



TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

Pekka Jääskeläinen

**From Parallel Programs to Customized Parallel
Processors**



Julkaisu 1086 • Publication 1086

Tampereen teknillinen yliopisto. Julkaisu 1086
Tampere University of Technology. Publication 1086

Pekka Jääskeläinen

From Parallel Programs to Customized Parallel Processors

Thesis for the degree of Doctor of Science in Technology to be presented with due permission for public examination and criticism in Tietotalo Building, Auditorium TB224, at Tampere University of Technology, on the 8th of November 2012, at 12 noon.

Tampereen teknillinen yliopisto - Tampere University of Technology
Tampere 2012

ISBN 978-952-15-2932-0 (printed)
ISBN 978-952-15-2966-5 (PDF)
ISSN 1459-2045

ABSTRACT

The need for fast time to market of new embedded processor-based designs calls for a rapid design methodology of the included processors. The call for such a methodology is even more emphasized in the context of so called *soft cores* targeted to re-configurable fabrics where per-design processor customization is commonplace.

The C language has been commonly used as an input to hardware/software co-design flows. However, as C is a sequential language, its potential to generate parallel operations to utilize naturally parallel hardware constructs is far from optimal, leading to a customized processor design space with limited parallel resource scalability. In contrast, when utilizing a parallel programming language as an input, a wider processor design space can be explored to produce customized processors with varying degrees of utilized parallelism.

This Thesis proposes a novel *Multicore Application-Specific Instruction Set Processor (MCASIP)* co-design methodology that exploits parallel programming languages as the application input format. In the methodology, the designer can explicitly capture the parallelism of the algorithm and exploit specialized instructions using a parallel programming language in contrast to being on the mercy of the compiler or the hardware to extract the parallelism from a sequential input. The Thesis proposes a multicore processor template based on the Transport Triggered Architecture, compiler techniques involved in static parallelization of computation kernels with barriers and a datapath integrated hardware accelerator for low overhead software synchronization implementation. These contributions enable scaling the customized processors both at the instruction and task levels to efficiently exploit the parallelism in the input program up to the implementation constraints such as the memory bandwidth or the chip area. The different contributions are validated with case studies, comparisons and design examples.

PREFACE

When I started to work on the *TTA-based Co-design Environment (TCE)* project about ten years ago as a research assistant, I could not have imagined what an interesting journey of learning the then part-time job would start.

I would like to thank my supervisor Professor *Jarmo Takala* for this challenging work opportunity, his guidance, and trusting me on my research ideas. Furthermore, I want to thank him and the two reviewers, Professor *Georgi Gaydadjiev* and Professor *Johan Lilius* for their invaluable comments that helped me to improve this Thesis.

The results presented in this Thesis would not have been possible without the many hard-working smart colleagues that have contributed code and ideas for TCE since the beginning of the project. I have a great deal of gratitude for all of you, but I am afraid there are too many to mention separately without still forgetting someone important.

However, I want to specifically thank the main collaborators in the work for this Thesis, which was conducted mostly in the past four years. I am grateful for the opportunity I had to work with *Carlos Sánchez de La Lama*, M.Sc., who brought a “fresh energy boost” and new ideas to TTA research after his first post to our mailing lists about four years ago. I appreciate the work on the compiler backend of *Heikki Kultala*, M.Sc., and *Vladimír Guzman*, M.Sc. Heikki’s remarkable ability to remember many details of commercial off-the-self processor architectures has been a useful source of information for me numerous times. I thank *Otto Esko*, M.Sc., Dr. *Pablo Huerta*, and Dr. *Erno Salminen* for providing their FPGA and hardware expertise without which the experiments presented in this thesis would have been much less concrete.

Now that the toolset we have developed all these years is known to have been used to design processors for silicon implementation, I must thank the numerous people who have helped to stabilize TCE by using it for various ASIP design cases and re-

porting bugs and inefficiencies. Again, there are too many to mention, but from the “pioneers” I must bring up Prof. *Olli Silvén* and his research group in University of Oulu, especially researcher Dr. *Jani Boutellier*. I thank you for extensive testing and validating the work in various design cases and for spreading the TTA/ASIP awareness especially in the Oulu area. In this context I must also thank *Heikki Berg*, M.Sc and his research group for the chance to evaluate and test the techniques presented in this Thesis in our collaboration projects of past couple of years, and for bringing an “industrial point of view” to the research.

I am most grateful for the funding sources that enabled me to focus on the interesting research topic. In the past four years, I have received financial support from the *Academy of Finland*, *Ulla Tuominen Foundation*, *Tuula and Yrjö Neuvo fund* and *Nokia Foundation*.

I am thankful for the solid foundations provided by my siblings and parents, especially my mother who has constantly encouraged me to finish this Thesis project sooner than later. Finally, I thank my dear wife *Terhi* who has provided me a steady source of warmth and happiness during these years. Thank you for tolerating my sometimes too long work days that in part made finishing this Thesis possible.

Tampere, September, 2012

Pekka Jääskeläinen

TABLE OF CONTENTS

<i>Abstract</i>	i
<i>Preface</i>	iii
<i>Table of Contents</i>	v
<i>List of Figures</i>	ix
<i>List of Tables</i>	xi
<i>List of Abbreviations</i>	xiii
1. Introduction	1
1.1 Research Objectives	3
1.2 Main Contributions	4
1.3 Author's Contribution and Collaboration	4
1.4 Thesis Outline	5
2. Parallel Computing	7
2.1 Expressing Parallelism in Software	7
2.1.1 Synchronization Primitives	11
2.1.2 Open Computing Language	12
2.2 Parallelism in Processor Architectures	14
2.2.1 Instruction Level Parallelism	14
2.2.2 Data Level Parallelism	16
2.2.3 Task Level Parallelism	18
2.3 Customized Parallel Processors	20

3.	<i>Exposed Datapath Static Multi-Issue Architectures</i>	23
3.1	Transport Triggered Architectures	23
3.2	Benefits of TTA in Single Program Multiple Data Computation	25
3.3	Review of Work in Exposed Datapath Architectures	28
4.	<i>Customizable Processor Design Methodology for Parallel Programs</i>	33
4.1	Related Work	35
4.2	Parallel Processor Design Flow	36
4.2.1	Software Development	36
4.2.2	Single Core Customization	38
4.2.3	Multicore Exploration	41
5.	<i>Customizable Parallel Processor Template</i>	43
5.1	Related Work	43
5.2	Customizable Multicore Processor Template	45
5.3	Memory Model	46
5.3.1	Shared Default Data Memory configuration	48
5.3.2	Local Default Data Memory configuration	49
5.4	Software Stack	49
5.5	Thread Scheduling	51
5.5.1	Dthreads: a Distributed Threading Library	51
5.5.2	Requirement: Low Instruction Memory Footprint	53
5.5.3	Requirement: Autonomous Execution	53
5.5.4	Requirement: Minimal Number of Shared Memory Accesses	54
5.5.5	Requirement: Scalability	54
6.	<i>Open Computing Language Support</i>	57
6.1	Related Work	57

6.2	Open Computing Language in Hardware/Software Co-Design	59
6.3	Compiling OpenCL for Static Customized Processors	59
6.3.1	Standalone and Hosted Setups	60
6.3.2	Parallelizing Work-Items	60
6.3.3	Work-item Chaining Algorithm	63
6.3.4	Instruction Level Parallelization of Work-Groups	65
6.3.5	Custom Operation Support	67
7.	<i>Hardware Accelerated Synchronization</i>	69
7.1	Related Work	69
7.2	Datapath Integrated Lock Unit	71
8.	<i>Experiments</i>	77
8.1	Experiment Environment	77
8.2	TTA vs. SIMT in OpenGL Shader Execution	79
8.2.1	The Shader Example	79
8.2.2	Benchmarking TTAs against a SIMT GPU	81
8.3	Single Core Customization for OpenCL Kernels	84
8.3.1	AES OpenCL Implementation	85
8.3.2	Instruction-level Parallelism	86
8.3.3	Custom Operations	87
8.3.4	Single Core Customization Using Standard Components	88
8.4	Task Level Scalability	90
8.5	Barrier Synchronization Overheads	92
8.5.1	The Benchmark Processor	93
8.5.2	Benchmark Program	94

9. <i>Conclusions</i>	97
9.1 Main Results	97
9.2 Future Work	99
<i>Bibliography</i>	101

LIST OF FIGURES

1	A parallel for-loop computing a vector addition using the OpenMP pragmas in a C program.	8
2	A snippet of a program with instruction-level parallelism.	9
3	Vector addition in ANSI C.	9
4	Spin lock and unlock implementations using the atomic Compare-And-Swap instruction.	11
5	Vector dot product in OpenCL C.	13
6	Illustration of how the instruction bit fields control a SIMD datapath.	16
7	Parallel instructions controlling an ILP architecture.	17
8	Example of a TTA processor datapath.	24
9	An operation scheduling example showing MIMD scheduling freedom.	27
10	Resource overcommitting example.	28
11	Example of a heterogeneous MPSoC with multiple multicore TTAs.	34
12	The toolset supported design flow of parallel customized processors.	37
13	The single core customization flow.	38
14	Two different configurations of the memory model depending on the placement of the default data address space.	47
15	The software stack of TCEMC.	50
16	Dot product using the proposed Dthreads API.	52
17	Simple example on work-item chaining.	61
18	Work-item chaining with barriers.	63

19	Work-item code replication algorithm.	64
20	Effect of avoiding register reuse in work-item register allocation. . .	66
21	Example of using a custom operation inside an OpenCL kernel in a portable way.	68
22	An example hardware design of the lock unit.	73
23	Two spin lock implementations using the lock unit instructions. . . .	74
24	Two barrier implementations using the lock unit instructions.	75
25	Overview of the TCE compiler internals.	78
26	Wood appearance fragment shader.	80
27	Output image for the shader benchmark.	82
28	Cycle counts for 32-wide workgroup execution.	83
29	Effect of the work-group size on the cycle count for running the kernel.	84
30	The OpenCL AES encryption implementation.	86
31	The effect to the minimum processor cycles from increasing the core count.	92
32	The organization of the benchmarked multicore ASIP using the lock unit through a SYNC function unit.	93
33	The microbenchmark that “stress tests” the shared memory overheads of the barrier alternatives.	94
34	The speedups obtained by using the <i>fast</i> barrier in comparison to the <i>basic</i> barrier version.	95

LIST OF TABLES

1	Required lock registers and shared memory read and write accesses for the alternative lock unit based synchronization implementations.	76
2	Resources in the TTAs used in the shader benchmark.	82
3	Effects of software bypassing on register file pressure in the shader example.	84
4	Effect of the parallel Work-Item count to the cycle count.	86
5	Resources in the AES TTA processor.	87
6	Speedups from using custom operations.	88
7	Processor resources in the FFT experiment.	89
8	Execution time results for the FFT core.	89

LIST OF ABBREVIATIONS

AES	Advanced Encryption Standard
ALM	Adaptive Logic Module
ALU	Arithmetic Logic Unit
API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
ASIP	Application-Specific Instruction-set Processor
ASP	Application-Specific Processor
ASSP	Application-Specific Standard Product
CAS	Compare-And-Swap (instruction)
CFG	Control Flow Graph
CISC	Complex Instruction Set Computer
CMP	Chip Multiprocessor
DILU	Datapath Integrated Lock Unit
DLP	Data-Level Parallelism
DSP	Digital Signal Processing or Digital Signal Processor
EDA	Electronic Design Automation
FFT	Fast Fourier Transform
FPGA	Field-Programmable Gate Array

FSM	Finite State Machine
FU	Function Unit
GLSL	OpenGL Shading Language
GPU	Graphics Processing Unit
GPGPU	General-Purpose computing on Graphics Processing Units
HLS	High-Level Synthesis
ILP	Instruction-Level Parallelism
ISA	Instruction Set Architecture
SDDM	Shared Default Data Memory
LDDM	Local Default Data Memory
LLVM	The LLVM Compiler Infrastructure (originally Low Level Virtual Machine)
LSU	Load-Store Unit
LTT	Local Thread Table
LU	Lock Unit
LUT	Lookup Table
MCASIP	Multicore Application-Specific Instruction-set Processor
MPSoC	MultiProcessor System-on-Chip
NoC	Network-on-Chip
OpenCL	Open Computing Language
OpenGL	Open Graphics Libray
OTA	Operation Triggered Architecture
PC	Personal Computer

RA	Register Allocator
RF	Register File
RISC	Reduced Instruction Set Computer
RMW	Read-Modify Write (instruction)
RQ	Ready Queue
SDDM	Shared Default Data Memory
SDR	Software-Defined Radio
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Threads
SMP	Symmetric MultiProcessor or Symmetric MultiProcessing
SoC	System-on-Chip
SPMD	Single Program Multiple Data
SPM	Scratchpad Memory
STT	Shared Thread Table
TCE	TTA-based Co-design Environment
TCEMC	TTA-based Co-design Environment for MultiCores
TLP	Task-Level Parallelism or Thread-Level Parallelism
TTA	Transport Triggered Architecture
VLIW	Very Long Instruction Word (architecture)
WG	Work-Group (OpenCL terminology)
WI	Work-Item (OpenCL terminology)

1. INTRODUCTION

Contemporary embedded systems can include *Multiprocessor System-on-Chips (MPSoC)* which integrate multiple processors and other peripherals on a single chip. Commercial examples of MPSoC platforms in 2012 include the OMAPTM [1] family from Texas Instruments, the SnapdragonTM family from Qualcomm [2], and the NVIDIA[®] Tegra[®] [3] family. Each of these platforms include multiple processors to be used for, e.g., general-purpose computing, graphics acceleration and *Digital Signal Processing (DSP)*.

In contrast to general-purpose multiprocessors with a large expected set of different applications to execute, the design of a processor targeted to an embedded system can be customized to the requirements of an application domain such as video coding or *Software Defined Radio (SDR)*. This can be seen in the special instruction sets of the processors or in the choices of the used processor architecture styles. For example, the OMAP 4 SoCs of Texas Instruments include a DSP core (C64x) designed for accelerating contemporary multimedia codecs. It has an instruction set optimized for fixed-point computation, and a statically scheduled multi-issue architecture which is expected to perform suboptimally for control-intensive and dynamic workloads, but is able to provide sufficient computation performance for algorithms used in mainstream multimedia codecs. [4]

The number of new hardware design engineering efforts has increased with the improvements in reconfigurable logic such as *Field-Programmable Gate Arrays (FPGA)*. The designs targeting FPGA implementation can include so called *soft cores* where per-design processor customization is commonplace. Thanks to the flexibility of the implementation platform, soft core architectures can be freely customized according to the software running in them, which is their main benefit in comparison to their “hard core” alternatives.

The need for fast time to market of new processor designs targeted for ASIC imple-

mentation of new embedded SoCs or softcores in FPGAs calls for a rapid *co-design methodology* of the included application (domain) specific processors.

More generally, *Electronic Design Automation (EDA)* tool vendors are developing methodologies that expand the usability of their offerings from the hands of relatively low number of hardware-skilled engineers also to the hands of software engineers or algorithm designers. The aim is to provide easy-to-use mechanisms for *High Level Synthesis (HLS)* of hardware starting from algorithm descriptions in programming languages instead of hardware design languages. [5]

The C language has been commonly used as an input to HLS and hardware/software co-design flows [5]. The C program is used to drive a design flow to produce, automatically or manually, hardware accelerators or customized processors executing the described algorithm faster than an off-the-shelf processor would. However, as C is a sequential language, its capability to express parallel operations to utilize naturally parallel hardware constructs is far from optimal, leading to hardware designs with limited parallel resource usage. In contrast, when utilizing a parallel programming language as an input, wider hardware design space can be explored to produce designs that exploit multiple levels of hardware parallelism [6, 7].

This Thesis proposes a *Multicore Application-Specific Instruction Set Processor (MCASIP)* co-design methodology that exploits parallel programming languages as the application input format. In the proposed methodology, the designer can explicitly capture the parallelism of the algorithm and exploit special instructions using a parallel programming language in contrast to being on the mercy of the compiler or the hardware to extract the parallelism from a sequential input.

The multicores produced using the proposed design methodology can be used as replacements to fixed function hardware accelerators and as more general “domain-specific“ co-processors. The design methodology is independent of the final implementation technique, thus is suitable both for soft core designs in FPGAs and hard core designs implemented as ASICs or ASSPs (application-specific standard products).

The Thesis proposes several enabling components required for customized multicore design: a multicore processor template tailored for *Single Program Multiple Data (SPMD)* programming languages, compiler techniques involved in static instruction-level parallelization of SPMD computation kernels with barriers, and a simplified

instruction set for low overhead software synchronization implementation. These contributions enable the designer to scale the customized processors both at the instruction and task levels to exploit the parallelism in the input program up to the implementation constraints such as the memory bandwidth or the chip area. The proposed techniques are validated with case studies, comparisons and design examples.

1.1 Research Objectives

The higher-level objective of the research conducted for this Thesis was to develop a methodology and its related techniques for customized parallel processor design. The scope of the methodology under research was limited to the case where the input is a program that expresses parallel execution in the *Single Program Multiple Data (SPMD)* style, and the output is an homogeneous parallel processor which can be potentially used in a larger heterogeneous system as an accelerating building block.

An objective within the research was to study the benefits of the *Transport Triggered Architecture (TTA)* in providing finer-grain parallelism (TTA is an exposed datapath static multi-issue architecture that has been proposed as a more scalable VLIW [8]). More generally, a research goal was to identify benefits of the *Multiple Instruction Multiple Data (MIMD)* programming model of static multiple issue architectures in comparison to the common *Single Instruction Multiple Data (SIMD)* model.

In addition to the single core customization aspects, a research goal was to propose a processor template that enables task level parallelism customization with minimal designer effort. The scope of this research was limited to shared memory architectures. Within this scope, efficient hardware accelerated synchronization between the cores was identified as a main topic to study in more detail.

In order to map the parallel software to the parallel processor hardware, a research objective was to produce static parallelization techniques for SPMD programs. In this part of the research, the scope was limited to programming languages with kernels that include barrier synchronization. The *Open Computing Language* [9] was used as an example input language in that work.

1.2 Main Contributions

The main contribution of this Thesis is a methodology for producing customized parallel processors that can support the parallelism available in parallel programs at multiple granularities. Within this work at least the following novel contributions are presented:

- A customizable multicore processor template tailored for scaling parallel processor resources at multiple levels. Part of this work was a comparison of the TTA to other parallel datapath design styles in the context of instruction or data level parallel workloads.
- Techniques for efficient scheduling of parallel barrier-synchronized kernels to statically scheduled instruction-level parallel datapaths.
- A simplified memory system isolated data path integrated hardware lock unit to reduce the overheads of the common software synchronization primitives and their impact to the shared memory traffic.

1.3 Author's Contribution and Collaboration

This Thesis is based on previous publications [10–13] in which the Thesis Author has acted as the main or a second author with major contributions. The Thesis contains edited and augmented text from these publications.

The author has been heavily involved in the design, implementation and team coordination effort of the *TTA-based Co-design Environment (TCE)*, a software toolset laying the foundations for the work presented in this thesis. For the TCE software implementation contributions, the author has been the main programmer of the instruction set simulator [14], the fundamental multicore features, the TCE OpenCL support, and parts of the retargetable compiler chain.

The collaboration with Mr. Carlos Sánchez de La Lama of Universidad Rey Juan Carlos (URJC), Madrid produced many of the ideas in this Thesis. This collaboration led to the research on using TTAs for highly parallel graphics-style workloads. Results from this work were published in [10]. In addition, the fundamental ideas on

how to exploit the (then recently released) OpenCL standard to extract abundance of static instruction-level (and data-level) parallel code by means of parallelizing code from multiple kernel instances were produced with Mr. Sánchez de La Lama. The initial results from this work were published in [12, 15]. For these publications, Dr. Pablo Huerta of URJC contributed the AES benchmark. The implementation work of the static OpenCL C kernel parallelization techniques was conducted in co-operation with Mr. Sánchez de La Lama.

The proposed multicore processor template was largely inspired by contemporary GPU designs which were also used as basis for the platform and memory models of the OpenCL standard. However, generalizing and adapting the template to a customizable processor design methodology was proposed by the Author. Implementation of the template in an architecture description language, and the first experiments were published in [11].

Finally, the author's research on reducing the effects of synchronization primitives to the shared memory traffic in customized manycore designs with a shared address space lead to the Author's idea of the datapath integrated lock unit, published in [13]. The first hardware design of the lock unit was produced by Dr. Erno Salminen and Mr. Otto Esko who also produced an FPGA prototype of the 48-core TTA processor used in the synthetic synchronization benchmark presented in Section 8.5.

1.4 Thesis Outline

The rest of the Thesis is organized as follows. Chapter 2 revises the concepts related to parallel embedded computing that are referred to in the rest of the Thesis. Further background is given in Chapter 3 where a class of processor architectures called "the exposed datapath architectures" is defined. In addition to studying the other work in exposed datapath architectures, the *Transport Triggered Architecture* which is the single core exposed datapath processor template used in the proposed design methodology is discussed in more detail in the chapter.

Chapter 4 describes the high-level of the proposed parallel processor customization methodology in which new processors are instantiated from a processor template detailed in Chapter 5.

While the processor template is designed to support multiple parallel programming

language standards, support for the OpenCL standard was the first priority in this research due to its increasing popularity. Its adaptation to the design flow and the related compiler techniques are described in Chapter 6.

The synchronization instructions that use a simplified lock hardware to minimize synchronization overheads in the thread execution are described in Chapter 7.

Benchmarks and experimental designs are presented in Chapter 8. Finally, the Thesis is concluded in Chapter 9 along with examples of ideas for future work.

2. PARALLEL COMPUTING

This Thesis refers to several concepts and techniques related to parallel computing on embedded devices. This chapter shortly revisits these concepts. The definitions can be started with the concepts of “parallelism” in comparison to the concept of “concurrency”. A *concurrent* program is defined to be a set of processes of execution that might communicate and synchronize with each other. *Parallelism* can be described as executing multiple operations from a program simultaneously for improved execution time, or for other benefit such as reduced power consumption. A *concurrent* program, on the other hand, can also be executed in an interleaved manner without actual simultaneous computation taking place, thus can be seen as a way to structure programs [16]. In the rest of this thesis, the term *parallelism* is used both when writing about programming languages that can be used to express parallel execution (with the goal of improving performance) and also when writing about the processing hardware that executes operations simultaneously.

The rest of the chapter visits concepts from the point of view of the parallelism support in software and processor microarchitecture techniques that implement parallel execution at multiple levels of granularity.

2.1 Expressing Parallelism in Software

Experience has shown that it is very difficult, computationally expensive, and often just plain impossible to automatically extract parallelism from sequential programs [17]. Even if the algorithm at hand was inherently parallel, a sequential programming language does not provide the expressiveness to communicate it to the compiler, thus it forces the compiler to extract the parallelism by means of complex compiler analysis. Therefore, exploiting parallel processor resources efficiently requires the use of parallelism programming languages and libraries [16].

```
#pragma omp parallel for
for(i = 0; i < width; i++) {
    result[i] = A[i] + B[i];
}
```

Fig. 1: A parallel for-loop computing a vector addition using the OpenMP pragmas in a C program.

Several parallel programming languages have been proposed. Their adoption has been hindered by the additional difficulty of parallel programming in comparison to sequential programming [16]. Afterall, the design of many parallel languages and libraries has been driven by the characteristics of the platforms at hand with less consideration to the programmer-friendliness [18]. For example, the POSIX threading library *pthread* [19] allows describing task level parallelism in C programs using shared memory communication. It provides means to spawn “threads of execution” in a single process which shares memory, thus provides a low-level API for describing *Thread Level Parallelism (TLP)*.

Describing massively parallel programs, such as ones that exploit *loop level parallelism*, using lower level threading libraries such as *pthread* is burdensome. Therefore, more programmer-friendly APIs such as OpenMP [20] have been proposed to ease this task.

For example, the OpenMP *parallel for* construct (see Fig.1 for an example) allows describing *Single Program, Multiple Data (SPMD)* parallelism. In SPMD, the same program code (in this case, instructions forming a loop iteration) is executed on parallel processing elements over different sets of data (different locations in the input arrays). This style of parallelism is called *Data Level Parallelism (DLP)*. Programs with DLP can be mapped to the computational resources of different processor architectures in multiple ways as can be seen later in this chapter.

Instruction Level Parallelism (ILP) can be seen as a superset of DLP. It loosens the restriction of *Single Instruction stream Multiple Data streams (SIMD)* performing the same operation to parallel data to *Multiple Instructions streams Multiple Data streams (MIMD)*. That is, a processing platform that can support ILP can execute multiple different operations to multiple different data simultaneously, therefore, widening the scope of types of programs that can be parallelized. Figure 2 presents a

```
sum = a + b;  
diff = a - b;
```

Fig. 2: A snippet of a program with instruction-level parallelism.

simple program that can be parallelized using ILP-capable hardware that can execute both the addition and the subtraction at the same time.

The examples of OpenMP and pthreads were given as ways to describe parallelism in the C language which is popular especially in the embedded domain. The C language itself is an example of an *imperative language* and is sequential in nature. That is, the compiler or the processor hardware has to extract the finer granularity parallelism from the program descriptions as there is no means to explicitly express parallel computation at the statement level.

