

Automatically Parallelizing Embedded Legacy Software on Soft-Core SoCs

Automatische Parallelisierung bestehender eingebetteter Software mit Soft-Core SoCs

Zur Erlangung des akademischen Grades Doktor-Ingenieur (Dr.-Ing.)

genehmigte Dissertation von Kris Heid aus Groß-Umstadt

Tag der Einreichung: 24.06.2019, Tag der Prüfung: 20.08.2019

Darmstadt — D 17

1. Gutachten: Prof. Dr.-Ing. Christian Hochberger

2. Gutachten: Prof. Dr.-Ing. Jeronimo Castrillon



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Fachgebiet Rechnersysteme
Fachbereich Elektrotechnik
und Informationstechnik

Automatically Parallelizing Embedded Legacy Software on Soft-Core SoCs
Automatische Parallelisierung bestehender eingebetteter Software mit Soft-Core SoCs

Genehmigte Dissertation von Kris Heid aus Groß-Umstadt

1. Gutachten: Prof. Dr.-Ing. Christian Hochberger
2. Gutachten: Prof. Dr.-Ing. Jeronimo Castrillon

Tag der Einreichung: 24.06.2019

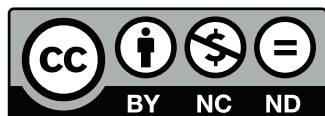
Tag der Prüfung: 20.08.2019

Darmstadt — D 17

URN: urn:nbn:de:tuda-tuprints-90205

URL: <http://tuprints.ulb.tu-darmstadt.de/id/eprint/9020>

Dieses Dokument wird bereitgestellt von tuprints,
E-Publishing-Service der TU Darmstadt
<http://tuprints.ulb.tu-darmstadt.de>
tuprints@ulb.tu-darmstadt.de



Die Veröffentlichung steht unter folgender Creative Commons Lizenz:
Namensnennung – Keine kommerzielle Nutzung – Keine Bearbeitung 4.0 International
<https://creativecommons.org/licenses/by/4.0/deed.de>

Erklärungen laut Promotionsordnung

§ 8 Abs. 1 lit. c PromO

Ich versichere hiermit, dass die elektronische Version meiner Dissertation mit der schriftlichen Version übereinstimmt.

§ 8 Abs. 1 lit. d PromO

Ich versichere hiermit, dass zu einem vorherigen Zeitpunkt noch keine Promotion versucht wurde. In diesem Fall sind nähere Angaben über Zeitpunkt, Hochschule, Dissertationsthema und Ergebnis dieses Versuchs mitzuteilen.

§ 9 Abs. 1 PromO

Ich versichere hiermit, dass die vorliegende Dissertation selbstständig und nur unter Verwendung der angegebenen Quellen verfasst wurde.

§ 9 Abs. 2 PromO

Die Arbeit hat bisher noch nicht zu Prüfungszwecken gedient.

Ort, Datum und Unterschrift



Abstract

Nowadays, embedded systems are utilized in many areas and become omnipresent, making people's lives more comfortable. Embedded systems have to handle more and more functionality in many products. To maintain the often required low energy consumption, multi-core systems provide high performance at moderate energy consumption. The development started with dual-core processors and has today reached many-core designs with dozens and hundreds of processor cores. However, existing applications can barely leverage the potential of that many cores.

Legacy applications are usually written sequentially and thus typically use only one processor core. Thus, these applications do not benefit from the advantages provided by modern many-core systems. Rewriting those applications to use multiple cores requires new skills from developers and it is also time-consuming and highly error prone. Dozens of languages, APIs and compilers have already been presented in the past decades to aid the user with parallelizing applications. Fully automatic parallelizing compilers are seen as the holy grail, since the user effort is kept minimal. However, automatic parallelizers often cannot extract parallelism as good as user aided approaches. Most of these parallelization tools are designed for desktop and high-performance systems and are thus not tuned or applicable for low performance embedded systems. To improve this situation, this work presents an automatic parallelizer for embedded systems, which is able to mostly deliver better quality than user aided approaches and if not allows easy manual fine-tuning.

Parallelization tools extract concurrently executable tasks from an application. These tasks can then be executed on different processor cores. Parallelization tools and automatic parallelizers in particular often struggle to efficiently map the extracted parallelism to an existing multi-core processor. This work uses soft-core processors on Field Programmable Gate Arrays (FPGAs), which makes it possible to realize custom multi-core designs in hardware, within a few minutes. This allows to adapt the multi-core processor to the characteristics of the extracted parallelism. Especially, core-interconnects for communication can be optimized to fit the communication pattern of the parallel application.

