compiler can be used to generate code from a functional description of a filter [68]. Mapping parallel computations to a variety of computational fabrics (including multicore CPUs, GPUs and FPGA) can be done using OpenCL [69].

### 4.2.2. FPGA acceleration

Mapping computations to FPGA can be performed in a number of ways. High-Level Synthesis [70] is becoming a standard tool in many FPGA design environments. Additionally, FPGA vendors are supporting OpenCL through for example SDAccel [71].

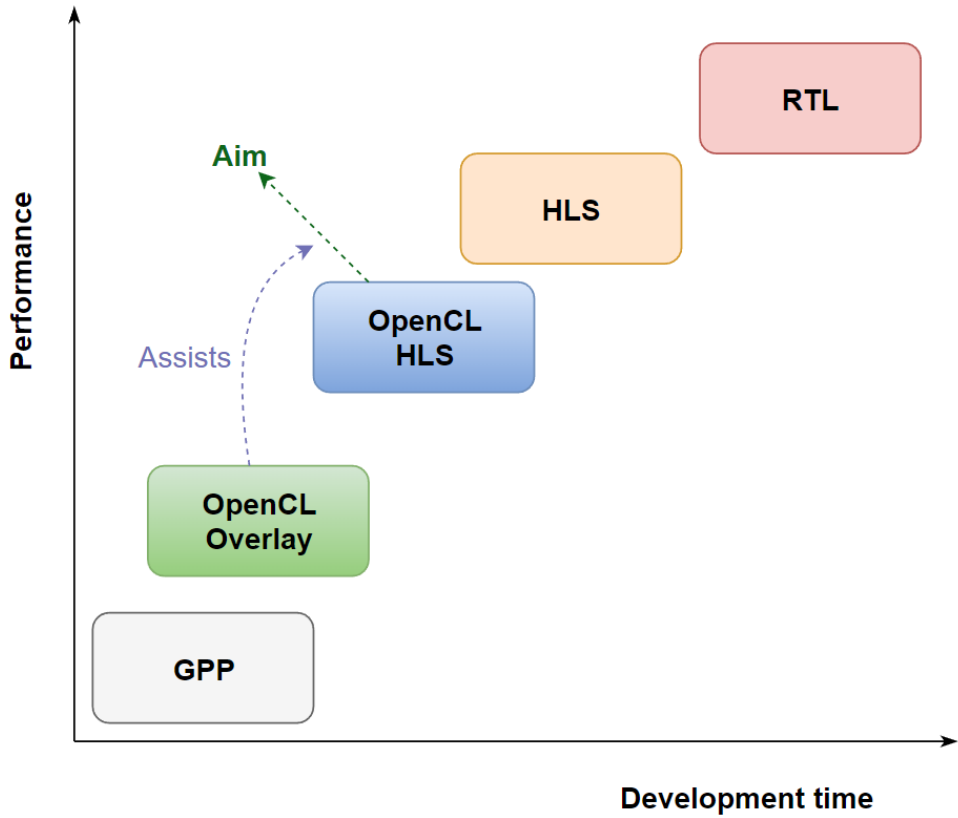


Figure 4.1: High-Level Synthesis (HLS) aims to reduce development time compared to a full-custom RTL design. Recently, FPGA vendors started to support OpenCL code. Starting from an OpenCL program, it costs less time to synthesize the first working design to FPGA, but it requires a considerable number of costly test, debug, and optimization cycles before it starts to perform comparable to an HLS design. Using an FPGA overlay that supports OpenCL code facilitates this process.

The traditional approach of developing datapath designs in VHDL is becoming rare and will be utilized only for very specific designs or if the HLS tools are not able to meet certain requirements. Still, the HLS toolflow has some drawbacks. Code modifications are often necessary, as it is not possible to write code in a frame-based way - the tools need to be able to identify buffers in such a way that it can be mapped to FPGA efficiently (stream-based) to prevent prohibitively slow main memory accesses. The ROCCC HLS compiler [72] is able to insert smartbuffers that can provide some data reuse, and in [73] this concept has been extended into a framework that can generate VHDL code for sliding window filters with optimized memory structure. Other related efforts exist, that aim to generate streaming designs from C code [74] or make use of Domain-Specific Languages (DSL) such as Halide are Darkroom [75] and HIPAcc. These approaches are able to generate hardware components for FPGA by providing an abstraction layer for HLS. Using

HLS and frameworks that generate HDL code, quality of result is not always consistent and synthesis times are still very long. This means there is still a gap to be bridged between the image processing code and FPGA development.

### 4.2.3. FPGA overlays

To reduce compilation time and enhance portability, FPGA overlays are becoming an interesting research area. Using an overlay on FPGAs would allow software programmers to target familiar architectures, without understanding the low-level details. MARC [76] is one such project where a multi-core architecture is used as an intermediate compilation target. It consists of one control processor and multiple processors (Cores) to perform computations. The data cores are used to run OpenCL kernels and the control core is used to schedule work to the data cores. The authors conclude that using such an overlay dramatically reduces development time and bridges the gap between hardware and software programs at an acceptable performance hit compared to hand-optimized FPGA implementation. Another related effort is OpenRCL [77]. The concept of using accelerators to speed up applications while retaining programmability is discussed in [60].

### 4.2.4. FPGA image processing overlays

In [64], a toolset is introduced for customized softcore image processing on FPGAs. Customizing the softcores is a concept that can be added to our proposed framework to improve performance. Resource-efficient processing elements are introduced by [46] and [63]. Our framework could make use of these processors if they were available, but our chosen processor supports OpenCL and we provide our own design-space exploration. This work introduces an FPGA framework that can be used for prototyping and debugging, and can form a basis to use HLS if the system does not achieve the target performance requirements. A similar framework has been introduced in [59], but instead of providing stream-based processing that allows scalability, they employ shared memory in a banked organization, accessible via a crossbar. This reduces scalability as will be evaluated in Section 4.6.

### 4.2.5. Integration frameworks

There have been related efforts in creating FPGA development frameworks that facilitate development and integration. One example is RIFFA [78], an open source project that provides communication and synchronization between host and FPGA accelerators using PCIe. As we are using I/O directly connected to the FPGA board, we do not require PCIe interfacing. A commercial framework that can incorporate HSL-generated accelerators, hand-written VHDL and IP is the DYNAMIC Process LOader or Dyplo from Topic Embedded Products [79]. It incorporates a network on chip which connects both software functions and FPGA accelerators together. The network can be re-routed at run time and designated areas of the FPGA can be re-configured with a different accelerator through Xilinx Partial Reconfiguration [80]. The Dyplo framework handles data transfers transparently by visualization of the accelerators on the FPGA, thus offering ease of programming and flexibility. Philips Healthcare has successfully used the Dyplo framework in their study for ALMARVI.



```

int a[128], b[128];

static int get(int addr)
{
    if (addr < 0 || addr >= 127)
        return 0;
    else
        return a[addr];
}

void fun()
{
    int i;
    for (i = 0; i < 128; i++)
    {
        b[i] = (get(i-1) + get(i) + get(i+1)) / 3;
    }
}

```

Figure 4.2: Code example of an OpenCL kernel.

### 4.3. Approach

This section will outline how we have used a slightly modified view on the OpenCL programming model to target our proposed streaming-based hardware framework while using ordinary OpenCL kernels.

#### 4.3.1. OpenCL's view on parallel computing

In many cases a compute 'problem' consists of a data-set for which each element needs to undergo a certain transformation and basically this transformation is the same for each element. Consider the code example in Figure 4.2:

Every  $b[i]$  is produced by the exact same code fragment and there is no dependency of an element of  $b[]$  to another element of  $b[]$ . Such an operation is called a 'kernel' in OpenCL terminology. Conceptually, all 128 computations could have been executed concurrently, on a platform that provides 128 processing elements. This kind of parallelism is the main target of OpenCL: execution of as many kernels in parallel as possible. Note that the code for each kernel is identical, but the execution flow can be different for example due to the boundary checking 'if' statement in the 'get()' function. The OpenCL framework tries to have many accelerators performing the same operation on many datasets independently. One element of such a dataset is called a 'work-item'.

#### 4.3.2. OpenCL memory model

OpenCL defines 4 types of memory objects:

1. Global Memory – read/write accessible from both the host and the execution device
2. Constant Memory – like Global Memory, but read-only for execution devices

3. Local Memory – only accessible within (a group of) execution devices
4. Private Memory – only accessible from a single execution device

OpenCL also defines a data cache between the Global / Constant memories and the execution devices. This cache is optional, but in practice it is always needed to avoid slow-down due to data transfers. This cache needs to be carefully designed, as many cores will try to access it simultaneously. If cache does not have enough access ports, it will quickly become a bottleneck. Figure 4.3 shows the OpenCL memory structure.

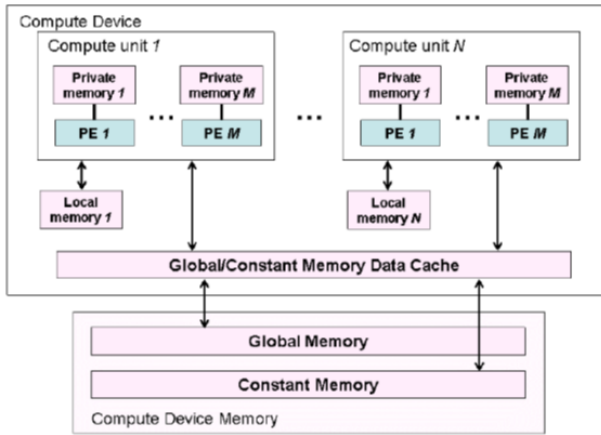


Figure 4.3: OpenCL memory structure

#### 4.3.3. Streaming data and OpenCL

The OpenCL model works with Single Instruction Multiple Data (SIMD) processing. One set of kernels is operating on the full data set. Other sets of kernels have to be programmed each time for iterative processing, where the data is stored to the global memory in between processing steps. This approach is shown in Figure 4.4. We would like to use OpenCL in a data pipelined, or streaming, implementation as explained in the previous section. This means that we would like a situation where we can program different sets of kernels where data is passed on from one set to the next, as depicted in Figure 4.5. Here, results are no longer written back to global memory, but passed to other accelerators through a connection mechanism that needs to be scalable (as we are targeting highly parallel workloads running on large numbers of compute devices) and able to provide sufficient bandwidth. This can be a Network on Chip or a certain connection topology that suits the application.

Note that an alternative approach could be to execute the different kernels consecutively on every compute device (essentially, we changing the kernel instead of moving the data - keeping the data set local). This requires every compute device to be capable of storing the instruction stream of each kernel in local instruction memory or cache. The storage capacity of these memories is an important design

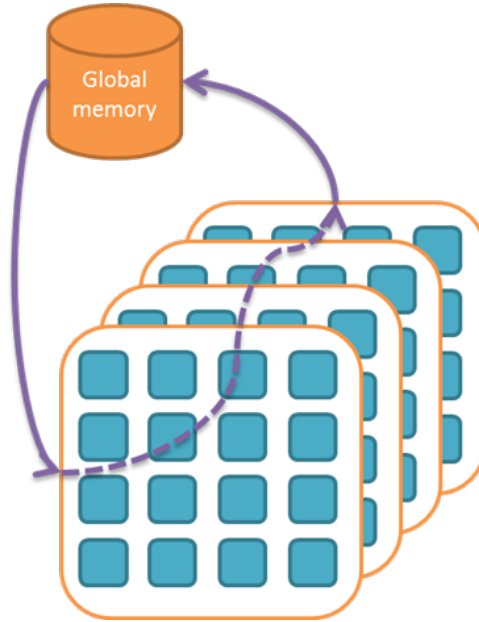


Figure 4.4: OpenCL data model

parameter as is explored in Section 4.4.1. Additionally, the size of the instruction stream will typically be larger than the size of a data block.

#### 4.3.4. OpenCL data architecture

In OpenCL a kernel always has a full view on the entire dataset but in most cases that is not necessary. Given that a certain kernel is operating on a block of data, this kernel only needs a limited view on the total working set as depicted in Figure 4.6. In this example (a 5x5 convolution kernel), the data located more than 2 lines above the current coordinates (x,y) are not needed anymore and the lines more than 2 lines below (x,y) are not needed yet. Assuming a set of compute devices are processing the data line by line, each device requires 5 lines of storage capacity to store their working set. A control mechanism should feed each compute device with the appropriate data in time and keep track of the locations of lines when assigning tasks to ensure the needed input lines are present and output results are only overwriting stale data. The OpenCL system provides a suitable basis to build such a mechanism: the command queue. This queue distributes work-items to the compute units, so it knows exactly which work-items are being processed. Consequently, it knows about the maximum view of each kernel and can compute the required data (sub)set as well.

Determining the maximum view size of the kernels in the image processing pipeline also influences certain parameters of the required hardware infrastructure (the storage capacity of the local memories of the compute devices). Conversely:

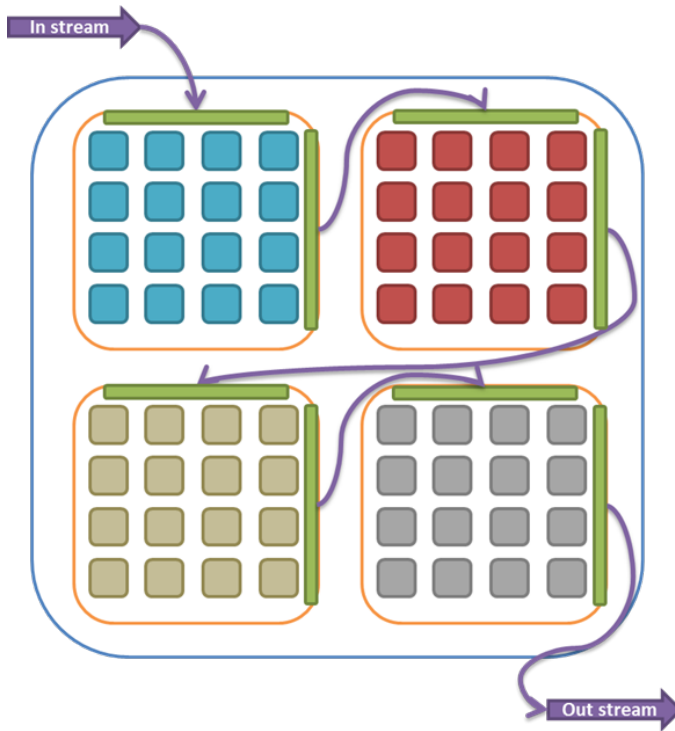


Figure 4.5: Streaming data model

for a given size of hardware buffers there is a maximum view size for each kernel. The process of finding the optimal buffer sizes between all kernels is important to prevent bottlenecks while minimizing the required sizes. There are numerous approaches to solve this, for example by performing simulations with iteratively decreasing buffer sizes, but this is outside the scope of this work. In our reference platform, we assume all filters have a maximal window size of 5x5 and will therefore need a buffer size of 5 lines for input and 1 additional line for output. The width of the lines depends on the stripe length (e.g., how the image is divided into vertical stripes), which also determines how many vertical lines of pixels need to be processed redundantly.

## 4.4. Implementation - Hardware

An FPGA-based platform targeting image processing pipelines needs a number of elements; a streaming memory structure, processing units, one or more DMA units, interfaces with off-board electronics (to receive the image and output it after processing), and control & debug interfaces with a central host. Additionally, run-time support is needed to move the image segments through the streaming memory structure. This should be done as transparently as possible in order to keep frame-

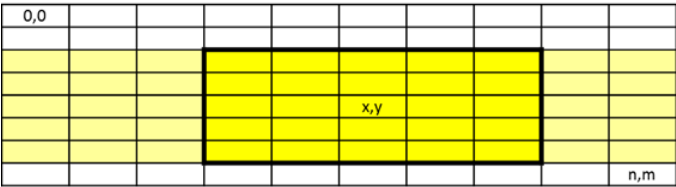


Figure 4.6: A kernel only needs a limited view on the total working set

based programmability.

4

4.4.1. Processing element

The processing elements used in this work are based on the  $\rho$ -VEX VLIW processor developed by TU Delft [21]. The implementation of this processor is written in a very generic way, so design space exploration can be performed. In our application domain, there is ample parallelism on both instruction level and data level (that can be exploited by SIMD or multithreading). The design-time configuration options available for the  $\rho$ -VEX are listed in Table 4.1. The processor will be configured

Configuration option	Area utilization	Code performance	Timing
Issue-width	-	+	-
Forwarding	-	+	-
Traps	+/-	+/-	+/-
Breakpoints	+/-	+/-	-
Perf. counters	-	+/-	-
Additional pipeline stages	+/-	-	+

Table 4.1: Design-time configuration options of the  $\rho$ -VEX processor and their effect on various exploration metrics.

in the smallest issue width to improve timing and limit area utilization as much as possible. Any decrease in area utilization may results in a larger number of cores, which will directly improve performance. Disabling forwarding in the pipeline and adding additional pipeline stages will impact code performance due to additional latency between operations, but this penalty can be reduced or even removed by using loop unrolling in the  $\rho$ -VEX compiler in order to fill the latency slots with other operations. This requires the cores to have sufficiently sized instruction memories, resulting in another trade-off as the memory sizes will impact timing and, to a certain extent, the number of cores that will fit on the FPGA.

4.4.2. Memory structure

The memory structure as used in our overlay is introduced in [37]. The concept is to organize the cores into streams of a configurable number of cores. Within

such a stream, each processor has a local (scratchpad) instruction memory and a local data memory. The sizes of these memories must be set at design-time and determine the maximum size of the program (.text section of the binary), and the maximum size and number of line buffers that cores can store. Similar to the design-space exploration of the instruction memory size, as discussed in Section 4.4.1, the size of the data memory buffers is an important parameter that should be carefully considered. Too large data memories can limit the number of cores that can be placed on the FPGA and create timing difficulties, too small memories can result in bottlenecks in the stream or prevent a certain core from supporting certain filters (for example, a convolution filter with a 5x5 pixel window size needs at least 5 buffered input lines and a buffer to write the output line).

In addition to its own local data memory, each core in a stream is able to access the data memory of its *predecessor* by means of an address decoder (see Figure 4.7). Each core in the stream will run a filter in the image processing pipeline. The local data memories are implemented using dual-ported BRAMs so that each core has single-cycle access to both memory regions. The first and last memories in the streams are connected to an AXI bus using a DMA unit. The image is segmented to distribute the workload over the available streams, while taking into consideration the necessary overlap to perform the window-based operations. Communication between the cores is realized in two ways; sending data, commands, parameters and synchronization is performed using the local memories (for example, convolution kernel parameters are propagated through the stream). Loading the instruction memories, debugging and resetting the individual cores is performed using a separate debug bus that is operating on a lower frequency to avoid timing difficulties. These last tasks are only performed during startup or debugging, therefore the lower frequency will not interfere with the performance.

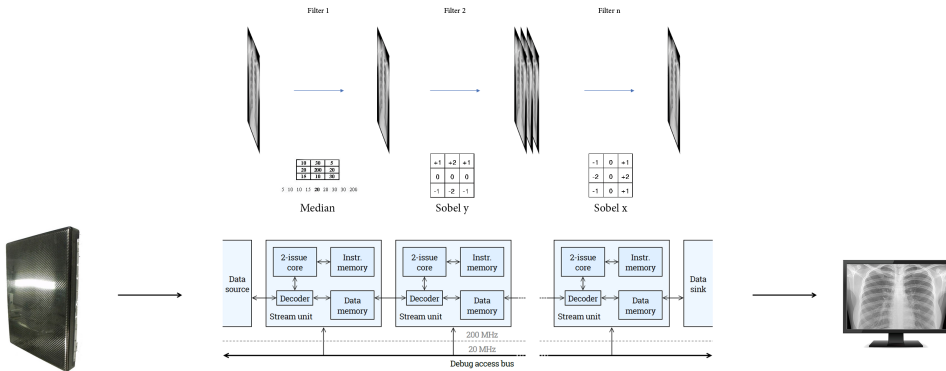


Figure 4.7: Overview of the streaming memory framework. Obtaining the image segment from the source and writing it to the sink is performed by a DMA unit that accesses framebuffers in the DRAM on the FPGA. Transfers between stream units are performed by local buses in the FPGA fabric, connecting local memories instantiated using BRAMs.

### 4.4.3. Interfaces

This section will discuss the hardware interfaces that move data to and from the processing elements and the outside world.

#### DMA unit

As stated, the first and last core of each stream is connected to a DMA unit that can transfer blocks of data to the AXI bus. This connection is implemented in a non-blocking way, to allow cores to send a request to the DMA unit without having to wait until it becomes available. Arbitration is performed by means of a simplified Network-on-Chip (NoC), as is depicted in Figure 4.8.

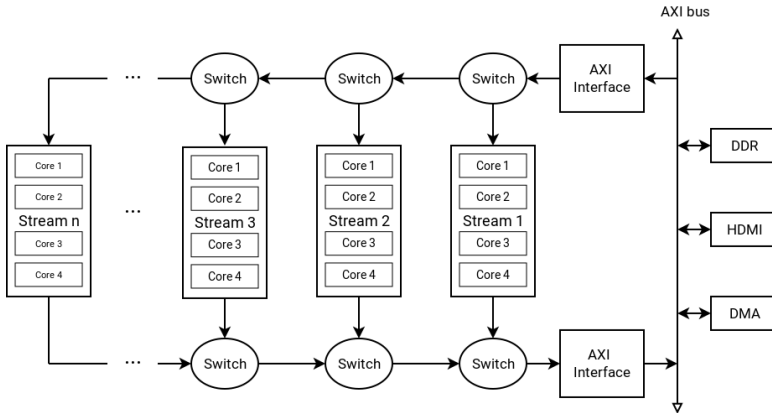


Figure 4.8: DMA unit that is in essence a simple NoC with switches leading up to an AXI bridge that bursts packets of streaming data to a framebuffer in DDR (or any memory address provided by the requesting stream).

The requesting core writes the target address and the data (or size) to be transferred into a BRAM, and flags the request in a control register. Each pair of streams is connected to an arbiter (router) in a fully unbalanced organization (this creates a better layout for Place and Route). Each arbiter contains registers in order to avoid a long timing path. When receiving a new request, an arbiter will lock itself until the payload has been fully transferred (similar to wormhole switching). This way, all payloads will be transferred undivided, so the burst mode of the AXI bus can be used most effectively. The priorities of the arbiters are set such that streams that are further away from the bus interface have precedence.

#### Debug bus

In order to support the extensive debugging capabilities that are offered by the  $\rho$ -VEX processor, a separate bus is created that operates on a lower frequency than the datapaths and memories. This is because this bus is not performance critical and the lower clock will facilitate timing on the FPGA. The bus is bridged to the AXI main bus by means of an AXI slave interface, and connected to all cores in the system (all of which are memory-mapped into the AXI slave's address space). The functionality of the debug bus is determined by the memory regions of each  $\rho$ -VEX

core that it is able to access - the instruction memory, data memory, and control registers. The set of control registers allow standard operations such as halting and resetting the core, but also more advanced requests such as register file access, setting watch/breakpoints and toggling single-step execution mode.

To be able to use all debugging functionality, the  $\rho$ -VEX must be configured with traps enabled. Whenever a trap occurs during execution, the core will store the cause, the location in the program (program counter value), and an argument in a control register. This way, it is possible to ascertain what went wrong before the core halted or trapped into the trap handler. For example, if the core performs a memory read to an invalid address (unmapped or unaligned address), it will show the cause associated with invalid data access, along with the program counter that contained the corresponding load instruction and the address it was trying to access.

## 4.5. Implementation - Software

This section describes the implementation from a software point of view, starting with the buffer management and how the workload is parallelized over multiple cores, how the system performs the necessary synchronization, the way that OpenCL support has been implemented, how the interfaces are programmed and lastly how to develop applications for the platform.

### 4.5.1. Compilation and operation

The process elements, as discussed in Section 4.4.1, are based on the VEX instruction set architecture [16]. There is a full toolchain available for these cores, that must be used to compile code for the platform. A C compiler is available, along with a port of binutils and an architectural simulator that can be used for initial debugging of the code during development. The process of writing parallel code for the platform will be discussed in more detail in Section 4.5.4.

To load the binaries into all the processing elements, the current platform implementation includes a management core. In principle, this can be a (hard) ARM-based device (in case of Zynq and comparable platforms), or even an additional  $\rho$ -VEX core, but in our current implementation it is a microblaze processor as it is used in Xilinx reference designs to configure the AXI-based platform and peripherals. This core is not running any filters, but only concerns itself with sending commands to all the processing elements. It is able to access all the processing elements' control registers, instruction memory and data memory by means of the debug bus. In addition, it can control the DMA unit using control registers.

At startup time, the management cores resets and halts all processing elements and loads the corresponding instruction stream into the instruction memory of each core in each stream. Then it populates a datastructure in the data memories of each core to in order to have them wait for instructions (see also Section 4.5.3) and releases the cores. During normal operation, the instruction memories only need to be loaded once, however, it is possible to change the image processing pipeline by simply halting the cores and uploading a different instruction stream.



### 4.5.2. Buffer management

As discussed in Section 4.4.2, the image processing workload will be distributed across the available cores in two dimensions - each core in a stream will perform a separate filter (task-level parallelism), and each stream will handle a part of the image (data-level parallelism). The image is divided by vertical stripes of a certain width. In our reference design, the total number of cores is 16 streams of 4 cores each and the default stripe width is 60 pixels. The stripes are buffered per line in the local data memories of the processing elements. The data structure of these buffers is depicted in Figure 4.9. The type of filters that are to be supported by a certain

4

```
// Line buffer.
typedef struct
{
    // Position on the screen of the top-left pixel. x may be negative, as the
    // filters may need input pixels to the left of the
    short x;
    short y;

    // Data buffer.
    int d[95];
} PACK linebuf_t;
```

Figure 4.9: Struct that contains a single line of a stripe of the image that is being processed. It contains the position in the frame so it can be used as input for subsequent filters or written into a frame buffer.

core in the stream determines the number of line buffers that need to fit into the input data memory. If the filter operates on a 5x5 window to produce 1 output line, there must be enough storage capacity to store 5 lines in the input memory and 1 line in the output memory. Note that these storage requirements overlap between two adjacent cores. The buffers circulate in such a way that the required input lines are stable to be accessed by a core, and the line that is not needed anymore will be used as output storage for the previous core. The horizontal overlap is handled by extending the stripes on both sides when reading in the input from the framebuffer using DMA. This overlap is dependent on the filter size in the same way as the required number of buffered lines.

When distributing block-based filters among multiple streams, some amount of overlap is required between the stripes. Instead of communicating these pixels between cores (which is not supported by the memory structure), our platform redundantly computes them in every stream. The largest filter size determines the necessary amount of overlapping pixels, and these are added to the workload automatically by the management core. Edges can be dealt with in different ways; each core could implement the edge behavior and perform a check on each input line to identify whether it is an edge line. However, this will increase code size. Another option is to implement the behavior on a dedicated core and instruct the management core to send all edge lines to it. If the management core is fast enough, it could also perform the filters on all the edge lines directly.

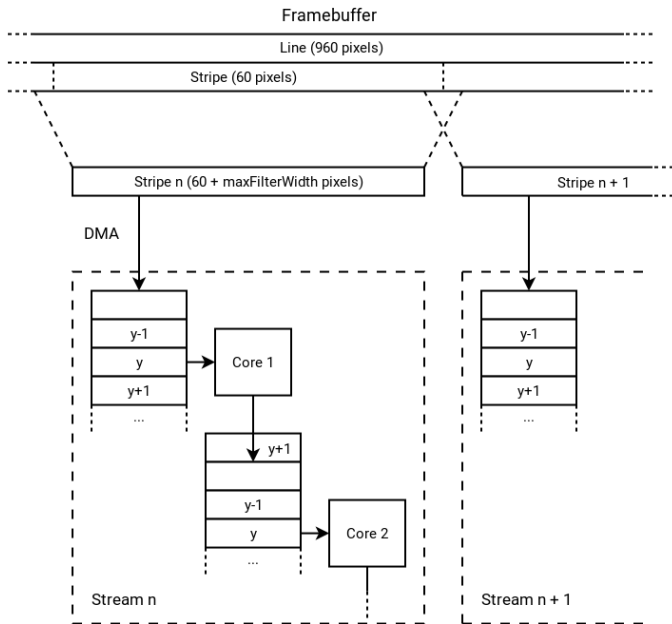


Figure 4.10: Diagram showing the overlap between stripes allocated to neighboring Streams, and lines that automatically overlap because they are allocated to the same stream.

### 4.5.3. Synchronization and communication

As the processing elements' data memories are not connected to the AXI bus directly, but to a slower debug bus, controlling each individual core from the microblaze using this debug bus would be too slow. However, each processing element is able to receive commands from the management core by means of a data structure that is depicted in Figure 4.11.

The struct contains a *state* member that is used for synchronization and to control buffer ownership. The predecessor core can take ownership of a line buffer by writing the index of that particular buffer into the state field. The buffers contain consecutive lines in a circular fashion and it is always implied that the buffers that are not reserved by the predecessor core contain valid lines (so that these can be used as input by the successor core). The successor core resets the state value to 0 when it is finished processing its line. The predecessor core will wait for this event before proceeding with the next line, so this mechanism can be used to apply backpressure.