An example of compiler analysis that is needed when automatically parallelizing C programs is the alias analysis. The example in Fig. 3 shows a basic case which already makes the compiler analysis non-trivial. In this case, the vector addition is a function which takes in three pointers to the input buffers and a result buffer. In order the compiler to be able to parallelize or reorder instructions from multiple iterations of the inner loop, it has to prove that the writes to the *result* buffer do not also write to some locations of *A* or *B* because of overlapping memory regions in the pointer buffers. Without *interprocedural alias analysis* it is impossible to know at compile time if the pointers given as the function arguments indeed point to overlapping regions in memory. [21]

```
void vec_add(float *A, float *B,  
            float *result, int width) {  
    int i;  
    for(i = 0; i < width; i++) {  
        result[i] = A[i] + B[i];  
    }  
}
```

Fig. 3: Vector addition in ANSI C.

This particular case was addressed in ISO C99 [22] with the introduction of the *restrict* qualifier. The qualifier can be used to mark pointers to be such that their memory region is accessed only through that pointer in the scope of the pointer declaration. This is a simple way to communicate hints to the compiler for improving parallelism. However, it is not a general solution as it applies only to the simple case. It cannot be used to improve parallelization of more complex access patterns such as independent accesses to different locations in the buffer or accesses to arrays with indices read from an another array, etc.

Functional languages describe the programs as stateless function calls which result in parallelizable computation trees. While functional programming is convenient for software engineers with a mathematical background, imperative programming, especially with the C-based languages, is still used by the majority of software engineers [23].

An important consideration with parallel programming is the means to communicate between the “units of computation” (later referred to as *threads*). In case of OpenMP and pthreads, for example, a random accessible shared memory is used to transfer data between the threads. Mutual accesses to the shared data are assumed to be synchronized using primitives such as *locks* and *semaphores*. Otherwise, data corruption can occur when two or more parallel threads modify the same shared memory locations at the same time (race condition).

Another prevalent communication method is “message passing”. In this method, explicit “messages” that contain the data are sent between the threads which can run in the same multiprocessor or even in different computers in a networked cluster. The explicit synchronization is avoided as the threads do not modify shared data directly during the communication.

The benefits of shared memory communication include the potential to reduce copying of data which is required in the message passing implementation. Additionally, *load balancing* is easier when there is a ready queue of threads in the shared memory from which to fetch threads to an idle processor. The main drawback is that the implementation of the shared memory hierarchy in hardware becomes expensive with larger number of processor cores. Therefore, a common solution is to use both methods: message passing for higher granularity communication (e.g., between computers in a cluster or separate processors in a heterogeneous MPSoC) and a shared

```
spin_lock(lock_var* A) {                               unlock(lock_var* A) {
    lock_var old;                                       *A = 0;
    do {                                               }
        old = CAS(A, 0, 1);
    } while (old == 1);
}
```

Fig. 4: Spin lock and unlock implementations using the atomic Compare-And-Swap instruction.

memory model inside a multicore for faster pointer based communication between tightly coupled threads [24].

2.1.1 Synchronization Primitives

Locks, barriers, and semaphores are basic synchronization primitives used to orchestrate the execution of a multithreaded program. Lock variables typically reside in shared memory and atomic *Read-Modify-Write (RMW)* instructions are needed for manipulating them without corruption. Locks are used to perform mutual exclusion to ensure critical sections that manipulate shared data structures are executed only by one thread at a time.

A common synchronization primitive heavily used in system code is the *spin lock*. It performs busy wait until the *lock variable* is marked “free” (usually by writing 0 to the variable). It can be implemented with a loop that “spins” until it manages to write 1 to the target lock variable before other threads do. The atomicity of the lock acquiring operation can be implemented with RMW instructions in the instruction set.

In case a spin lock is implemented using RMW instructions, the spin lock loops until it manages to swap a 0 to a 1 at address *A*, meaning it obtained the lock without it being locked before by another thread. The basic primitives can be implemented using the atomic *Compare-And-Swap (CAS)* operation as shown in Fig. 4. Note that the unlock can be implemented with a simple store to the lock address location in case the program is known to be well-behaved and there is no need to check for lock ownership.

Another common synchronization primitive used especially in SPMD programs is the *barrier*. Barriers are used to synchronize the control flow of all threads co-operating in the execution of the multithreaded program. The semantics of the barrier is to wait

at the barrier call site until all other threads have reached the barrier. After all threads have reached the barrier, the execution of the threads continues freely. The simplest barrier implementations use counter variables that are protected with locks [25]. They count how many threads have reached the barrier and block the waiting threads until the counter reaches the total number of threads.

2.1.2 Open Computing Language

In December 2008, a joint effort between major companies and other interest groups related to parallel heterogeneous programming led to the standardization effort of *Open Computing Language (OpenCL)* [26]. Albeit the background of OpenCL is clearly in the *General-Purpose computing on Graphics Processing Units (GPGPU)* community, and its version 1.2 highly resembles the proprietary CUDA language from NVIDIA [27], the aim of OpenCL is to become a universal programming standard for platforms with heterogeneous processing devices such as GPUs, CPUs and DSPs.

OpenCL programs structure the computation into *kernels* defined in OpenCL C kernel language, and specify that there shall be no data dependencies between the “kernel instances” (*work-items*, analogous to loop iterations) by default. The implementation is free to execute code from the different kernel instances sequentially, in parallel, or in an interleaved fashion, as long as the explicit synchronization primitives (e.g., *work-group barriers*) present in the kernel descriptions are respected.

The example OpenCL C kernel in Fig. 5 computes a dot product for a single location in two input buffers *a* and *b*, and places the result in the result buffer *c*. This kernel can be executed on different sized vectors in parallel, using as many *work-items* as there are elements in the vector. *get_global_id(0)* is used to query the index of the work-item in the *global index space* which in this case maps directly to the index in the buffers.

One point of difference to standard C notation in this example is the use of the *global* qualifier in the kernel arguments. This is used to mark the pointers to point to buffers in *global memory*. Other disjoint explicitly addressed memory spaces in OpenCL C include the *local memory* visible to single *work-groups* (groups of work-items within the global index space that can synchronize with each other) at a time, the *private*

```
kernel void
dot_product (global const float4 *a,
             global const float4 *b,
             global float *c)
{
    int gid = get_global_id(0);

    c[gid] = dot(a[gid], b[gid]);
}
```

Fig. 5: Vector dot product in OpenCL C.

memory visible only to single *work-items*, and the *constant memory* for storing read-only data.

The OpenCL runtime API (a separate API in standard C) is used to launch kernels and data transfer commands in one or more *compute devices* with *event* synchronization. Portability of OpenCL programs across a wide range of different heterogeneous platforms is achieved by describing the kernels as source code strings which are then explicitly compiled using the runtime API to the targeted devices.

While the OpenCL standard provides an extensive programming platform for portable heterogeneous parallel programming, the version 1.2 of the standard lacks in means to achieve *automatic performance portability*. It is burdensome to write an OpenCL program that performs efficiently on multiple heterogeneous computation platforms [28]. Recently it has been proposed that the performance portability of OpenCL could be improved with an introduction of a “virtual device” abstraction or a higher-level programming layer that would hide the device-specific optimization problems from the programmer [29, 30]. In addition, *auto-tuning* has been used to improve the performance portability of the OpenCL programs [29, 31].

OpenCL is an attractive candidate to act as an input for a customized parallel processor design flow such as the one proposed in this Thesis because it allows explicit definition of parallel execution at multiple granularities. Operations on OpenCL C vector data types express data level parallelism within a single work-item and the work-groups themselves implicitly describe data parallel execution across multiple work-items that is explicitly synchronized with barriers. Task level parallelism can be described at the higher level with parallel execution of multiple kernels and work-groups.

2.2 Parallelism in Processor Architectures

Parallelism is a means to improve the performance of a program without increasing the clock frequency. Processor architectures expose parallel computation resources to programs in different ways at the different granularities which are discussed in the following.

2.2.1 Instruction Level Parallelism

Pipelined execution in an in-order scalar processor can be seen as a way to exploit instruction-level parallelism by overlapping different *stages* of the instruction execution from multiple instructions. In such a case, data dependencies between instructions might cause pipeline stalls which can be reduced by compiler *instruction scheduling* [32], a compiler optimization that places independent instructions after each other to promote the overlapping for decreased pipeline idle time. Real parallel *execution* of multiple instructions requires, by definition, multiple parallel function units (*multi-issue*) [33].

Whether it is the responsibility of the compiler or the processor hardware to exploit the instruction-level parallelism places multi-issue processors somewhere between the classifications of *dynamic* and *static architectures*. Dynamic multi-issue architectures rely on processor hardware to extract and exploit the parallelism in a sequential input program while static multi-issue architectures rely on the compiler. This classification is not binary, that is, the real processor architectures have different levels of reliance on the compiler.

Superscalar processors can exploit hardware techniques such as hardware data dependency detection, out-of-order execution, and speculation to issue multiple instructions from the sequential program input to the multiple function units in the processor pipeline. The main benefit of such dynamic exploitation of parallelism include binary compatibility: old sequential programs run in the newer versions of the same architecture family without program recompilation. The main drawback is the added complexity in the control hardware to recognize the parallel instructions at runtime which leads to increased chip area and power consumption. [34]

Very Long Instruction Word (VLIW) architectures belong to the class of static multi-issue architectures. VLIW designs go towards the idea of *exposed datapath archi-*

itectures (discussed more thoroughly in the next Chapter) where the compiler (or the programmer) is given the responsibility to describe which instructions should be executed in parallel. The instruction latencies are also exposed to the programmer in VLIW architectures. Instructions have to be scheduled in such a way that they do not read their results too early or too late. The benefit of the VLIW style is clearly the simplified control logic which enables potentially very wide instruction issue widths which is achieved by placing the complexity to the compiler. In addition, the instruction windows of dynamic hardware are more limited in size while compilers see the whole function when they look for parallelism, which can lead to more parallelization opportunities. [35,36]

How much weight should be put to the drawback of the binary incompatibility issue of static architectures depends on the usage target. In general-purpose computing where source codes of all input programs are not commonly available, it naturally is problematic [37]. On the other hand, with embedded processors, especially in case of customized architectures where the hardware and software is often co-designed using an automatically retargetable toolchain, the harm is diminished. Moreover, with the advent of programming standards targeted to platforms with heterogeneous devices, compilation from an intermediate language to the target instruction set has been made a step in the application execution [9].

Another consideration between dynamic and static architectures is how well they can deal with dynamism in the input programs. For example, dynamic architectures can react better to variable length runtime events such as those from cache accesses. An out-of-order processor can hide the cache misses more naturally than a static architecture of which instruction execution is scheduled according to static instruction latencies. [38]

Branches in the program are another source of dynamism. Dynamic architectures can use techniques such as dynamic branch prediction [39] and speculation [34] to schedule instructions from a predicted path of the program ahead of time. Similar results can be achieved using the compiler by means of predicated execution (e.g., using if-conversion [40]) or speculative code motion (e.g., with trace scheduling [41]).

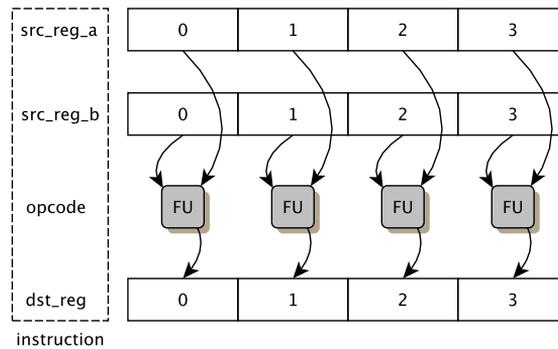


Fig. 6: Illustration of how the instruction bit fields control a SIMD datapath. SIMD instructions can refer to wide vector registers with subwords instead of defining the independent elements, thus compacting the instruction word. The opcode field controls all the four parallel function units (FU). In this case the SIMD instruction expresses four parallel operations on independent elements 0..3 of the vector registers.

2.2.2 Data Level Parallelism

Data level parallelism is a more restricted form of parallelism than ILP. In contrast to ILP where multiple different parallel operations can be applied to multiple different data, DLP is limited to same operation applied to multiple data.

A major benefit of limiting the parallelism to DLP in the processor architecture is requiring less instruction bits per executed parallel operation. In case of DLP, multiple parallel operations can be described by using a single opcode field and vector register identifiers (see Fig. 6). In comparison, the ILP architectures each operation requires a separate opcode field and one or more scalar register identifiers (see Fig. 7).

Single core architectural support for data-level parallel programs is provided by means of vector instruction set extensions (e.g., Intel[®] Advanced Vector Extensions [42] and the ARM[®] NEON[™] general-purpose SIMD engine [43]) or with more SIMD-oriented instruction sets (e.g., the Synergistic Processor Unit in the Cell Broadband Engine [44]).

In order to efficiently utilize a SIMD datapath in SPMD-style execution, parallel operations are extracted from multiple SPMD program “threads” or loop iterations with loop vectorization. The matching operations from the different iterations are scheduled to execute in lockstep in the *vector lanes* (parallel scalar function units controlled by the SIMD instruction). This results in parallel execution of N iterations using a single SIMD instruction stream, where N is the *issue width* of the SIMD

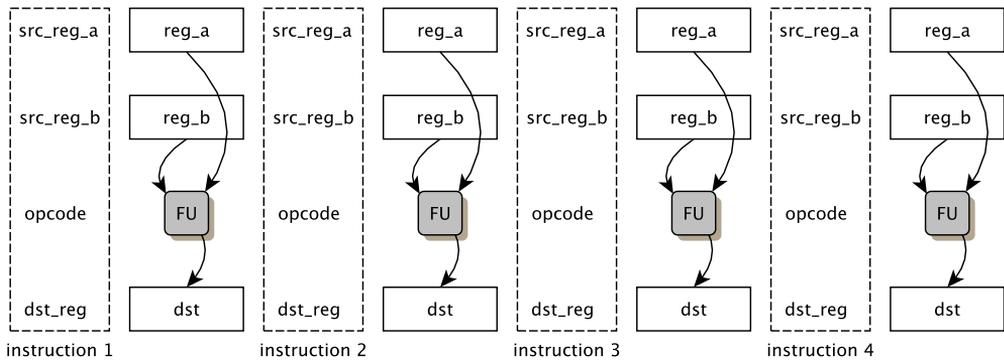


Fig. 7: Four parallel instructions controlling an ILP architecture with four parallel function units (FU). The instructions specify scalar operations. Each instruction can define a separate opcode for the computed data. The drawback is the program memory overhead of the four separate instructions.

hardware.

The “lockstep SIMD” style of execution is efficient as long as the iterations do not have diverging control paths such as an if-else-structure of which execution path depends on the iteration (thread id). In case of diverging branches, some of the vector lanes should be disabled in case all of the parallel iterations are not executing the branch at hand. This problem is discussed in more detail in the next chapter.

In some SIMD instruction sets, vector masks provide architectural support for diverging branches. In such cases, some of the vector instructions in the instruction set include a mask register operand which communicates the set of lanes that are enabled, thus providing the ability to “squash” the execution of the diverging iterations. Intel’s Advanced Vector Extensions, for example, support load/store masking which can be used to squash the loads and stores of the disabled lanes. In addition, its *blend instructions* can be used to pick selected elements from input vectors to implement conditional branches as conditional data selection. [42]

Contemporary manycore processors originally designed for graphics processing, which are nowadays used also for general-purpose high performance computing, have their architectures optimized for SPMD programs. [27, 45]

An example of a processor architecture feature tailored for SPMD programs is *Single Instruction Multiple Thread (SIMT)*. SIMT is a term used mainly by NVIDIA. It differs from the lock-step SIMD execution in its ability to handle diverging branches in

the program code transparently. SIMT hardware includes a separate program counter per each function unit that is used to disable the execution in case the processor is currently executing an instruction from a branch that should not be taken by the SPMD thread mapped to the unit. In the SIMT case, instead of decoding statically parallel wide SIMD instructions, the processor hardware inputs a single thread/iteration of the SPMD program which is distributed to the parallel scalar cores at processor runtime. [27, 46, 47]

2.2.3 Task Level Parallelism

Instruction scheduling, implemented by either the compiler, the processor microarchitecture, or both, can be used to hide latencies in the instruction pipeline. However, parallelism at the instruction-level has its limits. Most programs do not have enough independent instructions to hide very long latencies (hundreds or even thousands of clock cycles) that occur with cache misses. Furthermore, such latencies are usually dynamic, thus they cannot be efficiently alleviated with static compiler instruction scheduling. [36]

Multithreading is a technique that can be used to hide long latencies with useful computation by switching the execution to other independent threads of execution whenever long latencies occur (*coarse grained* or “*block*“ *multithreading* [48]) or even at every cycle (*fine grained multithreading* [49]) for hiding also shorter latencies.

In order to benefit from *Task Level Parallelism (TLP)* (also called *Thread Level Parallelism*), the program must be structured in independent threads of execution. The idea in TLP is to utilize the execution resources in the processor to advance other parts of the program (or even a completely different program running in another process) while another thread waits for a long latency operation to be served. Fast context switches require duplication of some resources, usually the register file, in the hardware to avoid slow copying and restoring of context data residing in a slower storage [48]. One interesting implementation of multithreading is *Symmetric Multithreading (SMT)*. SMT extends the dynamic processor hardware used for extracting ILP from a single thread to schedule instructions from multiple threads simultaneously to the function units in the machine, therefore interleaving execution from multiple threads on every cycle. [50]

Multithreading implements TLP by duplicating the processor resources only par-

tially. Fully parallel TLP requires duplicating the resources of the whole core to implement independent execution of multiple threads.

Multiprocessors include two or more full instances of processor cores to support executing multiple threads simultaneously. In case the processors are implemented in separate chips, the thread communication bottleneck is high. The increasing number of available transistors per chip has made it possible to include multiple cores in a single chip (*Chip Multiprocessors, CMP*). CMP allows cores to communicate with fast lower level caches for enhanced inter-thread data transfer speed. [51]

The connectivity, the level of memory sharing, and the datapath uniformity among the multiple cores leads to different categorizations of the multiprocessors. A *Symmetric Multiprocessor (SMP)* is a *homogeneous* multicore setup where the cores with identical instruction set architectures (ISA) access a shared memory. The shared memory is usually cached using cache coherence logic in hardware to maintain a consistent view to data from all the cores. This style of setup is easy to program as any thread can be assigned to any core uniformly which also eases load-balancing.

A major drawback in the SMP setup is the single shared memory bus and the associated cache coherence logic which becomes a bottleneck with increasing number of cores attached to it. *Non-Uniform Memory Access (NUMA)* improves the situation by dividing the memory hierarchy to *local memories* that are fast to a subset of cores, and *remote memories* that are slower to access (but are local to another subset of cores). [52]

In contrast to homogeneous multiprocessing where the cores are identical, *heterogeneous computation platforms* include multiple processors with different microarchitectures and varying memory hierarchies. The benefits from such setups are clear whenever the platform is supposed to run a varying set of algorithms from different application domains, or just a large application using a range of varying algorithms. The assumption is that a processor tailored to a specific application domain can execute the application more efficiently (e.g., with less power or area, or faster) compared to a more general-purpose core. This potentially leads to better utilization of the processing resources as the best matching cores can be picked for each algorithm. For example, running a serial control-oriented program in a 12-issue VLIW architecture is waste of the computational resources, while, on the other hand, a statically parallel algorithm can be executed very quickly on such an architecture after which the core

can be switched off for power saving. [53]

Heterogeneous platforms are common in the embedded domain. In embedded *System-on-a-Chip (SoC)* platforms, processors with different ISAs, hardware accelerators and peripherals are integrated to a single chip. A subcategory of SoCs is the *MultiProcessor SoC (MPSoC)* which means a SoC with more than one programmable processors on the same chip [54]. Another heterogeneous processing style that has received interest in the recent years is the GPGPU setup where a general-purpose processor runs an operating system and more serial parts of the program which is accelerated using a different processor originally designed for high performance graphics processing.

2.3 Customized Parallel Processors

In general, hardware-software co-design is a means to design an electronic system as a combination of application software running in instruction set processors and parts implemented with fixed function hardware. In hardware-software co-design methodologies, consideration of the hardware resources available in the designed system, the software organization, and the mapping of algorithms to the different resources in the system is performed concurrently. [55] In this Thesis, the focus is on a smaller subset of the co-design problem, that is, the co-design of instruction-set processors. Such processor co-design is performed at a phase of the electronic system design when a piece of application is mapped to a software-reprogrammable processor of which datapath resources should be customized to meet the requirements of the application at hand.

The line between the definitions of *Application Domain-Specific Processors (ADSP)*, *Application-Specific Instruction-set Processors (ASIP)* and *Application-Specific Processors (ASP)* is thin. The difference is defined to be in their level of specialization and compromises made to the generality in programmability. ADSPs are designed to be used for a wider range of algorithms within a single domain, such as processors tailored for popular multimedia codecs [56]. The concept of ASIP and its difference to ASP, however, is not so strictly defined in the literature. ASIP is commonly understood as a programmable hardware accelerator, a specialized replacement for a fixed function accelerator implemented as an ASIC or on an FPGA. Corporaal [8] proposes an additional level of specialization. He describes the difference between

an ASIP and ASP being in the software reprogrammability. In his definition, ASIP, while tailored for a single application, can still support (albeit perhaps not as efficiently) all programs defined in a higher-level language, while ASP can run only the targeted program, thus can be seen as merely a means to implement a fixed function accelerator.

The design methodology described in this Thesis supports all the above levels of processor specialization. Therefore, a common definition of *customized processor* will be used in the rest of this Thesis to describe the general case of a processor tailored for a *known set of targeted applications* (where the size of the set can vary from one to many).

A suggested difference between customized processors and “off-the-shelf” processors is that with customized processors the fabrication volumes can be lower and their “turn-around-time” shorter [57]. In other words, the design effort per manufactured chip is expected to be higher. In order to lower the design time, the customized processor design methodologies should be supported by design toolset with automatic toolchain retargeting. Especially important tool in such toolsets is an automatically retargetable higher-level language compiler, in order to avoid manual porting of assembly language programs to the designed processor variations, or manual retargeting of the compiler backend. Otherwise, the process of finding the desired processor alternatives (design space exploration) for the programs at hand is not feasible.

In order to make implementing a processor design toolset feasible, the *design space* of the customized processor alternatives is usually limited using a *processor template* which defines the set of parameters within which the new designed processors can differ [58]. Some processor templates limit the customization capabilities to one or more extra *special instructions* (also known as *custom operations* or *special function units*) attached to a predefined processor datapath. In some cases, the special instructions can also be runtime reconfigurable. Some templates allow customizing also the issue-width and the number of general purpose registers to match the provided ILP or DLP in the processor with the parallelism available in the programs at hand. In addition, the provided TLP can be varied in some templates (such as the one proposed in this Thesis) by defining the number of processor cores or the processor contexts available for multithreading. In these cases, the processor template supports the customization of the level of parallel execution resources provided by the processor, thus leading to the subcategory of *customized parallel processors*. [59]

3. EXPOSED DATAPATH STATIC MULTI-ISSUE ARCHITECTURES

The term “exposed datapath” is seen in the literature being attached to processor architectures where the processor datapath resources and even in some cases the data transports between the register files and function units are exposed for the direct control of the programmer. [60–62]

The *Transport Triggered Architecture (TTA)* which is used as the single-core processor template in the proposed processor customization flow can be classified as an exposed datapath architecture as it presents very low level details of computation to the programmer. The motivation for using an exposed datapath comes from the assumption of a parallel input program. As the input is parallel, as little additional hardware as possible should be dedicated to the runtime extraction of parallelism.

This chapter introduces the TTA, compares it to the other common parallel datapath paradigms, and reviews the work in the exposed datapath architectures.

3.1 *Transport Triggered Architectures*

VLIWs are considered interesting processor alternatives for applications with high requirements for data processing performance [63] and with limited control flow. *Transport Triggered Architecture (TTA)* is a modular processor architecture template which can be used as a design paradigm for wide static multi-issue architectures. The main difference between wide TTAs and VLIWs can be seen in how they are programmed: instead of defining which operations are started in which function units (FU) at which instruction cycles, TTA programs are defined as data transports between register files (RF) and FUs of the datapath. The operations are started as side-effects of writing operand data to the “triggering port” of the FU. Fig. 8 presents a simple example TTA processor [8].