Embedded applications are often structured as follows: receive input data, (multiple) data processing steps, data output. The multiple processing steps are often realized as consecutive loosely coupled transformations. These steps naturally already model the structure of a processing pipeline. It is the goal of this work to extract this kind of pipeline-parallelism from an application and map it to multiple cores to increase the overall throughput of the system. Multiple cores forming a chain with direct communication channels ideally fit this pattern. The previously described, so called pipeline-parallelism is a barely addressed concept in most parallelization tools. Also, current multi-core designs often do not support the hardware flexibility provided by soft-cores, targeted in this approach.

The main contribution of this work is an automatic parallelizer which is able to map different processing steps from the source-code of a sequential application to different cores in a multi-core pipeline. Users only specify the required processing speed after parallelization. The developed tool tries to extract a matching parallelized software design along with a custom multi-core design out of sequential embedded legacy applications. The automatically created multi-core system already contains used peripherals extracted from the source-code and is ready to be used. The presented parallelizer implements multi-objective optimization to generate a minimal hardware design, just fulfilling the user defined requirement. To the best of my knowledge, the possibility to generate such a multi-core pipeline defined by the demands of the parallelized software has never been presented before.

The approach is implemented for two soft-core processors and evaluation shows for both targets high speedups of 12x and higher at a reasonable hardware overhead. Compared to other automatic parallelizers, which mainly focus on speedups through latency reduction, significantly higher speedups can be achieved depending on the given application structure.



Zusammenfassung

Eingebettete Systeme werden heutzutage in vielen Bereich eingesetzt, um unseren Alltag zu erleichtern. Hierbei übernehmen diese immer mehr Aufgaben. Um die wachsende Anzahl an Aufgaben erledigen zu können werden Mehrkernprozessoren benötigt, welche eine hohe Leistungsfähigkeit bei gleichzeitig moderatem Energiebedarf bieten. Waren die ersten Mehrkernprozessoren noch mit zwei Rechenkernen ausgestattet, so existieren heute bereits Prozessoren mit dutzenden und hunderten Rechenkernen. Viele bestehende Anwendungen können jedoch ohne Anpassungen kaum von dieser hohen Anzahl an Rechenkernen profitieren.

Existierende Anwendungen haben meist einen sequenziellen Programmablauf und nutzen daher per se nur einen einzigen Rechenkern. Somit können sie nicht von den Vorteilen und der Rechenleistung moderner Prozessoren profitieren. Die Anwendungen müssten umgeschrieben werden, um das volle Potenzial von Mehrkernprozessoren zu nutzen, was jedoch neue Fertigkeiten und Denkmuster von Entwicklern fordert und zudem sehr mühsam und fehleranfällig ist. In den letzten Jahren wurden bereits eine Reihe an Programmiersprachen, Programmierschnittstellen und Compilern entwickelt, um Entwickler bei der Parallelisierung zu unterstützen. Dabei sind vollständig automatische Parallelisierer der heilige Gral der Parallelisierung, da sie dem Nutzer den Großteil der Arbeit abnehmen. Automatische Parallelisierer können jedoch teilweise nicht die Qualität der einer manuellen Parallelisierung von erfahrenen Entwicklern erreichen. Die Meisten der entwickelten Parallelisierungswerkzeuge sind außerdem für Desktop- oder Hochleistungsrechner entworfen worden und sind daher kaum an die Bedürfnisse eingebetteter Systeme angepasst. Daher wird in dieser Arbeit ein automatischer Parallelisierer für eingebettete Systeme vorgestellt, welcher oftmals die Qualität manueller Parallelisierungen übertrifft und auf Wunsch manuelle Anpassungen erlaubt.

Parallelisierungswerkzeuge sind in der Lage parallel ausführbare Aufgaben aus einer Anwendung zu extrahieren und diese dann auf verschiedenen Prozessorkernen auszuführen. Vor allem automatische Parallelisierer haben jedoch oft Probleme den gefundenen Parallelismus effizient auf die verfügbare beschränkte Anzahl an Kernen abzubilden. Daher werden in dieser Arbeit Soft-Core Prozessoren auf FPGAs verwendet, welche es ermöglichen ein angepasstes Mehrkernsystem innerhalb weniger Minuten zu realisieren. Hierdurch kann das System auf die Charakteristiken des extrahierten Parallelismus angepasst werden. Besonders die Kommunikationsinfrastruktur kann speziell auf das Kommunikationsmuster der parallelisierten Anwendung angepasst werden.