In addition to the state member, the structs contain a filter parameter struct for each following core. An example of this struct is depicted in Figure 4.12

If the state field is set to -1, a core will propagate these values to its successor. This way, new values can be loaded into each core. Lastly, the communication struct contains a number of line buffers. As discussed, the exact number must be set by the designer after careful consideration of the filters that a certain core needs to be able to perform.

```

// Data shared between core 1 and 2.
typedef struct
{
    // State:
    // - written to 0 by c2 initialization.
    // - written to -1 by c1 after updating line width/filter parameters. c2
    //   should then propagate the new parameters to the next core.
    // - written to x by c1 when the five lines following x-1 (modulo 6)
    //   contain valid data.
    //   - 1: c1 = line[0], c2 = line[1,2,3,4,5]
    //   - 2: c1 = line[1], c2 = line[2,3,4,5,0]
    //   - 3: c1 = line[2], c2 = line[3,4,5,0,1]
    //   - 4: c1 = line[3], c2 = line[4,5,0,1,2]
    //   - 5: c1 = line[4], c2 = line[5,0,1,2,3]
    //   - 6: c1 = line[5], c2 = line[0,1,2,3,4]
    //   - *: c1 = line[0,1,2,3,4,5], c2 = -
    // - written to 0 by c2 when it finishes its job.
    volatile int state;

    // Output line width (excluding extra data needed for the filters).
    volatile int line_width;

    // Framebuffer stride.
    volatile int stride;

    // Filter parameters (core 2).
    volatile filter_param_t core2_filter_params;

    // Filter parameters (core 3).
    volatile filter_param_t core3_filter_params;

    // Line buffer from c1 to c2. If state is positive, all buffers except
    // state-1 may not be modified by c1. Otherwise, c1 has access to all
    // buffers.
    volatile linebuf_t line[6];
} PACK c1_c2_t;

```

Figure 4.11: Example of a struct that is used to communicate between the cores, and to synchronize the streams. It can be used to change which filter should be performed, update filter parameters, and to apply backpressure to predecessor cores (using the *state* member). This struct is used to communicate between core 1 and 2 (therefore it does not contain any parameters for cores 0 and 1) and contains 6 linebuffers.

#### 4.5.4. Application development

One of the related work in terms of software development for image processing filters is Halide [68]. It can generate code by describing the relationship between an output pixels and its required input pixels. This is possible because the operations are the same for each pixel, using different input pixels. As discussed in Section 4.3, a work-item size of a single pixel is bad because of overhead (not only on a software level, but this would also prevent the DMA engine from achieving high bus throughput as this requires burst transfers). Therefore, our platform operates on lines instead of pixels. Developing filters for the platform consists of modifying the frame-based reference implementation into a line-based implementation. In the halide analogy, instead of describing how to calculate a single pixel, the programmer

```
// Parameters for the filters.
typedef struct
{
    // Filter type:
    // - 0: passthrough
    // - 1: 3x3 median
    // - 2: 5x5 median
    // - 3: 3x3 convolution
    // - 4: 5x5 convolution
    short mode;
    // Filter taps in a 5x5 grid. The last two values are not used, but exist
    // to force 32-bit alignment.
    short taps[25];
} PACK filter_param_t;
```

Figure 4.12: Example of a struct that is used to communicate filter parameters between cores. In theory, each core in the stream could have a specific struct because they can all have a distinct instruction memory containing different filters. In our reference implementation, the cores all support 2 different filters (median and convolution), and the kernel that is used for the convolution can be modified using the struct.

must specify how to compute a line of a given length.

The framework supplies the input linebuffers and an output linebuffer, the width of the line, and if applicable parameters for the filter. Kernels can be programmed in OpenCL and compiled with our LLVM backend from the Portable OpenCL (pocl) framework [81]. Alternatively, kernels can be programmed in C or VEX assembly code. The kernel code must be linked together with the control loop that polls for work.

## 4.6. Experiments/Evaluation

To evaluate the approach, we have developed a reference implementation using the framework and synthesized this for the Xilinx VC707 evaluation board with a Virtex 7 FPGA. The VHDL design is fully parameterized, and the fabric is organized in 16 streams of 4 consecutive cores with each 4 KiB of instruction and data memory. Figure 3.5 depicts the layout of the placed and routed design on the FPGA, after providing placement constraints for each individual stream of 4 cores while constraining the management core (microblaze) and interface logic such as HDMI and DDR controllers to the upper right corner.

In Figure 4.13, a comparison has been made with a related effort named Bio-Threads [59], showing the advantage of our memory structure regarding scalability.

The workload of the medical imaging platform consist of window-based image processing algorithms. For our evaluation, we have implemented the convolution operator that can be used with various filter kernels (which can be programmed into the processing elements as discussed in Section 4.5.3). The detector provides images with a resolution of 960 by 960 pixels. As our reference platform has 16 parallel streams, this results in a line size of 60 pixels. Table 4.2 depicts the performance of the platform for a convolution kernel using a 3x3 and 5x5 filter size. The implementation used here is naive regarding algorithmic optimizations; it computes each output pixel from all their respective input pixels (instead of re-using part of

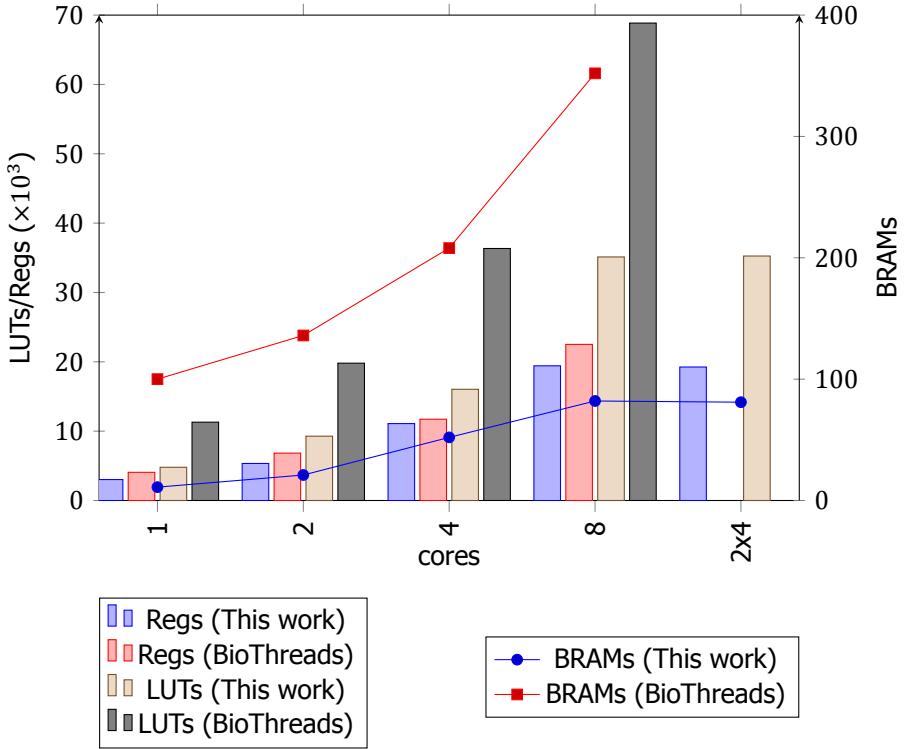


Figure 4.13: Resource utilization comparison with a related platform [59]. The advantage of this work is that it provides the option of adding parallel processing pipelines instead of increasing the number of cores connected to a central memory component (8 versus 2x4 cores). This is far more scalable and allows us to place a considerably larger number of cores on an FPGA.

the computation for the current pixel for the next pixel.

## 4.7. Conclusions

This work presents our approach of using a VLIW-based FPGA fabric that allows image processing kernels to be programmed in a frame-based fashion and processed efficiently in a stream-based fashion. Our memory structure is scalable and allows pipelines of different size with different filters to be mapped to the fabric. Instead of repeatedly synthesizing the platform, designers can explore and debug the design using this framework by only recompiling OpenCL or C code. If the platform does not provide enough throughput to satisfy the performance requirements, the code that is mapped onto the VLIW softcore processors can be passed through a HLS toolflow to produce a faster system that uses the same streaming memory structure. Results show that the platform is more scalable compared to a related image processing framework.

Algorithm	Cycles/line (60 pixels)	Cycles/img (960x960)	Frames/s (200 MHz)
Convolution 3x3	4005	3844800	52
Convolution 5x5	10216	9807360	20

Table 4.2: Performance for different filters of a single processing element per line, per image segment, and the total throughput assuming a 16-stream platform running at 200 MHz assuming no I/O stalls.

Future directions could include providing support for  $1 : n$  or  $n : 1$  connections to support processing steps that need input data from multiple sources or provide output to multiple sources. Additionally, a HLS flow for automatic accelerator generation could be created, and automatic design-space exploration using Halide or a similar tool.



# Part 2 - Dynamic workloads, dynamically reconfigurable platform

*Part 1 discussed highly static embedded image processing workloads. Part 2 focuses on dynamic environments, where workloads of varying intensity and characteristics must be executed with the highest possible throughput. In our opinion, dynamic workloads call for dynamic computing platforms. The overall aim of this part of this thesis is to demonstrate the ability of a dynamic processor to adapt itself to the workload, thereby achieving better performance than comparable (static) heterogeneous systems. Chapter 5 discusses ways to perform automatic characterization of a running program and to execute it as energy-efficient as possible. Chapter 6 examines a system executing generated workloads consisting of several tasks, optimizing for total throughput.*

	Hardware	Software	Scheduling
Classical	Static	Static & Dynamic	Static & Dynamic
Part 1	Static (fixed, design-time)	Static	Static (compile-time)
<b>Part 2</b>	<b>Dynamic (adaptable, run-time)</b>	<b>Dynamic</b>	<b>Dynamic (run-time)</b>
Part 3	Dynamic (adaptable, run-time)	Static & Real-time	Static (compile-time) & Dynamic (run-time)





# 5

## Evaluating auto-adaptation methods

*In the introduction, we proposed to use the  $\rho$ -VEX fine-grained adaptable processor, being a dynamic processing platform, to target dynamic workloads. The first step in evaluating the suitability of a fine-grained adaptable processor for dynamic workloads is to study different methods of exploiting this adaptability. This chapter focuses on how a simple hardware circuit is able to utilize the explicit parallelism encoded into VLIW binaries to detect the most suitable configuration to execute code sections. It uses the different methods to continuously and automatically optimize the processor configuration, with the aim to study if there is enough variability in code characteristics to exploit and to evaluate the efficiency of the different methods. The methods use run-time performance monitoring, and a hybrid approach that adds a compiler analysis step.*

## Abstract

To achieve energy savings while maintaining adequate performance, system designers and programmers wish to create the best possible match between program behavior and the underlying hardware. Well-known current approaches include DVFS and task migrations in heterogeneous platforms such as big.LITTLE processors. Additionally, processors have been proposed in literature that are able to adapt (parts of) their organization to the workload. These reconfigurations can be managed using hardware monitors, profiling and other compile-time information or a combination of both. Many current solutions are suitable for heterogeneous systems, as migration penalties pose a practical limit to the maximum adaptation frequency, but not for dynamic processors that can adapt much more fine-grained.

In this chapter, we present two novel concepts to aid these low-penalty reconfigurable processors - one requiring an ISA extension and one without. Our experimental results show that our approaches enable a dynamic processor to reduce the energy-delay product by up to 25% and on average 10% to 18% compared to the best performing static setups.

## 5

### 5.1. Introduction

With energy utilization as a new critical metric for computing systems, designers have devised numerous ways of configuring systems to run in various performance/power modes. The most notable examples are Dynamic Voltage and Frequency Scaling (DVFS), Heterogeneous Multicore Processors (HMPs) such as big.LITTLE, and polymorphic processors such as MorphCore [6]. In turn, researchers try to match program behavior to processor configurations in order to minimize both the energy utilization and the performance penalty associated with low-power configurations.

The time it takes to move an ARM big.LITTLE core in or out of sleep modes lies in the order of *milliseconds* and changing DVFS involves a latency of tens of microseconds. Furthermore, migrating a task to another core will introduce an additional penalty because of cold resources (cache, predictors) [82]. Because of these properties, a granularity of context-switch level (10 milliseconds) is adequate, as adapting to the workload any faster will only result in prohibitively large penalties.

In contrast to this, program characteristics can change at much higher frequencies [83]. Therefore, designs have been proposed that greatly reduce these penalties for heterogeneous systems [82] [84], and adaptable processors have been proposed that have very low adaptation penalties [6] [85]. These processing platforms have the potential of matching the program in a far more fine-grained way (in the time domain). However, currently used monitoring-based approaches are often based on measurement windows that are far too large to drive these high-frequency adaptations.

This work aims to determine what evaluation frequency is needed to profit from fine-grained adaptable processors. As sampling performance counters at this rate will create excessive overhead, we argue that an automatic evaluation circuit is required, moving the evaluation and adaptation control loop into hardware. Next

to sampling performance counters, we propose two additional auto-adaptation approaches. In one approach, we modified the compiler to insert instructions in locations that are likely to correspond with a phase boundary. When encountering this instruction, the processor starts a measurement and stores the results in a dedicated field in the same instruction word. The second approach involves a branch target buffer. At every branch, a measurement is started and results are stored in the buffer. When branching to the same target address again, the code characteristics have already been measured and can be retrieved. These two approaches aim to make adaptations more proactive.

We have applied the approaches to the  $\rho$ -VEX dynamic VLIW (very long instruction word) processor that is able to change configurations with a penalty of only 5 cycles (a pipeline flush). Results show that the  $\rho$ -VEX processor benefits from monitoring windows of approximately 75 cycles. Using the auto-adaptation approaches, the energy consumption of the adaptable processor can be reduced by 10% to 18% on average compared to the best static setup. The branch-based proactive approach slightly outperforms window-based solutions.

## 5.2. Approach

### 5.2.1. Target processor

In this work, we target the  $\rho$ -VEX processor, an open-source reconfigurable VLIW processor [54]. It can assign datapaths in pairs to one or multiple threads or disable them to conserve energy (see Figure 5.1). It has a reconfiguration penalty of 5

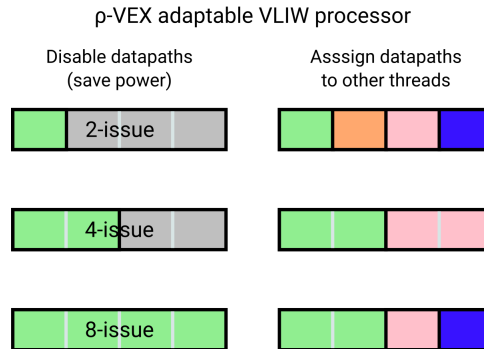



Figure 5.1: Conceptual depiction of the fine-grained reconfigurable VLIW processor targeted in this work. It consists of 8 datapaths that can be split or merged in pairs (i.e., each sub-block represents a 2-issue VLIW processor). These can be assigned to a thread or powered down to conserve power (left-hand side). Multiple blocks can be assigned to a single thread to exploit as much ILP as possible, or each block can be assigned to its own thread to exploit thread-level parallelism (right-hand side - the colors represent different threads).

cycles, because it needs to flush the pipeline. The processor can switch between a 2, 4, or 8-issue configuration without changing the binary it is executing, because it uses generic binaries [29]. In short, these work by ensuring that each VLIW bundle of 8 operations can also be executed in 2 or 4-issue mode, by removing

intra-bundle dependencies. (see Figure 5.2 for a simplified depiction of this).

### Original

```
;;
mov r2 = r3
mov r3 = r4
mov r1 = r2
;;
```



### Generic

```
;;
mov r1 = r2
mov r2 = r3
mov r3 = r4
;;
```




Figure 5.2: The *p*-VEX is able to switch configurations at any time, because the toolchain makes sure the code can be executed in every possible configuration. It does this by ‘re-sequentializing’ the code after it has been compiled for 8-issue. Each bundle is reordered such that the dependencies (shown as arrows) are met when executing the operations one by one.

## 5

VLIW architectures are widely adopted in embedded media and DSP applications, providing high energy efficiency (for example, in modem, audio and image processing subsystems in mobile phone SoCs) [86]. Code for VLIWs is statically scheduled by the compiler, decreasing hardware complexity. Instruction-level parallelism (ILP) is explicitly encoded in the binary. This makes it possible to measure performance of different core configurations, as we will see in Section 5.3. This makes the chosen VLIW platform very suitable to evaluate the proposed techniques.

### 5.2.2. Proposed auto-adapting method

The main idea behind our approach is that program characteristics change during the course of execution, but characteristics of code itself is fixed. In other words, the changes are due to the control flow through the different code sections in the binary. We propose to measure these characteristics once for every code section, and store this information in such a way that we can easily retrieve it whenever we revisit that section. For each section, a measurement only needs to be performed once for each core type (for HMPs) or configuration (for adaptable processors), after which the results for both are stored in their own field.<sup>1</sup> We are proposing two ways to store the measured code characteristics.

The first approach utilizes a structure that is similar to the branch target buffer (BTB) that is widely used in modern processors. Normally, the BTB is used to predict the branch target address early in the pipeline to reduce branch penalties. Our ‘Branch Target Configuration Buffer’ (BTCB) is a cache that is indexed by branch target addresses. Whenever a branch occurs, the BTCB is accessed to determine if there is information about the code that is being jumped to. If there is not, a measurement is triggered. When the next branch occurs, the measurement results

<sup>1</sup> On HMPs, measuring performance on one core type does not provide information about the performance on the other core type (see [87, Section 6.3]). To monitor which core type is the most efficient, the program needs to be migrated back and forth continuously. The same holds for different configurations of an adaptable processor.

are stored in the buffer. If there is information in the buffer, it can be used during the branch to reconfigure the processor to the most energy efficient configuration.

Our second approach introduces a special instruction we named `pchg` (phase change) that is added to the program by the compiler at certain locations that are likely to correspond with a longer, more stable phase (compared to the first approach, that operates on a basic block level). When encountering this instruction, a lookup is performed in a configuration buffer similar to the BTCB. This lookup can use the least significant bits of the PC (program counter) as index, or the compiler can assign indexes to code sections and place their index in the instruction.

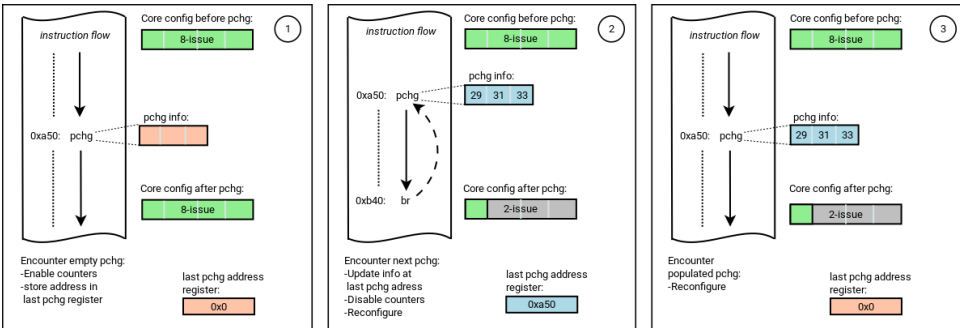


Figure 5.3: Overview of the `pchg` approach when encountering a loop, using the PC address as configuration buffer index.

During runtime, when the processor encounters this instruction for the first time, it keeps track of the index and starts the performance counters to evaluate the program characteristics in that phase. When the measurement has completed (when encountering the next `pchg` instruction), the results of the measurement are written back into the configuration buffer. Each time the processor encounters the instruction again, the information is available and the processor can use it to perform a reconfiguration immediately. An overview of the `pchg` approach is depicted in Figure 5.3.

Both approaches have their merits. The first approach is the most fine-grained but may trigger adaptations too often. The second approach requires recompilation of binaries (note that, if this is not possible, old binaries will still execute correctly but not trigger any adaptations) and results in runtime overhead because of the added instructions.

## 5.3. Implementation

This section discusses the implementation of the different approaches in the target platform. We start with the elements that the different approaches have in common, then we discuss the window-based monitoring approach, followed by the BTCB approach, and concluding with the phase change annotations.

### 5.3.1. Common

The target processor has a controller that handles reconfiguration requests. These requests can be performed via a memory-mapped control register writable by software (user or OS). Although the platform reduces adaptation overhead to only 5 cycles, sampling and evaluating performance counters in software introduces additional overhead. At the frequencies we are proposing in this chapter, this overhead becomes very significant. Therefore, we propose to use a hardware circuit to perform the evaluation and reconfiguration request directly. This section discusses this circuit.

We use a performance counter for each possible  $\rho$ -VEX core configuration. Using a scheme similar to [88], we increment these counters based on the location of a VLIW bundle marker. If a bundle is completely filled with 8 operations, the counter for the 2-issue configuration will increase by 4 and the counter for the 4-issue configuration will increase by 2 (see Figure 5.4). This scheme is enough to measure the performance of the configurations. However, we propose to estimate energy utilization.

ldw	r2 = symbol[r0]	CYC2	CYC4	CYC8
add	r3 = r3, 16			
mpyl	r4 = r3, r8	↗+1		
shl	r5 = r5, 7			
add	r6 = r6, 1	↗+1	↗+1	
add	r7 = r6, r13			
add	r8 = r6, r5	↗+1		
add	r9 = r6, r4 ; ;	↗+1	↗+1	↗+1

Figure 5.4: Measuring the performance for different configurations is done by decoding the location of the stop bit (VLIW bundle boundaries shown as `;;'). This bundle requires 4 cycles to execute on the 2-issue configuration and 2 cycles on the 4-issue. The 8-issue counter is equivalent to the bundle counter.

We have used the following energy estimation function:

$E = E_{static} + E_{dynamic}$  where

$E_{dynamic} = (SYL * E_{syl}) + (NOP * E_{nop})$  and

$E_{static} = (CYC_2 * E_{cyc2}) + (CYC_4 * E_{cyc4}) + (CYC_8 * E_{cyc8})$ .

Here,  $SYL$  is the number of execution syllables (individual operations of a VLIW bundle),  $NOP$  is the number of unfilled syllable slots, and  $CYC$  represents the number of executed cycles in 2-issue, 4-issue and 8-issue mode. The energy values depend on the hardware characteristics and should be set by the designer based on power estimations or measurements. For our evaluation we have used the values listed in Table 5.1. The dynamic part of the function is largely the same between configurations, so we can use a single cost value for each configuration.

Instead of multiplying the counter values with the energy estimation values (which would be expensive in hardware), we propose to use prescaler counters. The prescaler is increased using the configuration cycle count of the bundle (as

$E_{syl}$	$E_{nop}$	$E_{cyc2}$	$E_{cyc4}$	$E_{cyc8}$
4	1	2	3	4

Table 5.1: Used values for the energy estimation function in the simulator

depicted in Figure 5.4). When a configuration's prescaler exceeds its cost value, its energy estimation counter is increased by 1 and the prescaler is reset. The prescaler only needs enough precision to express the ratios between the cost values. The final energy estimation counters also needs limited precision, because 1) we are measuring relatively short sections of code and 2) if two estimations are very close to each other, both choices are equally suitable. In our current implementation, we are using 7 bits per configuration for the energy estimation counters. When any one of the counters overflows, all of them are right shifted by 1 position (the ratios between them stay intact). The required storage for the configuration buffer entries is  $7 \times 3$  bits (one for each possible  $\rho$ -VEX configuration).

5

### 5.3.2. Window-based monitoring

Window-based monitoring is not a novel approach proposed in this chapter but rather the current art to which we would like to compare. Using the hardware circuit from the previous section, our window-based implementation evaluates the energy estimation using a fixed period. The configuration with the lowest value is forwarded to the reconfiguration request register, and the counters are reset.

### 5.3.3. BTCB

For this approach we propose to add a buffer, the Branch Target Configuration Buffer (BTCB) that stores code information about branch targets. In case the processor already features a BTB, such as the Philips TriMedia VLIW [89], this structure can be widened to include the desired information.<sup>2</sup> When the processor executes a branch (conditional branches are only considered when taken), it will perform a lookup in the buffer to see if there is an entry with valid code information. If that is the case, it will perform a core adaptation.

If no such entry is found, the processor will start the performance counters. A register keeps track of the index of the entry. When a new branch is taken, this register is used to update the BTCB using the measured values. This can be done one cycle later than the new branch's BTCB lookup, to avoid requiring an additional access port. In our implementation, the BTCB is direct-mapped. Therefore, any collision (two branch addresses that map to the same BTCB entry) results in an eviction.

<sup>2</sup>Note that in that case, it is no longer indexed by the branch *target* but rather the PC of the branch itself; the buffer will return the predicted branch target and we propose to add the code information for that branch target to the entry.



### 5.3.4. Phase change annotations

In this approach, the compiler identifies locations that are likely to correspond to a phase. In these locations, it adds an instruction, named `pchg` (phase change). The processor performs a lookup in the configuration buffer when encountering this instruction, instead of at every branch. We have modified the  $\rho$ -VEX compiler to add a `pchg` instruction at the top of every loop and every leaf function. The compiler can choose to skip loops and functions that it estimates to have a total execution time lower than a certain threshold.

## 5.4. Evaluation

### 5.4.1. Experimental setup

To evaluate our approach, we have used the open source  $\rho$ -VEX polymorphic processor as discussed in Section 5.2.1. We have implemented our `pchg` approach in the compiler as discussed in Section 5.3 and modeled the monitoring hardware in the simulator. To measure only the behavior of the processor core, caches were disabled. Using this setup, the simulator is cycle-accurate regarding a  $\rho$ -VEX core attached to single-cycle instruction and data memories, as the code is completely statically scheduled. We will use MiBench [35] and SPEC CPUINT 2006 for our measurements. Not all programs could be used, as some are not supported by the  $\rho$ -VEX toolchain or libraries. We will use the modes listed in Table 5.2.

Type	Modes
Static core	2-issue, 4-issue, 8-issue
Dynamic core, windowed	10,000, 1,000, 500, 250, 100, 75, 50
Dynamic core, <code>pchg</code>	<code>pchg-0</code> , <code>pchg-100</code>
Dynamic core, BTCB	BTCB-inf, BTCB-2048

Table 5.2: Evaluated modes of execution.

Here, the static setups represent the supported  $\rho$ -VEX configuration modes, without any runtime adaptations. The windowed modes utilize performance monitoring with fixed windows of various sizes to perform core adaptations. The `pchg` modes utilize the proposed phase change annotations, with loop annotation thresholds of 0 and 100 cycles. BTCB uses the proposed branch target configuration buffer. We have evaluated a buffer with infinite entries and one with 2048 entries.

We will use the Energy-Delay Product (EDP) as metric and normalize to a static 8-issue configuration which represents the highest performing setup. Note that, due to the chosen values for the energy estimation function (see Table 5.1), the outcome for all measurements cannot be lower than 0.5, because no setup can execute faster than the 8-issue and the 2-issue energy estimation is  $0.5\times$  that of the 8-issue.

### 5.4.2. Results

### Overhead

Adding the `pchg` instructions into the programs results in runtime overhead. We have measured this overhead by running all 3 version of the binaries (not annotated, threshold 0, threshold 100 cycles) on a static 2-issue core. The results are plotted in Figure 5.5. On average, the runtime overhead is quite acceptable at approximately 0.5% on average.

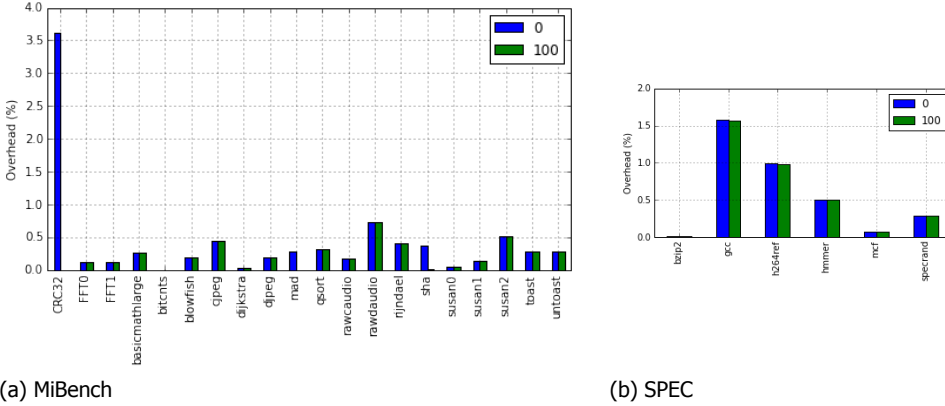


Figure 5.5: Overhead of adding the phase change instructions.