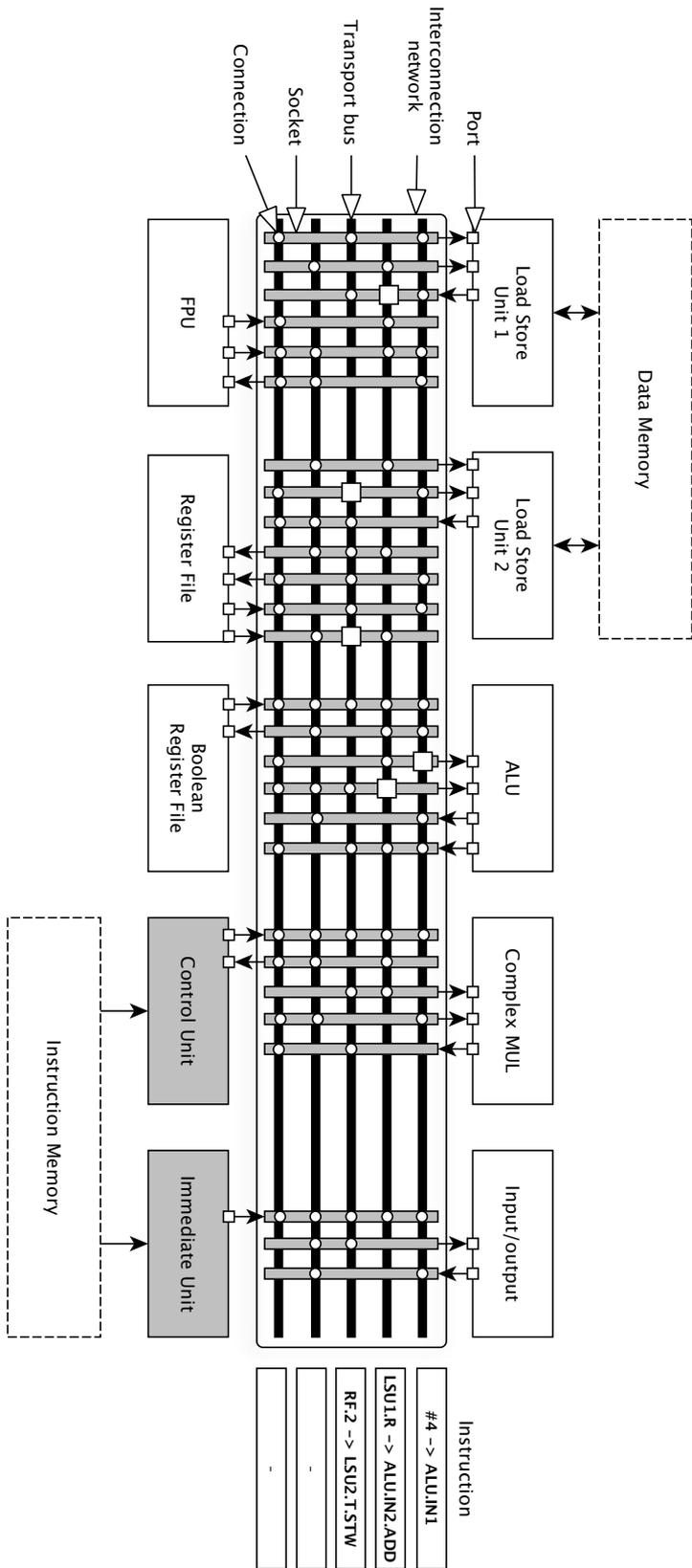


Fig. 8: Example of a TTA processor datapath. The datapath consists of function units, register files, boolean register files, and a customizable interconnection network. Data transports are explicitly programmed; a write to a special *trigger port* of a function unit starts the operation execution. The example instruction defines *moves* for three buses out of five, performing an integer summation of a value loaded from memory and a constant. In parallel, the third move stores a register to memory. The rest of the buses are idle. The connections enabled by the moves are highlighted with squares.

The programming model of VLIW imposes limitations for scaling the number of FUs in the datapath. Increasing the number of FUs has been problematic in VLIWs due to the need to include as many write and read ports in the RFs as there are FU operations potentially accessing it the same time. Additional ports increase the RF complexity, resulting in larger area, critical path delay and power consumption. Register file complexity of wide VLIW architectures has been identified as an obstacle in adoption of large VLIW architectures as soft-cores in FPGAs [64].

In case the VLIW supports register file bypassing, adding an FU to the datapath requires new bypassing paths to be added from the FU's output ports to the input ports of the other FUs, increasing the interconnection network complexity. Thanks to its programmer-visible interconnection network, TTA datapath can support more FUs with simpler RFs [65]. Because the timing of data transports between datapath units are programmer-defined, there is no obligation to scale the number of RF ports according to the worst case number of FUs accessing the RF at the same time [66]. As the register file bypassing is done in software instead of hardware, it is often possible to avoid the use of general-purpose registers as temporary storage, thus reducing both register pressure and register file port requirements even further. In addition, as the datapath connectivity is part of the architecture (visible to the programmer), it can be tailored according to the set of applications at hand, including only the most beneficial connections.

3.2 Benefits of TTA in Single Program Multiple Data Computation

The SIMD/vector or SIMT processors provide a good balance between hardware complexity and programming flexibility, but, in order to be efficient, the executed program must be data parallel, i.e. contain parts that can be efficiently vectorized with minimal branch divergence. Otherwise, diverging branches where the number of parallel instances of the SPMD program going into each branch is not a multiple of the issue width size will cause resource underutilization.

For example, let us assume a lockstep SIMD-programmed machine with 16 parallel function units, thus capable of running 16 instances ("threads") of a given program in parallel. Each thread is given a unique identifier (*thread_id* or an iteration number). In order to see the level of function unit idle time the SIMD execution model can cause, let us consider a couple of example branching conditions:

`(thread_id % 2 == 0)` Every two consecutive threads take different branches. This causes half of the FUs to be idle.

`(thread_id == 0)` Only the first thread enters the branch, rest of the 16 FUs are idle.

The underutilization results from the fact that SIMD/SIMT is limited to a single operation code per SIMD instruction. Therefore, in the case of branch divergence the diverged lanes must be “squashed” at run time using “vector masks” in case of static SIMD instructions or with per-FU program counters in case of SIMT scalar units. In other words, the squashed function units are waiting for the correct branch to be executed for their thread without doing any useful work.

On the other hand, when the program is scheduled for a TTA or VLIW with predicated execution support it is possible to reach higher utilization of FUs for programs with compile-time known control flow given that there are independent parallel operations to schedule at the idle function units. Two clear cases where TTA/VLIW machines can outperform the SIMD/SIMT machines with similar FUs can be isolated:

1. Number of function units in the architecture does not equally divide the number of “threads” in the program.

If a program cannot be split perfectly to the SIMD lanes in the core, TTA/VLIW processors allow using the extra FUs to execute independent parts of the code within the threads itself. Consider a simple program structure like the following:

```
A;
if (P)
    B;
C;
```

where A, B and C are data-independent. This program is to be run for two pieces of input data. Figure 9(a) shows how this example could be scheduled on a 3-way SIMD group. It can be clearly seen that one of the execution lanes is never used. Figure 9(b) shows how a predicated TTA/VLIW machine with

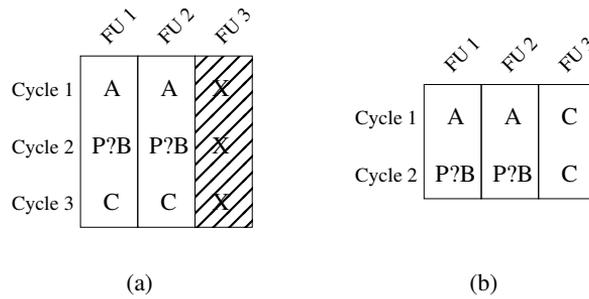


Fig. 9: An operation scheduling example showing MIMD scheduling freedom. Schedule for (a) predicated 3-way SIMD, and (b) predicated TTA/VLIW. Question mark denotes the predication of the succeeding operation with the preceding predicate register. Each function unit (FU) can execute all the operations (A, B, C).

the same number of function units can save a cycle out of three thanks to the scheduling freedom.

- Runtime conditions allow “overcommitting” of datapath resources by means of predicated execution.

A TTA/VLIW architecture that supports overcommitting of resources allows scheduling of two predicated operations to be executed in the same function unit at the same time cycle, in case it is known at compile time that the predicates of the two operations are never simultaneously true [67]. Code suitable for overcommitting is usually generated from *if...else* constructs. For example, consider the following program structure:

```

if (P)
    A;
else
    B;
    C;

```

Let us assume that all operations (A, B and C) in the program are data independent, and that there are two threads scheduled on two FUs. Fig. 10(a) shows the resulting schedule for a SIMD machine: one third of the execution time is wasted for each lane. An overcommitting schedule for VLIW/TTA, as shown in Fig. 10(b) saves that third cycle and results every computing element being used on each cycle.

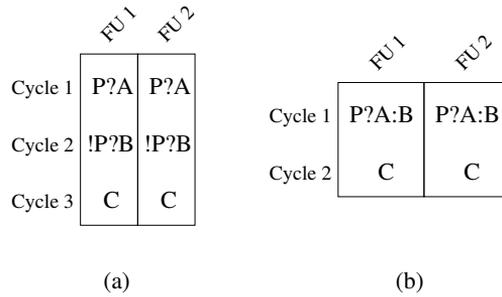


Fig. 10: Resource overcommitting example. Schedules for (a) predicated SIMD machine without overcommitting support, and (b) predicated TTA/VLIW with function unit overcommitting support. Question mark denotes the predication of the succeeding statement with the preceding predicate, the possible negated statement is given after a colon.

Clearly the programming freedom of overcommitting TTA/VLIW has its benefits in function unit utilization point of view. However, the obvious drawback is that in order to express the additional programming freedom, more program bits are required. The actual impact of this is program dependent and cannot be generalized. While the SIMD style provides excellent instruction density in comparison to the VLIW/TTA approach, the reduced scheduling freedom (single operation code per instruction) might mean that more SIMD instructions are needed due to the diverging branches to encode the whole program. The overcommitting predication support requires the capability to describe, in addition to the predicate, the execution of two operations per each FU, growing the instruction word even wider for the VLIW datapaths. In case of the bus programmed TTA, the number of buses per FU, thus the number of move slots, has to be doubled.

3.3 Review of Work in Exposed Datapath Architectures

The fundamental ideas in exposing most or all of the control of the processor internal components to the programmer can be traced back to the idea of *microcoding*. Microcoding was proposed in the early 1950s to be used to simplify and modularize the processor control unit implementation by allowing the processor designer to describe the processor control signals to be produced from programmer-visible instructions as small *microprograms* instead of hard-wired logic. In *horizontal microcoding* the microprogram for each instruction describes precisely what happens at each cycle of the execution of the coded instruction in the processor datapath, including the register

enable signals, data routing through multiplexors etc. In *vertical microcoding*, on the other hand, the microprogram contains instructions that need some level of further decoding by the processor hardware and expand to control signals for multiple execution cycles. One of the earliest commercially successful machines that employed microcoding was the System/360 series from IBM in the 1960s. [68, 69]

Microprograms are typically stored in read-only memories or programmable logic circuits (called *control stores*) inside the processor control unit. In some cases, the control store is writable, which enables on-the-field updates of the so called *firmware* of the processor. In the early days of microcoded machines the writable control stores were used for instruction set emulation and to implement instruction sets customized for different higher-level programming languages within a single processor microarchitecture. [69]

In the light of the microprogramming and writable control stores, the concept of the so called “exposed datapath architectures” can be interpreted to reflect the fact that the details that have been typically visible only to the microcode programmer are exposed to the common programmers of the processor. That is, an extremely “exposed architecture” fetches the datapath control signal values directly from the program memory instead of indirectly from a microcode control store. All the proposed exposed datapath architectures follow this principle to some degree. They mainly differ in the level of additional instruction decoding needed, which places their programmer-interface somewhere between horizontal and vertical microcoding.

The *Reduced Instruction Set Computer (RISC)* can be seen as a step towards “exposed datapath architectures”. It was proposed in the late 1970s with the argument that the earlier, so called *Complex Instruction Set Computers (CISC)* included overly complex instruction decoding and execution logic (implemented commonly using the microprograms) given the widespread use of higher-level programming language compilers that hide the details of the processor instruction set from the programmers. [70]

FPS-164 was a scientific co-processor introduced in 1981 that included an instruction set similar to the ones in horizontal microprograms. With this architecture it became clear that such detailed instruction sets demand an efficient higher-level language parallelizing compiler for it to be usable. A Fortran-77 compiler for FPS-164 was introduced in [71]. Research in horizontal microcode compaction (producing parallel microprograms automatically) led to the ideas on the *Very Long Instruction Word*

(*VLIW*) architectures and compilation techniques for global extraction of instruction level parallelism (trace scheduling) required by such statically parallel architectures in the early 1980s. In essence, the original *VLIW* was a statically scheduled multi-issue RISC, thus resembled the horizontal microcoded machines which expose parallel hardware resources to the programmer. [41]

The concept of data transport programmed architectures was first proposed for control processors in the mid-1970s. The work published by Lipovski can be seen as one of the pioneering research efforts in exposed datapath architectures and transport programmed processors [72,73]. This processor included only one instruction: a data move between memory mapped control registers. The ALU was attached to the core as an I/O device.

In the early 1990s, *Corporaal et al.* from the Delft University of Technology proposed their *MOVE* architecture for general-purpose applications and high performance computing [74]. One of the key discoveries they made was the potential of transport programming in reducing the register file complexity bottleneck of wide *VLIW* architectures [66, 75]. They proposed a new classification for processor architectures according to the way their instructions trigger the operation execution. In this classification, their *MOVE* architecture belonged to the class of *Transport Triggered Architectures (TTA)* while the “traditional” architectures were classified as *Operation Triggered Architectures (OTA)*. The *MOVE* project also produced an ASP design framework [76] with a retargetable C compiler, instruction set simulator and automated design space exploration tools.

The work from the Delft *MOVE* project was continued in the Tampere University of Technology (where the work for this Thesis was conducted). Part of this effort was to produce a more extensible rewrite of the *MOVE* tools, study the use of *TTA* processors in the context of low-power DSP applications and to research issues in ASP design flows, such as retargetable compilation and fast retargetable instruction set simulation. The toolset project was named *TTA-Based Co-Design Environment (TCE)* [77].

In 2003, a processor concept later to be called *FlexCore* [60] was proposed within the *FlexSoC* [78] research project. *FlexCores* are programmed using so called *Native-ISA (N-ISA)* instructions that are seemingly similar to horizontal microcode instructions. For instruction width reduction they propose the use of a “reconfigurable instruction

decoder” which is an instruction compression unit that expands instructions encoded in so called *Application-Specific ISA (AS-ISA)* format to the more detailed N-ISA format [79]. This is a concept similar to vertical microcoding with a writable control store. The reconfigurability of the instruction decoder/decompressor allows the emulation of multiple traditional ISAs with a single FlexCore datapath, an idea somewhat similar to the dynamic translation used in the Crusoe processors [80] where the translation is done using a more complex software layer. FlexCore is supported by a compiler with an instruction scheduler [81].

Similarly to FlexCore, the *No Instruction Set Computer (NISC)* proposed in 2005 utilizes the idea of horizontally microcoded programming with no additional decoding logic. Their compiler can also generate a *Finite State Machine (FSM)* based control logic, making their design flow an interesting candidate as an implementation technique for a C to RTL flow. [82]

The *Efficient Low-power Microprocessor (ELM)* project studied techniques to reduce power consumption of embedded processors [62]. In their work they identified that the main source of energy inefficiency in embedded RISC processors is the instruction and data supply, more specifically the caches. In order to reduce this bottleneck, they proposed the use of explicitly controlled instruction register files [83] and operand register files [84]. Both of these improvements increase the programmer-visibility, i.e., expose more details of the microarchitecture to the programmer. The operand registers enable explicit operand forwarding which is similar to the software bypassing optimization in case of TTAs. The main difference is that the TTA template used in the design methodology proposed in this Thesis includes at most only one register per function unit input or output.

MOVE-Pro proposed improvements to the original TTA template used by the MOVE project [61]. Similarly to the Stanford ELM, their architecture adds a small set of registers local to the function units to improve the possibilities of the software bypassing optimization. In their work the register file is located at the function unit output, in contrast to Stanford ELM where the register files store function unit input operands. However, it is not clear from this or the ELM work whether the additional complexity of multiple output or input registers is justified in the common case. In [61] the comparisons were made only against a simple RISC architecture instead of the previous TTA processors. For example, in [85] the authors observe that the vast majority of temporary values are short lived which can be read that they are used once or maybe

twice by the program, which speaks against the additional complexity.

Another improvement proposed by MOVE-Pro is to avoid a specific trigger port like in the original MOVE architecture, thus increasing instruction scheduling freedom by allowing arbitrary operand move ordering for each operation. The other way to look at it is that, in fact, all of the ports in their architecture are potentially triggering, which means the instructions need to encode the triggering info (add an opcode field) for all types of operand move instructions. The programming simplification of allowing all move instructions to any operand to trigger the operation execution has been proposed also in 2004 by another research group [86].

4. CUSTOMIZABLE PROCESSOR DESIGN METHODOLOGY FOR PARALLEL PROGRAMS

General-purpose multiprocessors are commonly implemented as *homogeneous* shared memory computers to achieve easier programmability and scalability for certain types of workloads. On the other hand, the multiprocessors in embedded systems can be customized more freely according to the application domain (e.g., video processing) which is possible in part thanks to the lack of the legacy instruction set support burden present in general-purpose computing. In addition to special instructions, the customized multiprocessors can provide varying degrees of instruction-, data- and task-level parallelism in order to meet the performance goals placed by the targeted set of algorithms while staying in a limited area and energy consumption budget. At the same time, programming such devices should be as simple as possible in order to reduce software porting and development costs.

Fuller et. al [87] suggest that getting more performance with additional cores will eventually reach its limits, e.g., due to excessive power consumption [88]. They consider application-optimized processing units as a potential solution in the longer run. However, in order for such a solution to be feasible, the multiprocessor customization should be supported with a flexible design methodology.

This Chapter describes the higher level overview of the proposed methodology supporting design of *Multicore Application-Specific Instruction Set Processors (MCASIP)*. The designed MCASIPs can be implemented, e.g., as standalone processors or as building blocks in heterogeneous systems. The design methodology is implementation technology independent, thus the MCASIPs can be implemented on FPGAs or as ASICs. Fig. 11 shows an example of a heterogeneous system where the processing nodes are composed of multiple MCASIPs with various issue widths and local memory sizes. In this picture, a separate host/control processor is depicted. The host processor can run a complex general-purpose or real time operating system. It should

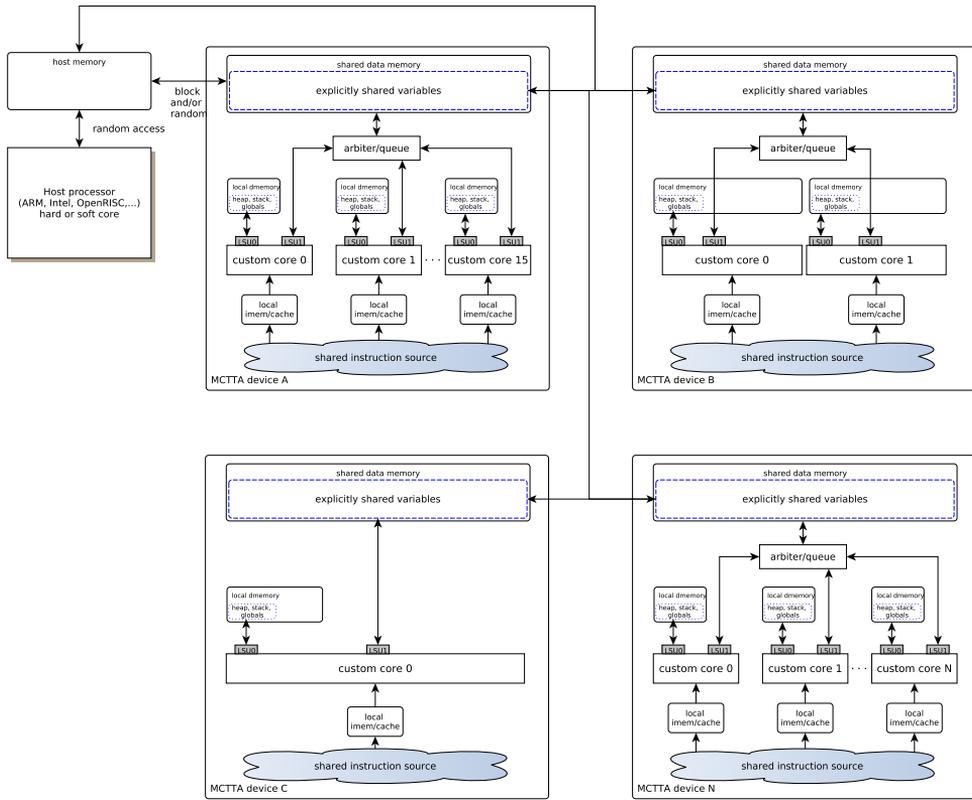


Fig. 11: Example of a generic heterogeneous system with multiple MCASIPs designed using the proposed design methodology accompanied with a control/host processor. Each MCASIP can differ in their instruction set, core count, single core issue width and local memory size. The details of the interconnection network between the MCASIPs is abstracted away in this picture.

be noted, however, that as the MCASIP template is fully programmable and incorporates an independent thread scheduler (described in Section 5.5), decentralized task coordination can be implemented to reduce the host-to-devices communication bottleneck.

The main goal in the design methodology is to enable scalable parallel processor architecture co-design for programs described in established parallel programming paradigms such as OpenCL or OpenMP. The design methodology is based on a compiler supported parallel processor template (described in Chapter 5) that combines the ease of programmability from using a common shared memory with the ability to optimize shared memory usage with core-private local memories. Supporting compiler

techniques for OpenCL C kernel parallelization on the proposed static multi-issue template are described in Chapter 6.

4.1 Related Work

The design methodologies for processor customization flows have only until recently concentrated on single core optimization aspects without paying much attention to exploring the homogeneous multicore processor design space [58]. A good survey of the current processor customization flows can be read from [59]. These customization methodologies can be used to produce multiple *decoupled* customized cores to form a heterogeneous multicore. An example of a paper proposing such a methodology is [89] where the Chess/Checkers ASIP design suite is proposed for customizing the single cores in MPSoCs.

In addition to the single core customization aspects also present in previous processor customization flows, the proposed design methodology enables the design of customized homogenous multicores with local and shared random access memories. The emphasis of the proposed design flow is on SPMD multiple address space programming languages. The benefits of the single-ISA multicore in comparison to multi-ISA include easier programming and dynamic workload balancing. However, it should be emphasized that the multicores produced with the proposed design methodology can be used as building blocks in larger heterogeneous systems, and connected to various interconnection topologies. The combination of homogeneous multicore processing elements in a heterogeneous multicore system has the potential to provide the benefits of the both alternatives to the designed system.

A design methodology for heterogeneous multicores for parallelizing dataflow applications using task level pipeline parallelism is proposed in [90]. Their implementation uses the Tensilica Xtensa LX configurable cores for processing elements. A heuristics for mapping applications organized in dataflow manner in such pipelined multicore system is presented.

Single-ISA heterogeneous multicore space was explored in [91]. In that work, processor cores were customized in a 4-core heterogeneous multiprocessor, unlike in the previous work which had used a given heterogeneous architecture. The paper shows that the best heterogeneous designs are found when the cores in the multiprocessor

are customized to the set of applications at hand. In the proposed methodology the scope of customizing the cores is similar but smaller. The end result that is optimized for a given application, is a parallel customized (multi)core that can be attached to a larger heterogeneous multi-core. In the proposed methodology, the input programs are assumed to express parallelism which the exposed datapath single core template supports with minimal control overheads. The processor template in the proposed design methodology adds the degree of task level parallelism available in each processing core as a parameter, therefore enabling customization of both fine-grained and coarse-grained levels of parallelism for each core in a heterogeneous system which executes highly-parallel SPMD workloads.

Regarding scalable multicore generation, related work has been conducted in the MOSART project [92]. In the MOSART project, the focus is on global interconnect and the memory bottleneck inherent in multicores and other heterogeneous multicore system design issues. It is widely acknowledged that the shared memory bottleneck is one of the main problems to solve in multicore scalability [52].

While acknowledging the importance of memory hierarchy design exploration in the focus of the MOSART project, the proposed customized processor design methodology relies on the idea of co-design where the hardware and software are designed mutually. The assumption of the use of a relaxed memory consistency model and the explicitly addressed scratchpad memories in the supported programming languages moves the problem of reducing the memory bottleneck more towards a software optimization challenge.

4.2 Parallel Processor Design Flow

Fig. 12 illustrates the MCASIP design flow that is described in the following subsections.

4.2.1 Software Development

The first phase is the software development phase. It is often faster to develop the basis for the software using native compilation and execution using a 3rd party *Software Development Kit*. This allows rapid implementation of the software along with

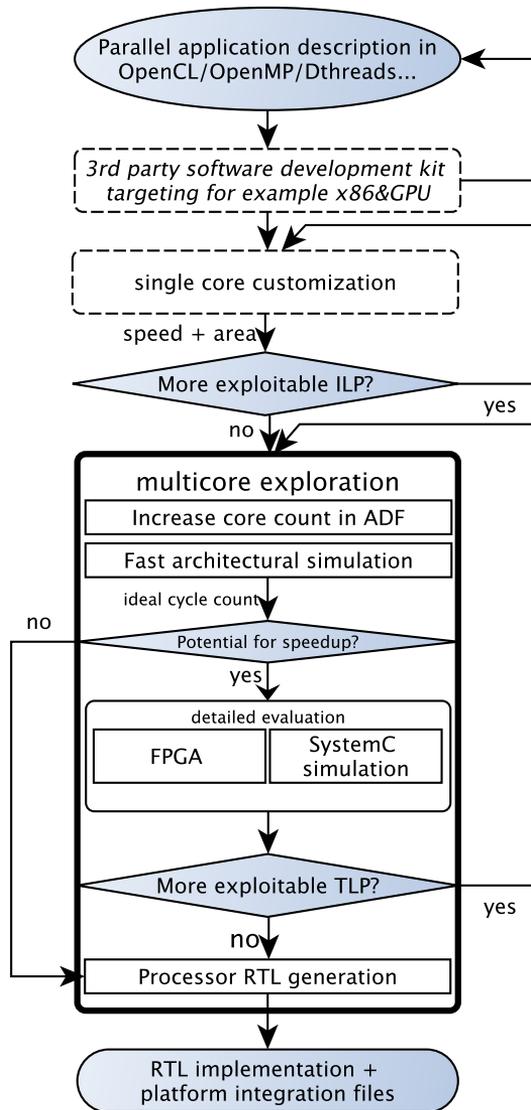


Fig. 12: The toolset supported design flow of parallel customized processors. The design flow adds an additional multicore exploration phase after the standard single core customization flow of TCE.

its verification data without using an instruction set simulator which is always slower than native execution. In practice, the parallel program can be implemented and tested, for example, on a desktop PC using one of the supported input languages while paying attention to program portability. The output of the software develop-

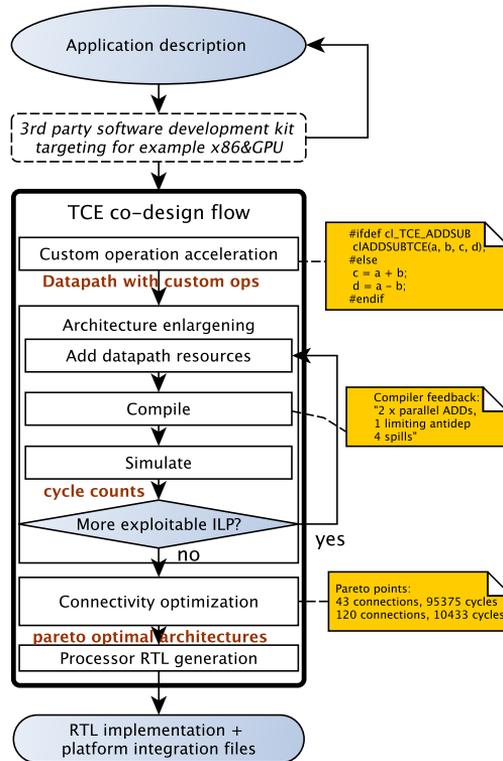


Fig. 13: The single core customization flow.

ment phase is a verified portable program description, which is then fed into the *TTA-based Co-design Environment (TCE)* single core customization design flow.