Anwendungen eingebetteter Systeme haben oftmals die folgende Struktur: Eingangsdaten empfangen, Verarbeitung der Daten (in mehreren Schritten), Ausgabe der Daten. Die verschiedenen Verarbeitungsschritte sind hierbei meist nur locker gekoppelte aufeinanderfolgende Transformationen der Daten. Die beschriebenen Schritte weisen somit die Struktur eine Verarbeitungs-Pipeline auf. Daher ist das Ziel dieser Arbeit diesen sogenannten Pipeline-Parallelismus aus der Anwendung zu extrahieren. Die einzelnen Verarbeitungsschritte werden dann zur Erhöhung des Datendurchsatzes auf verschiedene Kerne abgebildet. Hierbei passt eine Kette von Prozessorkernen mit direkter Kommunikation zwischen den Nachbarn ideal zur Charakteristik des extrahierten Pipeline-Parallelismus. Das Konzept des Pipeline-Parallelismus ist in heutigen Parallelisierungswerkzeugen eher selten vorzufinden, was auch daran liegt, dass aktuelle Mehrkernsysteme nicht die benötigte Struktur bzw. Flexibilität von Soft-Cores bieten, um die Pipeline-Muster ideal abzubilden.

Der Beitrag dieser Arbeit ist ein automatischer Parallelisierer, welcher in der Lage ist aus einer sequentiellen Anwendung Pipeline-Parallelismus zu extrahieren und diesen auf eine zuvor beschriebene Kette von Prozessorkernen abzubilden. Der Nutzer muss lediglich eine Verarbeitungsgeschwindigkeit vorgeben, welche die parallelisierte Anwendung erreichen soll. Der Parallelisierer extrahiert anschließend den nötigen Parallelismus aus der Anwendung und erstellt automatisch ein individuell angepasstes Mehrkernsystem. In diesem System sind neben der Kommunikationsinfrastruktur auch bereits alle genutzten

Peripherien enthalten, sodass es direkt einsatzbereit ist. Der Parallelisierer optimiert das System in verschiedenen Aspekten, um möglichst minimale Hardware zu generieren, die dennoch den Nutzervorgaben entspricht. Die Generierung einer Mehrkern-Pipeline, die individuell auf die parallelisierte Anwendung angepasst ist, wurde nach meinem besten Wissen noch nicht veröffentlicht.

Das Konzept wurde für zwei Soft-Core Prozessoren implementiert und die Evaluation weist einen hohen möglichen Geschwindigkeitszuwachs des Faktors 12 und mehr, bei moderat erhöhtem Hardwarebedarf auf. Im Vergleich zu anderen automatischen Parallelisierern, die sich lediglich auf eine Erhöhung des Durchsatzes durch Verringerung der Latenz fokussieren, kann ein weitaus höherer Geschwindigkeitszuwachs erreicht werden, falls die Anwendung die nötigen Charakteristiken aufweist.

Contents

Abbreviations	10
List of Figures	12
List of Tables	14
List of Code Listings	15
1 Introduction	17
1.1 Motivation	17
1.2 Problems & Goals	18
1.3 Work plan	19
2 State-of-the-Art	21
2.1 Multi-/Many-Core SoC Platforms	21
2.1.1 Embedded Multi-Core Architectures	21
2.1.2 Embedded Many-Core Architectures	22
2.1.3 Soft-core multi-/many-cores	24
2.1.4 Conclusion	25
2.2 Extracting Parallelism from Applications: Design Choices	26
2.2.1 Programming Paradigms	26
2.2.2 Types of Parallelism	27
2.2.3 Partitioning Level	27
2.2.4 Memory Architecture	28
2.2.5 Task Scheduling	28
2.2.6 Conclusion and Scope	28
2.3 Parallelization Tools	29
2.3.1 Tools out of Scope	29
2.3.2 DSLs/Language Extensions	31
2.3.3 APIs/Libraries	32
2.3.4 Annotations	33
2.3.5 Automatic	36
2.3.6 Summary	38
3 Target Platforms	40
3.1 SpartanMC	40
3.1.1 Inter-Core Communication	40
3.1.2 Performance-Counter	45
3.2 MicroBlaze	46
3.2.1 Inter-Core Communication	46
3.2.2 Timer - Performance Counter	47
3.3 Inter-Core Communication performance evaluation	48
3.3.1 1-to-1 Communication	48
3.3.2 1-to-N and N-to-1 Communication	48
3.4 Global Memory	49
4 Used Multi-Core Architectures and Execution Concepts	51
4.1 Required Application Structure	51