### Window sizes

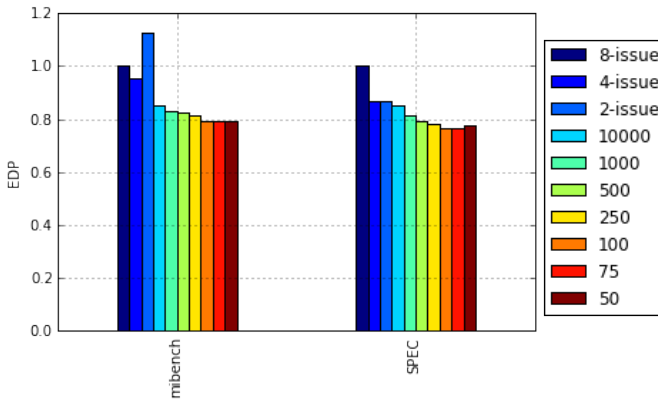
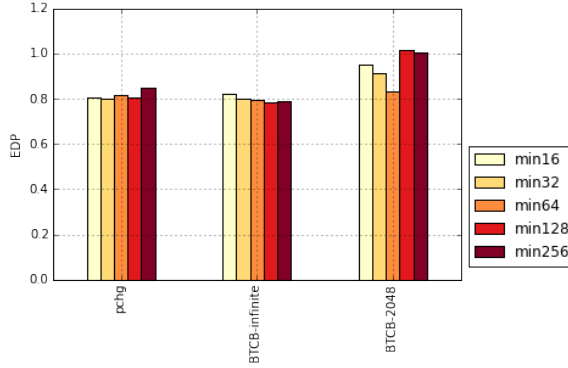


Figure 5.6: EDP for different window sizes. For both benchmark suites, 75 instructions performs best.

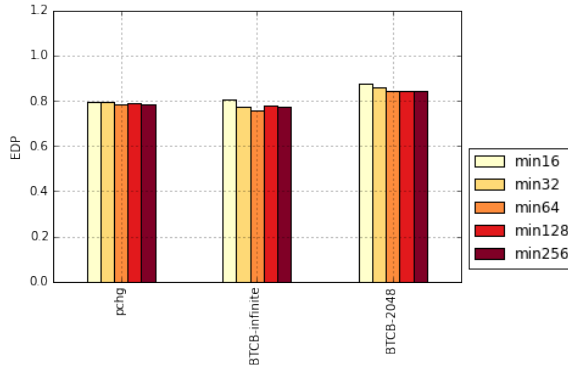
We evaluate windowed monitoring setups using various window sizes between 50 and 10,000 cycles. The results are plotted in Figure 5.6. For both benchmark suites, the disadvantage (overhead) surpasses the advantage of higher frequency adaptations at approximately 75 cycles. Our measurements reveal that using a

window size of 75 compared to 1000 cycles improves EDP up to 20% (for `specrand` and `rijndael`) and on average 6%, supporting our claim that code can change very frequently and a fine-grained reconfigurable processor is able to match these changes more closely.

### Runlength thresholds



(a) Mibench.



(b) SPEC.

Figure 5.7: EDP for different runlength thresholds.

The energy estimation counters can use a minimum runlength threshold for a measured code section. If this threshold is not reached when the measurement is finished (because of a new `pchg` instruction, or because of a branch), the core will not perform an adaptation. We have evaluated different threshold values and the results are depicted in Figure 5.7. In case the BTCB is limited in size to 2048 entries, there is a clear optimal threshold for MiBench of 64 instruction bundles and the relative loss in performance (compared to the best performing setup with an infinite buffer) is in this case 6%. The other setups, as well as the SPEC benchmarks, are not as strongly influenced by the threshold. The loss can be attributed mostly

to two outliers in the form of `basicmath` in MiBench and `specrand` in SPEC, that may suffer from a high number of collisions.

### Comparing the approaches

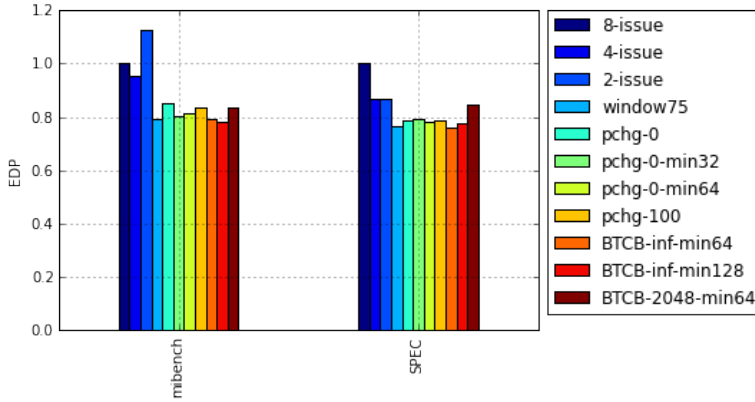


Figure 5.8: EDP for the best performing setups for each approach.

Using the best results for each approach as reported in previous sections, we have plotted the averages of the different techniques in Figure 5.8. The dynamic setups perform considerably better compared to the static cores. The first observation is that the window-75 setup performs relatively well, achieving 10% and 17% better EDP on average (for SPEC and MiBench, respectively), compared to the best performing 4-issue static core. The BTCB approach performs best, with on average 12% and 18% better EDP. The `pchg` annotations perform up to 26% and on average 10% (SPEC) and 16% (MiBench) better than the best performing static core.

For many programs, ILP variability is quite low, and the EDP for the dynamic approaches is not significantly lower than that of the best performing static setup. The largest gains are measured for the program `rawaudio` with all approaches achieving approximately 25% better EDP than static setups. However, the window-1000 approach performs similarly for this program (indicating that fine-grained approaches do not provide an advantage) In contrast, `rijndael` does not show any improvement when using a 1000 cycle monitoring window, while our proposed BTCB approach provides 20% lower EDP compared to the best static core and 8% over the best window approach (75 cycles).

## 5.5. Related work

The polymorphic processor used in our evaluations is discussed in more detail in [85]. Other dynamic processors that could make use of our proposed scheme are MorphCore [6], TRIPS [7] and CoreFusion [8]. Rodrigues et al. [90] propose a dynamic processor that morphs by allowing one core to take control over a func-

tional unit residing in a neighboring core. They introduce a dynamic phase classification scheme that uses a table to store and lookup phases. Guo et al. [88] built a windowed counter scheme for the  $\rho$ -VEX that predicts program phases and reconfigures the processor accordingly. Similarly, [91] tries to predict phases using statistical and table-based predictors. Chi et al. [92] show the advantage of combining static and dynamic profiling techniques to improve performance/energy tuning, focusing on disabling some processor resources and fetch throttling. Our approach uses compiler analysis instead of profiling as the static component.

In addition to dynamic processors, the scheme can be used by single-ISA heterogeneous multicore systems [13] such as ARM big.LITTLE processors [14], particularly, systems that were designed to have low migration penalties such as [82] and [84]. For schemes with similar objectives on HMPs see for example [93], [87] and [83]. Related work in autotuning are for example [94] and [95], where hardware modules are introduced that perform evaluation of power and performance on a softcore processor. However, the purpose is to perform dynamic partial reconfiguration, which is very different from how the  $\rho$ -VEX works.

Sherwood et al. [96] propose a similar technique of using an on-chip buffer to store detected phases based on branches, but focusing on long, stable phases. In addition, they evaluate “Dynamic Processor Width Adaptation” similar to the  $\rho$ -VEX (but supporting only a 2-issue and 8-issue configuration). They perform a short measurement in both configurations at every phase change, which is one of the problems that our proposed solution aims to solve (see Section 5.2.2).

## 5.6. Conclusions

When targeting a highly dynamic processor that has a low reconfiguration penalty (in this work, the  $\rho$ -VEX with a penalty of 5 cycles), improvements in energy efficiency can be gained by using very fine-grained automatic adaptations. Evaluations of window-based autotuning of the configuration show that using a window of 75 cycles results in the best EDP (up to 20% better than a 1000 cycle window). This confirms that code characteristics can change very rapidly, and that the dynamic processor is able to follow the changes more closely than traditional autotuning schemes that use relatively large window sizes. Not all programs show this highly dynamic behavior.

The proposed approaches open up the possibility of superscalar-based, single-ISA heterogeneous or adaptable processors with low penalties. Using a window-based approach is not possible in this case, because it would need continuous migrations between core types to evaluate the code characteristics, negating the advantages. Using our proposed methods to store information about code sections, measurements need to be performed once in every configuration, after which the information is stored and can be retrieved when revisiting the section again.

Overall, the approaches enable the reconfigurable processor to achieve up to 25% and between 10% and 18% on average better EDP compared to the best static platform. The proposed BTCB approach achieves the best results, slightly outperforming window-based autotuning.

# 6

## Adapting to dynamic workloads

*The previous chapter studied different methods to automatically adapt a processor to the code characteristics of a running program. While a single program can have very dynamic code properties, a large part of the dynamic behavior in embedded workloads is due to there being multiple tasks placed on a single processor. These tasks will not be active all at the same time and will all have their own code characteristics as well. This chapter aims to improve performance by continuously maximizing resource utilization. Similar to the previous chapter, we evaluate setups that rely solely on performance monitoring and setups that use compiler analyses.*

## Abstract

Dynamic (polymorphic) processors are designed to cope with dynamic workloads encountered in many modern-day embedded (signal processing) domains. They differ from their counterparts, i.e., homogeneous (employing SMT) and heterogeneous multi-processors, in how to achieve their purpose. Still, they are faced with the following challenges: (1) how to efficiently measure code characteristics on-the-fly, (2) how to determine the most suitable core (type), and (3) how to increase the sampling rate of performance monitors for more fine-grained behavior. In this chapter, we propose to use a VLIW-based architecture and exploit its characteristic of having explicit parallelism to dynamically match running code to core configurations (of the polymorphic processor) or core types (in case of using heterogeneous multi-cores). Additionally, we propose two methods to perform this match; using a compiler pass that annotates the code and using a monitoring-based scheme. Both methods are implemented and evaluated in terms of overhead, code coverage, and performance on randomly generated task sets. Results show that our methods allow a polymorphic processor to execute these task sets 20% faster on average compared to a heterogeneous multicore system with equal computational resources.

## 6

### 6.1. Introduction

Modern-day embedded signal processing platforms are faced with increasing workload complexity due to the dynamic nature of applications. An example is the code explosion in automotive systems. Currently, Heterogeneous Multi-Processors (HMP) (e.g., ARM big.LITTLE) are used to deal with these dynamic workloads. The use of high-performance and low-power cores are self-evident as suggested by their names. However, and less evident, an application without inherent Instruction-Level Parallelism (ILP) should avoid being executed on a high-performance core. Moreover, penalties are associated with the transfer of execution from one core type to another, e.g., (1) context saving/restoring and (2) “cold”-starting the caches and predictors after the migration. These factors severely limit the frequency at which tasks can be migrated to the most suitable core.

Based on these observations and supported by a theoretical foundation given by [9], dynamic (polymorphic) processors were proposed. They can merge cores to achieve high performance for single programs or split them for high total throughput for multiple programs. As internal states no longer need to be moved between the cores, they adapt much more quickly to dynamic workloads. Moreover, dynamic processors have more configuration options compared to current HMPSoCs (big.LITTLE has only 2 options - a big core and a little core). This allows the processing platform to more accurately match the application requirements. However, the challenge of characterizing the current workload and determining the most efficient processor configuration for it remains.

The dynamic processor targeted by this work is a polymorphic VLIW processor called  $\rho$ -VEX [21]. It allows for run-time adaptation of the issue-width with a latency of 4 and penalty of 5 clock cycles. Possible execution modes are a single 8-way,

two 4-way, four 2-way VLIW cores, or a combination. This way, the  $\rho$ -VEX can run tasks with high ILP on the full 8-issue processor, achieving high single-thread performance, and running lighter threads on multiple 2-issue cores, achieving high multi-threaded throughput. VLIW-based polymorphism provides a natural way of assigning execution resources (i.e., datapaths) to threads, as the compiler has already bundled instructions that can be executed in parallel. Unused slots can be assigned to execute another thread. It also enables quick determination of how many datapaths the program needs to achieve its maximum performance. This is not possible for superscalar architectures (where parallelism is implicit and needs to be extracted by hardware). More precisely, the compiler is aware of the instruction schedule and can therefore, precalculate execution times of code sections for different processor configurations.

In this chapter, we propose two methods to explicitly provide information on the available parallelism to the runtime environment, that subsequently determines the most efficient assignment of execution resources to threads. The first method utilizes performance counters to determine the performance of supported processor configuration by simply counting the operations encoded in the VLIW (instruction) bundles. The second method utilizes the compiler to encode this information into the binary. This approach can be more fine-grained than the first method and aims to make the core adaptations more *proactive* and reduce the time code is running on a mismatched core. We implemented a task scheduler for the  $\rho$ -VEX that periodically samples the datapath utilization information to select the most appropriate hardware configuration. Furthermore, we evaluated a fully automated scheme where the processor itself utilizes the compiler information from the currently running threads to continuously optimize its configuration. Our results show that our approach allows the dynamic processor to achieve 20% higher performance on average compared to a HMP with equal computational resources.

## 6.2. Background

The first goal of this work is to assign computational resources (datapaths) to threads to maximize resource utilization (thereby increasing throughput). The second goal is to do this in a fine-grained manner, to exploit high-frequency changes in code characteristics present in certain programs. There are different manners in which this assignment can be performed:

- **Simultaneous Multi-Threading** executes multiple execution contexts on a set of datapaths. Hardware performs the resource assignment to threads. Therefore, an intelligent resource assignment method is not necessary and we will not evaluate our approach on this type of system.
- **Heterogeneous processors** migrate tasks between larger/smaller cores to assign more/less computational resources. In current HMPs (based on superscalar architectures), this includes larger structures that are required to extract ILP, such as instruction windows, register renaming logic, and dependency checking circuitry. Furthermore, task migrations typically incur a considerable performance penalty.



- **Polymorphic processors** can split or merge (parts of) their resources. Performance penalties involved with splitting or merging is typically far lower compared to HMPs, because internal states (of running threads) do not need to be moved to another core.

Heterogeneous and polymorphic systems based on superscalar architectures face the challenge how to measure/evaluate the performance of a workload. Because ILP is implicit and must be extracted by the processor, performance can be measured on one core but this will not provide an accurate prediction of the code's performance on another core [87]. Compile-time profiling can improve the prediction, but it remains challenging at run-time without frequent migrations between the cores and, therefore, incurring the associated migration penalties.

In contrast to the superscalar (sequential) binaries, VLIW binaries do provide explicit ILP information by means of the bundle boundaries ('stopbits'). Stopbits signal the end of an instruction packet (bundle) indicating to the processor that these instructions can be executed in the same cycle. Even when running in a lower issue-width than the maximum bundle size, the processor can count these bundle boundaries and compare with the number of committed instructions to obtain the average number of operations per bundle. This is a strong indicator of performance, as running a program with a low average number of instructions per bundle will likely not execute any faster on a high issue-width processor. This relation, next to their prevalence in embedded signal-processing systems, is the key reason we chose to target VLIW architectures.

### 6.3. The $\rho$ -VEX polymorphic VLIW processor

In this section, we provide a brief summary of the  $\rho$ -VEX processor [85]. The design combines an 8-issue datapath organization similar to what can be found in the Texas Instruments TMS320C6x family of DSPs [97] with a multi-threaded approach as seen in the Qualcomm Hexagon DSP processors found in modern Snapdragon chipsets [86].

By assigning datapaths to threads, programs can run in 2-issue, 4-issue, or 8-issue mode or combinations thereof. It can change between configurations with a penalty of 5 cycles (to flush the pipelines and restart the datapaths). In contrast to previously proposed polymorphic processors [98] [99], the  $\rho$ -VEX processor can change configurations at any time during run-time without separate binaries and checkpoints. This is achieved by use of generic binaries [29]. Even though the original chapter reports performance penalties up to 30%, subsequent improvements have reduced this to below 5% (via a different register allocation algorithm and register renaming during assembly). Generic binaries are a requirement to use the  $\rho$ -VEX's polymorphism in general, not just to use the concepts proposed in this work. The final property of the  $\rho$ -VEX platform is its design-time configurability allowing static versions, without polymorphism, to be synthesized. We utilized these static configurations for our evaluations ("one 4-issue core with two 2-issue cores", similar to performance-asymmetric HMPs such as ARM big.LITTLE).

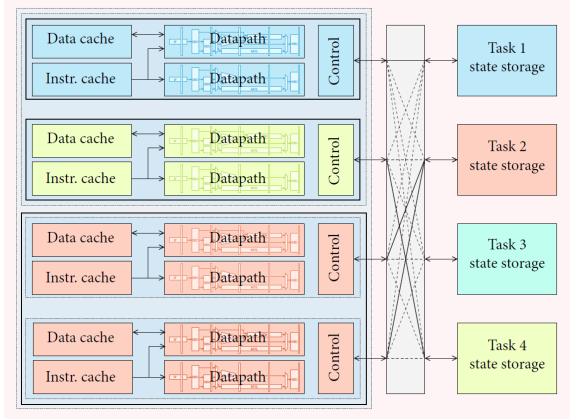


Figure 6.1: The  $p$ -VEX organization: The datapaths are connected to the task state storage (i.e., register file and complete set of control registers such as program counter) through a set of multiplexers. Performance is achieved by assigning more datapaths to a task and high throughput is achieved by running multiple tasks. The depicted '422' configuration shows tasks 1/2/3/4 running in 2/4/idle/2-issue mode, respectively.

## 6.4. Approach

### 6.4.1. On-line profiling

The use of stopbits in a VLIW architecture allows for quick determination whether an instruction bundle can execute faster on a high issue-width processor while running on a low issue-width processor and vice versa. For example, it takes 4 cycles to commit an 8-operation bundle on a 2-issue machine during which only a single stopbit is encountered. This clearly indicates that the same bundle could have been executed in a single cycle on an 8-issue machine. This information can be sampled over a longer period of time to measure the average amount of VLIW datapaths that a given code segment is able to utilize. This gives a strong indication on how many resources should be assigned to this task.

```
ldw  r2 = symbol[r0]
add  r3 = r3, 16
mpyl r4 = r3, r8 ----- ops +2
shl  r5 = r5, 7
add r6 = r6, 1 ----- ops +2
add  r7 = r6, r13
add r8 = r6, r5 ----- ops +2
add  r9 = r6, r4 ; ;
----- ops +2, stops +1
```

Figure 6.2: Execution of a single VLIW bundle consisting of 8 operations on a 2-issue core. The dashed line depicts the operations that are executed per cycle. By counting the number of committed operations and the encountered stopbits, the average datapath utilization of the code can be calculated easily.

Operation and bundle boundary counters were already present in the  $\rho$ -VEX design. In our current implementation, sampling the 'stopbit' and operation counters happens in software at a frequency of 100 Hz. As the  $\rho$ -VEX can be synthesized at 80 MHz on a Virtex-7 FPGA, this corresponds to a period of 800.000 cycles.

#### 6.4.2. Compiler annotations

Our second approach entails information passing from the compiler to the runtime regarding the datapath utilization. An additional compiler analysis pass inserts annotations into the binary code. This pass is rather simple, as we target a VLIW architecture, and is performed after the compiler has exhausted all ILP extraction possibilities. Furthermore, our analysis has access to the iteration counts of loops when they are fixed, or will otherwise estimate them based on heuristics. These iteration counts are normally used for static branch optimization (minimizing branch penalties for the most common path by reordering basic blocks). We will evaluate different thresholds using the iteration count; 1000, 100, or 0 (annotate every loop, regardless of length).

During the analysis pass, we accumulate the schedule lengths for the different  $\rho$ -VEX configurations (2-issue, 4-issue, 8-issue). If the number of total cycles of such a schedule (iterations  $\times$  schedule length) is larger than the threshold, a pair of instructions is added before the head of the code section (see Figure 6.3). These instructions write the schedule lengths (normalized to  $\leq 255$  so that the information for 4 possible hardware configurations can fit in a single 32-bit word) to a memory address. In our current implementation, this memory address points to a processor control register. Alternatively, it can point to a field in an OS data structure that is evaluated by the OS scheduler [100]. As the reconfiguration penalty of the  $\rho$ -VEX is extremely low, the read-out of these control registers and hardware configuration determination are moved to hardware resulting in significant overhead reduction. Therefore, we implemented this in our simulation for further evaluation. Furthermore, the compiler is able to clear the value after the loop, in order to 1) measure the portion of time that the processor is running in annotated loops (to measure the coverage) and 2) to inform the runtime that there is currently no information about the parallelism. Subsequently, the runtime can decide to configure the VLIW to the 2-issue mode, to keep the current configuration, or to give priority to other threads that do have valid ILP information. Resetting the value is done by added a single store instruction to all the basic blocks that have an exit arc from the annotated loop.

#### 6.4.3. Datapath assignment

As the  $\rho$ -VEX has 4 contexts, the runtime can evaluate the parallelism of up to 4 threads and decide how to best assign the VLIW datapaths to them. The scheduler works as follows. First, it divides the total number of operations in the schedules by the 3 different configuration settings to obtain the average resource utilization (for example, a loop with 18 operations requires 9, 6, and 3 cycles to execute in 2-, 4- and 8-issue modes, respectively, with resource utilization factors of 2, 3, and 6 - maximum total resource utilization is 8 equal to the number of datapaths). Second,

```

##<rVEX> loop schedule lengths: (48 | 17 | 20 | 30 ) = 0x3011141e
    mov    $r0.46 = 0x3011141e
    ;;
    stw    _SCHED_LENGTHS[$r0.0] = $r0.46
    ;;
##<rVEX> 2-issue loop schedule length: 30
##<rVEX> 4-issue loop schedule length: 20
##<rVEX> 8-issue loop schedule length: 17
.L_BB90_main:    ## 0x396
##<loop> Loop body line 39, nesting depth: 2, iterations: 600
##<loop> unrolled 2 times
##<sched>
##<sched> Loop schedule length: 17 cycles (ignoring nested loops)
##<sched>
##<sched>      2 mem refs      ( 11% of peak)
##<sched>      46 integer ops ( 33% of peak)
##<sched>      48 instructions ( 35% of peak)
##<sched>
##<freq>
##<freq> BB:90 frequency = 540000.000000 (heuristic)
##<freq> BB:90 => BB:90 probability = 0.99833
##<freq> BB:90 => BB:89 probability = 0.00167
##<freq>
##    .loc    0    42    0
    mpyhs    $r0.51=$r0.9, $r0.25
    mpylu    $r0.50=$r0.9, $r0.25
    add      $r0.40=$r0.19, $r0.9
    mpylhus  $r0.49=$r0.9, $r0.25
    mpyhhs  $r0.48=$r0.9, $r0.25
    ;;

```

Figure 6.3: Examples of assembler instructions that are added by the compiler before the loop head.

it determines the highest total resource utilization for every possible configuration.

## 6.5. Experiments/Evaluation

In our evaluations, we used programs extracted from the SPEC (CPU2006 Integer C), MiBench, and PolyBench benchmark suites. SPEC represents a general-purpose workload targeting workstations, MiBench focuses on the embedded domain, and PolyBench consists mostly of scientific workloads. We chose these different suites to be able to evaluate how well the compiler annotations perform on different types of workloads. For performance measurements, we mainly utilized benchmarks from MiBench, because these are the most relevant for the target application domain. The libraries used to link with the benchmarks (floating-point, division, newlib C standard library) have all been compiled with the same compiler as the respective platform under evaluation.

We used an internally developed architectural simulator to perform our evaluations. Without caches, our simulator is cycle-accurate when validated to our design prototyped on a Xilinx Virtex-7 FPGA.

### 6.5.1. Annotation overhead and coverage

The first evaluation metric is the amount of overhead by the annotations inserted during program compilation. We executed the unaltered versions of each program and compared the number of cycles to the versions that were compiled by our modified compilers. The average results are presented in Table 6.1. When annotating every loop and straight-line function that the compiler is able to find, the average overhead is 2.35% and coverage 73%. The compiler with loop threshold 100 provides a reasonable trade-off between coverage and overhead. The runtime overhead could be reduced by merging the added instructions into existing VLIW bundles (if there are empty slots available, which is usually the case). The measurements presented here use fully separate bundles for each added annotation, in order to identify the upper bounds on overhead.

The Polybench benchmark suite represents a class of programs that are highly structured, with a large fraction of stable, long-running loops. We observed that the compiler is able to achieve high coverage and the loop threshold has little influence. MiBench targets embedded systems and seems to have less structured longer loops to annotate. Using a lower threshold, decent coverage can still be obtained. SPEC represents the general-purpose domain and seems to be difficult to analyze. The highest achievable coverage is less than 50% at a cost of 3.8% overhead. The annotation phase in the compiler needs to be considerably more sophisticated to be able to properly annotate this code, therefore we cannot target this application domain with our compiler scheme at this time (regardless, note that VLIWs have flourished mostly in other domains than general-purpose).

Loop threshold	Overhead (%)			Coverage (%)		
	1000	100	0	1000	100	0
Mibench	1.50	1.90	2.30	33.7	57.9	63.7
PolyBench	2.03	2.03	2.09	84.1	84.1	84.6
SPEC	1.53	1.98	3.76	32.5	39.7	47.9
Total	1.77	1.98	2.35	59.3	69.3	72.6

Table 6.1: Average annotation overhead and coverage for the different benchmark suites

### 6.5.2. Throughput & Performance

The goal of using the datapath utilization to assign datapaths to tasks is to increase throughput for multi-task workloads. To evaluate this, we will execute randomly generated task graphs (randomly selected programs that are assigned random dependencies on other programs in the graph) on the execution platforms listed in Table 6.2.

The homogeneous multicores and the heterogeneous 422 are based on the static version of the  $p$ -VEX so there is no difference in instruction set. Note that all the platforms have the same number of datapaths. Although this chapter focuses on assigning execution resources, not caches, we have performed measurements with both a perfect memory system (single-cycle access) and caches (with a simple bus model) to provide some insight into their influence on the results. A design-space

Type	Organization	Scheduling
Homogeneous	four 2-issue cores (2222)	plain
	two 4-issue cores (44)	plain
	one 8-issue core (8)	plain
Heterogeneous	one 4-issue core, two 2-issue cores (422)	plain
	one 4-issue core, two 2-issue cores (422)	perfmon
	one 4-issue core, two 2-issue cores (422)	annotated
Polymorphic	8-issue Polymorphic	hardware
	8-issue Polymorphic	perfmon
	8-issue Polymorphic	annotated

Table 6.2: Evaluated setups

exploration of the best cache organization for the polymorphic processor is outside the scope of this chapter, therefore, we have chosen for a layout that shares similarities with the Hexagon and TMS320 VLIW processors. For cached setups, each 2-issue core equivalent is attached to a 32 KiB, 4-way set-associative cache so that total cache capacity is also equal for all platforms. The cache blocks are shared between 2 neighboring cores in case of the polymorphic processor (dual ported). The 422 platform represents heterogeneous platforms similar to ARM big.LITTLE (such as the Qualcomm Snapdragon 808 that has 2 large and 4 small cores). The platforms that make use of the compiler annotations are all running generic binaries, the other platforms are running un-annotated generic binaries. Regarding the resource assignment, we use the following scheduling types:

- Plain - straight-forward execution of the taskgraphs. The heterogeneous platform will migrate a task if a big core is idle.
- perfmon - periodically sampling performance counters and performing task migrations.
- annotated - periodically sampling the schedule info and performing task migrations.
- hardware - utilizing the schedule info, fully automatic core adaptations (only on polymorphic platform).

In the hardware scheduling mode, we simulate a hardware configuration scheduler that will adapt the core when the contents of the schedule info register change. This mode is not feasible for the 422 platform as the migration overhead will become prohibitive. Similarly, there is no *plain* mode on the polymorphic platform, as not performing any adaptations renders it equal to one of the other platforms (all possible individual configurations of the dynamic core are present). Making the schedule info available for periodic sampling will allow us to evaluate the efficacy of the compiler annotations on heterogeneous systems. We also measured this scheme on the dynamic core for comparison.

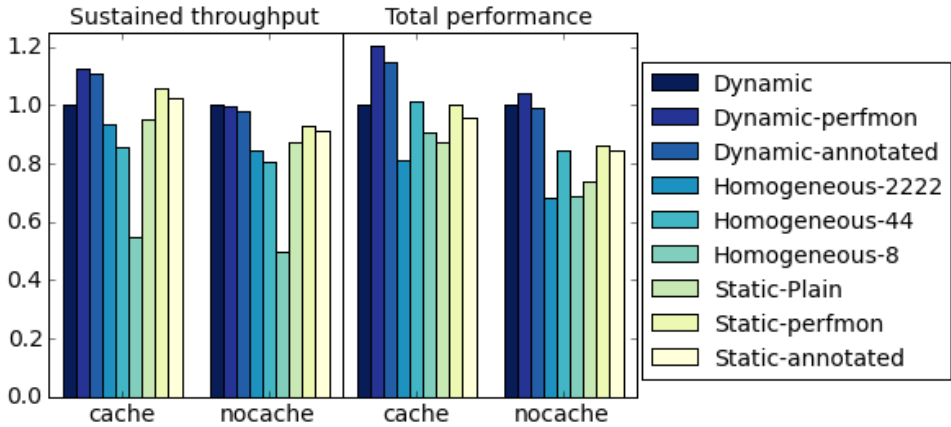


Figure 6.4: The left plot shows the average throughput (number of committed operations) during 100M cycles. The Dynamic processor achieves 8% more throughput compared to the HMP. The performance monitoring approach provides 10% more throughput than plain scheduling on the HMP. The right plot shows the normalized performance (1/execution time) of the full task graphs. Here the Dynamic processor achieves 20% better performance than the HMP. Each result set is normalized to the fully automated polymorphic processor.