4.2.2 Single Core Customization

Once the program description is completed and verified, the single core datapath is customized for the application. The steps in the single core customization methodology are illustrated in Fig. 13. The methodology usually starts from implementation and verification of the application using, for example, a GPU-based *Software Development Kit (SDK)* or just the native C compiler running in a PC, and continues using the TCE tools for the co-design of the customized single core processor that can execute the application as efficiently as possible.

In the first phase of the single core customization flow, the designer uses his or hers algorithm expertise to choose the potential custom operations that can accelerate the

implementation. The custom operations are called from the kernel code using macros hiding inline assembly blocks (in case of basic C code) or the vendor extension API (in case of OpenCL C) with fallbacks to software only execution in order to maintain the portability of the code.

In this phase, a generic architecture is used in order to isolate the benefits of the custom operations. In addition to calling special instructions with the extensions API, at this stage the designer should also ensure that the kernel does not call unwanted software floating-point emulation libraries in case of floating-point kernels. Hardware floating-point support can be added to the designed architecture by picking one or more floating point units from a hardware database. The units and the special function units can be added to the architecture using the graphical *Processor Designer (ProDe)* tool of TCE.

After the potential custom operations have been added to the generic architecture, the next phase includes growing the architecture with “standard datapath components” to exploit ILP. It should be noted, however, that the ordering of custom operation selection and architecture enlargening phases depends on the level of specialization the designer wants to apply. In case a processor with more generally usable resources is desired, it might make sense to first grow the ILP capabilities with FUs supporting standard operands from the input language (such as additions and multiplications) after which the custom operations are used only as “the last measure” if the performance goal is not reached.

The goal in the architecture enlargening phase is to find a minimal set of function unit (FU) and register file (RF) resources that satisfy the ILP demands of the executed kernels. This “architecture enlargening” phase starts from a minimal architecture that can still execute any program, or the generic architecture augmented with custom operations in the previous (optional) phase. The architecture is grown with additional FUs and RFs until the obtained kernel cycle count improvement is negligible. The iterations consist of architecture modification, recompilation of the code to the new architecture, and architecture simulation to analyze the potential cycle count decrease.

In order to assist the designer in finding the set of adequate FU and RF resources for the kernels, the retargetable compiler provides textual feedback that can guide the design decisions. The feedback includes statistics from an optimal resource un-

constrained schedule of the kernel critical path, the number of variable spills and the number of register anti-dependencies that limit the parallelism.

The optimal schedule of the kernel critical path is used to estimate the number of parallel FUs the kernel can efficiently utilize. The number of spills and the anti-dependencies in the *Data Dependence Graph* guide the choice of the number of registers. The designer usually wants to add enough registers to allow all variables to stay in registers for the whole kernel execution to get the best possible performance.

The number of anti-dependencies, on the other hand, might not be such a self-evident designer guidance. When a pre-pass register allocator is used, the anti-dependencies resulting from reuse of registers can ruin the parallelization opportunities presented to the post-pass instruction scheduler. Especially, when executing multiple C loop iterations or OpenCL C work-items in parallel, the anti-dependencies between the work-items might take away the benefits of inserting additional work-items or loop iterations to the code, thus only growing the instruction memory footprint. Therefore, the designer might want to try to increase the register count until the parallelism limiting anti-dependencies disappear. Due to the ability of the TTA template to partition register files efficiently, the complexity of a single register file can be kept low when the total register count is increased [75].

Ideally, after the kernel cycle count has been saturated by adding enough FU and RF resources, the kernel schedule should be limited only by the true dependencies in the program. In reality, the data memory bandwidth usually limits the architecture enlarging phase. The number of parallel *Load-Store Units (LSU)* is always limited by the memory system implementation in realistic designs. Thus, the saturation point of the kernel cycle count is often limited by the number of LSUs that can be supported by the memory implementation instead of the parallel arithmetic function units the kernel critical path could exploit.

The final stage in the single core customization methodology is the interconnection network optimization. Until this stage, the customized architecture is grown only by adding FUs and RFs, without paying attention to optimizing the interconnection network between them. A fully connected TTA interconnection network is not usually realistic to implement, especially with large architectures. Therefore, the network should be reduced to improve the cycle time, reduce the power consumption and to save chip area.

Finding a customized interconnection network for the application would be a tedious manual task, thus an automated interconnection network optimizer was developed to automate this phase. The *connectivity sweep* algorithm goes through the interconnection network by first removing connections to the RFs one transport bus at a time. The effects of each of the connection removal to the cycle count are measured and the least affecting connections are removed first. The algorithm is similar to the connectivity optimization done in [93] and in [94] except the proposed algorithm emphasizes the more expensive register file connections (the bus read/write muxes are in the same critical path as the register read/write selection muxes). The interconnection network optimizer produces a set of architectures with varying number of connections and statistics of the application cycle counts for each of the variation. As the number of these alternatives can be huge and it is too time consuming to synthesize them all to obtain the exact area and delay, the user is also presented a set of pareto optimal architectures with regard to their number of connections and the cycle count. In case of TTAs, the number of connections in the interconnection network has a strong relation to the consumed chip area and the longest path delay.

Finally, after the architecture has been designed, the hardware implementation can be generated automatically by using an RTL generator. The standard *Processor Generator (ProGe)* tool of TCE is used for implementation generation. ProGe generates both a VHDL implementation of the core and the necessary files for integrating it to various platforms such as FPGA chips. The final synthesis is conducted with a third party synthesis tool.

In case the designer wants to exploit task level parallelism in addition to the special instructions and the instruction-level parallelism, the customized single core architecture is entered to the optional multicore exploration phase.

4.2.3 Multicore Exploration

In essence, the multicore exploration phase of the design flow (later referred to as TCEMC as in TTA-based Co-design Environment for MultiCores) consists of increasing the *core count* parameter of the architecture description file and measuring its effects on the processor performance.

The designer can test the potential for speedup using the fast architecture simulator that does not include a detailed model of the shared memory microarchitecture but

assumes ideal memory access latencies without contention. This allows the designer to measure the potential benefits of adding more cores without going through the rest of the slower more detailed evaluation steps. In case the architecture simulation shows significant enough speedup potential, the designer continues to the detailed evaluation phase. Otherwise, the design flow exits to processor RTL generation.

In the detailed evaluation phase the designer has two options: FPGA evaluation and SystemC simulation. At this point the shared memory hierarchy must be described in more detail to evaluate its effects to the multiprocessor performance. In the case of FPGA evaluation, the processor RTL is automatically generated along with its shared memory hierarchy implementation. The detailed cycle counts are then obtained by running the design on an FPGA. Another alternative for more detailed performance evaluation is a system level simulation where also the memory hierarchy is modeled. This can be done by using SystemC-based simulation. TCEMC provides SystemC hooks for connecting the TTA core architecture simulation model to SystemC simulations, thus enabling incremental simulation detail level addition while still using faster higher-level simulation model [14] for simulating the core.

After the evaluation phase, the effects of the shared memory are known which can lead the designer to modify the memory hierarchy or to optimize the program to exploit local memories more efficiently. In case the FPGA evaluation was chosen, also the FPGA resource consumption and maximum execution frequency is now known which also adds a limit to the number of cores that can be still added. Depending if these results show potential for more task-level parallelism to be exploited, the designer either exits the design flow to RTL implementation generation or goes back to adding more cores to the multicore.

The final step in the design flow is the processor RTL implementation generation step. RTL generation is implemented with an hardware library-based approach where only the custom operations need to be described in a hardware description language [95]. The rest of the implementation is generated automatically for the designer. In addition to producing the RTL implementation, this step also generates the possible files and interfaces required for integration on, for example, different FPGA platforms. A more detailed description of this phase is available in [96].

5. CUSTOMIZABLE PARALLEL PROCESSOR TEMPLATE

This chapter describes the proposed parallel processor template from which new multicore ASIPs are instantiated in the proposed design methodology. Section 5.1 presents the work related to the proposed template, Section 5.2 describes the processor template concentrating on the resources that can be scaled to support the parallelism in the software and Section 5.3 provides a description of the memory model of the processor template. The implications of the processor template and the memory model to the supported programming models and the threading runtime are presented in Section 5.4 and Section 5.5.

5.1 *Related Work*

The proposed parallel processor architecture template and its memory hierarchy with private local memories are similar to the *Synergistic Processor Unit (SPU)* of the heterogeneous IBM Cell architecture [44]. A major difference with the SPU in comparison to the proposed template is that SPU concentrates on data level parallelism for vectorizable code with a SIMD datapath which includes only limited support for *Instruction-Level Parallelism (ILP)*. In the proposed parallel processor template, the emphasis is on the more general ILP. While ILP is easier to exploit as it does not require vectorizable code for parallelizing program operations, it has the drawback of potentially wider instruction word size in comparison to vector or SIMD instructions. This drawback has been addressed in research on instruction compression and encoding methods [97].

Another difference between the SPU architecture and the proposed one is with the access to the shared main memory. In case of SPUs, only the local store can be accessed randomly. The shared memory supports only block transfers. The memory architecture in the proposed template includes separate address spaces and two

LSUs for accessing the local memory and the shared memory simultaneously with load/store instructions.

The proposed template has similarities to the contemporary GPU architectures. For example, NVIDIA GPUs based on their G80 architecture include multiple *Streaming Multiprocessors (SMs)*. Each SM consists of several *Scalar Processors (SP)* than can execute one scalar instruction at a time. However, these scalar cores do not execute independent instruction streams as their operations are defined by instructions from a single data parallel program (the SIMT execution model). [27, 98]

The NVIDIA GPU approach is an efficient solution for massively data parallel applications as it minimizes the instruction stream bottleneck. The aim in the proposed template, on the other hand, is wider applicability and easier programmability. The cores are fully independent, fed with independent instruction streams. Certainly, in designs with many cores providing the instruction streams can become challenging. However, one of the primary implementation targets is FPGA where the designer can freely customize the instruction memory hierarchy, and, e.g., use private instruction memory ROMs for feeding the cores with instructions with adequate throughput. Another notable factor is that the instruction memories are read-only from the point of view of the program, thus they can be cached without a complex cache coherence logic. In the NVIDIA GPU data memory model, the set of scalar processors access a fast local memory that is shared among the cores. This resembles the memory model in the proposed template where multiple function units on a single core share the local memory.

Recent commercial GPU designs are moving towards more general purpose high performance programmability, blurring the old GPU vs. CPU roles apparent in the previous generations. For example, the Kepler™GK110 architecture of NVIDIA is labeled as a “high-performance computing architecture” instead of a GPU architecture. The architecture introduced several features easing programmability for less uniform workloads than the shader execution the original programmable GPU pipelines were designed for. The architecture allows the device to spawn threads dynamically, in contrast to the old model where the host processor orchestrated the work [99]. This enables the device to act more independently, executing part of the task schedule by allowing the device program to launch kernels and synchronize their execution without frequently going back to the host CPU. The flexibility of programmability to, e.g., allow offloading (part of) the task scheduling to the device has been the goal of

the proposed processor template from the start. This is accomplished by a software-defined task scheduler supporting dynamic thread creation.

5.2 Customizable Multicore Processor Template

The input programs to the design flow are assumed to express parallel computation, thus the main responsibility for the proposed processor template is to provide adequate level of parallel computing resources at the processor architecture side. Therefore, the support for scalable parallelism was the first priority goal for the designed processor template. The second goal was simplicity; as the compiled input program is assumed to express the parallelism explicitly, as little additional hardware as possible should be dedicated to the runtime extraction of it.

The customizable multicore processor template uses the TTA as the template for customizing the single cores in the homogeneous multicore. It has the same datapath customization properties as the single-core template augmented with extensions to support explicit access to multiple address spaces from higher level languages and minimal homogeneous multicore customization.

The single core datapath customization points in the *Architecture Description Format (ADF)* of the single-core customization flow are described in [77,100]. Summarizing, the most important data path customization points in the single core template follows:

- Function units (FU). Each core can have one or more FUs with one or more operations. The operations can be fully pipelined or implement complex FU pipeline resource sharing patterns.
- Register files (RF). The number and sizes of the general-purpose and boolean RFs can be customized.
- Operation set. The template supports custom operations with arbitrary number of operands and results.
- The interconnection network that connects the FUs and RFs. The connectivity matrix can be application-tailored.

TTA, as described in Section 3.1, fulfills well the simplicity requirement being one of the most “bare boned” processor design paradigms available [8]. Minimal control

and execution scheduling hardware logic is required after the simple decode stage of the instructions.

5.3 Memory Model

In the traditional terms, the processor template adheres to the *Harvard architecture* where data and instructions are stored in separate address spaces. The only constraint the processor template places to the instruction memory implementation is that each core must be able to read one instruction during the programmer-visible fetch stages of the control unit. In case the memory latency is dynamic, for example, due to a cached memory hierarchy implementation, the additional cycles to fetch an instruction cause the entire core to be locked, leading to reduced performance.

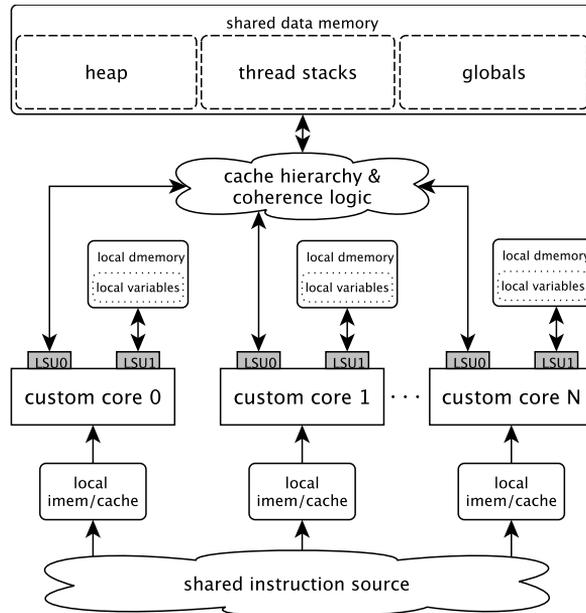
The address spaces were already a customizable feature in the original single core ADF proposed in earlier work [77, 100]. The described architectures could include one or more load-store units which could each access independent address spaces. The proposed extension to support multicore customization adds two new customization points to the ADF address space properties:

1. An integer identification number for each address space.

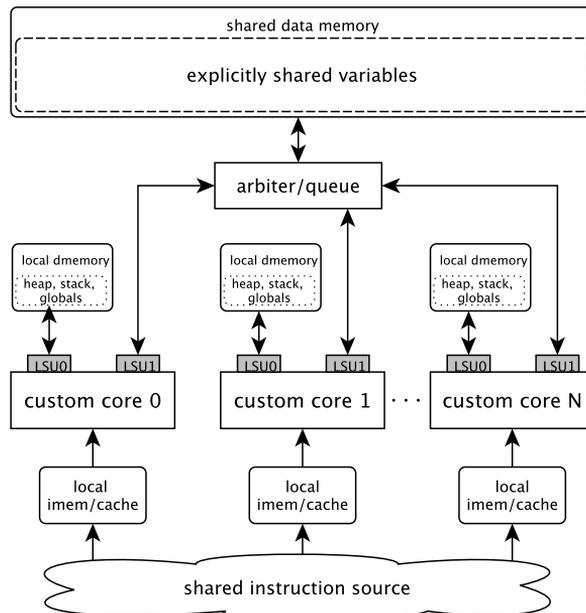
Address space #0 denotes the default address space and higher ids denote additional explicitly accessed address spaces. The address space ids are referred to using the explicit address space attributes supported by the used compiler front-end, thus enabling the use of multiple address spaces from higher level languages.

2. Shared/local attribute. In case an address space is marked “shared”, it is accessible by all cores in the multicore. Otherwise, the address space maps to a core local memory.

These new address space customization points, along with an integer parameter for the number of cores provides a minimal set of customization points for implementing homogeneous multicore application-specific processor co-design with support for programmer-optimized memory accesses.



(a) Shared Default Data Memory (SDDM) configuration.



(b) Local Default Data Memory (LDDM) configuration.

Fig. 14: Two different configurations of the memory model depending on the placement of the default data address space.

The data memory model of the processor template assumes two or more disjoint address spaces accessed using independent *Load-Store Units (LSU)*, thus separate memory operations. This is different from the usual *scratchpad memory (SPM)* setup where the SPM is mapped to the same address space as the shared/main memory and possibly shares the same address and data bus with it, or the SPM is the only random accessible memory seen by the core [101]. The dual-LSU setup allows accessing both memories simultaneously with compiler-exposed memory latencies. The choice in which address space the global variables of the program reside is made by the programmer using language-specific *pointer type qualifiers*.

One of the address spaces is mapped to a global memory that is shared across all the cores in the multicore. The *shared memory* can be used for communication, load balancing, and memory mapped I/O. In addition to the shared memory there is at least one private address space that maps a fast per-core *local memory*. Thus, the memory system is distributed, but not all the memory is visible to all the cores to reduce the memory hierarchy implementation complexity.

The shared memory implementation and microarchitecture is left open for customization. The threading runtime library presented in Section 5.5 assumes a *weak ordering model* [102], thus does not rely on specific ordering of non-synchronization memory operations leaving room for optimized parallel memory hierarchies.

The choice of mapping the “default address space”, the one which stores the stacks and the heap of the threads, to the shared memory or the local memory leads to two different memory configurations. This seemingly small choice also dictates the set of programming languages and interfaces that can be efficiently supported on the designed multicore. For example, Pthreads [19] assume the visibility of the thread stack to all the other threads, which requires the use of SDDM. This configuration choice is not so relevant in case the different address spaces are not explicitly accessed by the programmer and, for example, when an automatic data scratch pad memory partitioning technique [103] is used to optimize the local memory usage.

5.3.1 Shared Default Data Memory configuration

Figure 14(a) represents the generic TCEMC template in a configuration where the default address space is mapped to the shared memory (*Shared Default Data Memory, SDDM*). In this case, the stacks of all threads and the shared heap reside in the

global memory leading to increased, and often unnecessary, shared memory traffic. This creates higher demand for a cache hierarchy with potentially complex coherence logic [104] and unpredictable latencies. The main benefit of this configuration is the easy programmability for engineers familiar with shared memory threading libraries and not comfortable declaring variables shared only when necessary.

5.3.2 Local Default Data Memory configuration

The second configuration is shown in Fig. 14(b). When the private local memory of the core is set as the default address space, the cores use their local memories for thread stacks and heaps (*Local Default Data Memory, LDDM*). The main benefit of this is reduced unnecessary (accidental) shared memory traffic. As all shared memory accesses are explicitly defined by the programmer, and data is local by default, higher shared memory access latencies can be potentially tolerated than with SDDM. This means that a simple shared memory queue/arbitrator without data caching might be sufficient for controlling the shared memory accesses. Of course, this distinction is not strict since both configurations can use a complex cache hierarchy, or just a simple queue for the shared data memory access. LDDM merely forces the programmer to pay better attention to the shared memory accesses. It should be emphasized that shared memory is available for communication also in LDDM but must be explicitly accessed by marking the shared variables with the shared qualifiers in the program.

LDDM is the recommended memory configuration of TCEMC in case the chosen programming language supports it. The common shared memory threading programming models cannot be used because the default address space is not shared. For example, passing pointers to stack or heap objects to other threads breaks when the receiver thread is executing on a different core. A threading library supporting the LDDM configuration that can be used for implementing various other programming models is proposed in Section 5.5.

5.4 Software Stack

Small instruction memory footprint is a major requirement for the utilized software stack because providing the instruction streams to potentially high number of independent cores is challenging. Size optimized software stack allows more designs to

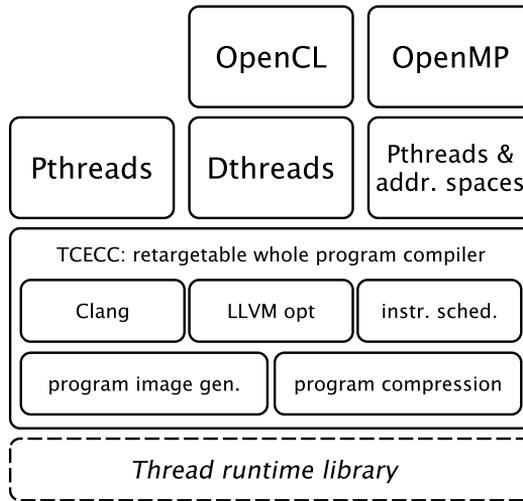


Fig. 15: The software stack of TCEMC. At the highest level, the application is described in one of the supported parallel languages. The LLVM-based whole program compiler compiles the application and the required threading libraries to LLVM bit code which is then converted to executable bit images with the retargetable TCE code generation tools. The compiled program is executed without operating system.

use private local instruction memories where large parts of the whole executed software is replicated.

Fig. 15 shows the software stack in the generated MCASIPs. The application is described using one of the supported parallel programming models. In addition, the *Dthreads* library is an API implemented for this design methodology that can be used directly for programming. In the picture *Dthreads* is drawn below *OpenCL* as it can be used to implement the execution of multiple *OpenCL work-groups* and kernels in multiple (distributed) threads. The retargetable LLVM-based [105] compiler is responsible for mapping the whole input program along with its threading library to the customized instruction set in the single core customization methodology. The code generation supports optional instruction compression to further reduce the required instruction memory size.

5.5 Thread Scheduling

In order to execute, coordinate and load balance threads in the designed multicores, an efficient thread runtime scheduler is needed. In the proposed LDDM memory configuration, the threads run in their own private cores accessing local memories that are visible only to the threads executing in the same core while still being able to random access the shared data address space.

The multithreading in the proposed customizable multicore template is assumed to be implemented fully in software with the management structures optimized to exploit the randomly accessible private and shared memories.

The threads run until completion unless they yield execution time to other threads, for example, while waiting at a barrier. Real time features and pre-emption emulation are supported by means of compiler assisted threading, as proposed in [106].

The design choices and the optimizations in the threading runtime on the proposed processor template are discussed later in this Chapter.

5.5.1 *Dthreads: a Distributed Threading Library*

Traditional C-based threading libraries such as *Pthreads* (POSIX threads [19]) assume that all threads share a single common address space where also the stacks of the threads and the heap reside. This assumption can be seen for example in thread creation; the thread argument data is passed as a pointer to some arbitrary data without size information. The launched thread can then read the data on demand through the passed pointer which can even point to the stack of the launcher thread.

Pthreads work well in the SDDM configuration where the default address space is visible to all the threads in the system. In case of LDDM, each thread is created in a local context and cannot make assumptions in which core it ends up executing.

From the programmer's point of view, the API of *Dthreads* resembles the customary *Pthreads* API. In fact, the main API difference is the aforementioned thread argument passing example; the thread argument data must be copied to the thread table as there is no guarantee the thread will execute in the same core (thus access the same local memory) where it was created in. Therefore, in thread creation, the programmer must pass the size of the argument data along with the argument data pointer.

Another additional feature not present in the standard Pthreads is the possibility to pin a thread to the core it was created in, enabling fast creation and execution of threads inside a single core.

```

#include <dthread.h>

#define V_N 800
#define V_WIDTH 2048
#define THREADN V_N

volatile __shared__ float a[V_N][V_WIDTH];
volatile __shared__ float b[V_N][V_WIDTH];
volatile __shared__ float products[V_N];

void* dot_prod(void* args) {
    int i;
    int thread_id = *(int*)args;
    products[thread_id] = 0.0f;
    for (i = 0; i < V_WIDTH; ++i) {
        products[thread_id] +=
            a[thread_id][i] * b[thread_id][i];
    }
    return NULL;
}

int main() {

    int tid;
    dthread_t threads[THREADN];
    dthread_attr_t attr;
    dthread_attr_init(&attr);
    for (tid = 0; tid < THREADN; ++tid) {
        dthread_attr_setdetachstate(
            &attr, DTHREAD_CREATE_DETACHED);
        dthread_attr_setargs(&attr, &tid, sizeof(tid));
        dthread_create(&threads[tid], &attr, dot_prod);
    }
    return 0;
}

```

Fig. 16: Dot product using the proposed Dthreads API.