4.2	Pipeline	51
4.2.1	Pipeline Hardware Limitations	52
4.3	Pipeline with Replication	53
4.3.1	Replicated Pipeline Hardware Limitations	54
4.4	Shared Global Memory	54
4.5	Communication Overhead	54
4.6	Latency	55
5	Automatic Parallelization	57
5.1	Overall toolflow	57
5.1.1	AutoPerf: Application Profiling	57
5.1.2	AutoStreams: Automatic Annotations	57
5.1.3	μ Streams: Annotated Source-Code Transformation	59
5.1.4	Refine Timing Constraints	59
5.2	Common Software Infrastructure	60
5.2.1	Cetus	60
5.2.2	Common Transformation Infrastructure	62
5.3	AutoPerf	64
5.3.1	Traditional Approaches	64
5.3.2	Implementation	66
5.3.3	Credibility of Measured Results	67
5.4	LoopOptimizer	69
5.4.1	Loop Parallelization Techniques	70
5.5	AutoStreams	73
5.5.1	Optimization Points	73
5.5.2	Implementation	74
5.6	μ Streams	78
5.6.1	Usable Pragmas	79
5.6.2	Unsupported Constructs	80
5.6.3	Implementation	81
5.7	PeripheralDetector	89
5.7.1	Workflow	89
5.7.2	Implementation	91
5.7.3	Sources of False Detection	92
5.7.4	Automatic Peripheral Detection on Multi-Core Systems	93
6	Evaluation	94
6.1	Test Applications	94
6.1.1	ADPCM	95
6.1.2	MJPEG2000	95
6.1.3	IIR Butterworth Filter	96
6.1.4	Firewall	96
6.2	Application Profiles	97
6.2.1	Benchmark Characteristics	98
6.3	Possible Parallelization & Performance Gain	100
6.3.1	Parallelization without Optimizations	100
6.3.2	Parallelization with Replication	104
6.3.3	Parallelization with DMA Interconnects	107
6.3.4	Parallelization with LoopOptimizer	110

6.4	AutoStreams Estimation Accuracy	115
6.4.1	Hardware Estimation	115
6.4.2	Application Runtime Estimation	117
6.5	Parallelization with Peripheral In-&Output	119
6.5.1	Firewall	119
6.5.2	ADPCM with IO	124
6.6	Manual vs. Automatic Parallelization	126
6.7	Maximum Frequency Multi-Core Designs	128
6.7.1	Speedup vs. Performance Loss through Lower Frequency	130
6.8	Latency in the Generated Pipelines	131
6.9	Dynamic Verification: System Tests	133
6.10	Comparison with Related Work	133
6.11	Best Practice Proposals	134
7	Conclusion & Future Work	135
	References	139
	Supervised Students' Theses	146
	Own Publications	147

Abbreviations

AHB	Advanced High-performance Bus
AI	artificial intelligence
APD	Activity and Pattern Diagram
API	application programming interface
ASIC	application-specific integrated circuit
AST	Abstract Syntax Tree
BRAM	Block RAM
CDFG	control data flow graph
Cell B.E.	Cell Broadband Engine
CFG	control-flow graph
CGRA	coarse-grain reconfigurable architecture
CPN	C for Process Networks
DAG	directed acyclic task graph
DMA	direct memory access
DMCG	Directive-Based MPI Code Generator
DSE	design-space exploration
DSL	domain specific language
DSP	digital signal processing block
EMB ²	Embedded Multicore Building Blocks
FMC	FPGA Mezzanine Card
FPGA	Field Programmable Gate Array
FSL	fast simplex link
GCC	GNU Compiler Collection
GPIO	general purpose input/output
GPU	graphics processing unit
GUI	Graphical User Interface
HDL	hardware description language
HLS	high-level synthesis
HPC	high performance computing
ICC	Intel C/C++ Compiler
IDE	integrated development environment
ILP	instruction level parallelism
IoT	Internet of Things
ISA	instruction set architecture
ISR	interrupt service routine

JTAG	Joint Test Action Group IEEE 1149.1
KPN	Kahn Process Network
LUT	look up table
MCAPI	Multicore Communications API
MDM	MicroBlaze Debug Module
MIMD	multiple instruction multiple data
MP-SoC	multi-processor system-on-chip
MPI	message passing interface
MRAPI	Multicore Resource Management API
MTAPI	Multicore Task Management API
NoC	Network-on-Chip
NUMA	nonuniform memory access
OpenHMPP OS	Open Hybrid Multicore Parallel Programming operating system
PPE	PowerPC processor element
RISC	reduced instruction set architecture
SANLP	static affine nested loop program
SIMD	single instruction multiple data
SMP	symetric multi processor
SoC	System-On-Chip
SPARC	Scalable Processor ARChitecture
SPE	synergetic processing element
SUIF	Stanford University Intermediate Format
TBB	Intel Threading Building Blocks
TLP	thread-level parallelism
TPL	task parallel library
UART	Universal Asynchronous Receiver Transmitter
VLIW	very long instruction word
WCET	worst-case execution time