## 6

Figure 6.4 depicts the normalized average throughput of all platforms. The fully automated dynamic core has been used for normalization. For setups with perfect memory (nocache), it provides the highest throughput by a small margin. When using caches, it suffers a considerable penalty (13% on average) because the cache is not shared between all datapaths. This means that excessive adaptations may cause a program to be repeatedly cut off from the cache block that may contain live data. A solution is to share the cache blocks between all the datapaths, but this will likely result in higher access latency or infeasible requirements (e.g., 4 access ports).

Of the sampling-based setups, the performance monitoring approach performs slightly better compared to the compiler-annotated approach. This can be explained partially by the overhead of the annotations (according to Table 6.1, around 2% overhead is expected). Additionally, some programs have relatively low code coverage, which may severely limit the quality of the information on which the resource assignment is based. We can see that task migrations based on performance monitoring allows the static core to achieve 10% better performance on average compared to plain task execution. Comparing the platform types, the polymorphic processor provides approximately 8% higher throughput on average compared to the HMP. The advantage varies considerably between task graphs, and can get up to a factor  $2\times$  higher or down to 20% lower. This can be attributed to the degree of variability in the tasks. When the code characteristics are very stable, a static platform is well capable of executing it efficiently. In the presence of a highly dynamic workload, the dynamic processor is able to adapt without suffering from excessive migration penalties.

The performance results, on the right side of Figure 6.4, are slightly more pronounced, with a 20% advantage shown by the polymorphic processor over the HMP. This is due to the amount of parallelism that changes during the course of the execution of the task graph (sometimes multiple tasks can be run in parallel, sometimes only one). The polymorphic is able to exploit more parallelism (TLP or ILP) in all these sections.

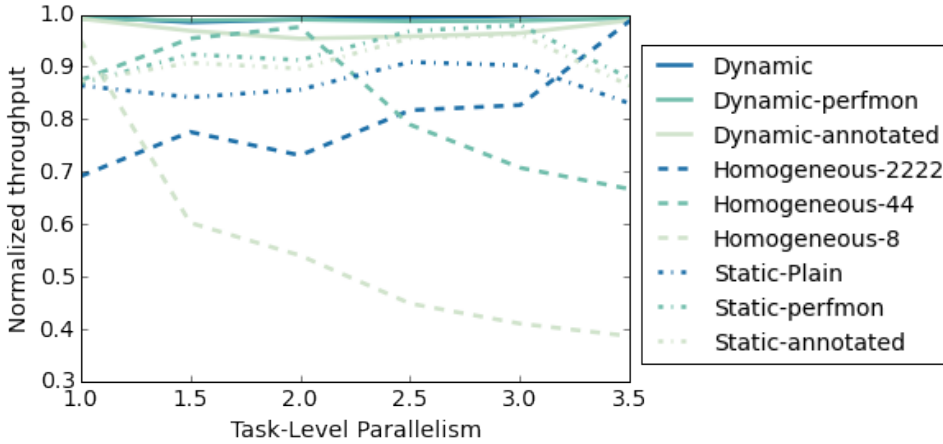


Figure 6.5: Task-level parallelism versus normalized throughput during 100M cycles.

Figure 6.5 shows the relation between Task-Level Parallelism (TLP) and the throughput of the setups. This has been calculated by monitoring the number of tasks that were active during execution and is therefore limited to at most 4 tasks. The single-core setup performs adequately when the task graph is very sequential (TLP  $\approx 1$ ) and similarly, the 4-core homogeneous platform is at its best when it can execute a task on each core (TLP  $\approx 4$ ). The HMP setups perform best with task-level parallelism around 3 (note that this platform has 3 cores). Therefore, we have tuned our task graph generation algorithm to create graphs with average task-level parallelism of approximately 3, to provide a lower bound on the advantages of the polymorphic processor. The graph shows that the polymorphic setups deliver high throughput independent of the task-level parallelism and helps to interpret the difference between the throughput and performance results in Figure 6.4. The throughput measurements show a snapshot of a workload with an average TLP of 3 (represented by the corresponding point on the X-axis of Figure 6.5), while the performance results measure the behavior during the entire lifespan of the task graphs, during which the TLP will continuously change (visualize this by moving left and right in the TLP graph of Figure 6.5). Combined with the ability to better adapt to code variability, this results in the observed 20% increased performance in Figure 6.4.



## 6.6. Related work

### 6.6.1. Phase detection and workload characterization

In order to tune application performance (be it automatically by using online adaptive approaches or manually), an important step is to measure performance characteristics of a workload. An important realization is that these characteristics change significantly during the course of a program's execution [4]. There are two general approaches in phase detection: online and offline. There have been efforts to combine the two [92] [101]. This is fully supported by our proposed scheme and will likely increase the quality of our results [4], but it is outside the scope of this chapter and regarded as future work.

Online methods use counters to profile applications at run-time [102] [103]. Every scheme that makes use of these has to balance the sampling rate and overhead. For current systems, this overhead can be kept tolerable because the rate at which the hardware can adapt (DVFS, core migration) is also low. However, our target hardware platform can adapt at much higher rates, rendering most online approaches unusable.

Most work on compiler-aided application tuning focuses on cache performance (e.g., [104]), for example by improving cache partitioning between simultaneously executing threads. A related technique that aims to reduce power consumption by reconfiguring a processor using compiler annotation is [105]. Their technique focuses on identifying regions of code that have a high cache miss ratio, and scaling down processor frequency. These techniques are orthogonal to ours, as we focus on utilization of execution resources (functional units).

### 6.6.2. Polymorphic processors

A number of previous projects have designed polymorphic processors in the past, most notably [7], [6]. Most of these efforts were based on a superscalar architecture targeting general-purpose computing. The  $\rho$ -VEX is a VLIW-based architecture that targets embedded systems. Earlier polymorphic processors based on VLIW architectures [98] [99] needed separate binaries for each supported configuration and/or are limited to configuration changes determined at compile-time. Operating system support for dynamically reconfigurable architectures is introduced by Chameleon [100], which could be used in conjunction with this work. In [88], performance monitoring is used to adapt the  $\rho$ -VEX to a single program in order to achieve higher energy efficiency. This work instead targets a multi-tasking environment, adds the compiler approach, and also allows measurements when running in lower issue-widths than the full 8-issue configuration. In [92], a hybrid approach is introduced, combining compile-time profiling and run-time monitoring to reduce the issue width of a hypothetical superscalar processor. Sherwood et al. [96] briefly discuss an adaptable issue-width processor where they sample each configuration option every time their phase classifier detects the start of a new phase. This work attempts to prevent this by proposing to exploit the parallel nature of a VLIW architecture.

## 6.7. Conclusions

In this work, we introduced and evaluated two methods to steer migrations in a heterogeneous multicore and adaptations in a polymorphic processor. Using these methods, a polymorphic processor shows a 20% improvement in performance on average over a heterogeneous platform with an equal amount of computing resources. On the heterogeneous platform, they provide 10% better performance compared to plain task scheduling. Without these methods, user intervention, manual resource assignment, or compile-time profiling is required to steer the task migrations or core adaptations. The additional value of using compiler annotations in combination with automatic core adaptations in hardware depends strongly on the workload and can provide up to a factor  $2\times$  more throughput. In the presence of caches, the performance of the most adaptive setup is reduced because the cache blocks are not shared between all datapaths of the polymorphic processor. Devising a proper cache organization that can cope with this adaptive architecture is a complex topic outside of the scope of this chapter and a possible future research direction.



## Part 3 - Real-time and mixed-criticality systems

*Part 2 discussed dynamic workloads where the optimization goal was throughput or energy-efficiency. Each task was assumed to be of equal importance. In many embedded applications, this is not possible because some tasks have real-time requirements. This means that the platform must be able to give strict performance guarantees to some tasks.*

*Part 3 of this thesis starts with discussing the architectural properties of the  $\rho$ -VEX processor that create advantages for real-time systems in Chapter 7. Then, we will discuss how they can be used to improve the static schedulability for real-time workloads in Chapter 8. Additionally, it proposes a full system architecture that is suitable for mixed-criticality systems. The platform is designed to guarantee time-safety for critical tasks while still providing as much throughput as possible to non-critical tasks.*

	<b>Hardware</b>	<b>Software</b>	<b>Scheduling</b>
Classical	Static	Static & Dynamic	Static & Dynamic
Part 1	Static (fixed, design-time)	Static	Static (compile-time)
Part 2	Dynamic (adaptable, run-time)	Dynamic	Dynamic (run-time)
<b>Part 3</b>	<b>Dynamic (adaptable, run-time)</b>	<b>Static &amp; Real-time</b>	<b>Static (compile-time) &amp; Dynamic (run-time)</b>



# 7

## Evaluating real-time properties

*In the introduction, we note that many embedded applications are faced with strict timing requirements. This chapter discusses the properties of the  $\rho$ -VEX that are of particular interest to the real-time application domain. Some complex components that are needed for the processor's adaptability can also be used to remove context switching penalties and decrease interrupt latency. These are important aspects of a computing platform for real-time systems, as the maximum interrupt latency is often limited and context switch penalties must be taken into account when scheduling multiple tasks on a processor.*

## Abstract

The register file is an expensive component in the design of any processor, especially, when considering the additional ports that are needed to support multiple datapaths within a wide-issue VLIW processor. In a recent work, these additional resources were used to dynamically reconfigure the register file to support a dynamically reconfigurable VLIW core. The design can be perceived as a single 8-issue, two 4-issue, or four 2-issue VLIW cores. Consequently, the multi-ported design can operate in different modes, namely as *one*, *two*, or *four* register files, respectively, corresponding to the active number of cores. The implementation of the register file design on FPGAs using Block RAMs still results in unused resources due to the coarseness of the Block RAMs.

In this chapter, we propose to re-purpose these unused BRAM resources to additionally support multiple contexts next to earlier-mentioned modes. In this manner, the 8-issue, 4-issue, and 2-issue cores have access to 4, 2, and 1 contexts, respectively. Consequently, we can avoid saving and restoring of the task states in a multi-task environment, turning context switching from a traditionally time-consuming event to an almost instantaneous event. The advantage of this is the reduction of interrupt latency and task switching latency, which are important in real-time and embedded systems.

Our results show that our technique can improve interrupt latency by a factor of  $17.4\times$  compared to using a software register spill routine, depending on the behavior of the memory system. Likewise, the task switching time can be improved by  $6.7\times$ .

## 7

### 7.1. Introduction

The  $\rho$ -VEX processor [21] is a dynamically reconfigurable VLIW processor that can adapt its organization to the requirements of different workloads. One of its most important run-time parameters is the issue-width that allows for adaptation towards the ILP of the task(s) at hand. The design can be configured as a single 8-way ( $1\times 8$ -way), two 4-ways ( $2\times 4$ -way), four 2-way ( $4\times 2$ -way) VLIW processor core(s), or combinations of those: e.g., two 2-ways and one 4-way. This capability requires the design of an extensive register file to support these different modes. In the worst case, the register file must provide:

- 8 write ports and 16 read ports when running in the  $1\times 8$ -way mode
- 4 architecturally separate register files when running in the  $4\times 2$ -way mode

To design a register file that satisfies these requirements we use techniques such as Block RAM (BRAM) duplication and a Live Value Table (LVT), which we will discuss in Section 7.2.

A major drawback of the current design is the large resource utilization. The BRAMs used to implement the register file on the FPGA need to be duplicated multiple times to provide the necessary amount of read and write ports. Every BRAM has a capacity of 512 32-bit words (2 KiB); however, the architecture only requires

64 32-bit registers. Because of this, the resulting design has an enormous storage capacity of which at most an eighth is used by the processor in any particular configuration.

The design presented in this chapter aims to convert the drawback of the high BRAM usage of the register file for wide-issue VLIW softcore processors into an advantage by using the overcapacity to store different execution contexts. The actual utilization of the BRAM storage capacity will increase from  $\frac{1}{8}$  to  $\frac{1}{2}$ . Support for multiple contexts in hardware relieves the core from having to spill and restore its entire register file contents to and from memory in the event of a task switch or interrupt. In a multi-tasking environment, this concept changes task switches, which are traditionally very time-consuming, into a virtually instantaneous event. Faster context switching has advantages in numerous computing scenarios, as it will increase responsiveness for interactive workloads and improve interrupt latency and task switching speeds in real-time systems. In the following, we illustrate several cases in which our work can improve performance:

- Frequently used threads: Kernel threads, like schedulers, must be frequently executed. In a traditional core implementation, timers interrupt the core and trigger context switching in order to execute such threads. In our work, these threads can be maintained within the core and thereby remove the need for context switching. For example, an application is executing in the 8-issue mode using 1 out of 4 contexts. When the scheduler needs to execute, the current thread can be scheduled to run on a 4-issue core - this mode switch only takes several cycles when using generic binaries [29]. In the remaining 4-issue core, the execution of the scheduler can be resumed by using its own context that remained "dormant" within the core.
- Dynamic switching of execution by different cores: When threads require more resources, e.g., when their ILP increases, our processor design allows for it to claim additional datapaths to execute the code more efficiently. This does mean that another thread must be stalled for a while. However, in our case, the context of the second thread does not need to be saved into the memory and can remain within the core until it is resumed. In the latter, another context switching operation is saved.
- Context-cycling after cache misses: When our processor is running in the 8-issue (4-issue) mode, it can have up to 4 (2) contexts stored within each core. This means that when one thread is encountering a cache miss, thus execution is stalled, the core can easily switch to another thread (context) and continue execution, i.e., Switch-on-Event Multi-Threading SoEMT.
- Embedded real-time systems with multiple tasks that require stringent real-time constraints (e.g., control loops with sensors and actuators). A single core can process more events using multiple contexts [106]. Therefore, a softcore can be used as microcontroller on an FPGA which would save the designer from having to design hardware circuits to handle some events or having to



resort to a multi-core system where distinct events are handled by a dedicated core.

The register file of our  $\rho$ -VEX is a complex topic, as it is also instrumental in supporting the core's dynamic reconfigurability [22]. We limit the scope of this chapter to evaluating the benefits from multiple hardware contexts. It must therefore be noted that the costs of this design (see Table 7.1) are paid not only for multiple contexts, but also to support the dynamic reconfigurability. Our approach in this chapter gives us a 17.4 $\times$  reduction in interrupt latency and 6.7 $\times$  reduction in context switching time.

## 7.2. Background

The multi-ported register file is a challenging component in the design of softcore VLIW processors. Wide-issue VLIW processors like the  $\rho$ -VEX need register files with a large number of read and write ports. The VEX instruction set architecture (ISA) supports operations that use two source registers and one destination register. Because of this, the number of write ports required is equal to the issue-width, and the number of read ports is equal to twice the issue-width. Creating such complex register files using FPGA LUT resources is very expensive and scales very poorly with the number of ports. The reconfigurable  $\rho$ -VEX design and the implementation of its multi-ported register file are introduced in [107]. Moreover, in [108], the idea of using a Live Value Table (LVT) is discussed that enables the use of banked memories with duplication to create multi-ported BRAM memories. The ideas presented in this chapter are built upon a register file design that is implemented using this technique. We will discuss the concepts and challenges briefly in this section.

Creating RAM memories that have more read ports is straightforward and achieved by duplicating the BRAM and writing data into each block simultaneously. In this way, each BRAM contains the same data, and their read ports can be used independently of each other. Increasing the number of write ports, however, is more difficult. Several solutions exist in literature. The simplest solution is to divide the register file into banks, each connected to one of the write ports [109]. This solution restricts the range of registers each write port can write to and thus reduces the freedom the compiler has to schedule instructions. Another solution introduced in [110] increases the size of each bank to the original register file size and renames the registers in between the compiler and assembler. This solution enables a banked design with the same scheduling freedom as an actual multi-ported register file but utilizes a multiple of the number of registers. Note that this technique does not necessarily require more BRAMs since their size is a lot larger than the 64 registers specified in the VEX ISA. It does, however, increase the number of bits required to specify the source and destination registers in instructions.

The register file used in the  $\rho$ -VEX uses the technique introduced by [108]. This scheme also duplicates the register file for each write port. However, instead of uniquely naming the registers in each bank, a Live Value Table (LVT) keeps track of which bank holds the most recent value of each register. It uses this information

to multiplex the right bank to the read ports, as shown in Figure 7.1. The LVT needs to be implemented as a multi-ported LUT based RAM because it still needs one write port per register file write port. However, since it only needs to hold a bank address, it is much narrower than the original register file that the scheme seeks to replace. While this technique enables the register file to be implemented mostly with BRAMs instead of LUTs, it still scales poorly with the number of ports. The number of BRAMs required is equal to the product of the number of read and write ports. The depth of the LVT scales linearly with the number of registers in the register file while the width scales logarithmically with the number of write ports. The number of ports required for the LVT is equal to the number of ports on the register file.

### 7.3. Related work

In [111] the authors analyzed the high requirements that wide-issue VLIW processors pose on the register file. They discuss hypothetical FPGA primitives similar to existing BRAMs but featuring many more read and write ports. These primitives do not exist in current FPGAs, therefore, the use of large BRAM or LUT-based structures is required to emulate this behavior [108].

In [112], it is stated that “the context switch time is one of the most significant overhead factors in any operating system” and shows that high timer interrupt handling latency can impede schedulability of real-time tasks. In [106], it is measured that using a multi-threaded architecture with 4 register sets allows an autonomous guided vehicle to run at a 28% higher velocity. In [113], measurements were performed to quantify the interrupt latency of several embedded Linux distributions running on a Xilinx Microblaze.

There are numerous examples of processors which use the concept of multiple register files to enhance the context switching time and interrupt latency in hardware. In [114], comparisons are made (by means of simulations) between increasing the number of cores and increasing the number of register sets in terms of increasing performance for a parallel workload. In [115], the MIPS architecture is extended by duplicating the register file multiple times and adding special instructions to switch between them when a context switch is required. In [116], the authors propose a novel architecture, which also supports holding multiple contexts in hardware simultaneously, and extend it with a dedicated cache to hold contexts to prevent spilling to main memory. Among other things the effects of the additional contexts on interrupt latency is investigated. Storing multiple contexts is also a requisite for (Simultaneous) Multi-Threading (SMT) [27]. An example of a VLIW processor with SMT support is the Itanium [117]. These technologies target high-end ASIC processors while this work targets the embedded (FPGA) domain.

The synthesizable ARPA-MT [118] and RTBlaze [119] processors also use SMT to improve schedulability and performance for embedded real-time systems. However, all the resource investments in this core are only used for SMT. The ARPA-MT core has a single execution pipeline. The fetch and decode circuits as well as the register file need to be duplicated for each thread slot.

In contrast, the  $\rho$ -VEX uses the additional resources to support: 1) a very wide

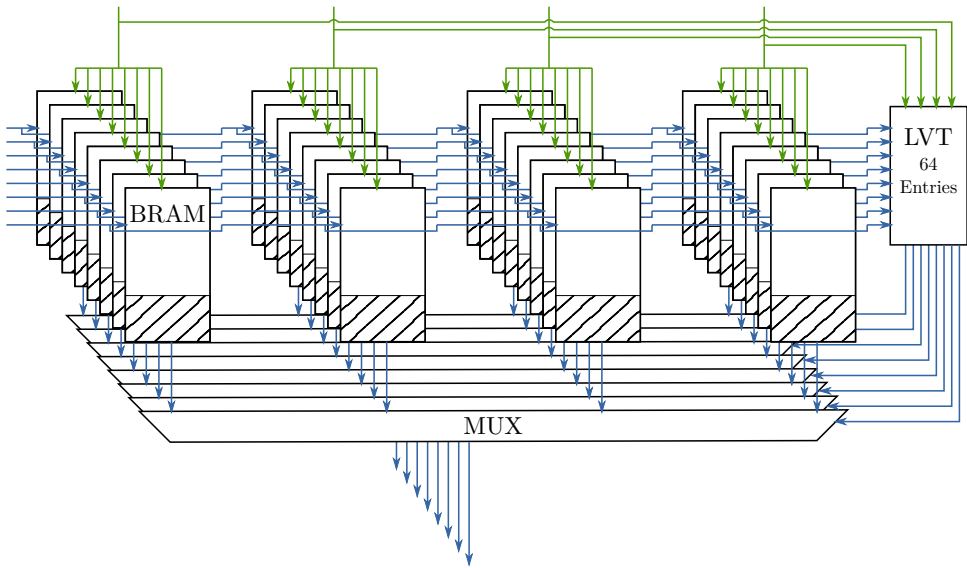


Figure 7.1: Block diagram of register file implementation using multiple banks of BRAMs. The green arrows indicate write ports, while the blue arrows indicate read ports. The shaded area represents the portion of the BRAM used for storing a single context.

## 7

VLIW to exploit ILP, 2) multiple hardware contexts and 3) a multi-core configuration (in other words, all contexts can be active and executing at the same time). Therefore, it uses the additional resources in a more efficient way compared to the previous work.

### 7.4. Implementation

Figure 7.1 shows the implementation of a register file with four write ports and eight read ports ( $4W \times 8R$ ), using BRAMs and an LVT. The  $8W \times 16R$  version would be 4 times as large. The hatched area represents the part of the BRAM that is actually used to store the 64 registers used by the  $\rho$ -VEX. The figure shows that a large part of the BRAMs is unused. Because the  $\rho$ -VEX can be configured as four independent processors, it also needs four separate register files. However, the total number of read and write ports is the same for one large 8-issue processor or four separate 2-issue processors. Because of this characteristic, the same multi-ported register file can be used in each configuration. The number of registers, however, needs to be quadrupled, for a total of 256 registers, since each core needs a separate register file of 64 registers. The BRAM resources on contemporary FPGA boards provide more than sufficient storage capacity to accommodate this, so there is no added cost in BRAM resources. However, the LVT does need to increase in size, to keep track of the most recent location of all 256 registers.

Figure 7.2 shows how the multiple contexts can be stored in the previously unused space of the BRAMs. Creating four separate register spaces is a necessary

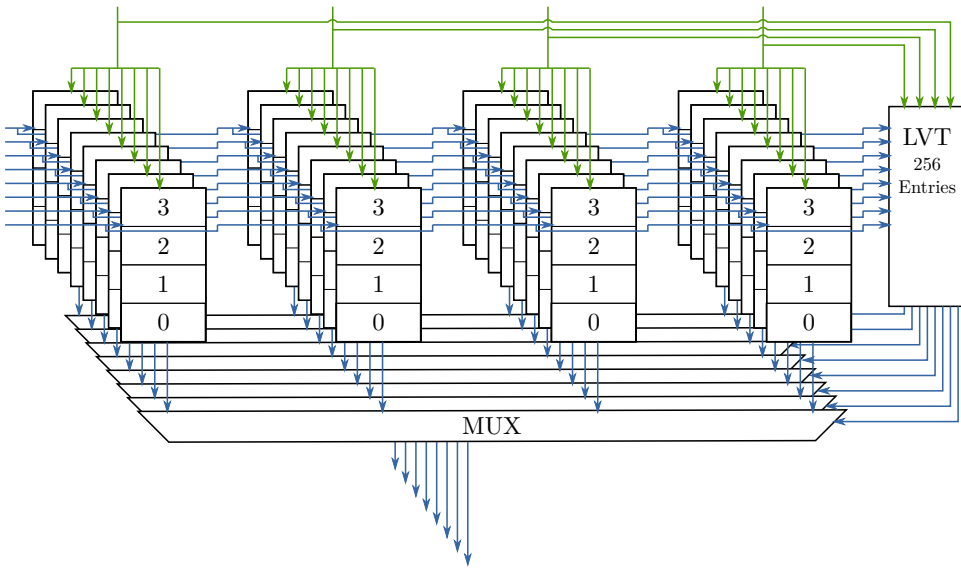


Figure 7.2: Block diagram of register file implementation supporting multiple contexts. Here the number of BRAMs is the same, but the LVT is larger.

cost to enable the  $\rho$ -VEX to be split into four separate processors. However, not all of the register spaces are used when the core is configured as a single 8-issue processor or two 4-issue processors. This creates the opportunity to re-purpose these unused register spaces as alternative register windows, which can be used to store the register context of inactive processes. Since the four register windows are implemented as a larger continuous address space, the uppermost bits can be used to select one of the four register windows.

The  $\rho$ -VEX utilizes more registers than just the 64 general purpose registers. It also has the following registers, that must be stored for a context switch:

1. A special 32-bit register used to store the return address for a function call (the link register).
2. Eight 1-bit registers used for conditional branching.
3. The program counter.
4. Various control registers, used for example for interrupt handling.

These registers cannot easily be stored in BRAMs, as the control logic needs to be able to access all these registers at once. Therefore, these registers are implemented in LUTs. To support running as  $4 \times 2$ -issue processors, all these registers need to be duplicated as well, and can thus be used as part of the hardware contexts. Some additional hardware is required to use these registers for context switching, as not every lane would necessarily need access to all duplicates of the registers for reconfiguration only, while this is necessary for context switching.

However, when this is done, the only registers which need to be spilled and restored are those registers which are used by the context switching routine, or scheduler itself. Because the additional hardware cost is small, our context switching design incorporates this feature.

A hardware context switch is not entirely free in terms of cycles in the current  $\rho$ -VEX design. To avoid complicating the forwarding logic, context switches are only possible when the pipeline is empty. Because the  $\rho$ -VEX has a five stage pipeline, five cycles are needed to flush the pipeline before a context switch can occur. In addition, the context switches are currently controlled by the dynamic reconfiguration controller, which takes three additional cycles to decode and commit a new configuration. Two of these are spent still executing instructions in the old context.

## 7.5. Experimental Setup

Our measurements are carried out using the  $\rho$ -VEX VLIW softcore processor clocked at 37.5 MHz running on a Xilinx ML605 development board, which incorporates an XC6VLX240T Virtex 6 FPGA. We use a timer connected to the interrupt request input of the processor to generate interrupts at different rates to measure the impact of our approach on the performance of the system.

We quantify the impact on performance by measuring two different values, namely:

1. *Interrupt Latency*: The number of cycles elapsed between the moment an interrupt request is received by the core, and the first instruction of the interrupt handler being executed.
2. *Context switching latency*: The number of cycles elapsed between the moment a context switch is requested (due to an interrupt), and the first instruction being executed in the new context.

Figure 7.3 shows what these latencies are made up of, namely: pipeline flushing, saving context registers, running the interrupt service routine (in our case the task scheduler), and finally restoring the context registers. By using hardware contexts the latency of saving and restoring registers can be eliminated. We measured these quantities by creating a workload of four programs. At every timer interrupt a scheduler selects a different program to execute, and performs the context switch to that program. The programs themselves have no impact on the measurements, since they are purely dependent on the time it takes to save and restore all context registers.

In order to measure the difference between hardware and software context switching, we wrote a software and a hardware context switching routine. The software version saves the complete context to the stack of the currently running task, stores the stack pointer to a predefined memory location, and starts executing the interrupt handler. The interrupt handler then calls the scheduler in order to schedule the next task. The current stack pointer is then replaced with the stack pointer of the new task. Next, the application context of the newly selected task is

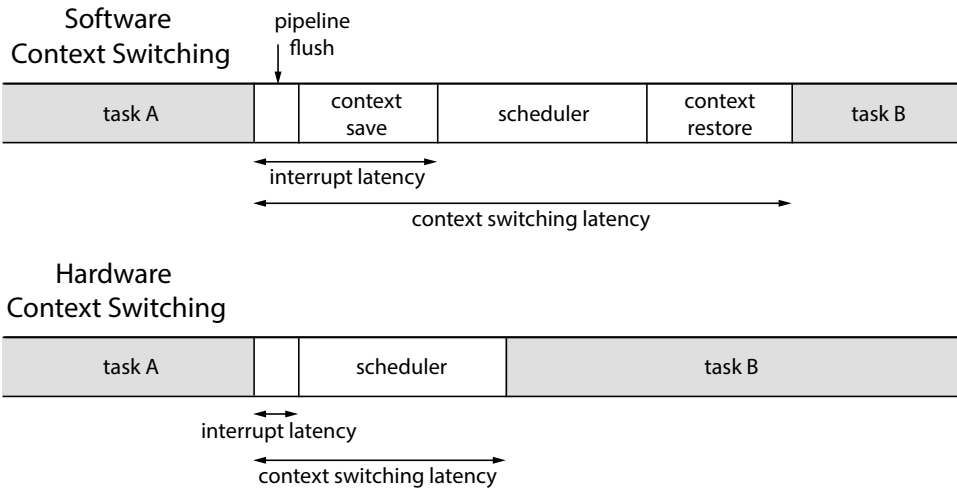


Figure 7.3: Context switching and interrupt latency definition.

restored from the stack, after which control is handed back to the application. The hardware switch routine does not need to save or restore all registers. Instead it only has to do so for the registers used by the interrupt routine, in this case the scheduler.

The scheduler utilizes a linked list in memory to determine which task to switch to; each entry representing a task, with a mapping to another task. When a task completes, the linked list is rebuilt such that the context switching code does not switch back to the completed task, and a context switch is requested immediately using a software trap instruction. When the last task completes, it signals completion to the platform.

Because cache behavior will impact the latencies for saving and restoring the contexts we perform the measurements for different memory access latencies. We measure using latencies from 0 (single cycle memory access) to 30 cycle memory access on cache miss. The cache itself consists of a separate instruction and data cache, respectively 32 KiB and 8 KiB in size. The size has intentionally been kept small, because the programs under test had to be small as well for the entire memory to fit on the FPGA; it is assumed that, under normal circumstances, larger caches will be used, but the running programs will also use wider regions of more memory. Both caches have single-cycle hit latency for reads. The data cache has a two-cycle latency for writes for both hits and misses, as long as one of the four write buffers is vacant.