Fig. 16 shows an example of a Dthreads program that computes a dot product with multiple threads and a parallelizable for-loop. It performs the dot product for 800 of 2048-wide *float* vectors using 800 threads. A reader familiar to Pthreads should notice the API similarity. In addition to the aforementioned difference in thread cre-

ation (argument data is passed with an additional API call), another notable detail is that the input and output buffers are explicitly marked to reside in the shared memory using the `__shared__` keyword. Other data such as the automatic variables reside in the core local memories.

In addition to the similarity with Pthreads, other design requirements for Dthreads with their implementation details are presented in the following subsections.

5.5.2 Requirement: Low Instruction Memory Footprint

The goal of having as low instruction memory footprint as possible for the threading runtime was reached by including only the most important thread programming interfaces in the library and avoiding the use of complex standard library functions such as `malloc()`. The supported functionality include thread creation, joining, the mutex synchronization primitive, thread argument passing, and a thread scheduler with threads running always to completion (no pre-emption). The aggressive full program dead code elimination of the LLVM compiler infrastructure [105] used in the TCEMC compiler removes threading functions not used by the program from the final program image.

The measured ROM footprint for a minimal threaded program (such as the one in the example of Fig. 16) is in the order of ten kilobytes, depending on the width of the instructions and the instruction compression scheme of the designed core. The thread scheduling and initialization functionality takes approximately half of this space. Data memory consumption is highly dependent on the maximum thread function argument size and the thread stack size.

5.5.3 Requirement: Autonomous Execution

All cores in the multicore are assumed to be identical and can execute autonomously without any “master control/scheduler processor” orchestrating the thread execution. Hence, each core can create and fetch new threads freely from the thread table residing in the shared memory. The implementation involved a start-up function that chooses one of the cores to execute the first thread that runs the *main function*. All cores also initiate an idle thread which is executed when there are no active threads in

the system. Thus, the other cores wait in their idle threads until the main function creates new threads for them to execute. The idle thread functions serve as places where to implement power saving schemes to reduce power consumption during thread idle time.

5.5.4 Requirement: Minimal Number of Shared Memory Accesses

Especially in the LDDM configuration, the shared memory accesses are assumed to be much more expensive than the local memory accesses, and present a potential bottleneck for the throughput of the multicore. Therefore, the threading library should minimize accesses to its book keeping data structures in the shared memory and rely on local memories as often as possible.

Dthreads splits the thread bookkeeping to two: the *Shared Thread Table (STT)* and the *Local Thread Table (LTT)*. The former resides in the shared memory and contains only unstarted or dead threads. As soon as a core requests for threads to execute, a thread is obtained from the STT to the LTT where also the thread stack is initialized. The thread accesses the STT next time only at its exit or when joining another thread. At that point it informs the STT that it can be joined or its resources freed, depending on the *detach state* of the thread.

In order to support synchronization of the shared memory accesses, the cores must include either an atomic *Compare-And-Swap (CAS)* operation or the proposed datapath integrated lock unit instructions as described in Chapter 7. Both alternative synchronization instruction sets are supported by the Dthreads runtime implementation and can be chosen at the program link time.

5.5.5 Requirement: Scalability

The threading runtime library must adapt to the number of cores available without software modifications. This enables fast experimentation with different number of cores as recompilation of the code for each variation is not necessary. In addition, the performance of the threading routines must not degrade dramatically when the number of cores or threads are increased.

In the proposed multicore template, the number of cores is a single integer parameter in the architecture description format. Dthreads implementation is unaware of this

parameter and does not need any special core identification instruction to be present in the multicore's instruction set. The core identification and scaling is implemented in Dthreads with a global counter in the shared memory. Each core gets their core id from this global counter and copies it to their local memory for later use. The sizes of threading book keeping data structures are not dependent on the total core count, thus making the recompilation unnecessary.

Performance degradation when increasing the core or thread count is avoided by using constant time thread creation and scheduling routines, and by minimizing the critical sections to reduce the time multiple cores wait to access the STT.

In order to avoid a lock contention problem of a single *Ready Queue (RQ)* implementation, a solution was found by combining the *work sharing* and *work stealing* techniques [107]. In the thread scheduler, each core has its own RQ in the shared memory, guarded with a separate lock. When new threads are created, they are distributed to different RQs in a round robin manner (work sharing). In the case a core runs out of tasks to execute, it tries to steal work from another core's RQ (work stealing). What is important in improving the performance is that neither the thread creation nor work stealing *waits* for locks, but only *tries* to acquire one. In case a core wanting to add or steal threads cannot acquire a lock to a core's RQ at the first attempt, it simply proceeds to the next core's RQ until it manages to lock an RQ. This distributes the lock contention problem to the number of cores in the system, thus improves the scalability of the Dthread runtime scheduler when there are tens or even hundreds of cores competing for work.

6. OPEN COMPUTING LANGUAGE SUPPORT

This Chapter describes a methodology and compiler techniques involved in applying the Open Computing Language (OpenCL) as an input programming standard in the proposed design flow of customized parallel processors.

While the proposed design flow is input language independent, the first priority programming standard within the research has been OpenCL. The OpenCL execution model also has driven the design of the multicore template described in Chapter 5. The key motivation to focus on OpenCL is its wide industry adoption which helps program portability. Support for OpenCL has been announced from companies such as Apple, NVIDIA, AMD, Intel, and S3. Thus, it seems OpenCL is an important parallel programming standard to support in the future.

In this Chapter, compiler techniques for scalable static parallelization of OpenCL kernels are proposed. The parallel nature of OpenCL C is utilized in the proposed kernel compiler by extracting instruction-level parallelism (ILP) from the kernel instances to improve the utilization of the available hardware resources in the statically instruction scheduled processor architectures designed with the proposed methodology. In addition to the parallel execution, an additional important feature is a method for explicitly executing custom hardware operations from the OpenCL C kernels using the OpenCL vendor extension API.

6.1 *Related Work*

In the recent years, there have appeared a few publications on using GPGPU programming paradigms for generating code for non-GPU devices. Many of those papers describe the use of the proprietary CUDA [27] language as the input while the proposed work is based on the standardized OpenCL. However, as OpenCL and CUDA are very similar, these projects are considered related to the proposed work.

MCUDA [108] is a framework that aims to replace the suboptimal CUDA to x86 compilation tool of the NVIDIA SDK with a version that parallelizes the execution on multiple host cores. The framework creates loops out of multiple work-item execution to retain work-group barrier semantics. FCUDA [109] is a source-to-source translator built on techniques implemented in MCUDA.

In MCUDA and FCUDA, the parallelization is considered only at the task level while in our work the focus is on ILP. Exploiting the ILP within a single wide statically scheduled core has its benefits as there are less off-core synchronization and communication required because more of the shared data between work items can be stored in fast general-purpose registers.

CUDA is used as a starting point for hardware accelerator generation for FPGAs in [110]. The approach is exemplified with a kernel used to implement the *MrBayes* algorithm. The paper shows a procedure on how to map the relatively simple kernel of this algorithm to a pipelined hardware design. The approach differs from the proposed one mainly in the programmability. While the proposed approach is based on a processor template with simplistic control logic, their approach generates directly hardware constructs with control implemented as state machines. In addition, the approach they present is not automated while the aim in the proposed work is to provide a fully operational co-design tool flow.

Soon after the OpenCL specification was published, interest arose to apply the programming standard for FPGA design flows. Fletcher et al. [111] compare a hand-optimized FPGA implementation of an OpenCL-based Bayesian inference application between FPGAs and GPGPUs to study the performance potential from the customizable hardware in comparison to the more general-purpose processing platforms of GPU architectures. They conclude that the customization of the data paths and the memory hierarchy can bring significant performance gains (speedups from 3.14 to 4.25). However, they also state that in order to make OpenCL to FPGA design engineering effort feasible and easy enough, such flows call for a customizable many-core template-based approach. Such an approach is presented in this Thesis.

The techniques proposed in this Thesis were one of the earliest published work (2010) in the area of OpenCL-based customized processor implementations [15]. OpenRCL [112] was proposed around the same time. The main difference is that OpenRCL uses a simple scalar core based processor template while the proposed design

flow exploits a fully customizable static multi-issue TTA processor template that can also support fine-grained parallelism. SOpenCL [113, 114] uses a datapath template that is very similar to the bare boned TTA processor template. However, their work is aimed at generating non-programmable hardware accelerators while the proposed design flow aims at retaining software programmability.

6.2 Open Computing Language in Hardware/Software Co-Design

In the context of hardware/software co-design, while OpenCL provides the most useful characteristics of C, such as the low level programming model, the following differences and additional features of OpenCL C stand out:

- *Explicit independence between work-items and work-groups.* As the execution, including memory accesses not only to the “private” storage but also to shared “local” and “global” memories, is assumed to be independent between work-items, it is possible to parallelize code from multiple work-items at different granularities of parallelism. All synchronization is done by explicit barrier and memory fence calls.
- *Support for multiple disjoint address spaces* helps in intra-work-item alias analysis and enables explicit access to multiple separate memories.
- *No dynamic memory allocation.* The data memory demands of the kernels can be more accurately estimated at compile time.
- *Vector data types.* Allows defining vector computation which can be trivially parallelized.
- *Recursion not supported.* Enables aggressive procedure call inlining.

6.3 Compiling OpenCL for Static Customized Processors

The core algorithms and concepts used in efficient compilation of OpenCL kernels to the customized statically parallel processors are described in Sections 6.3.1-6.3.4. Section 6.3.5 describes the way the compiler uses the OpenCL vendor extension API to let the programmer access custom operations in the processor design.

6.3.1 Standalone and Hosted Setups

OpenCL is a computing language that is primarily meant for programming heterogeneous multicore platforms. However, as one of the goals of OpenCL is to enable portability across various platforms, it is possible to execute full OpenCL programs purely using a single processor. This notion leads to two different setups for the designs:

1. *Standalone.* The custom processor executes both the OpenCL host and device code. In this mode, the compiler compiles and links both the host and kernel programs together to a single processor binary to be executed on a standalone customized processor. No OpenCL support is required from the (possible) host processor of the custom processor. However, the whole source code of the kernel must be available for offline compilation to produce binaries of the kernels, unless the custom processor also includes an OpenCL C compiler, which is usually unrealistic.
2. *Host/device.* The custom processor executes only the kernels implemented with OpenCL C and is controlled by a host processor. This is the standard CPU/GPU setup and requires OpenCL runtime and platform APIs to be implemented in the host. It supports also kernel code modification at runtime as the kernels can be recompiled on the host.

In the experiments presented in Chapter 8, the standalone setup was used to produce standalone custom processors. I.e., in the terms of the OpenCL platform model, the customized TTA processors act as the *host*, the *compute device*, and the *compute unit* at the same time. The function units of the TTA can be considered to be *processing elements*.

In the terms of the OpenCL memory model, the *global memory* and the *constant memory* are mapped to the shared address space and the *local and private memories* map to the local memory and general-purpose registers of each core.

6.3.2 Parallelizing Work-Items

According to the standard, the OpenCL work-items are completely independent from each other. Thus, it is possible to chain code from multiple work-items in the same

sites. That is, whenever a single work-item reaches a barrier, it cannot proceed its execution until the rest of the work-items in the work-group have reached it. Thus, in case the work-items are to be parallelized statically, the kernel has to be split at the barrier call sites and the parallelization should be conducted independently in the split parts.

The example in Fig. 18 shows the chaining of two work-items in case of a kernel with a barrier call in the middle. In this case the kernels are chained by duplicating and appending the basic blocks before the barrier and connecting the last basic block in the copied chain to the “barrier pseudo basic block” (which is just an instruction scheduling barrier in this case) and similarly duplicate and chain the basic blocks after the barrier.

In this example, the code before the barrier also includes an if-else structure. In such case, each control flow structure needs to be duplicated as a whole for each work-item due to the single program counter execution. A succeeding if-conversion [40] pass attempts to convert these control structures to single instruction level parallelizable predicated basic blocks. However, the code after the barrier is a single basic block without branching, thus the chaining algorithm can join the basic blocks of the two work-items to a single one.

When there are barriers inside a conditional basic block or a loop body, the work-item chaining becomes more complex as the problematic nature of static compilation of independent execution using a single program counter becomes more apparent. According to the OpenCL standard, in case of a loop with barriers, each iteration of the loop is synchronized separately. Thus, when a single work-item reaches the barrier in an iteration, it waits for the rest of the work-items to complete the code before the barrier at that iteration. Conversely, when there is a barrier inside a loop, it can be assumed that all the work-items execute the loop the same number of times, otherwise the end result is undefined (the barrier causes a subset of work-items to lock up indefinitely).

The work-item chaining in the case of barriers inside loops can be done by treating the loop body independently from the loop construct as proposed in [108].

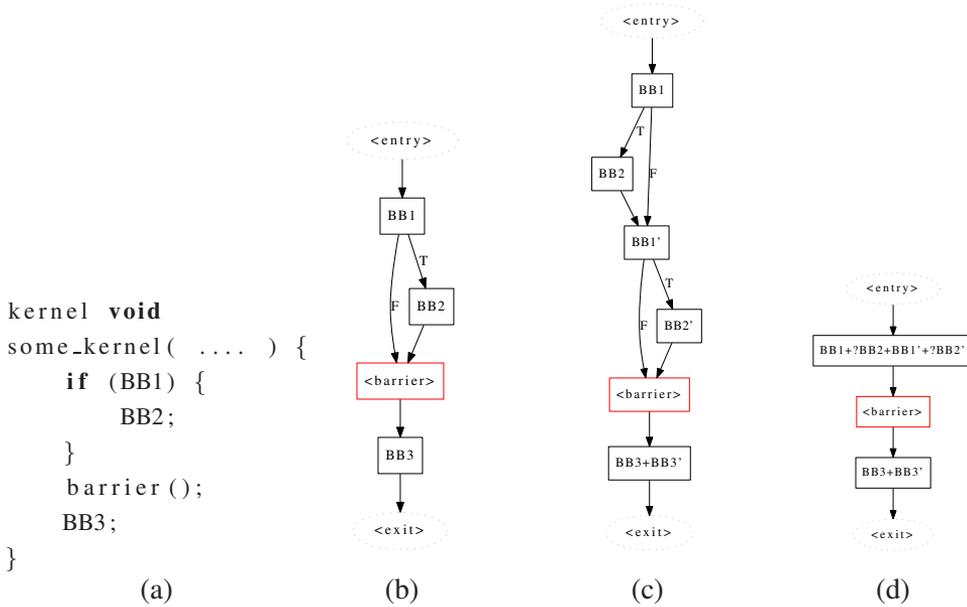


Fig. 18: Work-item chaining with barriers: (a) OpenCL C kernel source with three basic blocks (BB), (b) initial kernel CFG, (c) two work-items chained, and (d) the CFG after branching eliminated with if-conversion. Question marks denote dependencies on a predicate register.

6.3.3 Work-item Chaining Algorithm

The algorithm for statically generating code for every work-item (effectively replicating the kernel code the required number of times) was implemented as a set of closely related LLVM optimization passes. The high-level structure of the replication process is shown in Fig. 19.

The first step for the algorithm is to find the barriers, i.e., calls to OpenCL C *barrier()* API function, present in the kernel code. As the barriers do not need to be in the main kernel function code, but might have been placed by the programmer in some of the kernel called subfunctions, a prior “flattening” is required. This process performs aggressive function inlining for all non-kernel functions, thus ensuring that kernels themselves have no calls once flattened. Apart from easing the barrier detection, flattening also improves the results of the instruction scheduling due to creating larger basic blocks.

Each region between barrier calls is then processed independently. In order to follow the OpenCL programming model, the regions need to be executed a number of times

```

LLVMOPENCL(module)
1  for each Function  $f \in \textit{module}$ 
2      do if IsKernel( $f$ )
3          then FLATTEN( $f$ )
4              DETECTBARRIERS( $f$ )
5              for each Region  $r \in f$ 
6                  do LOOPREGION( $r$ )
7                      REPLICATECODE( $r$ )
8  return module

```

Fig. 19: Work-item code replication algorithm.

equal to the work-group size. This can be achieved by two different ways: creating loops or replicating the code for each work-item. The former has the advantage of keeping the code size small, but results in less ILP to be exploited, while the latter creates more ILP but can lead to huge programs needing plenty of resources and processing time to schedule.

In order to parametrize this tradeoff, the kernel compiler provides a runtime parameter to determine the maximum number of replications to be performed per region, thus the number of work-items potentially executed statically in parallel, and generates the remaining work-item executions using loop structures.

The region replication algorithm works like the basic loop unrolling that is modified to mark instructions belonging to different work-items with a unique annotation to help alias analyzer in recognizing independent instructions. Each basic block is replicated, maintaining the intra-region control flow structure, and an unconditional branch is then added at the end of the previously existing region to ensure the replicated code is run after the original. This process is repeated as many times as required according to the number of parallel work-items to be created.

When processing references to data, it should be noted that only *private* variable definitions must be replicated. Kernels can also have *local* variables, which are shared among work-items in the same work-group, and, therefore, must not be copied. This variable sharing, however, does not hinder parallelization, because the OpenCL standard specifies a relaxed memory consistency model in which accesses to local vari-

ables from different work-items are considered to be independent. The programmer must ensure that different work-items do not access the same variable at the same time by careful coding or by using explicit synchronization mechanisms such as barriers or memory fences.

The region chaining algorithm is designed to generate valid and easy-to-debug code, but it does not perform any optimizations. As such, it creates several basic blocks connected by unconditional branches, which can be combined into larger basic blocks.

After the region replication has been performed for each kernel, the entire code is linked with the host program (in case of the standalone execution mode) and a global optimization stage takes place to reduce this unoptimized code to a smaller and more efficient form.

6.3.4 Instruction Level Parallelization of Work-Groups

The processors generated with the proposed design flow are statically scheduled architectures with possibly hundreds of programmer visible general-purpose registers. In order to improve the parallelization opportunities presented to the post-pass instruction scheduler, the work-item independence should be communicated to the register allocator (RA) in order to avoid allocating overlapping registers between the work-items which cause additional restrictions to the scheduling freedom (also known as register anti-dependencies).

A simple example of Fig. 20 shows a schedule with (a) two work-items sharing registers, and (b) work-items containing new registers assigned to the produced variables. The first schedule is only partially parallelized, while the second produces a completely instruction parallel execution of the two work-items, saving one cycle in the execution time.

The register allocation can be performed using the default LLVM register allocator (RA) implementations by adding a pre-RA *variable partitioning pass* that assigns the variables used by the different work-items with different sets of registers. This works especially well with homogeneous clustered datapath designs where the datapath consists of symmetrical “cluster nodes” with their own register files and function units. In case of a clustered machine, the partitioner assigns the work-item variables in round robin manner to different cluster nodes according to the work-item id,

leading to minimal antidependencies due to register sharing between the independent work-items. In case of a non-clustered machine design, the variables are allocated in a round-robin manner to the possible multiple register files in the design.

```

cycle work-item 1      work-item 2
  0 add r1, r2, r3
  1 mul r4, r1, r3    add r1, r2, r3
  2 stw #100, r4      mul r4, r1, r3
  3                   stw #104, r4

```

(a)

```

cycle work-item 1      work-item 2
  0 add r1, r2, r3    add r6, r2, r3
  1 mul r4, r1, r3    mul r7, r6, r3
  2 stw #100, r4      stw #104, r7

```

(b)

Fig. 20: Effect of avoiding register reuse in work-item register allocation: a simple example when (a) reusing registers, and (b) using separate registers. The first operand of the instruction is the destination.

Another source for data dependencies in programs that lead to unnecessary serialization are the memory accesses. In case the program contains stores to memory, it is legal to move a succeeding load before or parallel with a store only if it is known that the store and the load never access the same memory location (alias). When scheduling instructions from multiple work-items of OpenCL C kernels in parallel there are several useful properties to assist the alias analysis.

1. Accesses to the different address spaces cannot alias. That is, even in case the *global* and *local* memories were mapped to the same physical address space, the compiler can treat them as disjoint areas and reorder the accesses freely.
2. Accesses through pointers to the constant memory can be assumed to be only reads. Thus, no overlapping with non-const pointers can happen. Furthermore, as the constant memory is known to be truly read-only no write can alias with constant memory reads, in contrast to the const pointers in C/C++, for example, which can point to memory that is modified by non-const pointers.
3. Most importantly, in the regions between work-group barriers, the memory accesses of different work-items can be considered not to alias due to the re-

laxed consistency model of the OpenCL specification. This allows treating the chained work-items as fully independent regions of code.

The alias analyzer of the instruction scheduler takes advantage of these special properties of OpenCL C to minimize the data dependencies that restrict parallelization of the work-item chained code, resulting in more scheduling freedom.

6.3.5 Custom Operation Support

The use of custom operations also known as special instructions or special function units (SFUs), is often the most important way to accelerate the execution of an application running in a customized processor. The capability of a processor template to support custom operations without restrictions to their complexity enables gradual optimization of the architecture by increasing the specialization and complexity of the custom operations until the performance is equal to an accelerator implemented as a tailored fixed function hardware block. However, more complex the custom operation at hand is, less likely it is that the instruction selector finds it automatically from the intermediate representation of the program. That is, in order to make complex custom operations feasible, it is crucial to provide seamless support for programmers to call custom operations at the source code level.

The OpenCL standard defines an API to provide support for vendor specific extensions (see [26], Chapter 9). This API is used in the proposed implementation as a means to access the custom operations available in the target processor. The standard requires the OpenCL C compiler implementation to generate specifically named preprocessor macros when an extension is supported. In the proposed compiler, the required headers and macros to produce the inline assembly that triggers the custom operations are generated automatically from an architecture description file. Thus, it is possible to compile the same OpenCL C kernel code both to targets that support and to targets that do not support the custom operation in question by using the preprocessor to select the accelerated custom operation or the software-only version. One important benefit from this is that the custom operation accelerated program can still be compiled with a 3rd party OpenCL SDK without modifications in case a software alternative of the custom operation functionality is provided.

An example code snippet that uses a 2-input-2-output custom operation *ADDSUB*,

```
#ifdef cl_TCE_ADDSUB
    cl_ADDSUBTCE(a, b, c, d);
#else
    c = a + b;
    d = a - b;
#endif
```

Fig. 21: Example of using a custom operation inside an OpenCL kernel in a portable way.

which adds and subtracts its operands in parallel is shown in Fig. 21. The *#else* branch executes the same operation in software to maintain portability while the main branch executes the custom operation using the automatically generated intrinsics.

The compiler techniques and the custom operation support are evaluated in Chapter 8.

7. HARDWARE ACCELERATED SYNCHRONIZATION

The customization point of core count in the proposed customization flow enables matching the task level parallelism provided by the processor to the requirements of the application at hand. However, improving the performance by adding more cores to the design is often hindered by the overheads due to the need to synchronize the execution of multiple threads running in multiple cores. This Chapter describes a simple hardware unit called *Datapath Integrated Lock Unit (DILU)* that alleviates the overheads from synchronization.

7.1 Related Work

Techniques to reduce overheads of software based synchronization implementations have been studied widely. For example, *Adaptive Backoff Techniques* use simple heuristics to compute a time to wait before polling the barrier variables again to reduce the traffic [115–118]. These approaches usually rely on a cache coherent memory hierarchy to enable fast spinning on a local cached copy of the lock variable. However, cache coherence hardware brings additional chip area costs to the design which are avoided in embedded multicores with explicitly accessed local memories.

Another approach is to use dedicated synchronization hardware. One of the earliest works is presented by *Beckmann* and *Polychronopoulos* in [119] that supports barriers by means of a barrier register hardware. Each 1-bit register denotes whether each processor has reached the barrier or not. Their work relies on a synchronization primitive specific hardware, while the DILU approach leans towards the software side, thus adding more flexibility.

SGI Origin2000 implemented *read-modify-write (RMW)* operations inside memory controllers [116], an approach referred to as *Active Memory Unit* in [120]. These techniques implement simple RMW operations as atomic operations inside the mem-

ory unit without producing unnecessary cache invalidation traffic. However, these approaches still occupy the shared memory controller for the atomic memory operations while DILU can free the shared memory to be used purely for useful computation. In addition, they assume spinning can be done on a local cache, a luxury that is not available in non-cache coherent multicore systems.

In general, many software-based synchronization optimization algorithms assume availability of fast local globally accessible memory implemented by means of cache coherence or distributed memory [115]. While such memory hierarchies clearly improve programmer-friendliness in general-purpose computing, it is arguable that in case the applications executing in the customized processor do not otherwise benefit from such an expensive [104] memory hierarchy, synchronization should not pose the sole motivation for adding one to the MCASIP design at hand.

The *Synchronization-operation Buffer (SB)* [121] reduces the spinning overheads by performing the polling independently from the processor core using dedicated hardware unit in the memory block. The unit sends notifications to the processor after a memory location changes its content to a desired value. Like the proposed method, also SB avoids the need for coherent caches. However, SB monitors the shared memory bus for updates to the interesting variables, in contrast to DILU which decouples itself completely from the shared memory.

Synchronization counters (SC) [122] present a mechanism to attach synchronization hardware to heterogeneous systems where the included processor architectures do not support atomic operations. The memory mapped hardware implements atomic incrementing of counters when loads are issued to addresses mapped to them. Using the counters the waiting cores are assigned integer identifiers that denote each core's turn to access the lock. While the scheme is interesting as its hardware is highly simplified and can be easily integrated to existing systems, the checking for the turn (updated by the core that releases the lock) still requires polling the memory. An optimization that uses local memory for the turn variables to reduce the polling overheads is also presented in the paper. However, the optimization requires globally visible random accessible local memories (implemented via cache coherence or distributed memory) which are not available in the simplified memory hierarchy in the proposed multicore template.

Distributed Synchronization Controller (DSC) [123] is an approach close to DILU.