List of Figures

3.1	Core-Connector simplified schematic hardware design	41
3.2	Dispatcher simplified schematic hardware design	42
3.3	Concentrator simplified schematic hardware design	43
3.4	MemSwap Dual simplified schematic hardware design	44
3.5	MemSwap Multi simplified schematic hardware design	44
3.6	Alternative approach for MemSwap Multi with fewer BRAMs	45
3.7	Shared Memory simplified schematic hardware design	45
3.8	MicroBlaze Mailbox AXI-Stream simplified schematic hardware design	47
3.9	MicroBlaze shared memory simplified schematic hardware design	47
3.10	Transmission duration vs. data size for different 1-to-1 core-interconnects	49
3.11	Global memory throughput	50
4.1	Pure pipeline, hardware configuration	52
4.2	Replicated pipeline, hardware configuration	53
4.3	Pipeline with global memory, hardware configuration	54
5.1	Simplified Overall Automatic Parallelization Toolflow with Tool Section Reference	58
5.2	μ Streams concept: SW transformation	59
5.3	Simplified Cetus Abstract Syntax Tree (AST) generated from Listing 5.1	61
5.4	Simplified μ Streams transformation pass runner class diagram	63
5.5	Detailed AutoPerf toolflow	65
5.6	Detailed LoopOptimizer toolflow for different operation modes	69
5.7	Detailed AutoStreams toolflow	73
5.8	Search tree for design space exploration	76
5.9	Detailed μ Streams toolflow (dashed=optional)	78
5.10	Task dependency created from Listing 5.10	84
5.11	Generated pipeline structure and communication	88
5.12	Detailed Peripheral-Detector toolflow (dashed=optional)	90
6.1	Image tiles as processed by the JPEG 2000 encoder	96
6.2	Firewall zones	97
6.3	ADPCM 2x speedup requirement , no optimizations	101
6.4	MJPEG 2x & 4x speedup requirement , no optimizations	102
6.5	IIR 2x speedup requirement , no optimizations	103
6.6	ADPCM with replication	105
6.7	MJPEG with replication	106
6.8	IIR with replication	107
6.9	SpartanMC MJPEG replication with and without DMA-interconnects, 8x speedup require- ment	109
6.10	SpartanMC IIR replication with and without DMA-interconnects, 12x speedup requirement	110
6.11	ADPCM with loop optimization	111
6.12	MJPEG with loop optimization	112
6.13	IIR with loop optimization	113
6.14	IIR 2x speedup requirement, loop splitting VS loop fission	114
6.15	SpartanMC hardware estimation error	115
6.16	MicroBlaze hardware estimation error	116
6.17	SpartanMC cycles estimation error of different parallelized software parts	118
6.18	MicroBlaze cycles estimation error of different parallelized software parts	119
6.19	Firewall hardware design	120
6.20	Network throughput in Mbit/s for different system configurations	122
6.21	Network packet throughput for different system configurations	122

6.22	Duration per SpartanMC core with ADPCM 8x speedup requirement, core 1: 5x replication	125
6.23	ADPCM 12x speedup requirement with DMA and loop optimizations, manually parallelized, first try	126
6.24	ADPCM 12x speedup requirement with DMA and loop optimizations, manually parallelized, second try	127
6.25	ADPCM 12x speedup requirement with DMA and loop optimizations, manually parallelized after 16 tries	128
6.26	ADPCM and IIR maximum achievable frequency evaluation over multiple connected SpartanMC cores and interconnect types	129
6.27	ADPCM and IIR maximum achievable frequency evaluation over multiple connected MicroBlaze cores and interconnect types	129
6.28	MJPEG2000 maximum achievable frequency evaluation over multiple connected SpartanMC cores and interconnect types	130
6.29	MJPEG2000 maximum achievable frequency evaluation over multiple connected MicroBlaze cores and interconnect types	131
6.30	Latency increase compared to the sequential variant with MicroBlaze	132
6.31	Latency increase compared to the sequential variant with SpartanMC	132

List of Tables

2.1	Reviewed parallelization tools	30
4.1	Pipeline execution with 1-to-1 interconnects	52
4.2	Pipeline execution with 1-to-N, N-to-1 interconnects, replicated superscalar pipeline	53
4.3	Latency for pipeline execution	56
5.1	Produced performance-profile example	64
5.2	Detection accuracy with different applications	93
6.1	Benchmark processing step runtimes in cycles for SpartanMC and MicroBlaze	99
6.3	SpartanMC core and interconnect hardware cost on Artix-7 XC7A200T FPGA	108
6.4	Achieved speedups and AutoStreams DMA design choice	109
6.5	Achieved speedups and AutoStreams DMA design choice for previous replicated designs .	109
6.6	Estimation accuracy as relative estimation error in percent	123
6.7	SpartanMC ADPCM performance-profile with peripheral IO	125