To evaluate the context switching overhead in multi-process time-sharing systems, overall performance of the multi-task system is tested on hardware using the cached system. The timer is used to generate an interrupt at a fixed frequency, often referred to as the system “tick,” in which a context switch is performed. Clearly, the context switching overhead is directly related to the frequency of the system tick [112]. The frequency of the tick is usually in the order of 50 to 1000 Hz. A

	Register File		Core	Increase over Core
	1 Context	4 Contexts		
Slice Registers	806	1392	8529	6.9%
Slice LUTs	10764	15591	35148	13.7%
RAMB18E1	128	128	147	0%
RAMB36E1	0	0	128	0%

Table 7.1: Resource usage of register file with and without support for multiple contexts.

lower frequency will lead to lower switching overhead, but higher frequencies will result in a more responsive system. Systems that require more responsiveness will therefore have a higher tick frequency. For example, the Linux kernel uses a system tick of 1000 Hz for desktop systems, but this can be reduced to 100 Hz for server systems to reduce overhead. On the other hand, the Windows kernel uses 66 Hz. The frequency is varied between tests to evaluate its effect. In addition, the system is evaluated with varying bus latencies. The latencies used are estimates of what the average latency would be for a real off-chip memory system.

A cycle counter available within the  $\rho$ -VEX processor is used to measure the time from system reset to the program completion signal, which is given by the task switching implementation when all tasks have completed. For each timer and memory system configuration, both context switching implementations are evaluated. Because all other factors are kept constant, the difference in total execution time is only dependent on the context switching overhead. The speedup between the baseline and hardware context switching implementations is then determined to quantify this overhead.

## 7

### 7.6. Results

In Table 7.1 we show the increase in resource utilization of the register file when adding support for four contexts. As expected the number of BRAMs used does not increase. Only the number of registers and LUTs increases, since these are used to implement the LVT. While these increases seems large, when compared to the total usage of the core they are less significant. Additionally, note that this increase in resources in the register file is required to support the dynamic reconfigurability of the processor.

As we can observe in Table 7.2, the interrupt latency is 87 cycles for software context switching. The interrupt latency when using hardware contexts is only 5 cycles, solely due to the pipeline flush performed by the trap handling logic. A full context switch, i.e., the time between a tick interrupt request and the execution of the first instruction in the new context, takes 174 cycles using the software implementation, compared to 26 cycles using the hardware contexts.

In Table 7.3, we can observe the results of the same experiments run using a cached memory system, with a bus latency of 20 cycles. We observe that the improvement due to the hardware context switching is greater in this system, with

	Software	Hardware	Reduction
Interrupt Latency	87	5	17.4×
Context Switch Latency	174	26	6.7×

Table 7.2: Interrupt and context switching latency with single-cycle memories in cycles.

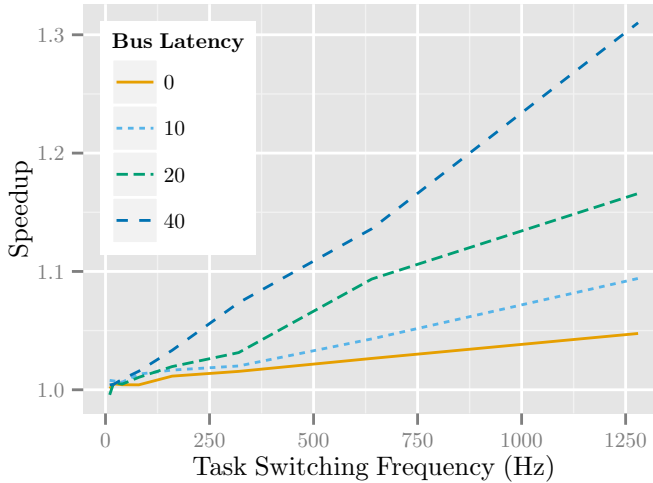


Figure 7.4: Speedup of the multi-task system due to the hardware context switching implementation.

the improvement in interrupt latency increasing from 17.4 to 23.5×, and the improvement of context switching time increasing from 6.7 to 14.8×.

	Software	Hardware	Reduction
Interrupt Latency	16798	713	23.5×
Context Switch Latency	31861	2148	14.8×

Table 7.3: Interrupt and context switching latency with cache and 20 cycles bus latency in cycles.

Figure 7.4 shows the speedup for different frequencies of the timer tick parameterized for different memory latencies, as measured on hardware using the cached system. It can be seen that in the region of higher task switching frequencies the difference between hardware and software context switching can be quite substantial depending on the memory system. A speedup of over 1.3× can be achieved for a bus latency of 40 cycles at a switching frequency of 1280 Hz.

## 7.7. Conclusions

The concept of using additional register files to speed up multi-threading performance has been applied in numerous designs in the past. In this chapter, we



apply the concept to an existing design, exploiting the overcapacity of the BRAMs in the existing implementation of the multi-ported register file and the additional logic required by the parameterized reconfigurability of the  $\rho$ -VEX softcore. We have demonstrated that the proposed design can decrease the interrupt latency by a factor of over 20 times in a realistic environment. Likewise, the total context switching time can be decreased by a factor of over 10 times. In a simple multi-task system the effect of this is apparent as the decrease in overhead results in a speedup of  $1.3\times$  in the most extreme case evaluated. For applications with few real-time requirements, where the system tick frequency would be relatively low, the speedup is negligible, as the task switching code would not be executed as often. However, embedded real-time systems that need to process large numbers of events will benefit most from the improvements.

# 8

## A platform for mixed-criticality systems

*In the previous chapter, we have shown how the architectural properties of the  $\rho$ -VEX result in improved interrupt latency and context switching penalties. Creating a real-time schedule, however, is considerably more involved and is the focus of the present chapter. It includes finding the worst-case execution time of the programs in the schedule, which can be difficult if the performance of the processor is highly unpredictable (for example, because it uses techniques such as caches and branch prediction).*

*We highlight the performance predictability of the VLIW-based architecture and present a broader view on designing a full platform that is suitable for mixed-criticality systems. In addition to the processor, we discuss spatial and temporal isolation of the memories as well. This is important, as the memory is also a part of the program state, and unpredictable memory access latency can create problems similar to unpredictable processor performance. A system must be able to provide guarantees on both sides, to ensure that no task is ever influenced by another task both in terms of functionality (not executing correctly because the state has been corrupted) and timing (missing a deadline because it unexpectedly needed to wait for another task).*

*We discuss the topic of schedule creation for an adaptable processor, and show that the static schedulability can be improved compared to static multicore processors. Additionally, we study whether it is possible to increase resource utilization within the static real-time schedule, by adding a dynamic scheduling component that re-assigns unused resources to another task.*

---

Parts of this chapter have been published in [42] and [43].

## Abstract

As embedded systems are faced with ever more demanding workloads and more tasks are being consolidated onto a smaller number of microcontrollers, system designers are faced with opposing requirements of increasing performance while retaining real-time analyzability. For example, one can think of the following performance-enhancing techniques: caches, branch prediction, out-of-order (OoO) superscalar processing, simultaneous multi-threading (SMT). Clearly, there is a need for a platform that can deliver high performance for non-critical tasks and full analyzability and predictability for critical tasks (mixed-criticality systems).

In this chapter, we demonstrate how a polymorphic VLIW processor can satisfy these seemingly contradicting goals by allowing a schedule to dynamically, i.e., at run-time, distribute the computing resources to one or multiple threads. The core provides full performance isolation between threads and can keep multiple task contexts in hardware (virtual processors, similar to SMT) between which it can switch with minimal penalty (a pipeline flush). In this work, we show that this dynamic platform can improve performance over current predictable processors (by a factor of 5 on average using the highest performing configuration), and provides schedulability that is on par with an earlier study that explored the concept of creating a dynamic processor based on a superscalar architecture. We measured a 15% improvement in schedulability over a heterogeneous multi-core platform with an equal number of datapaths. Datapaths that are not used by critical tasks can be assigned to non-critical tasks by adding a dynamic scheduling component. This allows the dynamic processor to assign up to 50% and on average 25% more resources to lower-priority threads during the execution of a static real-time schedule. Finally, our VHDL design and tools (including compiler, simulator, libraries etc.) are available for download for the academic community.

## 8

### 8.1. Introduction

In numerous real-time application domains such as automotive computing, emerging restrictions such as power limitations, cost, size, and maintainability push increasingly more tasks onto a single microcontroller. The timing properties of these tasks can vary in the degree of strictness, giving rise to the field of mixed-criticality systems [5]. Platforms executing these types of workloads have the following requirements:

- *Full predictability/analyzability* - It should be possible to perform Worst-Case Execution Time (WCET) analysis to extract tight bounds on the required computation time for each task.
- *Timing isolation* - Running tasks should not be affected by external influences, e.g., other tasks contending for resources, to ensure the timing validity of the system.

However, they also have the following desirable properties that are prone to contradict with above-mentioned requirements:

- *High-performance* - Tasks should not only be able to meet their own deadlines, but also to allow non-critical tasks to achieve a certain quality of service (for example, car entertainment media playback). Many architectural techniques to improve performance impede predictability and subsequently analyzability. Examples are caches, branch prediction, and out-of-order (OoO) processing.
- *Multicore/Multi-threaded* - Multiple tasks should be able to run concurrently, for reasons of performance and power (a single-core that is powerful enough to run all tasks will require more power than multiple cores with lower clock frequency and voltage), and improved schedulability (see Section 8.6). Multicore platforms often under-utilize processor resources, while multi-threaded platforms such as SMT processors fail to provide timing isolation [120].

In order to meet these requirements, numerous processor designs with predictable timing have been introduced. A recent example is FlexPRET [121], which is a fine-grained multi-threaded processor that provides full isolation and timing predictability. It is similar to a barrel processor but allows more flexibility in assigning cycles to threads in order to allow higher single-thread performance at the cost of adding forwarding paths (that need to distinguish the thread ID of instructions) to the design.

In this chapter, we propose to use the  $\rho$ -VEX polymorphic VLIW processor for mixed-criticality and real-time workloads. Its goal is to provide a high level of flexibility by being able to adapt to different workloads. We show that it has good properties in terms of performance and schedulability in this application domain. The runtime reconfigurable (polymorphic) version can choose to target programs with a high level of ILP in a high-performance single core 8-issue VLIW configuration, or multiple threads/programs in a multicore configuration with smaller issue widths. The key behind this runtime-adaptability is being able to split the processor's data paths into separate cores or combining them into a single larger core. When the processor is split, the independent datapaths provide full isolation from each other. The  $\rho$ -VEX provides multi-core processing capabilities without under-utilizing resources. The VLIW architecture, when used without caches, provides a high degree of time-predictability [122] as it offers static branch prediction (the compiler analyzes the most likely control flow and restructures the code accordingly), in-order execution and an exposed pipeline (all instruction latencies are fixed and known to the compiler - no pipeline interlocking or resource contention). Even when using caches, the  $\rho$ -VEX provides full performance isolation between tasks as the caches are split in the same fashion as the core, assuming that the backing bus interconnect provides isolation. Naturally, adding caches will reduce predictability and this chapter will therefore not evaluate cached setups. The designer can choose to use local memories and/or enable caches using VHDL generics. We evaluate the benefits of using this hardware platform for real-time workloads in terms of schedulability, throughput, and single-thread performance.

The contributions of this work are:

- We propose to use a polymorphic VLIW processor for real-time and mixed-criticality workloads.

- We discuss how to provide both temporal and spatial isolation.
- We perform an evaluation of the processor in terms of schedulability and performance using the Mälardalen real-time benchmark suite.
- We show that the number of randomly generated task sets that can be successfully scheduled on the processor is on par with an earlier study on a proposed dynamic superscalar architecture [26], and up to 15% higher compared to a heterogeneous multicore processor with an equal number of total datapaths.
- We perform an evaluation using the  $\rho$ -VEX proof-of-concept in terms of throughput using task graphs generated from the Mälardalen real-time benchmark suite.
- We show that the polymorphic VLIW is able to assign up to 50% more resources to non-critical tasks during execution of a static real-time schedule compared to heterogeneous processors with equal computational resources.

The remainder of this chapter is structured as follows. Section 8.2 introduces the execution platform and concepts necessary to understand the work. It also discusses the scheduling methodology that provides timing isolation between critical tasks while still being able to exploit processor polymorphism to increase schedulability. Section 8.3 presents a system architecture providing spatial and temporal isolation, and how to exploit processor polymorphism to dynamically assign execution resources to non-critical tasks when they are not utilized by critical tasks. Section 8.4 discusses how we use the scheduling methodology to create valid real-time schedules for the proposed platform. Section 8.6 presents the evaluation setup and discusses the results, Section 8.7 compares this work to existing literature and Section 8.8 concludes the chapter.

## 8.2. Background

This section briefly discusses concepts from earlier work that are needed to understand this work. First, we will introduce the dynamic processing platform used. Subsequently, we discuss the scheduling framework that we will use to schedule workloads for this dynamic processor.

### 8.2.1. Processing platform

In this work, we will use the  $\rho$ -VEX dynamically reconfigurable VLIW processor [21], that was implemented as a proof-of-concept for VLIW-based processor polymorphism. The main concepts of the processor will be discussed here as background for this work. The VEX ISA has been introduced by Fischer et al. in [16], as the academic sibling of the industrial st200 architectures that has been manufactured by STMicroelectronics. It is a family of architectures [31] that is parametrized regarding many design characteristics such as the number of registers and issue width. The  $\rho$ -VEX processor is a VLIW core that can contain multiple instances of the full

processor state (i.e., the general-purpose register file and control registers such as the program counter) creating 'virtual cores' called *contexts*. Between the 8 datapaths and the contexts, an interconnect is added that can be configured at run-time. When running in a single 8-issue mode, all datapaths are connected to one of the four contexts, and when running in  $4 \times 2$ -issue mode, each context is attached to a pair of datapaths (the instruction set architecture requires a minimum of two datapaths to support long immediates). A single pair of datapaths or multiples of these pairs can be re-assigned (i.e., reconfigured) to the contexts without the need to save/restore the contexts to/from main memory. Reconfiguring the interconnection can be performed within 9 cycles (4 cycles during which the new configuration is decoded and the core will continue running in the old configuration, 4 cycles to flush the pipeline and 1 cycle to start the new configuration). Datapaths that are unaffected by the reconfiguration command will continue running without stall cycles. A more in-depth discussion regarding the circuit complexity of this design and the benefits in terms of context switching and (reduced) interrupt latency can be found in [41].

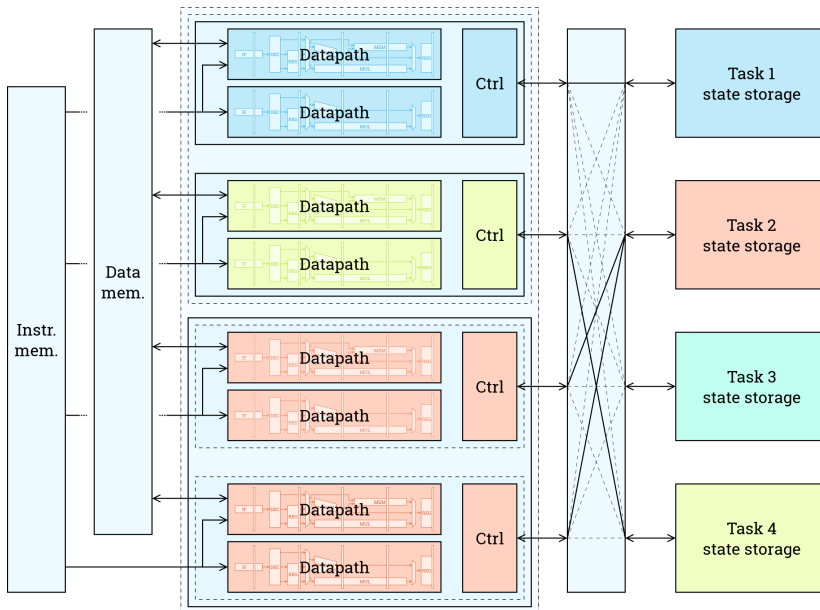
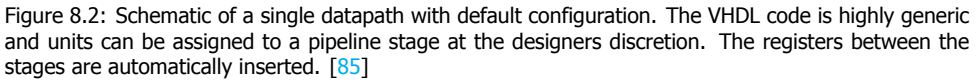


Figure 8.1: Schematic depiction of the concept behind the  $\rho$ -VEX polymorphism: multiple contexts can be connected to the datapaths in different fashions. In this configuration, Task 1 has been assigned a single pair of datapaths (a 2-issue VLIW), task 2 uses 2 pairs (forming a 4-issue VLIW), task 3 is inactive and task 4 has 1 pair of datapaths.

The core has an array of performance counters including cycle, operation, stall, and various cache-related counters. without stall cycles In order to provide high-precision timers, the size of these registers can be configured and are at most 56 bits (resulting in 10 days with single cycle accuracy at 80 MHz before the timer overflows). Using this width, they can be accessed by 2 ordinary load word instructions



The  $\rho$ -VEX pipeline has 4 stages by default (see Figure 8.2) and supports forwarding and variable-length VLIW instructions. It is also highly configurable at design time using VHDL generics, can be used with or without caches and is synthesizable to ASIC and FPGA targets. It supports up to 8 contexts. Using the default configuration, it can be synthesized at 80 MHz on the Xilinx VC707 evaluation board.

In [26], modifications to an Alpha 21164 processor are proposed to create a dynamically partitionable processor that can run 1 thread in 4-issue in-order superscalar mode, 4 threads in scalar mode, or a combination. The goal for this design is to be able to provide high performance for single threads but also analyzability and timing isolation between threads. Although this work is not directly comparable, being a proposed design without hardware implementation and also targeting the high-performance instead of embedded domain, it does provide us with a very useful scheduling methodology for our dynamic processor. It will be introduced here briefly.

The scheduling framework provides a way to create static schedules for a dynamic processor that supports high frequency reconfigurations. The problem with creating schedules for these processors is that 1) each task in the task set has its own period, so the hyper-period of the task set can become very large, and 2) the processor can be reconfigured at any time, resulting in an infeasible search space when combined with the length of the hyper period. In addition to this (although this problem is not discussed by [26]), a program can have different phases over the course of its execution, in which the Instruction-Level Parallelism (ILP) varies [4]. Depending on the current ILP of a program, changing the issue-width of the processor can have different effects on the performance (see Figure 8.3). Because of this, a WCET measurement (see Table 8.2) for a certain issue width is only valid if

the issue-width is not changed while the program is running. Extrapolating the performance between 2 issue widths (executing half of the program in 1 mode and the other half in another mode, and taking the mean of the WCET of the 2 modes) only works if the program is assumed to have uniform ILP (e.g., every VLIW instruction must have the same number of operations which is very unrealistic).

The scheduling methodology works by dividing the scheduling timeline into rounds of fixed length. The round length is a designer's choice, but should be short enough so that it fits a reasonable amount of times into each task's periods and long enough to spread the core's adaptation penalty. The execution of each task is spread in time, evenly over the rounds. Each task gets a fraction of each round equal to  $\frac{WCET}{period}$ . At the end of the task's period, it will have executed  $\frac{period}{roundlength}$  rounds. The number of cycles that have been assigned to a task is  $roundlength \times \frac{WCET}{period}$  (the number of assigned cycles per round)  $\times \frac{period}{roundlength}$  (the number of executed rounds during the tasks period). As this is equal to the WCET, it guarantees the validity of the schedule. In this fashion, we have spread out the executions of all tasks in the task set over their entire period resulting in a common sub-period (the round). Therefore, instead of having to analyze the entire hyper-period, we only have to create a schedule for a round and repeat it indefinitely.

In summary, creating a schedule for a workload corresponds to creating a valid schedule for a single round, resulting in a small search space for reconfigurations. This scheduling method ensures that reconfigurations always occur at the same time within a round and always coincide with a task switch. This way, every task always runs with a constant issue-width during its entire execution. From each

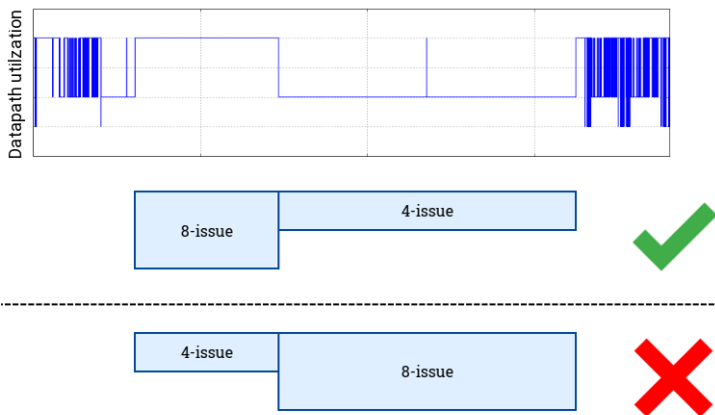


Figure 8.3: Depicted on top is the phase behavior of a MP3 encoder (LAME) during the course of its execution. As can be seen, the first stable phase requires an 8-issue configuration to achieve its maximum performance. the second stable phase can be executed on a 4-issue configuration without incurring a penalty. Running the first stable phase in 4-issue mode will result in a longer execution time that cannot be compensated by executing the second phase in 8-issue (because there is not enough ILP), and will therefore overshoot the WCET. However, this information is not known during execution. This is why we must ensure a task is always executed in the configuration that was used during scheduling.



task's point of view, it is always running in the same issue width despite of the reconfigurations. Because of this, the WCET for a task using that particular issue width is valid. It is the main point of how the scheduling method allows us to use a dynamic platform to schedule real-time workloads.

### 8.3. System architecture for Mixed-criticality systems

In this section, we describe how we use the adaptable processor combined with the scheduling methodology discussed in Section 8.2 to create a system architecture that provides temporal and spatial isolation for critical tasks and is able to provide high throughput for non-critical tasks.

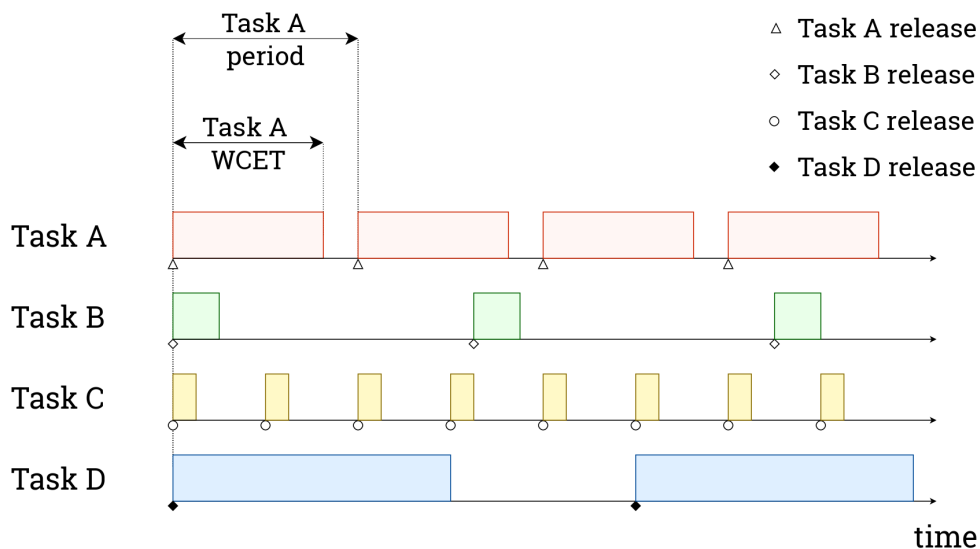
#### 8.3.1. Spatial isolation

In order to guarantee that critical tasks execute correctly, their full state needs to be protected from being (harmfully) modified by external factors, such as a faulty or malicious task. The state of a task consists of all the values of the internal processor registers and the memory in use by the task. The first part of this state, the processor registers, is the least challenging to protect, as it only needs to be accessible from the task itself. Its integrity is therefore guaranteed as long as the instructions read from the memory are valid. Memory isolation is more difficult, as every task is able to access memory, and a limited form of inter-task communication through memory may be required by the application. However, without protection mechanisms in place, any task could overwrite the instruction or data memory of any other task.

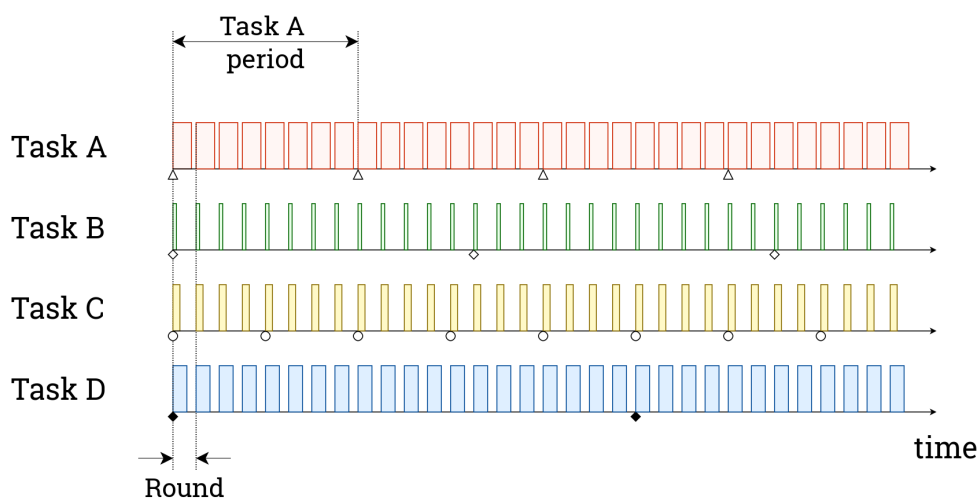
In our proposed platform, memory protection is implemented using two concepts. Firstly, by using local (scratchpad) memories instead of globally shared memory, and assigning a local memory to a context. Only that context is able to access this memory, as it is memory mapped only into the address space of that context. This is implemented by using the context id for multiplexer selection, instead of address decoders. Synchronization and communication between threads can be performed via global shared memory that is memory-mapped into every context's address space.

Additionally, we propose to be able to assign multiple software tasks to a hardware context, so that the platform facilitates running more threads than there are hardware contexts (in case of the  $\rho$ -VEX, there are four). In this case, one hardware context needs to divide its time between running different tasks. This means that either 1) there must be more local memories than contexts (keeping the 1:1 mapping between tasks and memories) or 2) that the local memory assigned to this context needs to store multiple program memory states and provide isolation between them (keeping the 1:1 mapping between hardware contexts and memories).

The first solution can be implemented by adding a processor register that contains the current task id, which can be written only by privileged code (e.g., the context switch routine). Its value will be used for multiplexing between scratchpad



(a) Task set with WCETs and periods.



(b) Task set spread over rounds.

Figure 8.4: Figure (a) presents an example task set, Figure (b) shows this set after the tasks have been equally spread over rounds.

memories. The second solution can be implemented by adding a processor control register whose value masks the accessible memory region (similar to what is used by [121], although they propose to use a low and high address register that both need a carry chain to perform the comparison, which is disadvantageous for timing). This register needs to be set by privileged code during task switches as well. Note that 1) the additional control register needs to be created for each context (virtual processor), and 2) the second solution can also be used in combination with a single globally shared memory instead of local memories for each context. In this case, however, contexts need to compete for access, requiring a multiple-access scheme such as round robin. This scheme needs to divide access to the memory ports, while also guaranteeing temporal isolation. This will be discussed in the following section.

### 8.3.2. Temporal isolation

In addition to providing memory protection for critical tasks, the system needs to guarantee that tasks cannot be interfered with regard to timing. Interference here means that a task must wait for a certain resource to become available, because it is in use by another task. These resources can be execution resources such as functional units in the processor pipeline, access to a bus, or a memory port. Generally, there are multiple ways to deal with interference; for example, by arbitrating between requests in a time-predictable way (using bounded delay or fixed time sharing), and by distributing resources over tasks in such a way that interference cannot occur altogether. Our proposed platform takes the latter approach. The most rigorous example of this is to run each task on a separate microcontroller. This distribution, when static, inherently impedes resource utilization. We propose to make the distribution dynamic, aiming to maintaining the time-predictable interference-free properties without limiting resource utilization.

By using a polymorphic processor, datapaths can be assigned to threads dynamically. They can be assigned to a single task, maximizing resource utilization. When assigned to separate threads, the distribution is fully separate, so that each thread can execute without any interference from the others. The memory layout discussed in Section 8.3.1 does not only provide spatial isolation, but also temporal isolation as each context can access its storage area with single-cycle access. Using local memories improves WCET analysis considerably, compared to a system using caches. Memory regions that are globally shared will use a round robin or time division access control protocol, so that all threads have bound latency on code sections performing communication or synchronization between different tasks. This way, busses, DDR memory and other shared resources can be used while still providing timing isolation (see for example [123]). Non-critical tasks can operate from main memory to conserve storage capacity in the local memories, if these tasks do not need to level of strictness regarding timing.

### 8.3.3. Assigning unallocated cycles to non-critical tasks

By exploiting the dynamic resource distribution, execution resources that are not required by critical tasks can be assigned to non-critical tasks in an efficient way.