Both consume two lock registers per barrier. The main difference is that in DSC the monitoring synchronization traffic and updating the synchronization registers happens in hardware while DILU generalizes the concept of lock registers and pushes the main synchronization implementation logic to the software side.

The idea of a set of memory mapped *test-and-set registers* implemented using a separate hardware unit is presented in [124]. Such registers are present in the *Intel's Single-chip Cloud Computer (SCC)* [125]. The lock registers are coupled with the memory hierarchy.

The *Lock Control Unit (LCU)* presented in [126], similarly to DILU, decouples the lock unit hardware from the shared memory hierarchy. The fundamental difference between LCU and DILU is that LCU aims to provide a complete (but more complex) solution for general-purpose computing while DILU is tailored for the customized processor co-design use case. That is, the base assumptions for the DILU design are that the multicore including the synchronization hardware runs well-behaved parallel programs without complex operating system support and that the DILU is used for synchronization only within a single multicore. Global synchronization of multiple processor nodes in an heterogeneous system, if needed, is assumed to be implemented with a separate synchronization hardware or another method. These assumptions resulted in simpler hardware requirements.

DILU resembles SCC, DSC, SB and LCU in their idea of isolating the synchronization logic to an independent hardware block. The distinctive feature of DILU is that it minimizes the hardware requirements and exposes the lock register manipulation operations to the instruction set of the processor. Moving complexity from hardware to software provides more flexibility for implementing the synchronization software library. Furthermore, isolating the synchronization hardware from the memory hierarchy ensures the same synchronization libraries and hardware can be used with various memory configurations.

7.2 Datapath Integrated Lock Unit

The polling during spinning in the lock variables used to implement the software synchronization primitives causes spurious traffic to the shared memory hierarchy causing unnecessary slowdown to other threads still executing their actual payload

code. The need to reduce shared memory contention due to synchronization and the varying synchronization needs among different customized multicore processors, led to the design of the proposed hardware lock unit. The novel feature is to expose a simple address lock book keeping hardware to the programmer by integrating it with the processor datapath. Using this *Datapath Integrated Lock Unit (DILU)* the programmer is free to implement various atomic operations in software using the lock variable book keeping functionality for mutual exclusion. The goal in the simplified hardware interface was to produce a simple lock unit that can be generated automatically according to the needs of the processor design at hand while providing enough flexibility to support various synchronization primitive implementations in software.

The *Lock Unit (LU)* hardware consists of a custom sized *lock register* file. The number of registers, connected cores and address bits are configurable. Each lock register stores the status of a shared memory address currently being locked. Unlocked shared memory addresses do not consume any registers. LU ensures there is at most one lock register reserved per address at a time. These requirements for the lock unit hardware can be implemented in multiple ways.

The example hardware design of the LU is shown in Fig. 22. The inputs from the cores are shown on the top and outputs on the bottom. Each core provides an address and a 3-bit command. LU can stall the core until it can be served, and after that, LU gives the status of the completed operation.

The example implementation of the LU needs two cycles for each operation (register update or status reporting). During the status cycle it uses round-robin to arbitrate which core gets the next access to the lock register file.

Most of the chip area of the LU hardware is consumed by the register file and the comparators that find the slot to be reserved or released. In addition, comparators are used to prevent double-locking of an address. The lock register file is split to two blocks in the picture: locked address and an associated status bit. The lock status(*i*) is 1 in case the address register (*i*) contains a valid locked address and 0 otherwise. The control logic is rather simple. It controls the stalling of the cores and produces the lock status based on the comparator outputs and the lock status bit.

The HW unit supports 4 basic operations: *read*, *lock*, *unlock*, and *wait until unlocked*. *Read* and *unlock* operations are non-blocking, while the *wait until unlocked* and *lock* can be implemented in both ways. The instruction set presented in this work utilizes

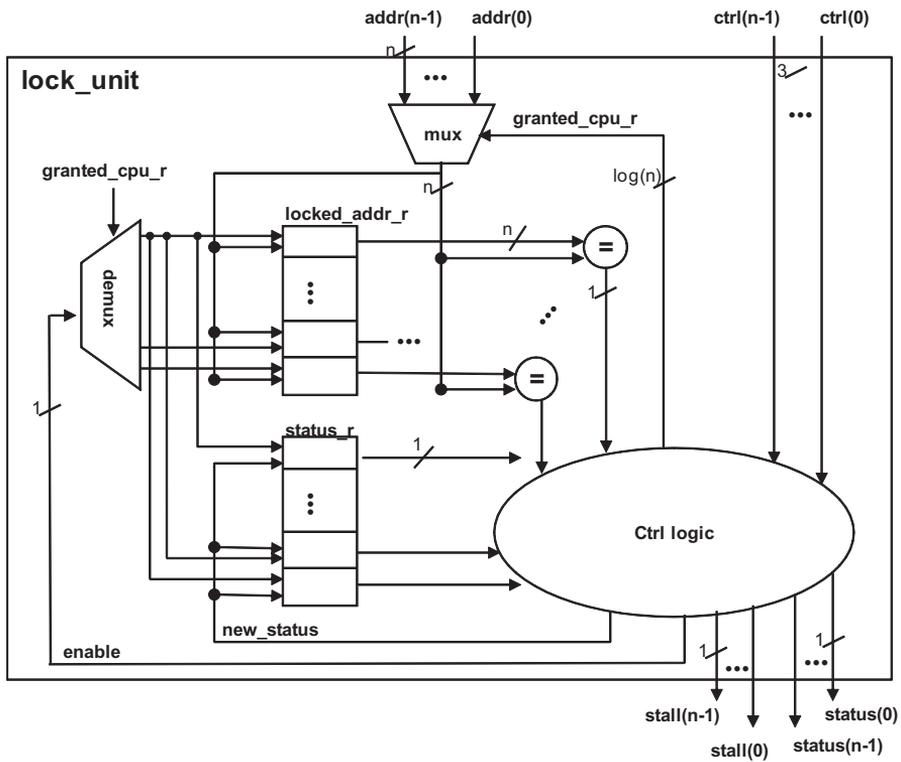


Fig. 22: An example hardware design of the lock unit.

only the non-blocking operations.

Each of the cores in the MCASIPs designed with the methodology includes a function unit, referred to as the *SYNC FU*, that is used to access the lock unit. Its operation set is as follows:

```

got_lock := TRY_LOCK A   Tries to acquire a lock at address A.
UNLOCK A                 Unlocks the lock at address A.
status := READ_LOCK A   Reads the lock status of the address A.

```

Additional blocking versions of the operations that stall the core until getting a lock could be supported as well in order to reduce instruction fetch overhead. However, the basic versions of the implemented software primitives presented in the following assume non-blocking lock instructions.

Two alternative implementations for both the spin lock and the barrier synchronization primitives were implemented using the above operations: one that minimizes the

```

// spinlock_fast                                // spinlock_basic :
lock_lu(lock_var* A) {                          lock_var global_lock;
    while (!TRY_LOCK(A));                       spin_lock(lock_var* A) {
}                                                try_lock:
wait_lu(lock_var* A) {                          lock_lu(&global_lock);
    while (!READ_LOCK(A));
}
                                                if (*A == 1) {
                                                    unlock_lu(&global_lock);
                                                    goto try_lock;
                                                } else {
                                                    *A = 1;
                                                    unlock_lu(&global_lock);
                                                }
}

```

Fig. 23: Two spin lock implementations using the lock unit instructions. The *fast* version uses the lock unit operations solely while the basic version uses a single global lock register to guard accesses to all lock variables. *wait_lu()* will be used in a later barrier example to implement spin waiting on a lock register without lock acquisition.

count of used lock registers per primitive and another that minimizes shared memory accesses by relying more on the lock unit registers.

The spin lock implementations are shown in Fig. 23. *spinlock_basic* uses a single global lock unit register to guard accesses to all lock variables residing in the shared memory. First, the function *lock_lu()* is called and it blocks until the global lock is acquired. If the actual lock variable contained in shared memory address *A* is locked (contains value 1), the global lock is unlocked and the lock acquisition is retried by jumping back. This consists the “spin loop” which is retried until the lock variable is 0 and can thus be locked by the core. Depending on the guarded critical section length and a potential “spin backoff algorithm” used, this version generates heavy traffic to the shared memory bus during its spin wait because the lock bit that is polled is stored in the shared memory.

On the other extreme, *spinlock_fast* in *lock_lu()* assumes that lock information can be stored fully into the lock unit registers, thus it does not access shared memory at all during mutual exclusion. All spinning is done by using the instructions of the lock unit. Thus, the number of these lock primitives that can be “alive” at the same time in a program is limited by the number of lock registers in the DILU. Another restriction is that the program should not rely on the shared memory value of the lock as that

```

barrier_fast(barrier_t* b) {
    int executing;

    lock_lu(&b->count_l);
    b->running -= 1
    executing = b->running;

    if (executing == -1) {
        /* init barrier */
        executing =
            b->running =
                b->total - 1;
        lock_lu(&b->barrier_l);
    }
    unlock_lu(&b->count_l);
    if (executing == 0)
        unlock_lu(&b->barrier_l);
    else /* spin wait */
        wait_lu(&b->barrier_l);
}

lock_var global_lock;

barrier_basic(barrier_t* b) {

    int executing;

    lock_lu(&global_lock);
    b->running -= 1
    executing = b->running;

    if (executing == -1) {
        /* init barrier */
        b->running =
            b->total - 1;
    }
    unlock_lu(&global_lock);

    /* spin wait */
    while (b->running > 0) {}
}

```

Fig. 24: Two barrier implementations using the lock unit instructions: *basic* uses a single lock register to guard all atomic operations, *fast* uses two lock registers; one to guard the thread counter, another for signaling the barrier completion.

value is not updated by the spin lock implementation at all.

The programmer or the compiler is responsible for deciding which synchronization function version to use in which occasion. A middle ground implementation between the *basic* and *fast* that would reduce contention on a single global lock register would be to use two or more lock registers that each guard an even share of lock variable memory addresses.

The barrier implementation alternatives are shown in Fig. 24. *barrier_basic* uses a counter variable in shared memory. The counter is used to count how many threads are still to reach the barrier. Updates to it are protected with a single global lock register. After acquiring the lock, a thread decrements the counter to denote that it has arrived. If it was the first one, the counter goes to -1 and the barrier must be set up. After releasing the global lock, it spin waits until the counter goes to zero.

The fast version, *barrier_fast*, uses a counter variable similarly, but consumes two lock unit registers per barrier, named *count_l* and *barrier_l*. The first lock guards the counter variable updates and the latter records the whole barrier status. The first

Table 1: Required lock registers and shared memory read (r) and write (w) accesses for the alternative lock unit based synchronization implementations. The shared memory accesses are per thread participating in the synchronization. C is a variable depending on the critical section length or the thread imbalance in case of the barrier. L is the number of locks or barriers in use at the same time during the program execution.

lock style	# of shared memory accesses			Required
	acquire	release	spin wait	# of lock regs
basic	$r + w$	w	$C \times r$	≥ 1
fast	0	0	0	$1 \times L$
barrier	init	reach	wait others	# of lock regs
basic	w	$r + w$	$C \times r$	≥ 1
fast	w	$r + w$	0	$2 \times L$

steps are similar to the basic algorithm except the barrier initialization locks also the *barrier_l*. It will be unlocked by the last thread reaching the barrier, i.e. when the count of threads still to reach the barrier goes to 0. This version accesses the shared memory when each thread reaches the barrier the first time (read, decrement, write). However, after that it spins on the second lock register by calling *wait_lu()* which minimizes the shared memory traffic while waiting for the other threads.

The presented barriers are of “one shot” type, i.e., assume the barrier is not reused after “discharged” once. Some additional logic, that was left out for simplicity, is needed for barriers used in loops to ensure all threads have exited the previous barrier call before any thread enters a new call.

Table 1 summarizes the shared memory and lock unit register costs of the alternative implementations. Locks can be implemented without shared memory access at all. In the both cases, the biggest difference is during spinning.

8. EXPERIMENTS

The experiments presented in this chapter evaluate the different parts of the proposed customizable parallel processor design methodology. First, in Section 8.1 the environment in which the techniques were implemented is briefly described. Section 8.2 compares the performance of the TTA approach, used as the single core processor template, to commonly used data parallel datapath design styles: SIMD and SIMT. Section 8.3 presents case studies where the single core customization capabilities of the methodology are tested. Section 8.4 investigates the task level scalability of the proposed multicore template and its accompanying threading library. Finally, Section 8.5 inspects the overheads of the proposed synchronization mechanism by means of a microbenchmark and a 48-core customized processor implementation on FPGA.

8.1 *Experiment Environment*

The methodologies and techniques proposed in this thesis have been implemented in an ASIP design toolset called TTA-based Co-design Environment (TCE) [77, 127]. TCE provides a full design flow from software written in higher-level language programs down to customized parallel program images and RTL implementations of the customized processors.

The single core processor customization flow was extended with the task level customization features accompanied with the proposed thread scheduler, the hardware accelerated synchronization primitives, and the compiler techniques to map OpenCL kernels to the designed TTA processors efficiently. This extended version was used for the benchmarks presented in this chapter.

The most important tools used in the design flow of TCE are its cycle-accurate instruction set simulator [14] which was extended for multicore simulation to retrieve

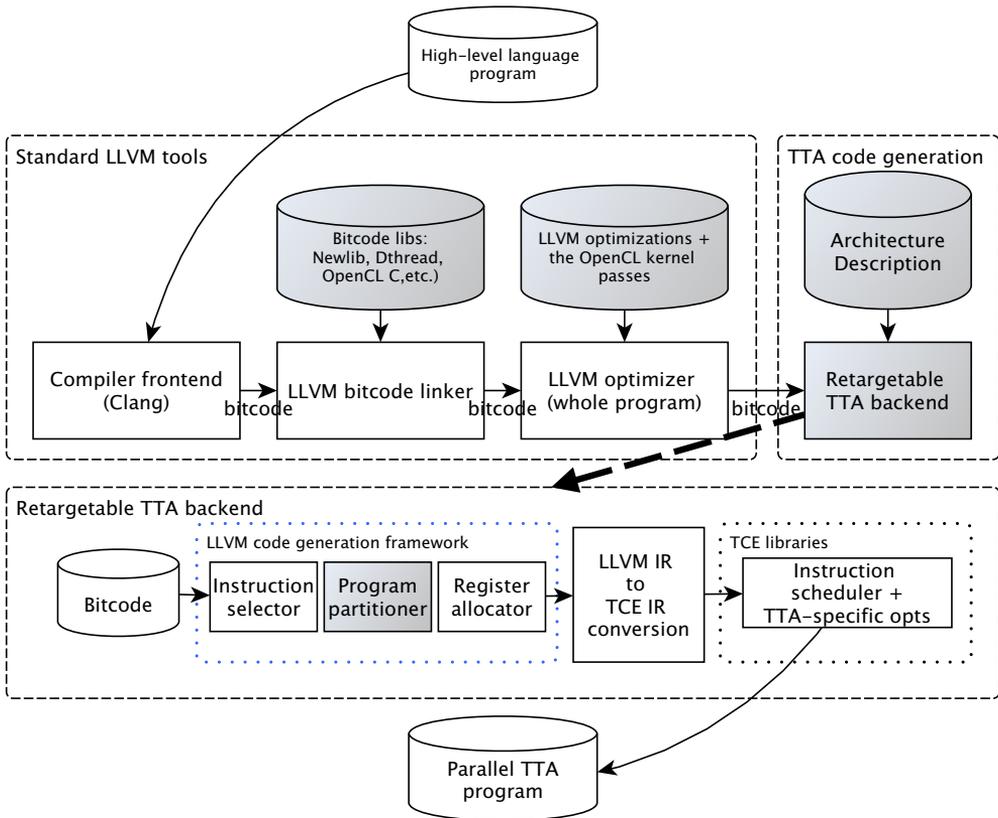


Fig. 25: Overview of the TCE compiler internals. The highlighted parts have been extended for performing the experiments in this Thesis.

the instruction cycle counts, and the retargetable compiler for implementing the compiler techniques for the OpenCL support. The *Processor Generator* [128] was used to produce the RTL implementations for the FPGA implementations.

The compiler techniques presented in this thesis were implemented as additions to the standard TCE compiler. Fig. 25 shows the overview of the TCE compiler internals. TCE compiler uses the *LLVM Compiler Infrastructure* [105] as the backbone, thus benefits from its interprocedural optimizations such as aggressive dead code elimination and link time inlining. In addition to the target independent optimizations provided by the LLVM toolchain, the TCE code generator includes an instruction scheduler with TTA-specific optimizations. The OpenCL-based program partitioner proposed in this Thesis is placed between the instruction selector and the register allocator and the LLVM passes that implement OpenCL C work group parallelization are performed at the whole program optimization phase. These passes have been

published in an open source implementation of the OpenCL standard called Portable OpenCL (pocl [129]) which also provided the OpenCL API implementations for the OpenCL benchmarks.

8.2 TTA vs. SIMT in OpenGL Shader Execution

In this section, the TTA approach is compared to the common SIMT approach in the context of graphics rendering. The programmable vertex and fragment shaders in the OpenGL graphics standard represent SPMD-style computation. An example shader is presented along with a benchmark against an existing implementation of the SIMT approach.¹

8.2.1 The Shader Example

In order to study the practical advantages of TTA over SIMT, let us consider a shader taken from [130] that gives the rendered surfaces a wood-like appearance by using an externally (host-side) generated noise texture. The vertex shader is relatively simple and just prepares some data for later per-fragment computations. The fragment shader, however, has some characteristics that make it adequate for showing how the additional scheduling freedom of TTA/VLIW improves SPMD performance under certain circumstances.

A streamlined version of the fragment shader code is given on Fig. 26. In the SIMT model, each of the lockstep SIMT function units will execute that code over a different fragment. This means that no vector operations are performed even when vector types are used in shader code. The “vectorization” happens by computing several fragments in parallel. In other words, each vector lane executes its instruction for a different fragment.

¹ Brief OpenGL terminology: A *vertex* is a point in a three dimensional space. Multiple vertices form *geometric primitives* such as polygons or lines. A *fragment* is a square in the rendered image that corresponds to a pixel in the final image written to a *framebuffer* (e.g., all the pixels between two vertices forming a line primitive). A *shader* is a piece of program written in, for example, the *OpenGL Shading Language (GLSL)* that performs customized computation on different programmable stages of the OpenGL graphics pipeline (such as the *vertex* or *fragment* processing stages) in order to produce customized visual effects. [130]

```
/* ... */

uniform sampler3D Noise;

/* ... */

void main(void)
{
    vec3 noisevec = vec3(texture(Noise, MCposition * NoiseScale) *
        Noisiness);

    /* ... */

    r = fract((MCposition.x + MCposition.z) * GrainScale + 0.5);
    noisevec[2] *= r;
    if (r < GrainThreshold)
        color += LightWood * LightGrains * noisevec[2];
    else
        color -= LightWood * DarkGrains * noisevec[2];

    color *= LightIntensity;
    FragColor = vec4(color, 1.0);
}
```

Fig. 26: Wood appearance fragment shader.

In general, the SIMT approach performs optimally when the number of vertices to be processed is an integer multiple of the number of SIMT lanes. As discussed in Section 3.2, in the other cases the MIMD TTA/VLIW machines can benefit from their capability to extract parallelism from within one thread. In this case, for example, the multiplication of the color vector by scalar can be parallelized easily in case there are free function units.

The other source of performance improvements comes from the *if...else* construct in the code, in the very likely case that not every fragment being processed in parallel happens to run through the same conditional block. While one of the blocks is executed, SIMT units corresponding to fragments which have to go through the other block are going to be idle. This phenomenon is called branch divergence that is caused by incoherent thread execution in an SPMD program. The proposed TTA with an overcommitting-capable compiler allows processing both blocks in parallel, reducing the resource underutilization.

These two situations correspond to the cases where the scheduling freedom in the proposed single core template based on the TTA can provide performance benefits over the common implementations of the SIMT execution model. Measurements of these improvements for the wood-appearance shader are given in the following.

8.2.2 Benchmarking TTAs against a SIMT GPU

An OpenCL C equivalent of the wood appearance fragment shader was developed in order to exploit the OpenCL C compilers available in both of the experimented platforms. The OpenCL version follows closely the GLSL shader code and generates the colored pixels of the image. The resulting image is shown in Fig. 27. This made it possible to abstract the rest of the rendering pipeline, thus isolating the performance of the shader at hand. The stimulus data was read from pre-generated buffers, and the shaders were executed on a number of different TTAs and an nVidia GeForce 9400 graphic card.

As the TTA template and the TCE compiler lacked (at the time of this writing) support for vector load/store units, and while nVidia cores can coalesce different buffer accesses into a single wide one, different TTA configurations with different number of 32-bit scalar LSUs were benchmarked. The 2-LSU case is practical, but as the resulting scheduled code is still slightly memory-access limited, results were generated

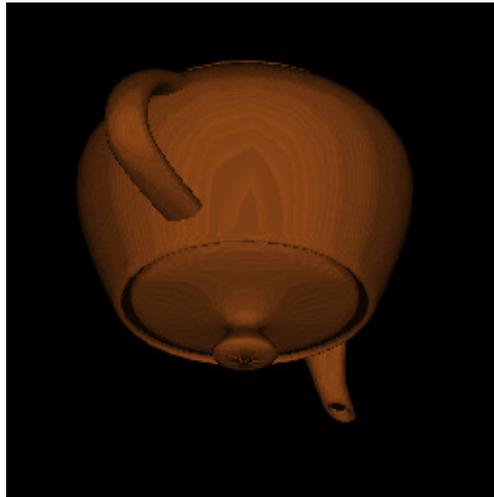


Fig. 27: Output image for the shader benchmark.

Table 2: Resources in the TTAs used in the shader benchmark.

resource	1 LSU	2 LSU	4 LSU
32 bit load-store units	1	2	4
floating point units		8	
extra floating point multipliers		8	
512 x 32 bit register files		8	
4 x 1 bit predicate register files		8	
full integer ALUs		8	
sqrt units		2	
transport buses		48	

also for an 4-LSU version, as the performance is expected to be similar to this once the vector load/stores are added to the architecture. Aside from this, the resources in the TTA used for the measurements closely match those present in a single core of the compared GeForce GPU. The resources are summarized in Table 2.

Fig. 28 presents cycle-count results for the execution of a single 32-wide work-group. The numbers for nVidia were obtained using the “Compute Visual Profiler” from their SDK, while the TTA cycle counts were produced using the instruction cycle-accurate simulator *ttasim* of TCE. The graph shows similar performance with the 2-LSU TTA and GeForce. TTA is slightly faster when 4 LSUs are present, alleviating the memory access bottleneck. Furthermore, when the execution causes the code to take diverging

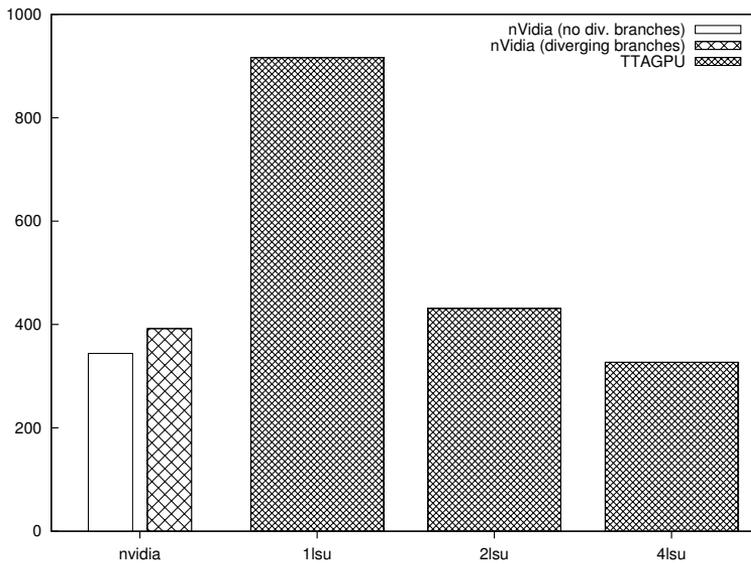


Fig. 28: Cycle counts for 32-wide workgroup execution.

branches, it causes a cycle count penalty on nVidia case of about 11%, while it has no effect whatsoever on the TTA architecture due to the aggressive if-conversion optimization and resource overcommitting.

In addition to reducing the effects of diverging branches to the throughput, the second case where TTA is assumed to improve over SIMT is when the number of parallel “threads“ does not efficiently cover the function units of the core. In order to measure the effect from this, the same benchmark was executed with different work-group sizes. Like expected, the cycle count does not change on the nVidia core when the number of work-items is smaller than the available SIMT lanes, but on TTA, the extra function units can be used to speed up the execution by scheduling parallel operations from the same thread in them. Fig. 29 shows how the cycle count decreases with the size of the work-group, with dashed lines marking the cycle count on the GeForce 9400. For work-group sizes of 16 and below, half of the function units on the nVidia core are idle, while the more free scheduling of the resources allow TTA to execute the kernel in much less cycles.