List of Code Listings

5.1	Example Cetus Input Program	61
5.2	Input source-code for profiling	64
5.3	Instrumented source-code (diff-style highlighting: green lines with + are added)	67
5.4	Original loop	70
5.5	Fissioned loop	70
5.6	Split loop	70
5.7	Break loop	72
5.8	Break loop transformed	72
5.9	Usable μ Streams pragmas	79
5.10	Example code to visualize task pipeline creation	83
5.11	Simplified abstract XML hardware description	86
5.12	Simplified main.c Freemarker task template	87
5.13	Usage of different Peripherals in a SpartanMC C-application	89
6.1	Generated assembler code, IIR benchmark processing step 0, parallelized variant	104
6.2	Generated assembler code, IIR benchmark processing step 0, single-core variant	104



1 Introduction

Computers have become a major part of our everyday live. Even though they are not always directly visible or identifiable as a computer. They are embedded into many products that we daily use. Today, way more so-called embedded computers or embedded systems exist than traditional desktop computers. Embedded systems are used in many areas, such as car industry, avionics, manufacturing industry, multimedia entertainment systems, health care and household items. With the Internet of Things (IoT) boom in the last years, almost everything contains embedded systems and is connected. We live in a world where coffee makers and dish washers can be controlled over the internet and cleaning robots tidy up your home while you are at work. These embedded systems take over more and more jobs and also the complexity which these systems handle increases. With more complex jobs, also the demanded processing power increases. For example, a cleaning robot continuously scans the room with a 360 degree distance measurement, generates a map of the room and calculates an ideal cleaning route covering all areas. New obstacles may appear on the route and new rooms could become visible, requiring an adaptation of the map and the route. At the same time the robot has to interact with a smartphone to display the status and receive commands. These tasks require high processing power from a battery driven device.

Traditionally, the processing power of a processor increases with a higher working frequency e.g. clock frequency, besides other methods found in the past decades of research in this field. However, a higher clock frequency and thereby increased operation voltage result in disproportionately high power consumption [1]. Performance scaling with frequency has also physically reached its limits with today's chip manufacturing techniques. At the same time, embedded devices are often battery powered and demand high processing power at an extremely low energy consumption.

These combined demands are fulfilled through multi-core processors nowadays. Multiple processors are combined on one chip and process workloads together. Multi-core processors theoretically increase processing power with each additional core, while the operation frequency and thereby the power consumption can be kept low. Thus, a multi-core system can maintain the same processing power of a single-core system at lower power consumption [2].

Writing software to use multiple concurrent processors is not that easy. Firstly, many existing algorithms in software are not written to process data concurrently and not all algorithms hold such concurrency. Secondly, software developers must learn new techniques to write new concurrent applications or adapt existing applications.

1.1 Motivation

A lot of legacy applications already exist for embedded systems which could benefit from multi-core devices. Especially legacy software that grew over the time requires adaptation since the additional tasks cannot be fulfilled anymore with a single-core processor. Parallelism must be extracted from the application. This parallelism is represented by different tasks that are mapped to different processor cores. Several possibilities to detect and leverage parallelism already exist. The challenge is to find enough parallelism and to map it efficiently to the multi-core platform [3]. Since multi-core platforms have unique characteristics, like for example communication cost, several objectives have to be optimized to successfully parallelize software. If for example an extracted parallel task is very small, it might take more time to tell another processor to start this task and collect the results than to execute it together with the original task on one processor. To aid the programmer in this process, already plenty of programming languages and language constructs exist. These techniques require manual effort from a developer. Alternatively, some automatic parallelization tools exist to relieve the developer. Such tools imply low effort for the developer but sometimes hand tuned parallelizations from skilled programmers result in a better performance. Language constructs for parallelization often exhibit great parallelization

possibilities. This gives a developer a great choice for possible parallelization, but also great chances for inexperienced developers to generate bad parallelizations.

Thus, automatic parallelization is desirable if it works well. And it works better when narrowed down to specific use cases. Development for parallelization tools has mainly been driven by the high performance computing (HPC) community, without focus on embedded systems. By targeting embedded systems for parallelization, restrictions as well as new opportunities for automatic parallelization tools apply in this narrowed field. In the domain of HPC, the maximum amount of parallelism is often desired. The higher the parallelization is, the faster the application runs, the better the solution is considered. Maximum parallelization is not necessarily optimal for an embedded system. Embedded systems often have a minimum required processing speed. As long as this speed can be achieved by parallelization, everything is fine. Higher parallelization is not required and might even lead to less energy efficiency.

With regard to hardware efficiency and task mapping to processors, another aspect can come into play for embedded systems: FPGAs. FPGAs are special circuits to realize almost arbitrary other digital systems. FPGA development has advanced so far that it becomes possible to even realize multiple embedded processors (called soft-cores) on one FPGA. The flexibility and reprogrammability of FPGAs make it possible to use arbitrary custom multi-core designs. It is thinkable to generate hardware to better fit the needs and structure of the parallelized application. One could even adapt the multi-core system with changing application requirements.