This can lead to increased throughput for these tasks, while still guaranteeing the real-time schedule. This is what makes this platform especially suitable for mixed-criticality systems, where a system runs critical tasks as well as non-critical tasks that can have a soft real-time requirement or a performance requirement. An often used example is an Unmanned Aerial Vehicle (UAV) that has a critical control loop maintaining stability of the aircraft, combined with communications and media encoding tasks that have some tolerance for delay, a dropped frame or a decrease in video quality.

Soft real-time tasks can be added into the schedule in the same fashion as hard real-time tasks. Best effort tasks can be added after creating the schedule (slack scheduling) as there will usually be much more free cycles because the typical execution time will often be lower than the worst case, especially if the execution time depends on the input). The  $\rho$ -VEX is also able to utilize these cycles as follows. As soon as a task finishes, it will request the scheduler to give its resources to one of the non-critical tasks as depicted in Figure 8.5. Naturally, the scheduler must ensure that tasks cannot take away resources from other tasks. From that moment, the portion of resources that was reserved for that task in each round will be given to the other task. When the original task is released again, the interrupt routine will restore the original schedule, thereby again guaranteeing the task's processing time. This means that the critical task is still fully isolated.

Depending on how much resources are left and how many threads are active, the core can run tasks in either high-performance 8-issue mode or high-throughput multicore mode, unlike other analyzable processors such as [121] that is limited to scalar execution. This is one of the key points of the  $\rho$ -VEX processor - it is able to provide timing isolation, in addition to exploiting its dynamic properties to adapt to the characteristics of the workload (Instruction-level or Thread-level parallelism as discussed in [85]). When using a heterogeneous multicore system, a task can be migrated from a little core to a big core when it becomes available to increase resource utilization as well (depicted in Figure 8.5). However, when the critical task is triggered that is normally assigned to that large core, the core must first save the state of the non-critical task before the critical task can resume. This penalty must therefore be added to the WCET. As long as there are enough hardware contexts available, this is not required by the polymorphic core.

## 8.4. Scheduling approach

This section discusses how the scheduling methodology presented in Section 8.2.2 is used to create valid static schedules for the proposed platform. We then propose two ways in which performance of one of the tasks can be improved beyond merely meeting its deadline in the worst case.

### 8.4.1. Worst-case schedule creation

To create a valid schedule for a single round as per the methodology described in Section 8.2.2, the scheduler needs to assign the required number of cycles and amount of compute resources (datapaths) to each task, drawing from a pool of

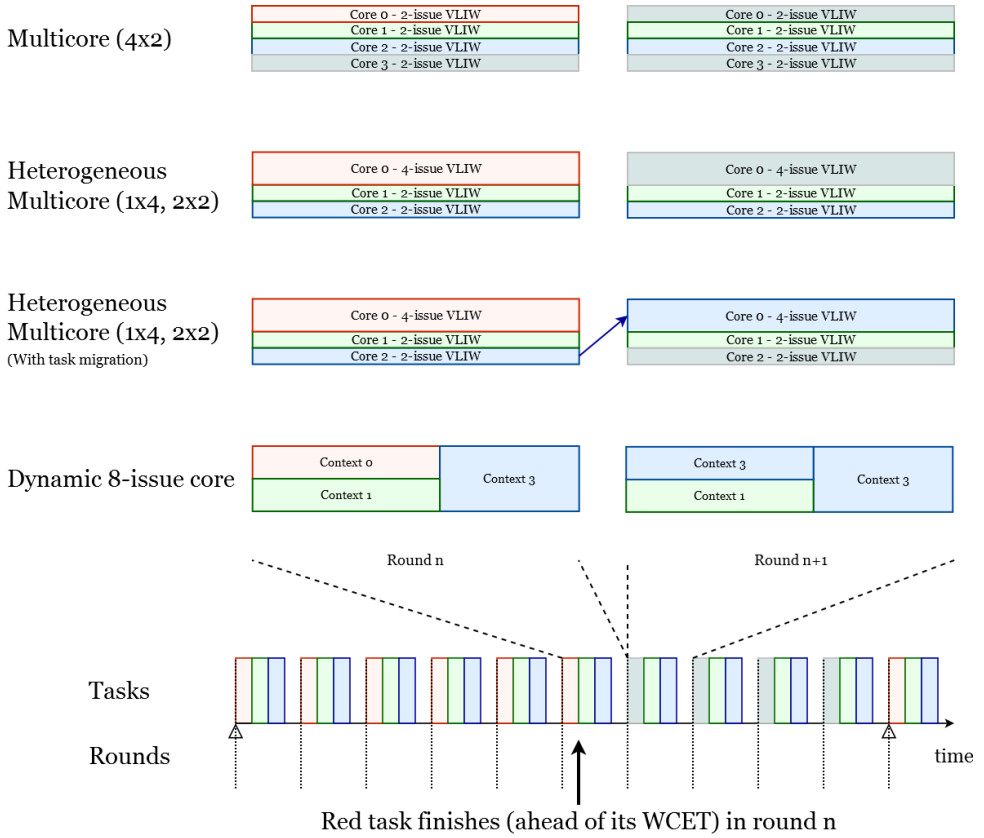


Figure 8.5: A Hard Real-Time Task (red) giving its resources to a Soft Real-Time Task when it has finished. When the original task is triggered again, it will take its resources back. Static multicore systems are not able to utilize all resources. A heterogeneous multicore can achieve a higher utilization by migrating a task to a larger core when it becomes available. However, this needs state saving and restoring (not needed by the dynamic core).

available resources. To solve this, our scheduler uses a 2-dimensional binpacking algorithm, where the number of cycles is one dimension (X) and the number of datapaths the second (Y).

The figures and examples in this chapter use a round length of 200 cycles, because a number of the benchmarks from benchmarks have runtimes starting from 2000 cycles (see Table 8.2). In our experimental evaluation, we will use longer round lengths to decrease reconfiguration and task save/restore overhead.

The amount of available datapaths depends on the processor. In case of the  $\rho$ -VEX, it is a design-time parameter and can be 2, 4, or 8. The polymorphic core has 8 datapaths by default. For our evaluations, we will compare an 8-issue polymorphic core to a number of static (fixed) core configurations, each having the same total amount of datapaths. These fixed configurations are 4x 2-issue, 1x4 + 2x2-issue, 2x4-issue, and 1x8-issue.

The fixed configurations differ in terms of schedulability. For example, if a task has a period between the 4 and 8-issue WCET, the 1x8-issue platform will be able to schedule it, while the other platforms cannot provide sufficient single-thread performance. Conversely, a task set with four tasks that each have a period that is slightly longer than the respective 2-issue WCET is schedulable on the 4x2-issue platform, but will most likely not be on the other platforms due to ILP limitations of the tasks.

In theory, the polymorphic platform is capable of scheduling all task sets that are schedulable on at least one of the static platforms. This is because the polymorphic core can ‘mimic’ them. Furthermore, there are also task sets that are only schedulable on the polymorphic core; see Figure 8.6 for an example. Therefore, the set of task sets that are schedulable on the dynamic core is a superset of the set of task sets that are schedulable on any of the evaluated static platforms. However, in practice, there is a small fraction of task sets where the current dynamic platform scheduler fails, while one of the static platform schedulers succeeds. This is due to the heuristic nature of the binpacking algorithm used in the evaluation process.

Our solution to the 2-D binpacking problem is implemented as follows. For each task in the task set, a list of two-tuples is created. This list contains an entry for issue widths of 2, 4, and 8; the two-tuples represent the issue width and the corresponding WCET divided by the task period. Only issue widths for which the WCET is smaller than or equal to the task period are included. The tasks are now sorted by area, a common pre-heuristic for binpacking algorithms. The area is defined as follows:  $\min_{width} \left( \frac{WCET_{width}}{period} \cdot width \right)$ . Before running the binpacking algorithm, a feasibility check is performed by simply testing whether the sum of all the areas is less than or equal to the total computational resources. If a task set is not feasible, it is ignored. The tasks are now packed one by one, by descending area. Packing is first attempted using the narrowest core (in case of the 1x4, 2x2 heterogeneous platform) and using the narrowest issue width compatible with that core. The packing algorithm utilized is bottom-left first (BLF) [124]. If packing fails, wider compatible issue widths are attempted first. If all possible run configurations on the narrowest core in a multi-core system fail, packing is attempted on the next core. If packing fails on all cores in the platform, the task set is considered to not be schedulable on that platform.

The output of the scheduler is a time-varying mapping from datapath to context/static core and from context/static core to task. Such a schedule is illustrated in Table 8.1.

Activation cycle	Context to datapath	Context to task
0	0 → 4..7	0 → A
	1 → 2..3	1 → B
	2 → 0..1	2 → C
60	0 → 4..7	0 → A
	3 → 0..3	3 → D

Table 8.1: Schedule table corresponding to the example in Figure 8.6.

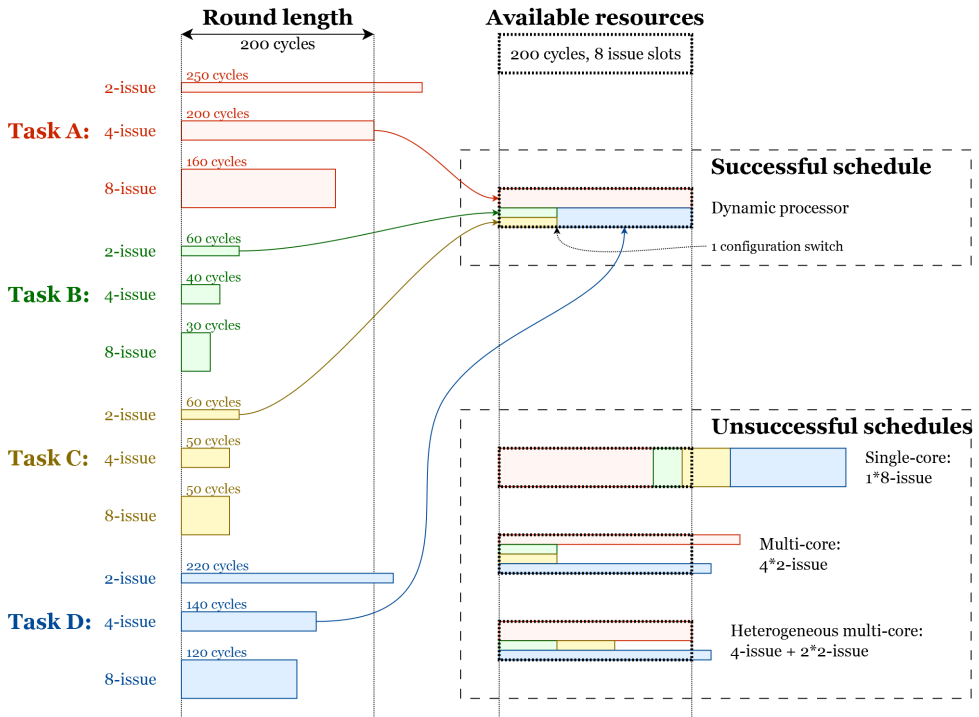


Figure 8.6: Example of how to create a valid schedule for a dynamic processor.

### 8.4.2. Improving average-case performance

Typically, the worst-case schedule of a task set created using the methods described above does not utilize all datapaths all the time. This would only happen if the task set were perfectly matched to the processing system. More likely, there will be unused resources, even in the worst-case. These resources can be used to accelerate tasks that benefit from increased performance, beyond what they need to meet their deadline. For instance, a compression algorithm may have a WCET for giving its first output, but may be able to successively refine the quality of the output when given more computational resources.

In this work, we select one task from the task-graph to give the unused resources to. We refer to this task as the non-critical task, within the context of a mixed-criticality system where a number of critical tasks have deadlines and the non-critical task has a minimum performance requirement, but it could also be a critical task that benefits from additional performance. We propose two methods for doing this. The first method simply assigns the unused processor time in the worst-case schedule to the non-critical task. The second method uses runtime information to also use processor time from the critical tasks, in case the actual runtime of these critical tasks is less than the worst case. We refer to the first method as the static scheduler, as it does not use runtime information, and to the second method as the dynamic scheduler.

Furthermore, we propose that the processing system is automatically reconfigured by a hardware circuit. This prevents a large amount of context save and restore time needed for a software solution based on timer interrupts. The proposed circuit for the static scheduling method consists of a timer that resets at the end of each round, a local memory that contains the schedule in the form of timestamp to configuration mappings, and a simple state machine. The state machine reads the schedule entry from the local memory and waits until the round timer reaches the schedule entry timestamp. When it does, it triggers a reconfiguration based on the datapath to context mapping (only for the polymorphic core), and writes the context to task mapping to the 'requested soft context' (RSC) control register of each  $\rho$ -VEX context (for both the polymorphic and static processing systems). The state machine then reads the next schedule entry, or resets itself if it reached the end of the list.

A write to the RSC register of the  $\rho$ -VEX triggers an interrupt if the new value does not match the value in the 'current soft context' (CSC) register. The handler for this trap then saves the state of the current task to memory, sets CSC to RSC, and then loads the requested task from memory. This prevents the need for DMA-like hardware.

The dynamic scheduling method requires additional hardware. Specifically, we propose to give each critical task a 1-bit 'yield' register. This register indicates whether the critical task currently needs to execute or whether it has met its deadline already, 'yielding' to the non-critical task. This flag may be reset by a timer interrupt at the start of the period of the respective task, and set when the task completes.

When a reconfiguration and/or set of context switches is performed, the hardware must read these yield registers to determine which of the tasks that are to be mapped to the processor in the worst-case schedule have already met their deadlines. It can then assign as many datapaths to the non-critical task as possible without negatively affecting the execution of remaining critical tasks.

## 8.5. Experimental setup

For the measurements, we are using a cycle-level architectural simulator for flexibility, and base our results on an FPGA prototype of the  $\rho$ -VEX processor clocked at 80 MHz running on a Xilinx VC707 development board. We will use 23 programs from the Mälardalen benchmark suite [125]. Execution times, listed in Table 8.2, were measured on a single core for each of the 3 possible configurations, in a platform without caches and using single-cycle memories (implemented by FPGA RAM blocks). These execution times are assumed to be the worst-case execution times, as they are always executed using the same input. However, standard WCET analysis or measurement techniques can be applied [122].

The programs were compiled using our port of the Open64 compiler using optimization level 3. Programs were run bare-metal, with our port of the newlib embedded standard C library. Our UART driver was modified so it did not wait when the output buffer is full, in order to remove the influence of serial output. Output is written into a reserved memory region so that it can still be examined



Benchmark	Worst-Case Execution Time (cycles)		
	2-issue	4-issue	8-issue
adpcm	350009	315107	309206
cnt	3626	2937	2664
compress	5236	4479	4241
cover	2210	2006	1815
crc	17879	14985	14667
edn	23184	18581	17505
expint	10800	9926	9512
fft1	50389	34061	26614
fir	183221	139559	129815
lms	4986376	3372138	2709911
ludcmp	55388	41112	34928
matmult	93211	85882	84649
minver	19289	13406	10844
ndes	31120	26502	24457
ns	5212	4253	4116
nsichneu	6357	6330	6316
prime	25788	23170	23158
qsort-exam	2995	2241	1922
qurt	21211	14494	11591
sqrt	19587	13877	11574
st	3631939	2313149	1731382
ud	11579	10720	10420
whet	29519872	18704428	13694591

Table 8.2: WCETs of the benchmarks for the possible processor configurations (2-issue, 4-issue, 8-issue).

after execution.

In order to measure the ‘schedulability’ (how many task sets can be successfully scheduled) of the processor and compare to [26], we have implemented a program that mimics the task set generator according to their description. We will briefly describe it here for clarity. We will generate task sets consisting of 4 tasks randomly selected from the benchmark set. For each of the tasks, a period is randomly chosen using the following constraints:  $WCET_{8issue} \leq period < (1.5 \times n_{tasks}) \times WCET_{2issue}$ . These boundaries guarantee any single task to be schedulable on a single 8-issue core (highest performing processor in our design space), and that a sufficient number of schedules are generated that are schedulable on a single 2-issue core (lowest performing processor in our design space).

The task set are divided into 4 bins of varying ‘difficulty’; every set is categorized in terms of total 2-issue utilization (the sum of the 2-issue utilization of each task in the set;  $\sum_{task} \frac{WCET_{2task}}{period_{task}}$ ). Bin 0 contains the lightest task set with a total utilization of 0 - 1, and bin 3 has the most difficult task sets with highest utilizations (between 3 and 4). Each task set is randomly generated and assigned to its corresponding bin until each bin has 2500 task sets. In [26], the bins consist of 25 task sets each. However, using that size, we found the results to vary significantly between runs. Therefore, we increased it to get more consistent outcomes. In addition to comparing to [26], we also generate task sets consisting of 8, 12 and 16 tasks that are randomly selected from the benchmark set.

Resource utilization was evaluated as follows. For each generated task set, one task was selected to operate as a non-critical task. This task is repeated continuously. During the execution of the task graph, the execution time of the instances of all the other (critical) tasks in the graph are randomly varied between  $0.5 \times WCET$  and  $WCET$ , to mimic the behavior of tasks with different inputs that have influence on the actual execution time. We have measured the resource assignment (number of cycles  $\times$  datapaths) and throughput (committed number of operations) for the non-critical task in a cycle-accurate simulation model of the processor executing the task graphs, using per-cycle execution traces of the benchmarks. We will evaluate 2 algorithms; one static algorithm that only assigns as much resources to the non-critical task as possible during the creation of the static real-time schedule, and a dynamic algorithm that re-assigns resources using the runtime scheduler as described in Section 8.4.2.

We will consider a number of platforms in our evaluation, all having equal aggregate resources (i.e., 8 datapaths in total), to demonstrate the effectiveness of the dynamic processor in utilizing these resources. The exception is the 1x2-issue platform, to match the “scalar processor” in [26].

## 8.6. Results

This section presents the measurement results in three evaluation metrics: schedulability, performance, and resource utilization.

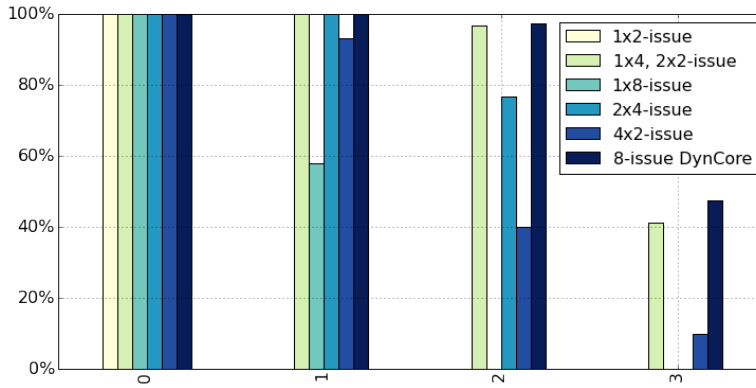


Figure 8.7: Plot of the number of successful schedulings for each of the evaluated hardware platforms. On the x-axis are the 4 bins with increasing total utilization. Results are relative to the number of feasible task sets in the bins (infeasible task sets are ignored).

### 8.6.1. Schedulability

Figure 8.7 plots the number of task sets that can be successfully scheduled on the different platforms. The four groups on the x-axis each list the results for a certain schedule bin, from 0 (task sets with lowest total utilization) to 3 (task sets with highest total utilization). Every result is relative to the number of feasible schedules in the bin (which is plotted in Figure 8.8).

A single 2-issue core can, by definition, only schedule tasks from bin 0 (total 2-issue utilization must be  $< 1$ ). This can be clearly seen in the graph, where all other bins have 0 successful schedules for that platform. As the difficulty increases, the advantage of using the dynamic processor becomes clear; in bin 2 it is able to schedule 97% of the feasible schedules and in bin 3 50%. The homogeneous multi-core platforms can schedule considerably smaller numbers of task sets, with the 4x2-issue being able to accommodate only 10% of the task sets from bin 3. These sets consist of tasks with periods that are close to, but not shorter than, the 2-issue execution time. If there is a single task in the set that requires a larger core to meet the deadline, this platform cannot schedule it. The 2x4-issue platform showing 0 successful schedules in bin 3 is due to the often small difference between 2-issue and 4-issue execution times (see Table 8.2). For this platform, running the programs in 4-issue mode is the only choice, resulting in more total 'area' utilization for a task, even if it does not need the additional performance. The same applies to the 1x8-issue platform, but the effect is even more pronounced. The heterogeneous 1x4,2x2-issue platform provides a very adequate schedulability, with the dynamic processor beating it by only 15%. This means that, if the higher single-thread performance that the 8-issue dynamic core can deliver is not needed, a heterogeneous platform is a good alternative for designers to consider.

The number of feasible task sets per bin is plotted in Figure 8.8. It drops as the difficulty of the task set bin increases to 80% for bin 2 and 20% for bin 3. Of these 500 feasible task sets in bin 3, the dynamic core is able to schedule around

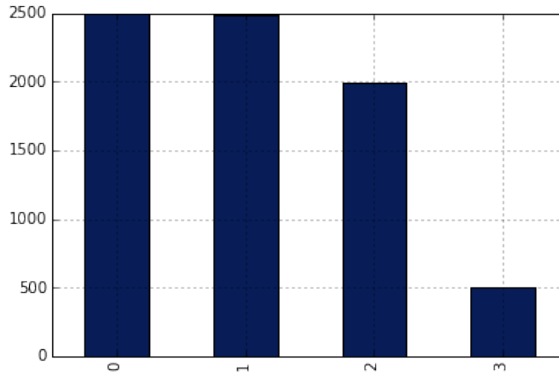


Figure 8.8: Plot of the number of feasible task sets for each of the task set bins.

50%, which could indicate that 1) there is room for improvement in the scheduling framework and binpacking algorithm, or 2) the requirement that every individual task must always run using a constant issue-width (see Section 8.2) could be a limiting factor.

Comparing to the results from [26], plotted in Figure 8.9, we see a similar curve over the task set bins, but our scheduler performs somewhat better for the dynamic processor. In bin 3, it can schedule almost twice the number of task sets at 50% vs. 28%. An equivalent of the heterogeneous 1x4, 2x2 platform has not been evaluated so we cannot make a comparison.

Figure 8.10 shows a different representation of the schedulability results, plotting the fraction of successful schedulings in relation to the total system utilization. The lower graph counts 10 cycles penalty for a reconfiguration (latency pipeline flush) and 150 cycles penalty for a migration. In both cases, the polymorphic processor (DynCore) provides a clear advantage in schedulability over the other platforms. At high utilization, the 4x2-issue platform is able to schedule a relatively large portion of the task graphs and performs equally well as the dynamic core. Here, the dynamic core will run in 4x2 configuration the majority of the time, not benefiting from adaptations. At lower utilizations, the dynamic core is able to schedule some tasks that require higher performance, thereby being able to schedule more task graphs than the 4x2 platform. Similarly, the 1x4, 2x2 platform is able to keep up with the dynamic core up to some point, where the dynamic core can benefit from being able to run an additional task in parallel.

### 8.6.2. Performance & area utilization

To evaluate the performance of the  $\rho$ -VEX processor, a comparison was performed with a 32-bit RISC-V processor on which the FlexPRET time-predictable processor [121] is based. As we are only measuring performance, the timing extensions are not needed. Measurements are based on cycle counts from the spike simulator executing the RV32I instruction set. All benchmarks are compiled using optimization level 3. The speedups are calculated assuming a target clock frequency of 80MHz

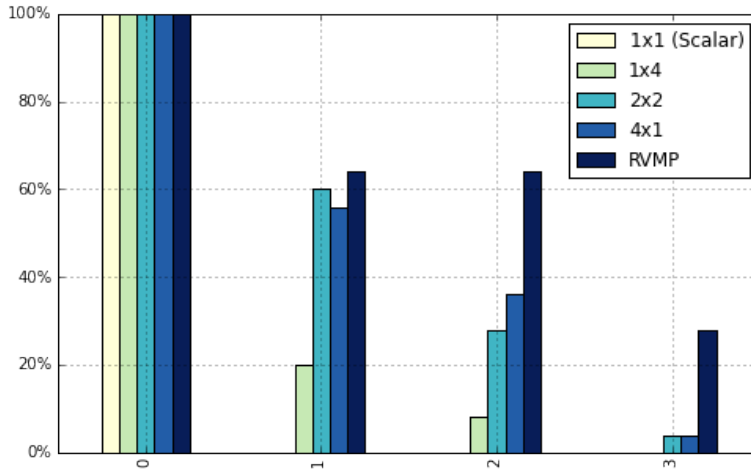


Figure 8.9: Schedulability results from [26], to be compared with Figure 8.7. RVMP (Real-time Virtual MultiProcessor) should be compared to dyncore.

for both processors. It must be noted, though, that the RISC-V will likely be able to achieve higher clock frequencies. Also note that there are 3 benchmarks that have been removed from the results (fft1, ns and sqrt). This is because the RISC-V compiler completely optimized them away, resulting in a program that only returns the answer.

When weighted to execution time, the highest performing 8-issue  $\rho$ -VEX is  $4.69\times$  faster than the RISC-V. As this benchmark suite is being dominated by the execution time of *whet*, we also report the non-weighted average of  $5.54\times$  speedup. Table 8.4 lists the FPGA utilization for the two processors when prototyped on a FPGA. The utilization of Block RAMs is relatively high in case of the  $\rho$ -VEX, because the multi-ported register file implementation requires duplicated storage combined with a Live Value Table [108]. Compared to the FlexPRET, the  $\rho$ -VEX utilizes approximately 5 times more resources. As can be seen in Table 8.3, this is similar to the increase in performance. This compares quite favorably as single-thread performance normally does not scale linearly with area utilization.

The size of the  $\rho$ -VEX, however, is a factor that designers will need to consider as an 8-issue VLIW will be overkill for many application scenarios. However, for some domains such as media or digital signal processing, VLIWs are known to provide significant performance gains over scalar RISC processors, therefore, in these cases the  $\rho$ -VEX is a suitable platform.

### 8.6.3. Resource utilization and throughput

In Figure 8.11, resource utilization and throughput for the non-critical task is depicted. The resource utilization (shown in the top graph) is calculated from the maximum possible number of resources (8 datapaths) while executing 500 rounds of 20.000 cycles. When total utilization of the task graph is low, more resources

Benchmark	RISC-V	Speedup		
		2-issue	4-issue	8-issue
adpcm	1732860	4.95	5.50	5.60
cnt	9554	2.63	3.25	3.59
compress	6917	1.32	1.54	1.63
cover	1808	0.82	0.90	1.00
crc	21633	1.21	1.44	1.47
edn	818203	35.29	44.03	46.74
expint	6726	0.62	0.68	0.71
fir	956526	5.22	6.85	7.37
lms	19926799	4.00	5.91	7.35
ludcmp	194575	3.51	4.73	5.57
matmult	682965	7.33	7.95	8.07
minver	43312	2.25	3.23	3.99
ndes	32785	1.05	1.24	1.34
nsichneu	4260	0.67	0.67	0.67
prime	74616	2.89	3.22	3.22
qsort-exam	10283	3.43	4.59	5.35
qurt	92343	4.35	6.37	7.97
st	15210054	4.19	6.58	8.78
ud	4120	0.36	0.38	0.40
whet	49287174	1.67	2.64	3.60
Weighted avg. speedup	N.A.	2.27	3.52	4.69
Avg. speedup	N.A.	3.95	4.99	5.54

Table 8.3: Performance - RISC-V (RV32I) vs  $\rho$ -VEX.

can be assigned to a single thread, which is why the graph shows a decreasing slope. The platforms that have a 4-issue as highest performing core cannot assign more than 50% of its resources (4 datapaths) to a single thread, and for the 4x2 platform this is 25% (2 datapaths). The results from Figure 8.10 can be recognized as some platforms are not able to schedule task graph above a certain utilization (prematurely ending their graph here). On all platforms, the dynamic algorithm is able to assign significantly larger fractions of the resources to the non-critical task, which is to be expected as the difference between WCET and actual execution time

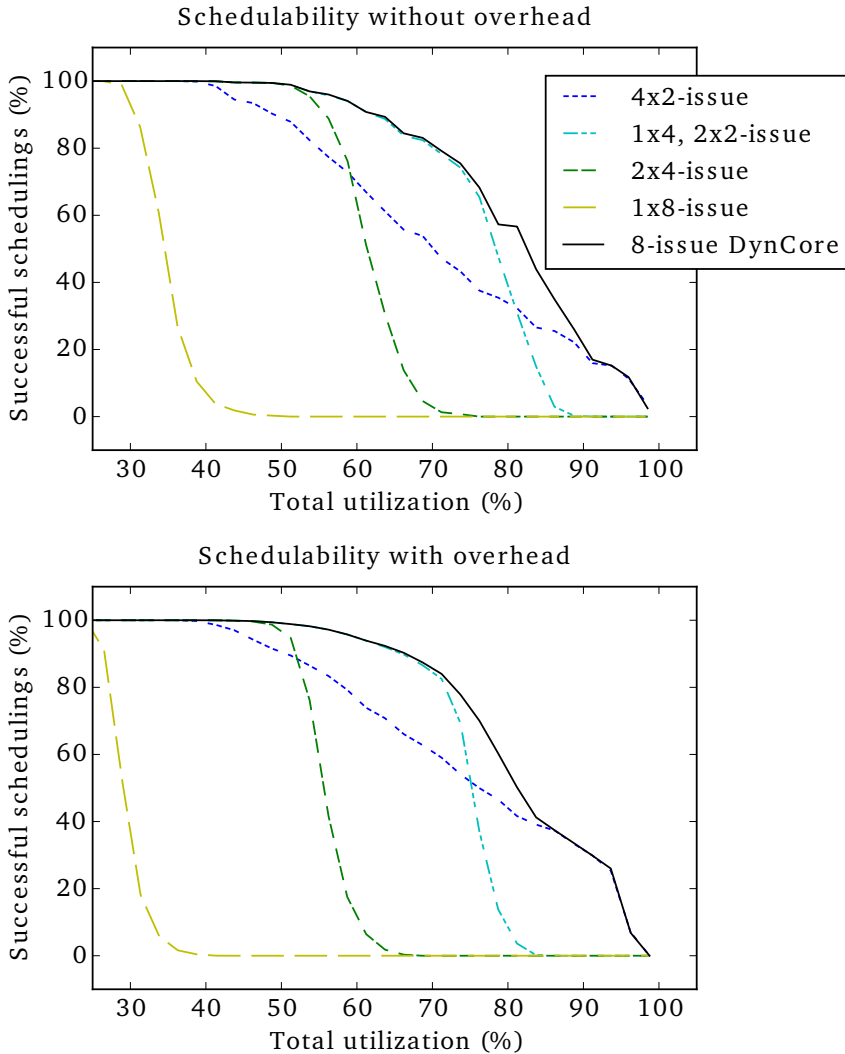


Figure 8.10: Schedulability plotted in relation to total system utilization.