While overcommitting and scheduling freedom can be provided by both the TTA and the VLIW approaches, the latter suffers more from increased register file pressure when scaling the number of function units to match the instruction level parallelism available in the shader code. The exposed datapath of TTA allows reducing this

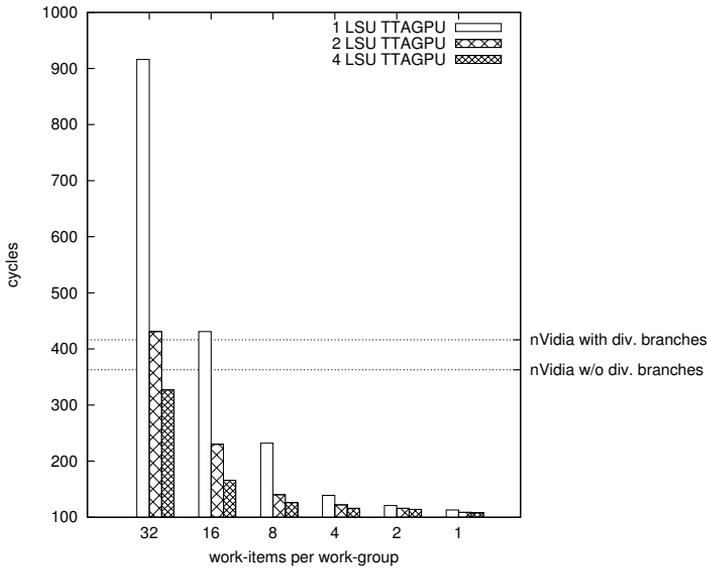


Fig. 29: Effect of the work-group size on the cycle count for running the kernel.

Table 3: Effects of software bypassing on register file pressure in the shader example.

	No bypassing	Bypassing	Reduction
Register reads	4897	3304	33%
Register writes	3483	2258	35%
total accesses	8380	5562	34%

pressure by means of software (register) bypassing, often avoiding the need to save intermediate results to general-purpose registers already at the compile time.

Results in Table 3 show about 34% decrease in register file accesses from software bypassing for the OpenCL kernel under consideration. This significant reduction in register file traffic highlights the benefit of TTA over a VLIW with similar datapath resources.

8.3 Single Core Customization for OpenCL Kernels

In order to validate and measure the performance and feasibility of the use of OpenCL as a starting point for customized processor design in practice, two example applications using the proposed design methodology were used. The first application, an

Advanced Encryption Standard (AES) encoder was used to verify the scalability of the static OpenCL kernel parallelization at the instruction level, and the ease of use of custom hardware operations. The second application, the radix-2 *Fast Fourier Transform (FFT)* was implemented to test customization purely using standard architecture components and to verify the automated interconnection network optimization tool.

8.3.1 AES OpenCL Implementation

AES algorithm uses a data block of 128 bits and a key size of 128, 192 or 256 bits. For the implementation a 128-bit key size was chosen. The operations involved in the algorithm are substitutions, rotations and permutations, using the 128 bits of data as a 4x4 array of bytes. Many software implementations of the algorithm manage the data to be processed as a buffer of *chars*, and all the operations are done in char size. For minimizing the number of memory accesses a variation of the Gladman's implementation [131] was used. It packs each four bytes of data in 32 bits unsigned values and uses other similar optimizations in other steps of the algorithm for reducing memory read and write operations.

The algorithm is divided into two steps: key expansion and encryption/decryption. The key expansion takes a 128-bit key and generates a 1408-bit expanded key. This step has to be done only once if the key does not change, and, therefore, this functionality was implemented in the host main program.

The encrypt and decrypt steps are done for each 128-bit block on the source data. These functions were implemented as OpenCL kernels. The encryption kernel receives several parameters from the host side: the global buffer to be encrypted, the expanded key, the buffer to store the results, and the substitution tables needed by the algorithm. Using these parameters and its own global identifier each work-item executing the kernel calculates the piece of input data it must process.

The host program is responsible for copying the data, key, and substitution tables to the device global memory. Once data is on device memory, the host launches as many work-items as there are 128-bit blocks in the input data buffer that must be encrypted or decrypted, and finally when all the work-items have finished it reads back the results (see Fig. 30).

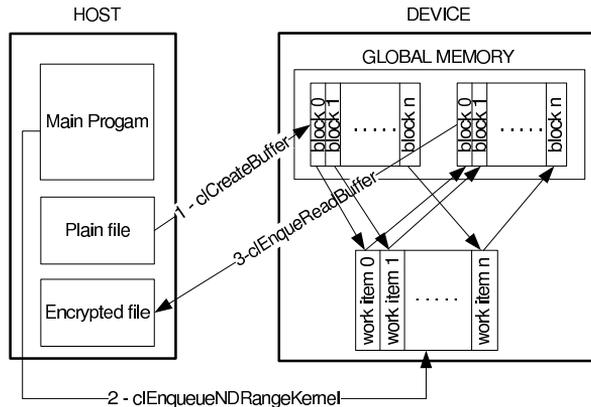


Fig. 30: The OpenCL AES encryption implementation.

# of parallel WIs	cycles	speedup
1	35,729	1.00
2	18,209	1.96
4	9,505	3.76

Table 4: Effect of the parallel Work-Item count to the cycle count.

8.3.2 Instruction-level Parallelism

The first experiment with AES was conducted to verify the ILP scalability of the OpenCL kernel compilation techniques. In order to measure this, an unrealistic architecture that provided enough resources (e.g., 250 FUs and 63 RFs) for the schedule length to be limited only by the data dependencies. The OpenCL kernel was compiled for this architecture with one, two and four parallel work-items. Due to starting to stress the limits of the compiler (scheduling time exploded and reached the maximum number of registers supported by LLVM) and the simulator of TCE, adding more units for additional work-items was stopped at the fourth parallel AES work-item.

The benchmark program encrypted 4KB of random data. The cycle counts with different number of parallel work-items are shown in Table 4. The numbers show that the compiler optimizations described in Section 6.3 are able to take advantage of the explicit parallelism in the OpenCL work-groups, and, given enough resources in the target machine, parallelizing the work-items perfectly producing close to linear speedup with relation to the number of parallel work-items.

resource	# of units	notes
Arithmetic-logic unit	3	1 cycle latency
Register file	3	16 registers per file
Load/Store unit	1	2 cycle load latency
32-bit multiplier unit	1	3 cycle latency

Table 5: Resources in the AES TTA processor.

8.3.3 Custom Operations

In the second experiment with the AES, the use of custom hardware operations to accelerate the application using the OpenCL C extension API (discussed in Section 6.3.5) was evaluated. For this experiment, a realistic base architecture was designed. The architecture, included datapath resources as shown in Table 5. The connectivity between the datapath units was clustered VLIW-like with FUs and RFs divided to three 1-FU-1-RF cluster nodes. The three nodes were interconnected with a fully connected transport bus.

In order to verify that the architecture is practical without, e.g., long critical paths ruining the performance due to low clock frequency, the architecture was synthesized on two FPGA chips: Xilinx Virtex 5 and Altera Stratix II. The maximum clock frequencies were 191 *MHz* for Virtex 5 and 149 *MHz* for Stratix II.

Two custom operations were designed and added to the base architecture:

- *MUL_GAL*, a multiplication of two integers in the Galois field $GF(2^8)$. The software implementation needs two reads from a logarithm table, a read from an antilogarithm table, an addition, and some control for performing this multiplication. In hardware, it can be done in a single clock cycle using two ROMs for the tables and an 8-bit adder.
- *SUBSHIFT* involves searching in a look-up table, substituting and mixing some elements of a 4×4 array. In software, it takes several clock cycles for reading the look-up table and mixing the elements of the array, but in hardware this operation can be done in a single clock cycle using a ROM and multiplexers.

The same encoding benchmark with the random 4KB input data set as in the previous experiment was compiled with two parallel work-items and simulated with the archi-

architecture	cycles	speedup factor	KB/s at 100 MHz
AESTTA	1,119,415	1.0	366
AESTTA+MUL_GAL	450,490	2.5	909
AESTTA+MUL_GAL+SS	286,778	3.9	1,428

Table 6: Speedups from using custom operations.

texture simulator to produce the cycle counts for the kernel execution. The speedups from using the two custom operations in comparison to the software-only AESTTA are shown in Table 6. The table includes also the computational encoding throughput scaled to 100 MHz clock frequency in order to exemplify the practical benefits of custom operations in this case.

The results show that adding custom operations using the extension mechanism works and can provide remarkable speedups in comparison to the architecture without custom operations, as expected. Adding both custom operations to the machine produces almost 4x speedup in comparison to the software-only version.

By inspecting the generated code it can be seen that the speedup is partially due to reduced general-purpose register pressure which results in less spills and less anti-dependencies that constrain the parallelism.

In this case, it would be possible to further accelerate the design with little effort, for example, by adding a fourth cluster to the base machine, increasing the number of general-purpose registers, or by adding more custom operations to the design. It can be seen from the previous experiment that given enough resources, the cycle count can be reduced considerably. However, the purpose of this experiment was not to design the fastest possible AES implementation, but to provide a proof-of-concept case for the proposed OpenCL-based processor customization methodology.

8.3.4 Single Core Customization Using Standard Components

In this experiment an architecture was customized by only using “standard components” with basic function unit operations. In addition, as an existing OpenCL FFT code was used, this experiment served also as an example of a fast processor customization cycle reusing existing OpenCL code as an input.

resource	#	notes
Integer arithmetic-logic unit	2	1 cycle latency
Register file	4	16 x 32 bit registers per file
Load/Store unit	1	2 cycle load latency
32-bit integer multiplier unit	2	2 cycle mul, add, sub

Table 7: Processor resources in the FFT experiment.

FFT size	parallel WIs	cycles	Without OpenCL AA info
256	1	38078	$\pm 0\%$
256	2	29390	+8%
1024	1	169595	$\pm 0\%$
1024	2	126155	+8%

Table 8: Execution time results for the FFT core. The last column shows the difference in cycle count when compiled without exploiting the additional OpenCL information in the alias analysis (AA).

For the FFT software implementation an existing OpenCL radix-2 FFT implementation was selected. The implementation by Eric Bainville is available freely in [132]. The original implementation used floating point arithmetics and was converted to use fixed-point number representation. The complex arithmetics of the algorithm is executed in software, i.e., 32-bit real and 32-bit imaginary parts were used. The arithmetic units used in the architecture were real-valued, complex-valued special multipliers were not considered.

The processor architecture was customized using the architecture enlarging method as described in Section 4.2.2. After several iterations, a machine consisting of resources as shown in Table 7 was produced. The interconnection network optimization tool was used to produce several architectures with varying connectivity. From this set three interesting pareto optimal architectures were chosen for FPGA synthesis. The best synthesis result on Xilinx Virtex 5 LX110T yielded an implementation with a maximum clock frequency of 130 *MHz*. The FPGA resource consumption was 3261 registers (4% of total available) and 6129 LUTs (8%).

The benchmark was executed for 256-point and 1024-point FFTs. The effect of an additional parallel work-item was also tested. The results from this experiment are shown in Table 8 which shows the numbers for the best performing interconnection network variation.

The used OpenCL implementation calls the kernel program iteratively from the host program loop multiple times for an FFT. Therefore, instead of benchmarking only the kernel execution time as in the AES benchmark, the whole call hierarchy from the host code to the kernel execution was included in the execution time to provide fair results. As the host and the kernel programs were running in the “standalone mode” (see Section 6.3.1), that is, on the same processor without a slow interconnect in between, such measurement is feasible.

The results are considered promising given that the total manual time spent on the architecture customization was less than a work day and no custom operations were used. In addition, it was confirmed that the additional memory access independence information from OpenCL improves the parallelization of multiple “iterations”. In this case of a resource constrained schedule the benefit was 8% with two parallel work-items.

8.4 Task Level Scalability

The design space of the multicore customizable processors is vast. At one end, there is a dual-core architecture with maximal per-core resources to satisfy the high level of instruction level parallelism (ILP) present in the programs. At the other end, there is an army of light-weight cores for a program with only thread level parallelism (TLP) or which benefits only from special instruction acceleration. Between these extremes there are countless variations where the reduced amount of single core resources gives room in the chip to higher number of cores, and vice versa, depending on the program and the restrictions placed by the implementation platform.

Additional region in the design space enabled by the proposed toolset is the single core processor design space for single threaded programs with ILP and/or which benefit from special instructions. The single core customization features were evaluated in the previous sections.

In order to ensure the interesting points in the design space can be reached, a benchmark with the capability to scale the parallelism at both the instruction and thread levels was created. The program implements the dot product of vectors using the LDDM model and the Dthreads API using 800 threads. The code is fully presented in the Fig. 16 in Section 5.5.1.

The single-core customization flow was used to design three different base architectures as follows:

swfp A simple core with only an integer ALU. Thus, performs the floating point (FP) operations in software. Other resources include a 16 x 32-bit general-purpose register file and 5 transport buses.

swfp2 Same as *swfp* except with double the number of integer datapath resources.

hwfp Same as *swfp2* but with an floating point unit (FPU) to speed up the FP computation. This exemplifies the operation set customization capability.

The core count of each of the single core architectures was increased from one core to 16 cores and the minimum processor cycles were obtained using the architectural simulator that simulates the LDDM hierarchy but assumes ideal latencies without access conflicts.

The results are illustrated in Fig. 31. The results show that the core count increase helps the software floating point architectures the most, as expected, because of their lower single thread performance. The additional ILP capabilities of *swfp2* is slightly visible with somewhat reduced cycle counts in each core multiplicities.

Due to the control complexity of the software FP emulation code, the compile-time exploitable ILP is rather limited in the input program. Adding the FPU for *hwfp* improved the performance drastically both because of the hardware acceleration itself and also due to increased static ILP from avoiding the need for the hard-to-parallelize FP emulation code. The maximum speedup from additional cores for *hwfp* started to decrease slightly sooner than with the software FP cores as faster thread execution resulted in running out of threads to execute quicker and also made the threading overheads more visible.

Naturally, the realistic speedups from multiple cores are highly dependent on the TLP available in the input application and its synchronization and communication needs. The used “embarrassingly parallel” dot product application was used merely to prove the scalability of the Dthread runtime library. The potential performance increased by adding more cores in all the cases, which shows that the threading runtime library implementation does not pose significant additional overheads that prevent the application speedups.

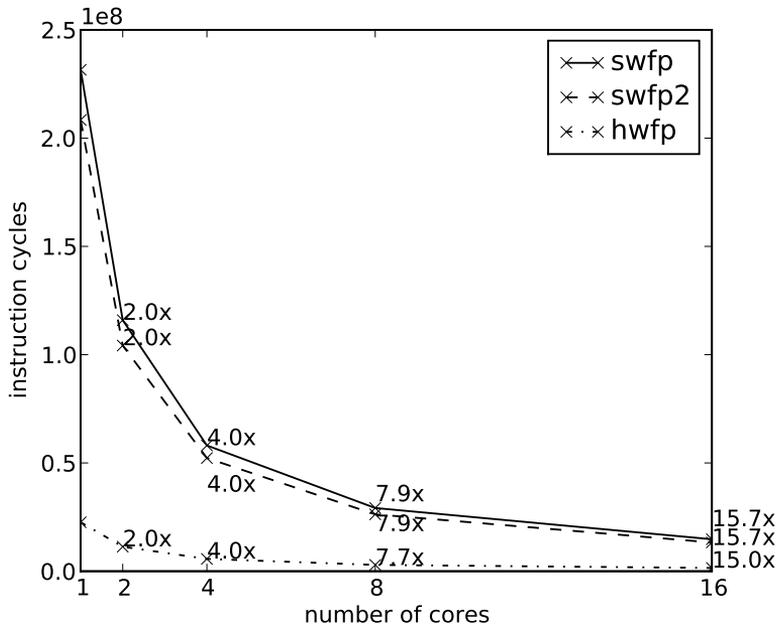


Fig. 31: The effect to the minimum processor cycles from increasing the core count. The potential speedup in comparison to the single core is written on each design point in the diagram.

8.5 Barrier Synchronization Overheads

The effect of using the barrier synchronization primitive implementation alternatives presented in Section 7.2 to the final performance of the application depends on various factors. Clearly, the synchronization operation per program operation ratio of the program is the primary factor. What is not so obvious is that the shared memory pressure of the program is another important factor. In case the program performs frequent accesses to the shared memory, it is hindered more by the avoidable traffic caused by the potential shared memory polling overheads of the synchronization primitives.

This Section presents an experiment that stress tests the proposed alternative barrier implementations in order to show their scalability differences.

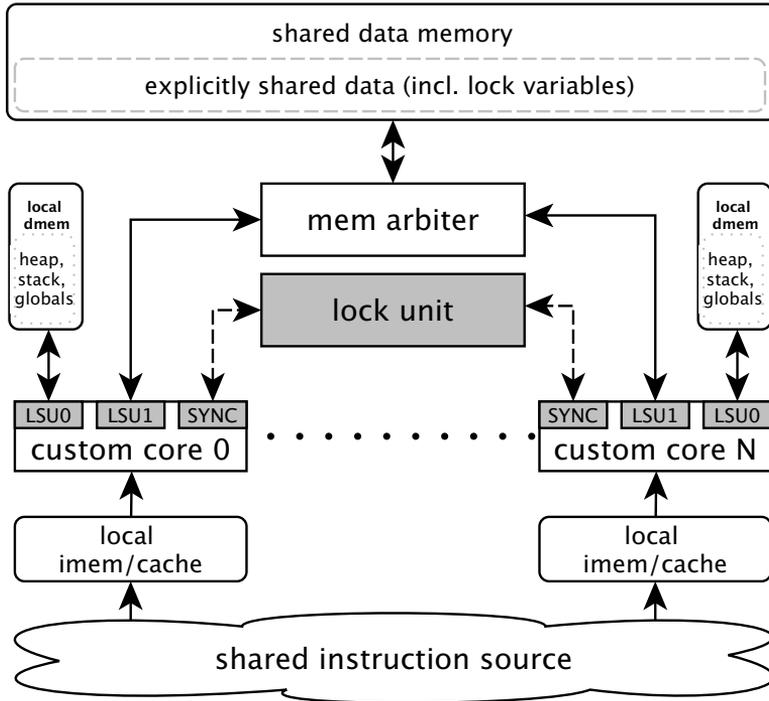


Fig. 32: The organization of the benchmarked multicore ASIP using the lock unit through a SYNC function unit.

8.5.1 The Benchmark Processor

For the evaluation of the proposed lock unit a 48-core MCASIP was designed using the proposed design flow. It is expected that the results are reproducible with any shared memory multicore architecture without dynamic caches.

All the cores in the designed MCASIP use the proposed datapath integrated lock unit for synchronization operations accessed using a SYNC FU as depicted in Fig. 32. The cores have a relatively simple architecture with an integer ALU, an integer multiplier and a register file with 16 32-bit registers. The memory configuration of the MCASIP was as described in Section 5.3. That is, each core had its own fast private data memory large enough for local stack and data in addition to an arbitrated shared memory. The shared memory access was implemented with a simple access queue which causes a dynamic load latency from 4 to 96 cycles, depending on how many cores are competing for memory access at the time instant.

```

for (int round = 0; round < ROUNDS; ++round) {
    if (core_id == 0) {
        for (int delay = 0;
            delay < smAccesses;
            ++delay, ++shared_value) {}
    }
    barrier(&b);
}

```

Fig. 33: The microbenchmark that “stress tests” the shared memory overheads of the barrier alternatives.

The design was synthesized to an Altera Stratix II FPGA with a clock rate of 50 MHz. The additional area overhead of the synchronization hardware was very low: a lock unit with four lock registers consumed about 1% (925 ALM) of the total logic utilization of the multicore (72 kALM).

8.5.2 Benchmark Program

The scalability of the proposed barrier alternatives was measured by implementing a synthetic microbenchmark that performs a tight barrier synchronized loop and executing it in the MCASIP in the FPGA. The benchmark loop is shown in Fig. 33.

The benchmark represents a “stress test” where the computation to synchronization ratio is extremely low and where the progress is heavily limited by simultaneous shared memory accesses from different cores. Artificial shared memory traffic was generated by adding an update to a counter residing in shared memory to the first core. This forces the other cores to wait that time in the barrier spin wait loop. While being a synthetic example, the case represents the worst case of *thread imbalance* where one shared memory heavy thread takes more time to complete than the others and the completed threads cause it to slow down due to the shared memory traffic from barrier spin waiting.

In this benchmark, the number of threads equals the number of cores, thus the context switch overheads were ruled out. If there were more threads per core, the barrier call would induce a thread context switch to allow the waiting threads to reach the barrier. This would reduce the total “unsuccessful” spin waiting for the time of each context switch, possibly reducing shared memory noise. However, the context switch time and its effect to the performance depends at least on a) the size of the thread context

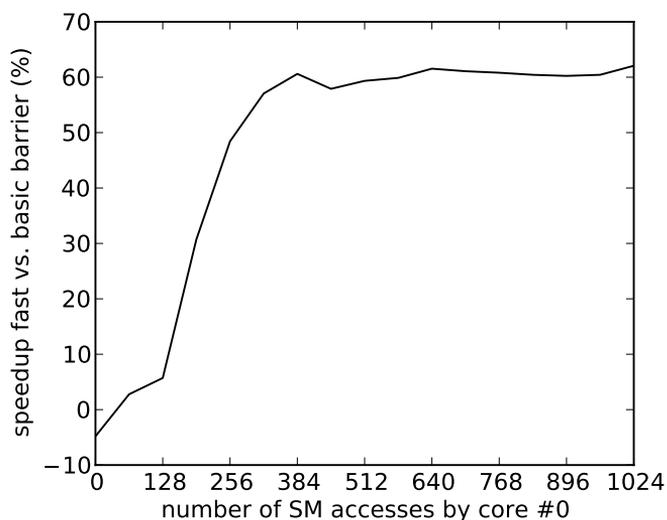


Fig. 34: The speedups obtained by using the *fast* barrier in comparison to the *basic* barrier version. The run times were measured as wall clock time with the FPGA implementation of the 48-core MCASIP, thus included all the dynamic latencies from memory access conflicts. The speedup was measured as a function of the number of memory accesses made by the executing thread to emphasize the effects of the barrier polling to the shared memory access conflicts.

to store/restore b) the location of the context data (shared memory or a local storage) c) the thread scheduler overhead d) if a pre-emptive scheduler is used, the length of the thread time slice, etc. Therefore, it was decided to simplify this benchmark to emphasize the effect of the synchronization overheads alone.

The results illustrated in Fig. 34 show that the speedup from using the fast barrier that eliminates shared memory polling is drastic. The speedup increases up to about 3000 shared memory accesses after which it stabilizes to around 64%. For less than 320 accesses the *basic* barrier is faster as the additional barrier software complexity of the *fast* barrier dominates the shared memory access reduction benefits.

9. CONCLUSIONS

In this Thesis a design methodology and the supporting design flow for Multicore Application-Specific Instruction Set Processors was proposed. The methodology enables rapid design cycles for efficient parallel processor-based application implementations starting from parallel programs. The customized processors can exploit instruction-level parallelism by means of an exposed datapath MIMD single core template, and task level parallelism with a shared memory multicore template that includes local scratchpad memories for memory access optimization.

For supporting single core customization in the methodology the Transport Triggered Architecture is used as the processor template. Although most of the TTA's advantages, such as the simplified control logic and the MIMD programming model, can be realized also with the traditional operation programmed VLIW architectures, TTA improves the instruction level parallelism scalability with its programmer-visible interconnection network. The transport programming model enables additional software optimizations, most importantly the software bypassing which reduces the register and register file pressure, and increases data locality by enabling programmer controlled data transfers between function units.

9.1 *Main Results*

The benefits of the additional scheduling freedom enabled by the MIMD programming model of the TTA/VLIW was compared to the common more restricted SIMD model in case of SPMD workloads. The conclusion of the comparison is that the MIMD model allows better utilization of the hardware resources in case of SPMD thread counts that do not match the width of the machine, but with the cost of extended instruction word width. In addition, in contrast to a strict SIMT with scalar lanes, MIMD enables scheduling multiple operations from the same thread in par-

allel. Predicated execution, especially when resource overcommitting is supported can be used to increase the resource utilization further, improving the performance of SPMD programs with diverging threads.

The proposed design methodology is generic enough to support a wide range of parallel languages. The OpenCL programming standard was used as an example input to the design flow. The compiler techniques needed in order to efficiently map the OpenCL kernels to parallel resources of the designed architecture were presented. The key problem in that work is with kernels that contain barriers which lead to challenges when parallelizing multiple work-items statically. Another challenge is to communicate the work-item parallelism to the instruction scheduler which was solved with a register partitioner that aims to assign each parallel work-item registers from separate register files.

For accelerating the synchronization primitives which are often highly utilized in parallel applications a datapath integrated lock unit hardware was proposed. The results with a 48-core MCASIP show the benefits of its flexibility. With a shared memory heavy micro-benchmark the proposed barrier alternative that uses two lock registers reduces shared memory contention up to 64% in comparison to a simpler version that consumes only one lock register and generates more shared memory traffic.

Overall, the preliminary simulation and FPGA implementation results show high potential for the proposed design methodology. The FPGA implementability of the designed processors was estimated in [11] with the conclusion that the design of soft multiprocessors is feasible using the design flow, with the limiting factor usually being the size of the local memory. MCASIPs with more than hundred of cores can be fit to the current high-end FPGA chips.

The Author considers the lack of support for fast exploration of different (parallel) memory systems the current main limitation in the proposed design methodology. In the methodology, manual work is required to get the final performance numbers that include also memory access conflict overheads. However, the once manually designed memory implementations for FPGA prototyping or the SystemC memory simulation models can be reused in later designs.

9.2 Future Work

The proposed design flow presents a fully working solution for parallel program driven customized multicore processor co-design. However, the Author does not claim the current proposal perfects all the techniques involved, but that it provides a functioning platform for further research.

An assumption in the proposed design methodology is that the designer driven co-design of the processors leads to results that are more reusable in the future designs, in contrast to the possibly very application-specialized results a fully-automated design space exploration would produce. However, especially for the high-level synthesis for producing FPGA-based implementations, most of the process of designing a new multicore processor with customized instructions could and should be automated. For this use case, a design database based automated design space exploration that also includes automated generation of instruction-set extensions has been planned as future work.