1.2 Problems & Goals

Different kinds of parallelism exist within applications that are coarse-grained enough to justify offloading to a different processor: Task-level, data-level and pipeline parallelism. The first two are widely researched and applied in parallelization tools. Pipeline parallelism is addressed by fewer tools, because it is not as widely applicable and has restrictions regarding application structure. Nevertheless, it is shown that parallelization for these applications on desktop computers[4, 5] gives promising results. Cordes et al.[6] showed the applicability of this concept for a simulated embedded system. Besides benchmark applications which mostly cover only one data transformation algorithm, embedded systems might execute multiple such transformations and also have to handle data in and output (not covered by Cordes et al.). Thus, the execution order is often: receive input data from peripherals, run (multiple) data transformation steps, send output data to peripherals. Such an application structure already exhibits different pipeline steps, which could very well be extracted and transformed into a processing pipeline to increase the throughput of the application.

An automatic parallelization tool with the following characteristics would be well suitable for embedded systems:

- Extract only necessary parallelism and not as much as possible. This would not result in the fastest system, but a sufficiently fast system with a small hardware footprint.
- Consider full system parallelization and not only concentrate on parallelizing loops.
- Consider the influence of peripheral interaction during parallelization, even though embedded systems always have peripheral interaction.
- Target low-performance embedded systems running bare-metal, incapable of running an operating-system.
- Adapt a configurable hardware system such as soft-cores on FPGAs to the extracted parallelism characteristics.

Currently there is no automatic parallelizing compiler for embedded systems covering the described aspects.

There are several open questions which arise from the set objectives:

- How good are current soft-core communication interconnects and how big is the communication overhead in contrast to computation complexity. Can interconnects be improved to better support the pipeline concept?
- How much parallelism is extractable from this concept? Previous approaches were only using processors with up to four cores.
- Can parallelization be done with distributed or shared-distributed memory systems to overcome the bottleneck of a global common memory?
- Can automatic parallelization keep up with hand parallelized variants from experienced developers?
- Can multi-core soft-core designs be automatically created and tuned to the application characteristics to not bother software developers with hardware design?
- What is the impact on the latency of a pipeline parallel design, since an upper bound for latency is important in some embedded systems?
- Do multi-core designs have a negative impact on the maximum achievable frequency in contrast to single-core designs, when realized on an FPGA?

1.3 Work plan

A work plan is elaborated to design an automatic parallelization tool, covering the aforementioned aspects. The work plan also incorporates the described uncertainties and open questions to verify the applicability of an automatic parallelization tool in this environment.

The following work packages are stated:

1. Find applications for low-performance embedded systems that benefit from the aspired pipeline concept.
2. Analyze current multi-core capabilities of soft-cores and the performance of supported interconnects. If necessary or possible, the multi-core capabilities and interconnect performance should be improved to support pipeline parallelism.
3. Investigate the pipeline concept with low-performance embedded systems through a manual annotation based parallelization tool at first. With an annotation-based parallelization, different applications can be parallelized and the resulting performance can be measured. The parallelization tool should work on the granularity of functions or multiple statements to deliver coarse-grained enough tasks to justify offloading to a different processor core. Also, functions imply a clear, easily analyzable interface for input and output data. This first step should reveal, if this concept is applicable and where possible improvements can be made.
4. The tools should be implemented as source-to-source tools. This makes the tools more independent of the target architecture and the compiler implementation. Additionally, this gives the user the freedom to easily analyze and manipulate the generated design without touching the implementation.
5. Design a profiler to give the user an idea of which parts to parallelize. Otherwise, the user would need to guess and parallelize with trial and error.

-
6. Investigate concepts to also parallelize loops, which are often steps consuming much processing time.
 7. Develop concepts to detect used peripherals from the source-code. On the one hand, this allows the parallelizer to adapt to peripheral interaction. On the other hand, the necessary hardware infrastructure in terms of cores, interconnects and peripherals per core can be inferred.
 8. In the last step, an automatic parallelization tool can be designed, using the previously developed tools. The different intermediate tools allow an easy exchangeability to adapt for new target platforms. An application performance profile should be automatically analyzed and annotations can automatically be set for the previously developed parallelization tool. The automatic parallelization tool should extract only as much parallelism as necessary to fulfill user defined requirements. The user must specify a minimum input processing rate as often demanded by embedded systems. Besides the parallelized software, a multi-core hardware design should be provided by the parallelization tool. Different interconnects should be evaluated to get the necessary performance with resource efficient interconnect. The tool should be able to obey restricted hardware bounds since differently sized FPGAs exist or other digital systems are also desired on the same FPGA.