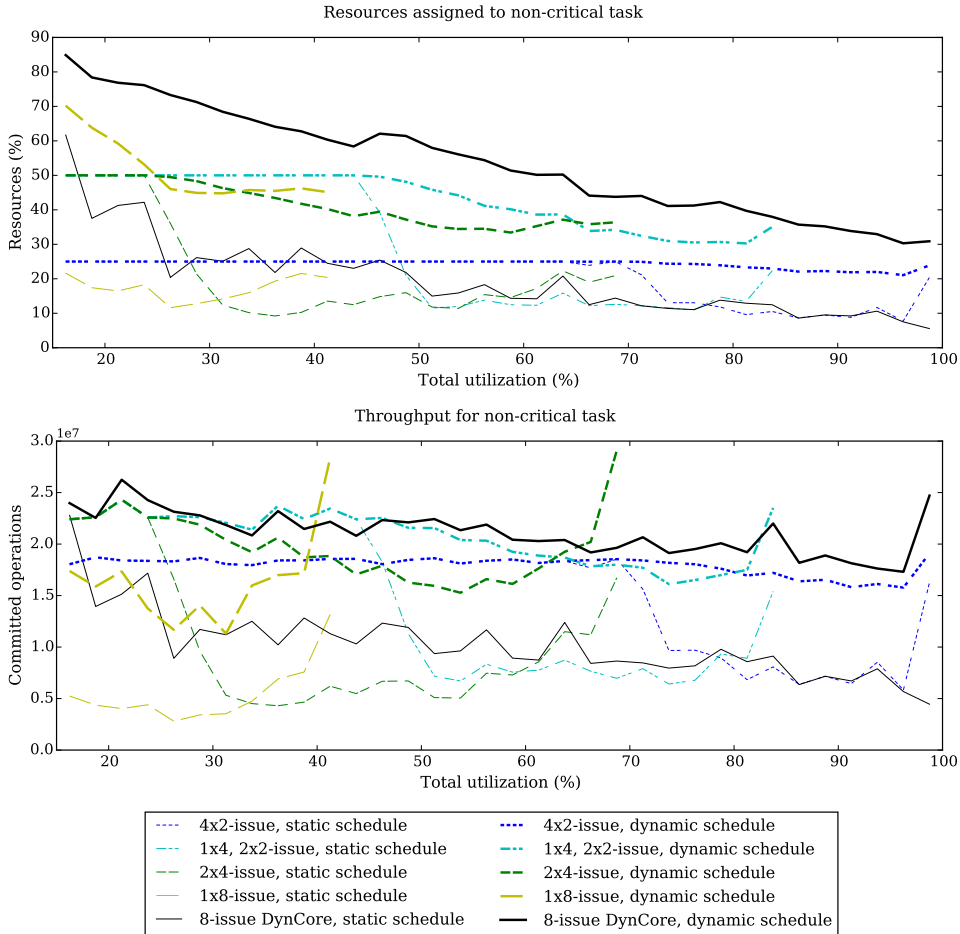


Figure 8.11: Resource assignment (top) and resulting throughput (bottom) for the non-critical task plotted in relation to total system utilization. Dynamically re-assigning resources leads to significantly better results. The polymorphic core is able to assign up to 50% and on average 25% more resources to the non-critical task compared to the second best performing platform.



	$\rho$ -VEX (4-threads)	FlexPRET (4-threads)	Increase (factor)
Slice Registers	8529	2687	4.72
Slice LUTs	31839	5661	5.6
BRAMs	128	Not reported	N.A.

Table 8.4: Resource usage of  $\rho$ -VEX vs. the FlexPRET timing-analyzable multi-threaded processor

for all the task instances cannot be exploited using the static algorithm.

Comparing the platforms, the dynamic core is at a clear advantage. It assigns up to 50% and on average 25% more resources to the non-critical task compared to the second best performing platform, the heterogeneous platform with one 4-issue and two 2-issue cores. Interestingly, when looking at the results in the lower graph, this increase in computational resources results in very modest throughput increases. This is because 1) we are using a single task, thereby limiting the processor to exploiting ILP (and preventing it from exploiting TLP) and 2) the tasks from the benchmarks suite do not have significant ILP. Tasks that typically run in the background on a VLIW will often be signal or media processing applications that will offer more inherent parallelism. Concluding, the top graph in Figure 8.11 provides an upper bound and the lower graph provides a lower bound on the expected increase in throughput by using the polymorphic processor. The seemingly anomalous spikes at the end of the throughput graphs for some of the platforms can be explained by the observations that, at these high utilization, only a small fraction of the task graphs can actually be scheduled on that particular platform (decreasing the dampening effect of averaging), and these graphs will likely contain one or more tasks with relatively high ILP.

By averaging the resource utilization for the non-critical task over all the task graph types (with all different numbers of tasks per graph and utilization levels), we obtain the graph depicted in Figure 8.12. It shows how many datapaths (lanes) each platform is able to assign by using either algorithm. Note that some platforms do not have datapoints for all utilization levels. The 8-issue platform has a high average in this graph because it can only schedule task graphs with a very low utilization (see Figure 8.10 and 8.11). Still, it provides a clear overview of the advantages of re-assigning unused cycles dynamically, and using the polymorphic processor (DynCore).

## 8.7. Related work

This work discusses (static real-time) schedulability, multi-threaded architectures, and time-predictable processors. In [126] and [112], predictability and schedulability is discussed. Examples of processors with multiple contexts/threads for the purpose of real-time systems are [106] [118]. In [114], performance comparisons are made between increasing the number of cores and increasing the number of register sets. A related VLIW architecture that has multiple hardware contexts is

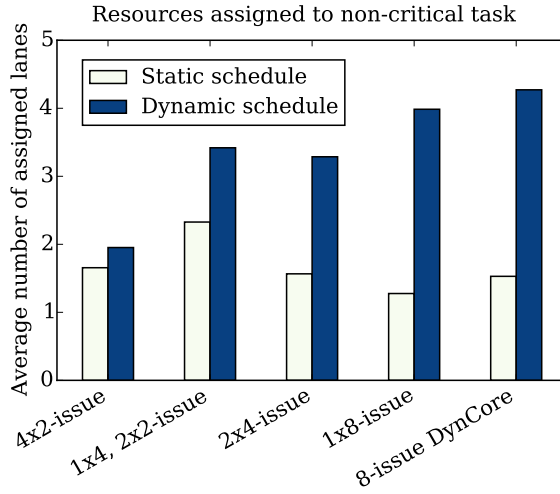


Figure 8.12: Average resource assignment for the non-critical task per evaluated setup.

the Itanium [117]. There, it is used to increase throughput, mostly by overlapping long-latency events (such as L2 cache misses) with computations from other threads. This is not directly comparable to the  $\rho$ -VEX used in this work as it targets the high-performance instead of embedded domain, and furthermore their SMT approach does not provide performance isolation. An SMT architecture with bounded performance interference is proposed in [120]. The multiple contexts of the  $\rho$ -VEX are discussed in [41]. In [26], modifications to an Alpha 211164 are proposed that makes use of multiple isolated multi-threaded contexts (virtual processors) to improve static real-time schedulability. These modifications share some similarities with the  $\rho$ -VEX, and their scheduling method is also used in this work (see [42] for a more in-depth discussion on the relation with our work). Some of the contributions we provide over this work is the use of a platform that is implemented in VHDL and available for download for the academic community, and which targets the embedded instead of high-performance domain.

Although it is not discussed by [26], the scheduling method is similar to period transformation [127] and p-fair scheduling [128]. P-fair also relies on rounds to assign resources to tasks. However, in p-fair scheduling, a round is atomic regarding resource assignment (a resource can only be assigned to a task for a full round). The scheduling method used in this work divides the resources *within* a round in order to reduce the search space of a dynamic processor that can change at any time during runtime.

In [129], a multi-processor static and dynamic scheduling method is approached that aims to increase utilization for mixed-criticality systems. The effect of resource sharing on application execution time is researched by [130].

In the realm of time-predictable processors, the most notable example is arguably the recently introduced FlexPRET software [121] that is also available for

download and can be prototyped on an FPGA. Instead of assigning execution lanes to threads, it can assign cycles to threads in a fine-grained multi-threaded fashion. By assigning it a larger number of the cycles, it can also throttle the performance of a single thread. The advantage of the  $\rho$ -VEX is that it is a VLIW architecture that can provide a performance advantage over the scalar FlexPRET (see [42]), particularly in certain embedded domains such as signal processing or media applications. Other time-predictable multi-threaded processors are PTARM [131] and Merasa [132]. In [133], a time-predictable 2-issue VLIW processor is introduced combined with caches that include hardware support for WCET analysis. See [122] for a study about time-predictability of VLIWs and their compilers. In a related effort, [134] aims to construct a timing-predictable platform using existing processors and using a number of design principles similar to our work (for example using separate memories).

## 8.8. Conclusions

This chapter introduces the  $\rho$ -VEX polymorphic processor to the field of real-time and mixed-criticality systems. We showed that it can exploit its dynamic properties to 1) improve schedulability over fixed execution platforms, while still providing execution time guarantees when using a round-based scheduling methodology, and 2) efficiently assign resources to lower-priority threads when high-priority threads finish ahead of their WCET (up to 50% and on average 25% more). The resulting increase in throughput depends on the tasks characteristics such as ILP. Due to the 8-issue VLIW architecture, it can also provide significant performance gains compared to scalar RISC architectures such as time-predictable RISC-V processors.

The nature of VLIW architectures provides a high degree of predictability as it uses static branch prediction and an exposed pipeline. This makes it possible to establish relatively tight WCET bounds, either using measurements (if it is possible to fabricate an input that activates the longest execution path) or using static analysis techniques. The processor's polymorphism creates the opportunity for systems to quickly adapt to the environment, a property identified to be desirable for future cyber-physical systems by [135], to be able to quickly react to an emergency situation (for example, by assigning all computational resources to avoiding an imminent collision). These advantages make it a suitable platform for mixed-criticality systems, especially when the workload contains media and/or signal processing applications.

The cost of increased area utilization is a trade-off that designers must make when choosing an execution platform. Keep in mind that, to achieve full predictability in a complete system, a predictable interconnect (and main memory system, if applicable) must be used such as [123]. When using caches instead of local memories, the system still provides performance isolation because the caches are split in the same fashion as the datapaths, but the predictability is severely impacted (one would need to assume that every cache access results in a miss). The  $\rho$ -VEX comes with VHDL code, toolchain (consisting of multiple compilers, binutils, newlib, etc.), a fast architectural simulator, extensive debug hardware and interface tools. It can be downloaded for academic use at [www.rvex.ewi.tudelft.nl](http://www.rvex.ewi.tudelft.nl).

# 9

## Conclusion

This section summarizes the topics discussed in this thesis, provides some concluding remarks and tries to identify a number of research directions for the future.

### 9.1. Conclusions

In Chapter 1, we have proposed to use static (design-time) reconfigurability to target static workloads and dynamic (run-time) parametrizability for dynamic workloads in the embedded domain. From the evaluations performed in this work, we can conclude the following:

#### **Part 1 - Static workloads, statically reconfigurable platform**

In Part 1, we introduced a design-time customizable computation fabric based on VLIW softcore processors and a streaming memory hierarchy:

- We found a VLIW processor to provide up to a factor of  $3.2\times$  better performance with similar resource utilization compared to the industry-standard MicroBlaze processor in Chapter 2.
- We showed that, for a static workload consisting of a chain of image processing filters, streaming data directly between cores results in considerably better performance compared to a bus-based topology in Chapter 3
- This stream-based platform can be programmed using OpenCL in a frame-based fashion, abstracting away the hardware complexity from software programmers as discussed in Chapter 4. This platform can be used for rapid prototyping, debugging and optimization to bridge the gap to High-Level Synthesis (HLS). It features a wide number of configuration parameters that can be explored by the designer without needing to program or modify HDL code. Additionally, it has improved scalability compared to a similar platform targeting the biomedical imaging domain, allowing it to increase the core count and operating frequency on an FPGA.

## Part 2 - Dynamic workloads, dynamically reconfigurable platform

In Part 2, we introduced mechanisms that allow a polymorphic processor to automatically evaluate code characteristics of a highly dynamic workload and adapt accordingly:

- First, we showed that fine-grained reconfigurable processors are able to closely match dynamic program characteristics using high frequency adaptations in Chapter 5. The  $\rho$ -VEX, with a configuration penalty of 5 clock cycles, achieves up to 20% better energy-delay-product when using a reconfiguration window size of 75 cycles compared to using a window of 1000 cycles. The performance is, as is to be expected, highly dependent on the amount of ILP variability present in the code.
- Automatic reconfigurations allow the dynamic processor to achieve up to 25% and on average 10% better EDP for a single program compared to the best performance static configuration with equal computational resources.
- Chapter 6 shows that automatic reconfigurations enable the reconfigurable processor to achieve 20% better performance on dynamic workloads compared to a static heterogeneous configuration with equal computational resources.
- Using compiler annotations to steer reconfigurations can provide up to a factor of  $2\times$  more throughput. However, the results are highly dependent on the workload and the additional value over using performance monitoring is, on average, limited.

## Part 3 - Real-time and mixed-criticality systems

In Part 3, we added real-time requirements to the workload, and explore mixed static and dynamic workloads and scheduling:

- In Chapter 7, we show that the additional hardware contexts allow the  $\rho$ -VEX to achieve up to a factor  $10 \times$  lower context switch latencies. Advantages are noticeable primarily in scenarios where large numbers of real-time events must be handled or when the system tick frequency is very high.
- Chapter 8 discusses a method to increase static real-time schedulability by leveraging the dynamic properties of the run-time parameterizable processor. In addition, we discuss that the VLIW architecture can provide high performance without sacrificing time-predictability.
- The proposed platform architecture targeting mixed-criticality systems increases throughput for non-critical static tasks while still guaranteeing time-safety for critical tasks.

## 9.2. Future research directions

The liquid architectures research direction is a large project within the Computer Engineering Laboratory and consequently, there have been various collaborations

with numerous researchers regarding different topics. Many ideas have been conceived during this time that we have not been able to pursue. We will list some of them here. For the static  $\rho$ -VEX:

- **SIMD** or other **instruction set extensions** are a proven method to increase individual core performance for image processing workloads. For example, the Hexagon Vector Extensions (HVX) use 1024-bit wide datapaths, supported by **Halide**. This could even lead to some form of **automatic design-space exploration** regarding code generation for image processing filters.
- Explore options for a **Network-on-Chip (NoC)**, or at least supporting 1:n and n:1 connections between streaming cores. This allows filters that require more input data, for example a global contrast enhancement algorithm.
- A **high-level synthesis (HLS)** flow for the code that is running on the cores. These can be synthesized to specialized accelerators that require less area and may achieve higher frequencies, but are no longer programmable.

For the dynamic  $\rho$ -VEX:

- **Validating simulation results** for performance and energy-efficiency of high-frequency core adaptations on actual silicon.
- **Switch-on-Event (coarse-grained) Multithreading**, commonly utilized method to hide memory latency. When a context must wait for a long latency event such as a L2 cache miss, it is swapped out by another context (whose cache miss may have been resolved while it was waiting). The  $\rho$ -VEX already has multiple contexts in hardware, which is the most costly component in terms of area utilization.
- **Register file banking or clustering**. A unified register file represents the largest component of a large issue-width VLIW processor. This makes the 8-issue  $\rho$ -VEX very expensive in terms of area utilization. A common technique to mitigate this is to split the register file into banks or clusters [18].
- Explore different options for the **cache organization**. Currently, the caches consist of blocks that split and merge together with the datapaths. When splitting the core, live data may become unreachable as it resides in a different cache block [136]. Another possibility is to create a unified cache organization, where all datapaths access a single cache block. This has some limitations (it will not be possible for all 4 sub-cores to perform a memory access simultaneously), and it will require set-associative cache to prevent different contexts to evict each other's live data, but will allow high-frequency adaptations without contexts losing access to their cached data.
- **Auto-tuning OpenMP support**. In a recent development, the  $\rho$ -VEX team has implemented experimental support for OpenMP. This opens up the possibility to automatically tune the number of threads based on their characteristics. A similar concept, auto-tuning SMT modes for a big data framework, has been evaluated on the POWER8 architecture in [137].

- To exploit the dynamic nature of the  $\rho$ -VEX on a system-wide basis, **SMP support** is required in the Linux kernel. Additionally, the kernel needs to be able to handle core split and merge events. This functionality has been implemented in Chameleon [100], whose creators share our vision for running dynamic workloads on dynamic computing platforms. Their implementation targets a theoretical dynamic architecture, a position that can be taken by the  $\rho$ -VEX.
- **Secure computing.** A program could be made more resilient to side-channel attacks by filling all parallel datapaths of the  $\rho$ -VEX with instructions, and selecting the correct outcome at the end of the execution block using a predicate instruction. Each possible flow requires the same amount of time and energy. An advantageous property of the  $\rho$ -VEX is that it can still provide high performance for code sections that do not require protection. A similar scheme is possible for **fault tolerance**.

# List of Publications

1. **J.J. Hoozemans**, J. Van Straten, S. Wong, *Increasing resource utilization in Mixed-Criticality Systems using a polymorphic VLIW processor*, Journal of Systems Architecture **84** (2018)
2. **J.J. Hoozemans**, J. Van Straten, Z. Al-Ars, S. Wong, *Evaluating Auto-adapting Methods for Fine-grained Adaptable Processors*, 31st International Conference on Architecture of Computing Systems (ARCS 2018)
3. **J.J. Hoozemans**, A.A.C. Brandon, J. Van Straten, S. Wong, *Compiler-driven versus Monitoring-based Processor Polymorphism* (in preparation)
4. **J.J. Hoozemans**, J. Van Straten, T. Viitanen, A. Tervo, J. Kadlec, *ALMARVI video processing SoC platform on Zynq* (under review)
5. **J.J. Hoozemans**, R. De Jong, S. Van der Vlugt, J. Van Straten, U.K. Elango, Z. Al-Ars *Frame-based programming, stream-based processing* (under review)
6. A. L. Sartor, P. H. E. Becker, **J.J. Hoozemans**, S. Wong, A.C.S. Beck, *Dynamic Trade-off among Fault Tolerance, Energy Consumption, and Performance on a Multiple-issue VLIW Processor*, IEEE Transactions on Multi-Scale Computing Systems **99** (2017)
7. **J.J. Hoozemans**, J. Van Straten, S. Wong, *Using a Polymorphic VLIW Processor to Improve Schedulability and Performance for Mixed-criticality Systems*, 23rd IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2017)
8. **J.J. Hoozemans**, R.W. Heij, J. Van Straten, Z. Al-Ars, *VLIW-based FPGA computational fabric with streaming memory hierarchy for medical imaging applications*, 13th International Symposium on Applied Reconfigurable Computing (ARC 2017)
9. A.A.C. Brandon, **J.J. Hoozemans**, J. Van Straten, S. Wong, *Exploring ILP and TLP on a Polymorphic VLIW Processor*, 30th International Conference on Architecture of Computing Systems (ARCS 2017)
10. **J.J. Hoozemans**, A. F Lorenzon, A.C.S. Beck, S. Wong, *Improved dynamic cache sharing for communicating threads on a runtime-adaptable processor*, 11th HiPEAC Workshop on Reconfigurable Computing (WRC 2017)
11. **J.J. Hoozemans**, J. Johansen, J. Van Straten, A.A.C. Brandon, S. Wong, *Multiple Contexts in a Multi-ported VLIW Register File Implementation*, 2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig 2015)
12. A.A.C. Brandon, **J.J. Hoozemans**, J. Van Straten, A. F Lorenzon, A. L. Sartor, A.C.S. Beck, S. Wong, *A Sparse VLIW Instruction Encoding Scheme Compatible with Generic Binaries*, 2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig 2015)



13. **J.J. Hoozemans**, S. Wong, Z. Al-Ars, *Using VLIW Softcore Processors for Image Processing Applications*, 15th International Conference On Embedded Computer Systems: Architectures, Modeling, And Simulation (SAMOS 2015)

# References

- [1] S. Chakraborty and S. Ramesh, *Guest Editorial Special Section on Automotive Embedded Systems and Software*, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **34**, 1701 (2015), doi: 10.1109/T-CAD.2015.2488378.
- [2] G. Buttazzo, *Research Trends in Real-time Computing for Embedded Systems*, *SIGBED Rev.* **3**, 1 (2006), doi: 10.1145/1164050.1164052.
- [3] P. Gai and M. Violante, *Automotive embedded software architecture in the multi-core age*, in *21th IEEE European Test Symposium (ETS)* (2016) pp. 1–8, doi: 10.1109/ETS.2016.7519309.
- [4] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder, *Discovering and Exploiting Program Phases*, *IEEE Micro* **23**, 84 (2003), doi: 10.1109/MM.2003.1261391.
- [5] S. Baruah, H. Li, and L. Stougie, *Towards the Design of Certifiable Mixed-criticality Systems*, in *16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)* (2010) pp. 13–22, doi: 10.1109/RTAS.2010.10.
- [6] Khubaib, M. A. Suleman, M. Hashemi, C. Wilkerson, and Y. N. Patt, *MorphCore: An Energy-Efficient Microarchitecture for High Performance ILP and High Throughput TLP*, in *45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2012) pp. 305–316, doi: 10.1109/MICRO.2012.36.
- [7] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore, *Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture*, in *30th Annual International Symposium on Computer Architecture (ISCA)* (2003) pp. 422–433, doi: 10.1109/ISCA.2003.1207019.
- [8] E. Ipek, M. Kirman, N. Kirman, and J. F. Martinez, *Core Fusion: Accommodating Software Diversity in Chip Multiprocessors*, in *34th Annual International Symposium on Computer Architecture (ISCA)* (ACM, 2007) pp. 186–197, doi: 10.1145/1250662.1250686.
- [9] M. D. Hill and M. R. Marty, *Amdahl's Law in the Multicore Era*, *Computer* **41**, 33 (2008), doi: 10.1109/MC.2008.209.
- [10] D. Liu, *Baseband ASIP design for SDR*, *China Communications* **12**, 60 (2015), doi: 10.1109/CC.2015.7188525.

- [11] M. Wijnvliet, L. Waeijen, and H. Corporaal, *Coarse grained reconfigurable architectures in the past 25 years: Overview and classification*, in *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)* (2016) pp. 235–244, doi: 10.1109/SAMOS.2016.7818353.
- [12] X. Fan, W.-D. Weber, and L. A. Barroso, *Power Provisioning for a Warehouse-sized Computer*, in *34th Annual International Symposium on Computer Architecture (ISCA)* (ACM, 2007) pp. 13–23, doi: 10.1145/1250662.1250665.
- [13] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen, *Single-ISA heterogeneous multi-core architectures: the potential for processor power reduction*, in *36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2003) pp. 81–92, doi: 10.1109/MICRO.2003.1253185.
- [14] P. Greenhalgh, *big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7*, ARM White paper (2011).
- [15] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fourth Edition: A Quantitative Approach* (Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007).
- [16] J. A. Fisher, P. Faraboschi, and C. Young, *Embedded Computing: A VLIW Approach to Architecture, Compilers, and Tools* (Morgan Kaufmann Publishers, 500 Sansome Street, Suite 400, San Francisco, CA 94111, 2005).
- [17] L. Codrescu, *Architecture of the Hexagon 680 DSP for mobile imaging and computer vision*, in *2015 IEEE Hot Chips 27 Symposium (HCS)* (2015) pp. 1–26, doi: 10.1109/HOTCHIPS.2015.7477329.
- [18] A. S. Terechko, *Clustered VLIW architectures: a quantitative approach*, *Ph.D. thesis*, Eindhoven University of Technology (2007), url: <http://repository.tue.nl/59f95276-a38c-4d4b-b48b-4a0f6c2f733d>.
- [19] O. Esko, P. Jaaskelainen, P. Huerta, C. S. de La Lama, J. Takala, and J. I. Martinez, *Customized Exposed Datapath Soft-Core Design Flow with Compiler Support*, in *International Conference on Field Programmable Logic and Applications (FPL)* (2010) pp. 217–222, doi: 10.1109/FPL.2010.51.
- [20] H. Corporaal, *Transport Triggered Architectures: Design and Evaluation*, *Ph.D. thesis*, Delft University of Technology (1995), url: <http://resolver.tudelft.nl/uuid:9ec25f3e-7879-4c1e-a7c7-9452a75032a2>.
- [21] S. Wong and F. Anjam, *The Delft Reconfigurable VLIW Processor*, in *17th International Conference on Advanced Computing and Communications (ICACC)* (2009) pp. 244–251.

- [22] F. Anjam, M. Nadeem, and S. Wong, *Targeting code diversity with run-time adjustable issue-slots in a chip multiprocessor*, in *Design, Automation Test in Europe Conference Exhibition (DATE)* (2011) pp. 1–6, doi: 10.1109/DATE.2011.5763219.
- [23] A. Brandon, J. Hoozemans, J. V. Straten, A. Lorenzon, A. Sartor, A. C. S. Beck, and S. Wong, *A sparse VLIW instruction encoding scheme compatible with generic binaries*, in *International Conference on ReConfigurable Computing and FPGAs (ReConFig)* (2015) pp. 1–7, doi: 10.1109/ReConFig.2015.7393361.
- [24] J. van Straten, *A Dynamically Reconfigurable VLIW Processor and Cache Design with Precise Trap and Debug Support*, *Master's thesis*, Delft University of Technology (2016).
- [25] H. Amano, *A survey on dynamically reconfigurable processors*, *IEICE Transactions on Communications* **E89-B**, 3179 (2006), doi: 10.1093/ietcom/e89-b.12.3179.
- [26] A. El-Haj-Mahmoud, A. S. AL-Zawawi, A. Anantaraman, and E. Rotenberg, *Virtual multiprocessor: An analyzable, high-performance architecture for real-time computing*, in *International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)* (ACM, 2005) pp. 213–224, doi: 10.1145/1086297.1086326.
- [27] D. M. Tullsen, S. J. Eggers, and H. M. Levy, *Simultaneous multithreading: Maximizing on-chip parallelism*, in *22nd Annual International Symposium on Computer Architecture (ISCA)* (1995) pp. 392–403.
- [28] B. Sinharoy, J. A. V. Norstrand, R. J. Eickemeyer, H. Q. Le, J. Leenstra, D. Q. Nguyen, B. Konigsburg, K. Ward, M. D. Brown, J. E. Moreira, D. Levitan, S. Tung, D. Hrusecky, J. W. Bishop, M. Gschwind, M. Boersma, M. Kroener, M. Kaltenbach, T. Karkhanis, and K. M. Fernsler, *IBM POWER8 processor core microarchitecture*, *IBM Journal of Research and Development* **59**, 2:1 (2015), doi: 10.1147/JRD.2014.2376112.
- [29] A. Brandon and S. Wong, *Support for dynamic issue width in VLIW processors using generic binaries*, in *Design, Automation Test in Europe Conference Exhibition (DATE)* (2013) pp. 827–832, doi: 10.7873/DATE.2013.175.
- [30] F. Anjam, *Run-time Adaptable VLIW Processors – Resources, Performance, Power Consumption, and Reliability Trade-offs*, *Ph.D. thesis*, Delft University of Technology (2013), doi: 10.4233/uuid:850eb79f-b8de-47b5-832f-7c2c138787be.
- [31] P. Faraboschi, G. Brown, J. A. Fisher, G. Desoll, and F. M. O. Homewood, *Lx: a technology platform for customizable vliw embedded processing*, in *27th International Symposium on Computer Architecture (ISCA)* (2000) pp. 203–213, doi: 10.1109/ISCA.2000.854391.