It is a generally acknowledged fact that the memory bottleneck limits the performance of processor-based designs [133]. This is emphasized with multicore designs that use a shared memory to share data and communicate with each other. The efficient conflict-free use of multibank parallel memories is important in unleashing the performance potential of parallel programs. [134] Work is ongoing in our group to develop methodologies for the design space exploration of parallel memory systems.

Multithreading relates to the memory system as it can be used to hide long memory operation latencies executed by a thread with computational operations from other threads. The additional challenge present in the proposed work is the use of the TTA, an exposed datapath architecture, as the single core template. Due to the abundance of programmer-visible details, the processor context that needs to be saved in a full thread context switch is costly to implement. In our previous work [106], compiler-assisted context saving techniques to find a compromise between co-operative context switches and hardware implemented pre-emptive threading have been researched. This work is to be continued with the focus on implementing coarse-grained threading focused on hiding long memory access latencies.

BIBLIOGRAPHY

- [1] Texas Instruments Incorporated. OMAP™ mobile processors, 2012.
- [2] Qualcomm Incorporated. Snapdragon™- the all-in-one mobile processor, 2012.
- [3] NVIDIA Corporation. Nvidia® tegra®, 2012.
- [4] Texas Instruments. *TMS320C64x/C64x+ DSP CPU and Instruction Set Reference Guide*, December 2007. <http://www-s.ti.com/sc/techlit/spru732>.
- [5] G. Martin and G. Smith. High-level synthesis: Past, present, and future. *Design & Test of Computers, IEEE*, 26(4):18–25, July-Aug. 2009.
- [6] Altera Corporation. Implementing FPGA Design with the OpenCL Standard. Technical white paper, Nov 2011.
- [7] M. Owaida, N. Bellas, C.D. Antonopoulos, K. Daloukas, and C. Antoniadis. Massively parallel programming models used as hardware description languages: the OpenCL case. In *Proceedings of the International Conference on Computer-Aided Design, ICCAD '11*, pages 326–333, 2011.
- [8] H. Corporaal. *Microprocessor Architectures: From VLIW to TTA*. John Wiley & Sons, Chichester, UK, 1997.
- [9] Khronos Group. *OpenCL Specification v1.2r15*, Nov. 2011.
- [10] C.S. de La Lama, **P. Jääskeläinen**, H. Kultala, and J. Takala. Programmable and scalable architecture for graphics processing units. *Transactions on HiPEAC*, 5, 2010. Available online.
- [11] **P. Jääskeläinen**, E. Salminen, C.S. de La Lama, J. Takala, and J. Martinez. TCEMC: A co-design flow for application-specific multicores. In *Proceedings*

- of International Conference on Embedded Computer Systems (SAMOS)*, pages 85–92, July 2011.
- [12] **P. Jääskeläinen**, C.S. de La Lama, P. Huerta, and J. Takala. OpenCL-based design methodology for application-specific processors. *Transactions on HiPEAC*, 5, 2011. Available online.
- [13] **P. Jääskeläinen**, E. Salminen, O. Esko, and J. Takala. Customizable datapath integrated lock unit. In *Proceedings of International Symposium on System-on-Chip (SoC)*, pages 29–33, Oct–Nov 2011.
- [14] **P. Jääskeläinen**. Instruction Set Simulator for Transport Triggered Architectures. Master’s thesis, Tampere University of Technology, Finland, Sep 2005.
- [15] **P. Jääskeläinen**, C.S. de La Lama, P. Huerta, and J. Takala. OpenCL-based design methodology for application-specific processors. In *Proceedings of International Conference on Embedded Computer Systems (SAMOS)*, pages 223–230, July 2010.
- [16] H. Sutter and J. Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, September 2005.
- [17] W. Blume, R. Eigenmann, J. Hoeflinger, D. Padua, P. Petersen, L. Rauchw-erger, and P. Tu. Automatic detection of parallelism: A grand challenge for high performance computing. *IEEE Parallel & Distributed Technology: Systems & Applications*, 2(3):37–47, 1994.
- [18] P. Stotts. A comparative survey of concurrent programming languages. *SIG-PLAN Not.*, 17(10):50–61, October 1982.
- [19] IEEE. *POSIX, IEEE Std 1003.1, 2004 Edition*.
- [20] OpenMP Architecture Review Board. *OpenMP Application Program Interface v3.0*, May. 2008.
- [21] J. Barth. A practical interprocedural data flow analysis algorithm. *Commun. ACM*, 21(9):724–736, September 1978.
- [22] ISO. *ISO/IEC 9899:1999 - Programming languages - C*, Dec. 1999.

-
- [23] TIOBE Software BV. Tiobe programming community index for april 2012. world wide web, April 2012.
- [24] T. LeBlanc and E. Markatos. Shared memory vs. message passing in shared-memory multiprocessors. In *Proceedings of the Fourth IEEE Symposium on Parallel and Distributed Processing*, pages 254–263, December 1992.
- [25] S. Berger and A. Stamatakis. Assessment of barrier implementations for fine-grain parallel regions on current multi-core architectures. In *IEEE International Conference on Cluster Computing Workshops and Posters (CLUSTER WORKSHOPS)*, pages 1–8, September 2010.
- [26] Khronos Group. *OpenCL Specification v1.0r48*, Oct. 2009.
- [27] T. Halfhill. Parallel Processing with CUDA. *Microprocessor Report*, Jan. 2008.
- [28] S. Rul, H. Vandierendonck, J. D’Haene, and K. De Bosschere. An experimental study on performance portability of OpenCL kernels. In *Symposium on Application Accelerators in High Performance Computing (SAAHPC ’10)*, July 2010. Available online.
- [29] K. Rupp, J. Weinbub, and F. Rudolf. Automatic performance optimization in ViennaCL for GPUs. In *Proceedings of the 9th Workshop on Parallel/High-Performance Object-Oriented Scientific Computing, POOSC ’10*, pages 6:1–6:6, 2010.
- [30] K. Spafford, J. Meredith, and J. Vetter. Maestro: data orchestration and tuning for OpenCL devices. In *Proceedings of the 16th international Euro-Par conference on Parallel processing: Part II, Euro-Par’10*, pages 275–286, 2010.
- [31] 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 Computing*, October 2011. Available online.
- [32] P. Gibbons and S. Muchnick. Efficient instruction scheduling for a pipelined architecture. In *Proceedings of the SIGPLAN 1986 Symposium on Compiler construction, SIGPLAN ’86*, pages 11–16, 1986.

- [33] B. Rau and J. Fisher. Instruction-level parallelism. In *Encyclopedia of Computer Science*, pages 883–887. John Wiley and Sons Ltd., Chichester, UK.
- [34] B. Rau and J. Fisher. Instruction-level parallel processing: History, overview, and perspective. In *Instruction-Level Parallelism*, volume 235 of *The Kluwer International Series in Engineering and Computer Science*, pages 9–50. Springer US, 1993.
- [35] H. Tanaka. Toward more advanced usage of instruction level parallelism by a very large data path processor architecture. In *Proceedings of Third International Symposium on Parallel Architectures, Algorithms, and Networks (I-SPAN '97)*, pages 437–443, December 1997.
- [36] H. Lee, Y. Wu, and G. Tyson. Quantifying instruction-level parallelism limits on an EPIC architecture. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 21–27, 2000.
- [37] J. Fisher. Customized instruction-sets for embedded processors. In *Proceedings of the 36th annual ACM/IEEE Design Automation Conference, DAC '99*, pages 253–257, 1999.
- [38] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach, 3rd edition*. Morgan Kaufmann Publishers Inc., 2003.
- [39] W. Hwu, T. Conte, and P. Chang. Comparing software and hardware schemes for reducing the cost of branches. In *Proceedings of the 16th annual international symposium on Computer architecture, ISCA '89*, pages 224–233, 1989.
- [40] J. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Proceedings of ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 177–189, January 1983.
- [41] J. Fisher. Trace scheduling: A technique for global microcode compaction. *Instruction-level Parallel Processors*, pages 186–198, 1995.
- [42] Intel Corporation. *Intel[®] Architecture Instruction Set Extensions Programming Reference*, February 2012.

-
- [43] ARM Ltd. The ARM[®] NEON[™] general-purpose SIMD engine, April 2012. Web page.
- [44] M. Gschwind, H. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. Synergistic processing in Cell's multicore architecture. *IEEE Micro*, 26:10–24, March 2006.
- [45] B. Gaster, L. Howes, D. Kaeli, P. Mistry, and D. Schaa. *Heterogeneous Computing with OpenCL*, chapter 6, pages 123–150. Morgan Kaufmann, 2012.
- [46] NVIDIA. CUDA programming guide v2.1. Tech. Report, 2008.
- [47] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, March–April 2008.
- [48] W.-D. Weber and A. Gupta. Exploring the benefits of multiple hardware contexts in a multiprocessor architecture: preliminary results. *ACM SIGARCH Computer Architecture News*, 17(3):273–280, April 1989.
- [49] J. Laudon, A. Gupta, and M. Horowitz. Interleaving: a multithreading technique targeting multiprocessors and workstations. *ACM SIGOPS Operating Systems Review*, 28(5):308–318, November 1994.
- [50] D. Tullsen, S. Eggers, and H. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of 22nd Annual International Symposium on Computer architecture (ISCA '95)*, pages 392–403, 1995.
- [51] K. Olukotun, B. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. *SIGPLAN Notices*, 31(9):2–11, 1996.
- [52] D. Patterson and J. Hennessy. *Computer Organization and Design: The Hardware/Software Interface, 3rd edition*. Morgan Kaufmann, San Francisco, US, 2005.
- [53] R. Kumar, D. Tullsen, N. Jouppi, and P. Ranganathan. Heterogeneous chip multiprocessors. *Computer*, 38(11):32–38, November 2005.
- [54] W. Wolf, A. Jerraya, and G. Martin. Multiprocessor system-on-chip (MPSoC) technology. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(10):1701–1713, October 2008.

- [55] G. De Michell and R. Gupta. Hardware/software co-design. *Proceedings of the IEEE*, 85(3):349–365, March 1997.
- [56] M. Arnold and H. Corporaal. Designing domain-specific processors. In *Proceedings of the Ninth International Symposium on Hardware/software Code-sign*, CODES '01, pages 61–66, 2001.
- [57] A. Alomary, T. Nakata, Y. Honma, J. Sato, N. Hikichi, and M. Imai. PEAS-I: A hardware/software co-design system for ASIPs. In *Proceedings of European Design Automation Conference, 1993 (EURODAC), with EURO-VHDL '93.*, pages 2–7, sep 1993.
- [58] L. Pozzi and P. Paulin. A future of customizable processors: Are we there yet? In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '07, pages 1224–1225, 2007.
- [59] I. Paolo and R. Leupers. *Customizable Embedded Processors: Design Technologies and Applications*. Elsevier Inc., 2007.
- [60] M. Thuresson, M. Sjalander, M. Bjork, L. Svensson, P. Larsson-Edefors, and P. Stenstrom. FlexCore: Utilizing exposed datapath control for efficient computing. In *Proceedings of International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, pages 18–25, 2007.
- [61] Y. He, D. She, B. Mesman, and H. Corporaal. MOVE-Pro: A low power and high code density TTA architecture. In *Proceedings of International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, pages 294–301, July 2011.
- [62] W. Dally, J. Balfour, D. Black-Shaffer, J. Chen, R. Harting, V. Parikh, J. Park, and D. Sheffield. Efficient embedded computing. *Computer*, 41:27–32, 2008.
- [63] R. Colwell, R. Nix, J. O'Donnell, D. Papworth, and P. Rodman. A VLIW architecture for a trace scheduling compiler. In *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 180–192, 1987.

-
- [64] M. Purnaprajna and P. Ienne. Making wide-issue VLIW processors viable on FPGAs. *ACM Transactions on Architecture and Code Optimization*, 8(4):33:1–33:16, 2012.
- [65] H. Corporaal. TTAs: missing the ILP complexity wall. *Journal of Systems Architecture*, 45(12-13):949–973, 1999.
- [66] J. Hoogerbrugge and H. Corporaal. Register file port requirements of Transport Triggered Architectures. In *Proceedings of Annual International Symposium on Microarchitecture*, pages 191–195, November-December 1994.
- [67] M. Smelyanskiy, S. Mahlke, E. Davidson, and H.-H. Lee. Predicate-aware scheduling: A technique for reducing resource constraints. In *Proceedings of International Symposium on Code Generation Optimization*, pages 169–178, March 2003.
- [68] M. Wilkes. The growth of interest in microprogramming: A literature survey. *ACM Computing Surveys*, 1(3):139–145, September 1969.
- [69] R. Rosin. Contemporary concepts of microprogramming and emulation. *ACM Computing Surveys*, 1(4):197–212, December 1969.
- [70] R. Smith. A historical overview of computer architecture. *IEEE Annals of the History of Computing*, 10(4):277–303, October 1988.
- [71] R. Touzeau. A Fortran compiler for the FPS-164 scientific computer. In *Proceedings of the 1984 SIGPLAN symposium on Compiler construction*, SIGPLAN '84, pages 48–57, 1984.
- [72] G. Lipovski. The architecture of a simple, effective control processor. In *2nd Euromicro Symposium on Microprocessing and Microprogramming*, pages 7–19, October 1976.
- [73] D. Tabak and G. Lipovski. MOVE architecture in digital controllers. *IEEE Journal of Solid-State Circuits*, 15(1):116 – 126, February 1980.
- [74] H. Corporaal. Transport triggered architectures examined for general purpose applications. In *Proceedings of Sixth Workshop Computer Systems, Delft*, pages 55–71, 1993.

- [75] J. Janssen and H. Corporaal. Partitioned register file for TTAs. In *Proceedings of 28th Annual Workshop on Microprogramming (MICRO-28)*, pages 303–312, 1996.
- [76] H. Corporaal and H. Mulder. MOVE: A framework for high-performance processor design. In *Proceedings of ACM/IEEE Conference on Supercomputing*, pages 692–701, 1991.
- [77] **P. Jääskeläinen**, V. Guzman, A. Cilio, and J. Takala. Codesign toolset for application-specific instruction-set processors. In *Proceedings of SPIE Multimedia on Mobile Devices*, pages 65070X–1 – 65070X–11, January 2007.
- [78] K. Hughes, P. Jeppson, M. Larsson-Edefors, M. Sheeran, P. Stenstrom, and L. Svensson. "FlexSoC: Combining flexibility and efficiency in SoC designs". In *Proceedings of the 21st IEEE NorChip Conference*, 2003. Available online.
- [79] A. Bardizbanyan, M. Sjalander, and P. Larsson-Edefors. Reconfigurable instruction decoding for a wide-control-word processor. In *IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)*, pages 322 –325, May 2011.
- [80] A. Klaiber (Transmeta Corporation). The Technology Behind Crusoe Processors, January 2000. Web page.
- [81] T. Schilling, M. Sjalander, and P. Larsson-Edefors. Scheduling for an embedded architecture with a flexible datapath. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*., pages 151 –156, May 2009.
- [82] M. Reshadi, B. Gorjiara, and D. Gajski. Utilizing horizontal and vertical parallelism with a no-instruction-set compiler for custom datapaths. In *Proceedings of the 2005 International Conference on Computer Design, ICCD '05*, pages 69–76, 2005.
- [83] D. Black-Schaffer, J. Balfour, W. Dally, V. Parikh, and JongSoo Park. Hierarchical instruction register organization. *Computer Architecture Letters*, 7(2):41 –44, July-December 2008.
- [84] J. Balfour, R. Halting, and W. Dally. Operand registers and explicit operand forwarding. *Computer Architecture Letters*, 8(2):60 –63, February 2009.

-
- [85] L. Lozano and G. Gao. Exploiting short-lived variables in superscalar processors. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 292–302, November–December 1995.
- [86] G.Cichon, P. Robelly, H. Seidel, E. Matúš, M. Bronzel, and G. Fettweis. Synchronous transfer architecture (STA). In *Computer Systems: Architectures, Modeling, and Simulation*, volume 3133 of *Lecture Notes in Computer Science*, pages 193–207. Springer Berlin / Heidelberg, 2004.
- [87] S. Fuller and L. Millett. Computing performance: Game over or next level? *Computer*, 44(1):31–38, 2011.
- [88] L. Kish. End of Moore’s law: Thermal (noise) death of integration in micro and nano electronics. *Physics Letters A*, 305(34):144–149, 2002.
- [89] G. Goossens, D. Lanneer, W. Geurts, and J. Van Praet. Design of ASIPs in multi-processor SoCs using the Chess/Checkers retargetable tool suite. In *Proceedings of the International Symposium on System-on-Chip*, pages 1–4, November 2006.
- [90] S. Shee and S. Parameswaran. Design methodology for pipelined heterogeneous multiprocessor system. In *Proceedings of the 44th Annual Design Automation Conference, DAC ’07*, pages 811–816, 2007.
- [91] R. Kumar, D. Tullsen, and N. Jouppi. Core architecture optimization for heterogeneous chip multiprocessors. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques, PACT ’06*, pages 23–32, 2006.
- [92] B. Candaele, S. Aguirre, M. Sarlotte, I. Anagnostopoulos, S. Xydis, A. Bartzas, D. Bekiaris, D. Soudris, Zhonghai Lu, Xiaowen Chen, J. Chabloz, A. Hemani, A. Jantsch, G. Vanmeerbeeck, J. Kreku, K. Tiensyrja, F. Ieromonimon, D. Kritharidis, A. Wiefrink, B. Vanthournout, and P. Martin. Mapping Optimisation for Scalable Multi-core ARchiTecture: The MOSART Approach. In *VLSI (ISVLSI), 2010 IEEE Computer Society Annual Symposium on*, pages 518–523, 2010.
- [93] J. Hoogerbrugge and H. Corporaal. Automatic synthesis of transport triggered

- processors. In *Proceedings of the First Annual Conference of ASCI*, pages 1–11, 1995.
- [94] T. Hoang, U. Jalmbrant, E. der Hagopian, K. Subramaniyan, M. Sjalander, and P. Larsson-Edefors. Design space exploration for an embedded processor with flexible datapath interconnect. In *Proceedings of the 21st IEEE International Conference on Application-specific Systems Architectures and Processors (ASAP)*, pages 55–62, July 2010.
- [95] O. Esko, **P. Jääskeläinen**, P. Huerta, C.S. de La Lama, J. Takala, and J. Martinez. Customized exposed datapath soft-core design flow with compiler support. *Proceedings of International Conference on Field Programmable Logic and Applications*, pages 217–222, 2010.
- [96] O. Esko. ASIP Integration and Verification Flow. Master’s thesis, Department of Information Technology, Tampere University of Technology, Tampere, Finland, June 2011.
- [97] J. Heikkinen, A. Cilio, J. Takala, and H. Corporaal. Dictionary-based program compression on transport triggered architectures. In *Proceedings of IEEE International Symposium on Circuits and Systems (ISCA)*, pages 1122–1124, May 2005.
- [98] NVIDIA[®]. NVIDIA’s Next Generation CUDA[™] Compute Architecture: Fermi[™] (whitepaper). Technical white paper (PDF available on web), 2009.
- [99] NVIDIA Corporation. NVIDIA’s Next Generation CUDA[™] Compute Architecture: Kepler[™] GK110 (whitepaper). Technical white paper (PDF available on web), 2012.
- [100] A. Cilio, H. Schot, and J. Janssen. Architecture Definition File: Processor Architecture Definition File Format for a New TTA Design Framework. Project Document, Tampere Univ. of Tech., Tampere, Finland, 2003-2006. Available on web.
- [101] Dake Liu. *Embedded DSP Processor Design*. Morkan Kaufmann Publishers, 1st edition, 2008.
- [102] S. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, December 1996.

-
- [103] P. Panda, N. Dutt, and A. Nicolau. On-chip vs. off-chip memory: The data partitioning problem in embedded processor-based systems. *ACM Transactions on Design Automation of Electronic Systems*, 5:682–704, July 2000.
- [104] P. Stenstrom. A survey of cache coherence schemes for multiprocessors. *Computer*, 23(6):12–24, June 1990.
- [105] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation Optimization*, pages 75–87, March 2004.
- [106] **P. Jääskeläinen**, P. Kellomäki, J. Takala, H. Kultala, and M. Lepistö. Reducing context switch overhead with compiler-assisted threading. In *Proceedings of IEEE/IFIP International Conference on Embedded and Ubiquitous Computing*, pages 461–466, December 2008.
- [107] R. Blumofe and C. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46:720–748, September 1999.
- [108] J. Stratton, S. . Stone, and W.-M. Hwu. MCUDA: An efficient implementation of CUDA kernels for multi-core CPUs. In *Languages and Compilers for Parallel Computing*, volume 5335 of *LNCS*, pages 16–30. 2008.
- [109] A. Papakonstantinou, K. Gururaj, J. Stratton, J. Cong, D. Chen, and W.-M. Hwu. FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs. In *Proceedings of IEEE Symp. Application Specific Processors*, pages 35–42, July 2009.
- [110] F. Pratas and L. Sousa. Applying the stream-based computing model to design hardware accelerators: A case study. In *Proceedings of Embedded Computer Systems: Architectures, MOdeling, and Simulation (SAMOS)*, pages 237–246, 2009.
- [111] C. Fletcher, I. Lebedev, N. Asadi, D. Burke, and J. Wawrzynek. Bridging the GPGPU-FPGA efficiency gap. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA ’11, pages 119–122, 2011.

- [112] M. Lin, I. Lebedev, and J. Wawrzynek. OpenRCL: Low-power high-performance computing with reconfigurable devices. In *Proceedings of the 2010 International Conference on Field Programmable Logic and Applications*, FPL '10, pages 458–463, 2010.
- [113] M. Owaida, N. Bellas, K. Daloukas, and C. Antonopoulos. Synthesis of platform architectures from OpenCL programs. In *IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 186–193, May 2011.
- [114] M. Owaida, N. Bellas, C. Antonopoulos, K. Daloukas, and C. Antoniadis. Massively parallel programming models used as hardware description languages: The OpenCL case. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 326–333, November 2011.
- [115] J. Mellor-Crummey and M. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9:21–65, February 1991.
- [116] D. Nikolopoulos and T. Papatheodorou. The architectural and operating system implications on the performance of synchronization on ccNUMA multiprocessors. *International Journal of Parallel Programming*, 29:249–282, June 2001.
- [117] T.E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.
- [118] A. Agarwal and M. Cherian. Adaptive backoff synchronization techniques. In *Proceedings of the 16th annual international symposium on Computer architecture*, ISCA '89, pages 396–406, 1989.
- [119] C. Beckmann and C. Polychronopoulos. Fast barrier synchronization hardware. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, pages 180–189, 1990.
- [120] Z. Fang, L. Zhang, J. Carter, L. Cheng, and M. Parker. Fast synchronization on shared-memory multiprocessors: An architectural approach. *Journal of Parallel and Distributed Computing*, 65:1158–1170, October 2005.

-
- [121] M. Monchiero, G. Palermo, C. Silvano, and O. Villa. Efficient synchronization for embedded on-chip multiprocessors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14:1049–1062, October 2006.
- [122] M. Moudgill, V. Kalashnikov, M. Senthilvelan, U. Srikantiah, T. Li, P. Balzola, and J. Glossner. Synchronization on heterogeneous multiprocessor systems. In *Proceedings of the 9th international conference on Systems, Architectures, MOdeling and Simulation (SAMOS)*, SAMOS'09, pages 133–139, 2009.
- [123] C. Yu and P. Petrov. Low-cost and energy-efficient distributed synchronization for embedded multiprocessors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 18(8):1257–1261, August 2010.
- [124] Bi. Akgul and V. Mooney III. The system-on-a-chip lock cache. *Design Automation for Embedded Systems*, 7:139–174, 2002.
- [125] T. Mattson, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, and S. Dighe. The 48-core SCC processor: the programmer's view. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC '10*, pages 1–11, 2010.
- [126] E. Vallejo, R. Beivide, A. Cristal, T. Harris, F. Vallejo, O. Unsal, and M. Valero. Architectural support for fair reader-writer locking. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO '10*, pages 275–286, 2010.
- [127] TCE: TTA-based codesign environment. Web page: <http://tce.cs.tut.fi>.
- [128] L. Laasonen. Program Image and Processor Generator for Transport Triggered Architectures. Master's thesis, Tampere Univ. Tech., Finland, 2007.
- [129] pocl: Portable OpenCL. Web page: <http://pocl.sourceforge.net>.
- [130] R. Rost. *OpenGL Shading Language*. Addison-Wesley, 3rd edition, 2010.
- [131] G. Bertoni, L. Breveglieri, P. Fragneto, M. Macchetti, and S. Marchesin. Efficient software implementation of AES on 32-bit platforms. In *Cryptographic Hardware and Embedded Systems - CHES 2002*, volume 2523 of LNCS, pages 159–171. Springer-Verlag, 2003.

- [132] E. Bainville. OpenCL Fast Fourier Transform: A simple radix-2 kernel, October 2010. Web page: http://www.bealto.com/gpu-fft_opencl-1.html.
- [133] C. Ding and K. Kennedy. The memory bandwidth bottleneck and its amelioration by a compiler. In *Proceedings of the 14th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 181 –189, 2000.
- [134] C. Gou and G. Gaydadjiev. Elastic pipeline: Addressing GPU on-chip shared memory bank conflicts. In *Proceedings of the 8th ACM International Conference on Computing Frontiers, CF '11*, pages 3:1–3:11, 2011.

Tampereen teknillinen yliopisto
PL 527
33101 Tampere

Tampere University of Technology
P.O.B. 527
FI-33101 Tampere, Finland

ISBN 978-952-15-2932-0
ISSN 1459-2045