- [32] J. Hoozemans, *Porting Linux to the rVEX reconfigurable VLIW softcore*, Master's thesis, Delft University of Technology, Delft, Netherlands (2014), url: <http://resolver.tudelft.nl/uuid:329eba52-453e-4339-9bd4-8230952446fc>.
- [33] J. Ray and J. C. Hoe, *High-level Modeling and FPGA Prototyping of Microprocessors*, in *Eleventh International Symposium on Field Programmable Gate Arrays (FPGA)* (ACM, 2003) pp. 100–107, doi: 10.1145/611817.611833.
- [34] S. Asaad, R. Bellofatto, B. Brezzo, C. Haymes, M. Kapur, B. Parker, T. Roewer, P. Saha, T. Takken, and J. Tierno, *A cycle-accurate, cycle-reproducible multi-fpga system for accelerating multi-core processor simulation*, in *International Symposium on Field Programmable Gate Arrays (FPGA)* (ACM, 2012) pp. 153–162.
- [35] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, *MiBench: A free, commercially representative embedded benchmark suite*, in *Fourth Annual IEEE International Workshop on Workload Characterization. (WWC)* (2001) pp. 3–14, doi: 10.1109/WWC.2001.990739.
- [36] J. Hoozemans, S. Wong, and Z. Al-Ars, *Using VLIW softcore processors for image processing applications*, in *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)* (2015) pp. 315–318, doi: 10.1109/SAMOS.2015.7363691.
- [37] J. Hoozemans, R. Heij, J. Van Straten, and Z. Al-Ars, *VLIW-Based FPGA Computation Fabric with Streaming Memory Hierarchy for Medical Imaging Applications*, in *13th International Symposium on Applied Reconfigurable Computing (ARC)* (Springer, 2017) pp. 36–43, doi: 10.1007/978-3-319-56258-2\_4.
- [38] J. Hoozemans, R. De Jong, S. Van der Vlugt, J. Van Straten, U. K. Elango, and Z. Al-Ars, *Frame-based programming, Stream-based processing (under review)*, (2018).
- [39] J. Hoozemans, J. Van Straten, Z. Al-Ars, and S. Wong, *Evaluating Auto-adaptation Methods for Fine-Grained Adaptable Processors*, in *Architecture of Computing Systems (ARCS)* (Springer International Publishing, 2018) pp. 255–268, doi: 10.1007/978-3-319-77610-1\_19.
- [40] J. Hoozemans, A. Brandon, J. V. Straten, and S. Wong, *Compiler-driven versus Monitoring-based Processor Polymorphism (in preparation)*, .
- [41] J. Hoozemans, J. Johansen, J. V. Straten, A. Brandon, and S. Wong, *Multiple Contexts in a Multi-ported VLIW Register File Implementation*, in *International Conference on ReConfigurable Computing and FPGAs (ReConFig)* (2015) pp. 1–6, doi: 10.1109/ReConFig.2015.7393329.
- [42] J. Hoozemans, J. V. Straten, and S. Wong, *Using a polymorphic VLIW processor to improve schedulability and performance for mixed-criticality systems*, in *23rd International Conference on Embedded and Real-*

- Time Computing Systems and Applications (RTCSA)* (2017) pp. 1–9, doi: 10.1109/RTCSA.2017.8046315.
- [43] J. Hoozemans, J. V. Straten, and S. Wong, *Increasing resource utilization in mixed-criticality systems using a polymorphic VLIW processor*, *Journal of Systems Architecture* **84**, 2 (2018), doi: 10.1016/j.sysarc.2018.01.003.
- [44] L. Russo, E. Pedrino, E. Kato, and V. Roda, *Image convolution processing: A GPU versus FPGA comparison*, in *VIII Southern Conference on Programmable Logic (SPL)* (2012) pp. 1–6, doi: 10.1109/SPL.2012.6211783.
- [45] A. Putnam, A. Caulfield, E. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Xiao, and D. Burger, *A reconfigurable fabric for accelerating large-scale datacenter services*, in *41st International Symposium on Computer Architecture (ISCA)* (2014) pp. 13–24, doi: 10.1109/ISCA.2014.6853195.
- [46] P. Wang, J. McAllister, and Y. Wu, *Soft-core Stream Processing on FPGA: An FFT Case Study*, in *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (2013) pp. 2756–2760, doi: 10.1109/ICASSP.2013.6638158.
- [47] S. Williams, A. Waterman, and D. Patterson, *Roofline: An insightful visual performance model for multicore architectures*, *Communication of the ACM* **52**, 65 (2009), doi: 10.1145/1498765.1498785.
- [48] C. Iseli and E. Sanchez, *Spyder: a reconfigurable vliw processor using fp-gas*, in *IEEE Workshop on Field-Programmable Custom Computing Machines (FCCM)* (1993) pp. 17–24, doi: 10.1109/FPGA.1993.279483.
- [49] A. K. Jones, R. Hoare, D. Kusic, J. Fazekas, and J. Foster, *An FPGA-based VLIW Processor with Custom Hardware Execution*, in *13th International Symposium on Field-Programmable Gate Arrays (FPGA)* (ACM, 2005) pp. 107–117, doi: 10.1145/1046192.1046207.
- [50] <http://www.trimaran.org>.
- [51] V. Brost, F. Yang, and M. Painsavoine, *A modular VLIW Processor*, in *IEEE International Symposium on Circuits and Systems (ISCAS)* (2007) pp. 3968–3971, doi: 10.1109/ISCAS.2007.378669.
- [52] M. A. R. Saghir, M. El-Majzoub, and P. Akl, *Customizing the Datapath and ISA of Soft VLIW Processors*, in *Second International Conference on High Performance Embedded Architectures and Compilers (HiPEAC)* (Springer, 2007) pp. 276–290, doi: 10.1007/978-3-540-69338-3\_19.

- [53] A. Lodi, M. Toma, F. Campi, A. Cappelli, R. Canegallo, and R. Guerrieri, *A VLIW processor with reconfigurable instruction set for embedded applications*, *IEEE Journal of Solid-State Circuits* **38**, 1876 (2003), doi: 10.1109/JSSC.2003.818292.
- [54] S. Wong, T. van As, and G. Brown,  *$\rho$ -VEX: A Reconfigurable and Extensible Softcore VLIW Processor*, in *International Conference on Field-Programmable Technology (FPT)* (2008) pp. 369–372, doi: 10.1109/FPT.2008.4762420.
- [55] F. Anjam, M. Nadeem, and S. Wong, *A VLIW softcore processor with dynamically adjustable issue-slots*, in *International Conference on Field-Programmable Technology (FPT)* (2010) pp. 393–398, doi: 10.1109/FPT.2010.5681444.
- [56] The HP VEX toolchain, <http://www.hpl.hp.com/downloads/vex/>.
- [57] *LEON/GRLIB*, <http://www.gaisler.com/index.php/downloads/leongrplib>, [Online; accessed 7-Sept-2016].
- [58] Images created by Michael Plotke, licensed CC BY-SA 3.0.
- [59] D. Stevens, V. Choularas, V. Azorin-Peris, J. Zheng, A. Echiadis, and S. Hu, *BioThreads: A Novel VLIW-Based Chip Multiprocessor for Accelerating Biomedical Image Processing Applications*, *IEEE Transactions on Biomedical Circuits and Systems* **6**, 257 (2012), doi: 10.1109/TBCAS.2011.2166962.
- [60] T. Nowatzki, V. Gangadharan, K. Sankaralingam, and G. Wright, *Pushing the limits of accelerator efficiency while retaining programmability*, in *IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2016) pp. 27–39, doi: 10.1109/HPCA.2016.7446051.
- [61] K. Ovtcharov, O. Ruwase, J.-Y. Kim, J. Fowers, K. Strauss, and E. Chung, *Accelerating Deep Convolutional Neural Networks Using Specialized Hardware*, (2015).
- [62] L. M. Russo, E. C. Pedrino, E. Kato, and V. O. Roda, *Image Convolution Processing: A GPU versus FPGA Comparison*, in *2012 VIII Southern Conference on Programmable Logic* (2012) pp. 1–6, doi: 10.1109/SPL.2012.6211783.
- [63] P. Wang and J. McAllister, *Streaming Elements for FPGA Signal and Image Processing Accelerators*, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* **24**, 2262 (2016), doi: 10.1109/TVLSI.2015.2504871.
- [64] B. Bardak, F. M. Siddiqui, C. Kelly, and R. Woods, *Dataflow toolset for Softcore Processors on FPGA for Image Processing Applications*, in *2014 48th Asilomar Conference on Signals, Systems and Computers* (2014) pp. 1445–1449, doi: 10.1109/ACSSC.2014.7094701.



- [65] P. Kristof, H. Yu, Z. Li, and X. Tian, *Performance Study of SIMD Programming Models on Intel Multicore Processors*, in *IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)* (2012) pp. 2423–2432, doi: 10.1109/IPDPSW.2012.299.
- [66] C. C. Chi, M. Alvarez-Mesa, B. Bross, B. Juurlink, and T. Schierl, *SIMD Acceleration for HEVC Decoding*, *IEEE Transactions on Circuits and Systems for Video Technology* **25**, 841 (2015), doi: 10.1109/TCSVT.2014.2364413.
- [67] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra, *From CUDA to OpenCL: Towards a Performance-portable Solution for Multi-platform GPU Programming*, *Parallel Comput.* **38**, 391 (2012), doi: 10.1016/j.parco.2011.10.002.
- [68] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, *Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines*, in *34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (ACM, 2013) pp. 519–530, doi: 10.1145/2491956.2462176.
- [69] J. E. Stone, D. Gohara, and G. Shi, *OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems*, *Computing in Science Engineering* **12**, 66 (2010), doi: 10.1109/MCSE.2010.69.
- [70] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, *High-Level Synthesis for FPGAs: From Prototyping to Deployment*, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **30**, 473 (2011), doi: 10.1109/TCAD.2011.2110592.
- [71] G. Guidi, E. Reggiani, L. D. Tucci, G. Durelli, M. Blott, and M. D. Santambrogio, *On How to Improve FPGA-Based Systems Design Productivity via SDAccel*, in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (2016) pp. 247–252, doi: 10.1109/IPDPSW.2016.171.
- [72] Z. Guo, B. Buyukkurt, and W. Najjar, *Input Data Reuse in Compiling Window Operations Onto Reconfigurable Hardware*, in *Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)* (ACM, 2004) pp. 249–256, doi: 10.1145/997163.997199.
- [73] Y. Dong, Y. Dou, and J. Zhou, *Optimized Generation of Memory Structure in Compiling Window Operations onto Reconfigurable Hardware*, in *Third International Workshop Reconfigurable Computing: Architectures, Tools and Applications (ARC)* (Springer, 2007) pp. 110–121, doi: 10.1007/978-3-540-71431-6\_11.
- [74] F. Plavec, Z. Vranesic, and S. Brown, *Towards compilation of streaming programs into FPGA hardware*, in *Forum on Specification, Verification and Design Languages (FDL)* (2008) pp. 67–72, doi: 10.1109/FDL.2008.4641423.



- [75] J. Hegarty, J. Brunhaver, Z. DeVito, J. Ragan-Kelley, N. Cohen, S. Bell, A. Vasilyev, M. Horowitz, and P. Hanrahan, *Darkroom: Compiling High-level Image Processing Code into Hardware Pipelines*, *ACM Trans. Graph.* **33** (2014), 10.1145/2601097.2601174, doi: 10.1145/2601097.2601174.
- [76] I. Lebedev, S. Cheng, A. Doupnik, J. Martin, C. Fletcher, D. Burke, M. Lin, and J. Wawrzynek, *MARC: A Many-Core Approach to Reconfigurable Computing*, in *International Conference on Reconfigurable Computing and FPGAs (ReConFig)* (2010) pp. 7–12, doi: 10.1109/ReConFig.2010.49.
- [77] M. Lin, I. Lebedev, and J. Wawrzynek, *OpenRCL: Low-Power High-Performance Computing with Reconfigurable Devices*, in *International Conference on Field Programmable Logic and Applications (FPL)* (2010) pp. 458–463, doi: 10.1109/FPL.2010.93.
- [78] M. Jacobsen, D. Richmond, M. Hogains, and R. Kastner, *RIFFA 2.1: A Reusable Integration Framework for FPGA Accelerators*, *ACM Trans. Reconfigurable Technol. Syst.* **8**, 22:1 (2015), doi: 10.1145/2815631.
- [79] Topic Embedded Products, *DYnamic Process LOader (DYPLO)*, Online (2017).
- [80] Xilinx, *Xilinx Partial Reconfiguration design tool*, Online (2017).
- [81] P. Jääskeläinen, C. S. de La Lama, E. Schnetter, K. Raiskila, J. Takala, and H. Berg, *pocl: A Performance-Portable OpenCL Implementation*, *International Journal of Parallel Programming* **43**, 752 (2015), doi: 10.1007/s10766-014-0320-y.
- [82] J. A. Brown, L. Porter, and D. M. Tullsen, *Fast thread migration via cache working set prediction*, in *IEEE 17th International Symposium on High Performance Computer Architecture (HPCA)* (2011) pp. 193–204, doi: 10.1109/HPCA.2011.5749728.
- [83] K. K. Rangan, G.-Y. Wei, and D. Brooks, *Thread Motion: Fine-grained Power Management for Multi-core Systems*, in *36th Annual International Symposium on Computer Architecture (ISCA)* (ACM, 2009) pp. 302–313, doi: 10.1145/1555754.1555793.
- [84] M. Rodrigues, N. Roma, and P. Tomás, *Fast and Scalable Thread Migration for Multi-core Architectures*, in *IEEE 13th International Conference on Embedded and Ubiquitous Computing (EUC)* (2015) pp. 9–16, doi: 10.1109/EUC.2015.36.
- [85] A. Brandon, J. Hoozemans, J. V. Straten, and S. Wong, *Exploring ILP and TLP on a Polymorphic VLIW Processor*, in *30th International Conference on Architecture of Computing Systems (ARCS)* (2017) pp. 177–189, doi: 10.1007/978-3-319-54999-6\_14.

- [86] L. Codrescu, W. Anderson, S. Venkumanhanti, M. Zeng, E. Plondke, C. Koob, A. Ingle, C. Tabony, and R. Maule, *Hexagon DSP: An Architecture Optimized for Mobile Multimedia and Communications*, *IEEE Micro* **34**, 34 (2014), doi: 10.1109/MM.2014.12.
- [87] M. Becchi and P. Crowley, *Dynamic Thread Assignment on Heterogeneous Multiprocessor Architectures*, in *3rd Conference on Computing Frontiers (CF)* (ACM, 2006) pp. 29–40, doi: 10.1145/1128022.1128029.
- [88] Q. Guo, A. Sartor, A. Brandon, A. C. S. Beck, X. Zhou, and S. Wong, *Run-time phase prediction for a reconfigurable vliw processor*, in *Design, Automation Test in Europe Conference Exhibition (DATE)* (2016) pp. 1634–1639, doi: 10.3850/9783981537079\_0644.
- [89] J. Hoogerbrugge, *Dynamic branch prediction for a VLIW processor*, in *International Conference on Parallel Architectures and Compilation Techniques (PACT)* (2000) pp. 207–214, doi: 10.1109/PACT.2000.888345.
- [90] R. Rodrigues, A. Annamalai, I. Koren, and S. Kundu, *Improving Performance Per Watt of Asymmetric Multi-core Processors via Online Program Phase Classification and Adaptive Core Morphing*, *ACM Trans. Des. Autom. Electron. Syst.* **18**, 5:1 (2013), doi: 10.1145/2390191.2390196.
- [91] E. Duesterwald, C. Cascaval, and S. Dworkadas, *Characterizing and Predicting Program Behavior and its Variability*, in *12th International Conference on Parallel Architectures and Compilation Techniques (PACT)* (2003) pp. 220–231, doi: 10.1109/PACT.2003.1238018.
- [92] E. Chi, A. M. Salem, R. I. Bahar, and R. Weiss, *Combining software and hardware monitoring for improved power and performance tuning*, in *Seventh Workshop on Interaction Between Compilers and Computer Architectures (INTERACT)* (2003) pp. 57–64, doi: 10.1109/INTERA.2003.1192356.
- [93] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer, *Scheduling Heterogeneous Multi-cores Through Performance Impact Estimation (PIE)*, in *39th Annual International Symposium on Computer Architecture (ISCA)* (2012) pp. 213–224, doi: 10.1109/ISCA.2012.6237019.
- [94] A. Otero, A. Morales-Cas, J. Portilla, E. de la Torre, and T. Riesgo, *A Modular Peripheral to Support Self-Reconfiguration in SoCs*, in *13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools (DSD)* (2010) pp. 88–95, doi: 10.1109/DSD.2010.100.
- [95] M. Aldham, J. Anderson, S. Brown, and A. Canis, *Low-cost hardware profiling of run-time and energy in FPGA embedded processors*, in *22nd IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)* (2011) pp. 61–68, doi: 10.1109/ASAP.2011.6043237.

- [96] T. Sherwood, S. Sair, and B. Calder, *Phase Tracking and Prediction*, in *30th Annual International Symposium on Computer Architecture (ISCA)* (ACM, 2003) pp. 336–349, doi: 10.1145/859618.859657.
- [97] *TMS320C66x CPU and Instruction Set Reference Guide*, Texas Instruments Literature Number: SPRUGH7 (2010).
- [98] H. Zhong, S. Lieberman, and S. Mahlke, *Extending Multicore Architectures to Exploit Hybrid Parallelism in Single-thread Applications*, in *IEEE 13th International Symposium on High Performance Computer Architecture (HPCA)* (2007) pp. 25–36, doi: 10.1109/HPCA.2007.346182.
- [99] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz, *Smart memories: a modular reconfigurable architecture*, in *27th International Symposium on Computer Architecture (ISCA)* (2000) pp. 161–171, doi: 10.1109/ISCA.2000.854387.
- [100] S. Panneerselvam and M. M. Swift, *Chameleon: Operating System Support for Dynamic Processors*, in *Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (ACM, 2012) pp. 99–110, doi: 10.1145/2150976.2150988.
- [101] C. Hu, D. A. Jiménez, and U. Kremer, *Combining Edge Vector and Event Counter for Time-Dependent Power Behavior Characterization*, in *Transactions on High-Performance Embedded Architectures and Compilers (HiPEAC)*, edited by P. Stenström (Springer, 2009) pp. 85–104, doi: 10.1007/978-3-642-00904-4\_6.
- [102] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl, *Continuous Profiling: Where Have All the Cycles Gone?* *ACM Trans. Comput. Syst.* **15**, 357 (1997), doi: 10.1145/265924.265925.
- [103] J. Lau, S. Schoenmackers, and B. Calder, *Transition phase classification and prediction*, in *11th International Symposium on High-Performance Computer Architecture (HPCA)* (2005) pp. 278–289, doi: 10.1109/HPCA.2005.39.
- [104] R. Sree, A. Settle, I. Bratt, and D. Connors, *Compiler-directed resource management for active code regions*, in *Seventh Workshop on Interaction Between Compilers and Computer Architectures (INTERACT)* (2003) pp. 85–93, doi: 10.1109/INTERA.2003.1192359.
- [105] C.-H. Hsu, U. Kremer, and M. Hsiao, *Compiler-directed dynamic voltage/frequency scheduling for energy reduction in microprocessors*, in *International Symposium on Low Power Electronics and Design* (2001) pp. 275–278, doi: 10.1109/LPE.2001.945416.
- [106] U. Brinkschulte, C. Krakowski, J. Kreuzinger, and T. Ungerer, *Interrupt service threads-a new approach to handle multiple hard real-time events on a*

- multithreaded microcontroller*, in *20th IEEE Real-Time Systems Symposium (RTSS)* (1999) pp. 11–15.
- [107] S. Wong, F. Anjam, and F. Nadeem, *Dynamically reconfigurable register file for a softcore VLIW processor*, in *Design, Automation Test in Europe Conference Exhibition (DATE)* (2010) pp. 969–972, doi: 10.1109/DATE.2010.5456908.
- [108] C. E. LaForest and J. G. Steffan, *Efficient Multi-ported Memories for FPGAs*, in *18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)* (ACM, 2010) pp. 41–50, doi: 10.1145/1723112.1723122.
- [109] M. A. R. Saghir and R. Naous, *A Configurable Multi-ported Register File Architecture for Soft Processor Cores*, in *Third International Workshop on Reconfigurable Computing: Architectures, Tools and Applications (ARC)* (Springer, 2007) pp. 14–25, doi: 10.1007/978-3-540-71431-6\_2.
- [110] F. Anjam, S. Wong, and F. Nadeem, *A multiported register file with register renaming for configurable softcore VLIW processors*, in *International Conference on Field-Programmable Technology (FPT)* (2010) pp. 403–408, doi: 10.1109/FPT.2010.5681446.
- [111] M. Purnaprajna and P. Ienne, *Making Wide-issue VLIW Processors Viable on FPGAs*, *ACM Trans. Archit. Code Optim.* **8**, 33:1 (2012), doi: 10.1145/2086696.2086712.
- [112] G. Buttazzo, *Hard Real-time Computing Systems: Predictable Scheduling Algorithms and Applications*, Real-time Systems Series, Vol. 24 (Springer, 2011) doi: 10.1007/978-1-4614-0676-1.
- [113] A. Ronnholm, *Evaluation of Real-Time Operating Systems for Xilinx MicroBlaze CPU*, *Master's thesis*, Malardalens University (2006).
- [114] R. Thekkath and S. J. Eggers, *The Effectiveness of Multiple Hardware Contexts*, in *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (ACM, 1994) pp. 328–337, doi: 10.1145/195473.195583.
- [115] N. I. Rafla and D. Gauba, *Hardware implementation of context switching for hard real-time operating systems*, in *54th International Midwest Symposium on Circuits and Systems (MWSCAS)* (2011) pp. 1–4, doi: 10.1109/MWSCAS.2011.6026348.
- [116] K. Tanaka, *Prestor-1: a processor extending multithreaded architecture*, in *Innovative Architecture for Future Generation High-Performance Processors and Systems (IWIA)* (2005) pp. 8 pp.–, doi: 10.1109/IWIA.2005.39.
- [117] R. Riedlinger, R. Bhatia, L. Biro, B. Bowhill, E. Fetzer, P. Gronowski, and T. Grutkowski, *A 32nm 3.1 billion transistor 12-wide-issue Itanium processor*

- for mission-critical servers, in *IEEE International Solid-State Circuits Conference (ISSCC)* (2011) pp. 84–86, doi: 10.1109/ISSCC.2011.5746230.
- [118] A. Oliveira, L. Almeida, and A. de Brito Ferrari, *The ARPA-MT Embedded SMT Processor and Its RTOS Hardware Accelerator*, *IEEE Transactions on Industrial Electronics* **58**, 890 (2011), doi: 10.1109/TIE.2009.2028359.
- [119] T. P. Wijesinghe, *Design and implementation of a multithreaded softcore processor with tightly coupled hardware real-time operating system*, *Master's thesis*, West Virginia University (2008).
- [120] M. Paolieri, J. Mische, S. Metzclaff, M. Gerdes, E. Quiñones, S. Uhrig, T. Ungerer, and F. J. Cazorla, *A Hard Real-time Capable Multi-core SMT Processor*, *ACM Trans. Embed. Comput. Syst.* **12**, 79:1 (2013), doi: 10.1145/2442116.2442129.
- [121] M. Zimmer, D. Broman, C. Shaver, and E. A. Lee, *Flexpret: A processor platform for mixed-criticality systems*, in *19th Real-Time and Embedded Technology and Applications Symposium (RTAS)* (2014) pp. 101–110, doi: 10.1109/RTAS.2014.6925994.
- [122] J. Yan and W. Zhang, *A Time-predictable VLIW Processor and its Compiler Support*, *Real-Time Systems* **38**, 67 (2008), doi: 10.1007/s11241-007-9030-5.
- [123] A. Hansson, K. Goossens, M. Bekooij, and J. Huisken, *CoMPSoC: A Template for Composable and Predictable Multi-processor System on Chips*, *ACM Trans. Des. Autom. Electron. Syst.* **14**, 2:1 (2009), doi: 10.1145/1455229.1455231.
- [124] B. Chazelle, *The Bottomn-Left Bin-Packing Heuristic: An Efficient Implementation*, *IEEE Transactions on Computers* **C-32**, 697 (1983), doi: 10.1109/TC.1983.1676307.
- [125] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, *The Mälardalen WCET Benchmarks: Past, Present And Future*, in *10th International Workshop on Worst-Case Execution Time Analysis (WCET)*, OpenAccess Series in Informatics (OASIs), Vol. 15, edited by B. Lisper (Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2010) pp. 136–146.
- [126] J. A. Stankovic and K. Ramamritham, *What is Predictability for Real-time Systems?* *Real-Time Systems* **2**, 247 (1990), doi: 10.1007/BF01995673.
- [127] L. Sha, J. P. Lehoczky, and R. Rajkumar, *Task Scheduling In Distributed Real-Time Systems*, *SPIE 0857, IECON'87:Automated Design and Manufacturing* (1987), 10.1117/12.943278, doi: 10.1117/12.943278.
- [128] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel, *Proportionate Progress: A Notion of Fairness in Resource Allocation*, *Algorithmica* **15**, 600 (1996), doi: 10.1007/BF01940883.

- [129] E. Yip, M. M. Y. Kuo, P. S. Roop, and D. Broman, *Relaxing the synchronous approach for mixed-criticality systems*, in *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)* (2014) pp. 89–100, doi: 10.1109/RTAS.2014.6925993.
- [130] G. Giannopoulou, N. Stoimenov, P. Huang, and L. Thiele, *Scheduling of mixed-criticality applications on resource-sharing multicore systems*, in *International Conference on Embedded Software (EMSOFT)* (IEEE, 2013) pp. 1–15, doi: 10.1109/EMSOFT.2013.6658595.
- [131] I. Liu, J. Reineke, D. Broman, M. Zimmer, and E. A. Lee, *A PRET microarchitecture implementation with repeatable timing and competitive performance*, in *30th International Conference on Computer Design (ICCD)* (2012) pp. 87–93, doi: 10.1109/ICCD.2012.6378622.
- [132] T. Ungerer, F. Cazorla, P. Sainrat, G. Bernat, Z. Petrov, C. Rochange, E. Quinones, M. Gerdes, M. Paolieri, J. Wolf, H. Casse, S. Uhrig, I. Guliashvili, M. Houston, F. Kluge, S. Metzlaß, and J. Mische, *Merasa: Multicore Execution of Hard Real-Time Applications Supporting Analyzability*, *IEEE Micro* **30**, *66* (2010), doi: 10.1109/MM.2010.78.
- [133] M. Schoeberl, P. Schleuniger, W. Puffitsch, F. Brandner, C. W. Probst, S. Karlsson, and T. Thorn, *Towards a Time-predictable Dual-Issue Microprocessor: The Patmos Approach*, in *Bringing Theory to Practice: Predictability and Performance in Embedded Systems*, Vol. 18 (2011) pp. 11–21.
- [134] C. Cullmann, C. Ferdinand, G. Gebhard, D. Grund, C. Maiza, J. Reineke, B. Triquet, and R. Wilhelm, *Predictability considerations in the design of multi-core embedded systems*, *Embedded Real Time Software and Systems*, **36** (2010).
- [135] M. S. Mollison, J. P. Erickson, J. H. Anderson, S. K. Baruah, and J. A. Scoredos, *Mixed-Criticality Real-Time Scheduling for Multicore Systems*, in *10th IEEE International Conference on Computer and Information Technology* (2010) pp. 1864–1871, doi: 10.1109/CIT.2010.320.
- [136] S. Hu, A. Brandon, Q. Guo, and Y. Wang, *Improving the performance of adaptive cache in reconfigurable vliw processor*, in *13th International Symposium on Applied Reconfigurable Computing: (ARC 2017), Delft, The Netherlands* (Springer, 2017) pp. 3–15, doi: 10.1007/978-3-319-56258-2\_1.
- [137] Z. Jia, C. Xue, G. Chen, J. Zhan, L. Zhang, Y. Lin, and P. Hofstee, *Auto-tuning Spark Big Data Workloads on POWER8: Prediction-Based Dynamic SMT Threading*, in *International Conference on Parallel Architectures and Compilation (PACT)* (ACM, 2016) pp. 387–400, doi: 10.1145/2967938.2967957.



# Curriculum Vitæ

Joost Hoozemans was born on July 22, 1987 in Delft, the Netherlands. He obtained a BSc. degree in Computer Science from Utrecht University in 2011 and a MSc. degree in Computer Engineering from Delft University of Technology in 2014. His master's research was on Operating System support for the  $\rho$ -VEX dynamically re-configurable VLIW processor. During his studies, he worked as an intern for TNO, evaluating a Wireless Sensor Network protocol. He continued his research on dynamic VLIW processors as a PhD. candidate within the EU-funded ALMARVI project. During this period, he has taught practical assignments of various post-graduate courses including Reconfigurable Computing Design and Modern Computer Architectures, for which he developed a new set of assignments based on the VLIW platform used in his research. In addition, he was involved with more than 10 master's thesis projects and 4 bachelor honors projects related to the  $\rho$ -VEX. He has contributed to demonstrators and deliverables for the ALMARVI project, and written several peer-reviewed conference contributions and journal articles. He was the general chair of the post-graduate study association 'Micro-Electronic Systems and Technology' and finance chair of the International Symposium on Applied Reconfigurable Computing (ARC 2017).