The following pages describe the state of the art, related work and the shortcomings of existing tools in more detail. Afterwards, the chosen target architectures are analyzed, the tool's implementation details are highlighted and design choices are described according to the work plan. The quality of the implemented automatic parallelization tool is evaluated with respect to different implemented optimizations to increase extractable parallelism. A conclusion is given at the end and suggestions for future work are made.

2 State-of-the-Art

In this section, firstly multi- and many-core architectures are reviewed in Section 2.1, with a specific focus on low-performance embedded domain. Afterwards, Section 2.2 discusses different possible design choices for extracting parallelism out of sequentially written applications. Last but not least, in Section 2.3 different parallelization tools are reviewed, categorized by their programming paradigm.

2.1 Multi-/Many-Core SoC Platforms

For many years the performance of processors was mainly increased through higher frequencies. When high-end processors surpassed the 4 GHz domain around year 2000, power consumption and wire delays became the dominant problems, limiting further scaling by frequency [1]. Researchers became aware of this dead-end by the 80s and had already researched multi- and many-core architectures. In 2001 the first dual-core processor (IBM POWER4) for personal computers was released. With the first multi-cores, the overall performance of the system was boosted, given that the application is able to leverage multiple cores. Also, multiple cores at lower clock rates achieved lower power consumption at the same level of performance, compared to a higher clocked single-core design [2]. Over the years, the core count of multi-core processors increased continuously from two up to hundreds or thousands of cores, which are then typically referred to as many-core processors.

The term "embedded processor" can be widely stretched to also cover high performance desktop/server processors embedded into a technical device which needs to be controlled or supervised. However, in the context of this work, the focus is on processors with low power consumption, small size at low cost which, implies more or less limited processing power.

In the following, the architectures are separated into multi- and many-core as well as configurable soft-core multi-processor system-on-chips (MP-SoCs), with the latter being able to cover both of the former domains, depending on user configuration.

2.1.1 Embedded Multi-Core Architectures

Today's embedded multi-core processors are widely dominated by ARM. ARM processors are widespread in many electronic systems, such as smartphones, automotive applications, sensors, medical devices or modems and routers. Devices requiring considerable processing power, such as smartphones, nowadays leverage System-On-Chips (SoCs) with multiple ARM high performance Cortex-A¹ series cores. Due to their popularity, there are many vendors producing Cortex-A multi-core SoCs: Freescale iMX², Apple Ax, Samsung Exynos³, HiSilicon Kirin⁴, MediaTek MTxxxx and Helio⁵, RockChip RK3xxx⁶, Qualcomm Snapdragon⁷, Nvidia Tegra⁸ etc., just to name a few. These SoCs are shared-memory architectures, typically with a bus-based cache coherency protocol and nowadays often contain between four and eight cores. ARM's newest interconnect (CMN-600⁹) is even implemented as a mesh network, allowing for good

¹ Product brief: <https://www.arm.com/products/silicon-ip-cpu>

² Product brief https://www.nxp.com/products/processors-and-microcontrollers/arm-based-processors-and-mcus/i.mx-applications-processors:IMX_HOME

³ Product brief: <https://www.samsung.com/semiconductor/minisite/exynos/products/mobileprocessor/exynos-9-series-9820/>

⁴ Product brief: <http://www.hisilicon.com/en/Products/ProductList/Kirin>

⁵ Product brief: <https://www.mediatek.com/products/smartphones/helio-x>

⁶ Product brief: <https://www.rockchip.nl/>

⁷ Product brief: <https://www.qualcomm.com/snapdragon>

⁸ Product brief: <https://www.nvidia.de/object/tegra-de.html>

⁹ Product brief: <https://developer.arm.com/products/system-ip/corelink-interconnect/corelink-coherent-mesh-network-family/corelink-cmn-600ae>

scalability beyond eight cores. Due to the variety of ARM's Cortex-A cores in terms of power and performance, ARM's big.LITTLE[7] concept is used in many mobile SoCs. High efficiency cores are used during low performance, low power scenarios and less efficient, high performance cores are switched on when required. The kernel scheduler implementation defines whether switching between the performance and efficiency cluster/cores is possible on a per-core basis or only for the whole cluster. Alternatively, heterogeneous task scheduling can be implemented. The performance of Cortex-A series processors is good enough to even run a recent full desktop operating system as shown with the Raspberry Pi project.

Besides the Cortex-A series, ARM also offers the Cortex-R and Cortex-M series targeting real-time and very low power, performance and cost. The multi-core SoCs for these processors are not as widely addressed by the vendors as for the Cortex-A series. NXP Semiconductors has the LPC4300¹⁰ containing one powerful ARM Cortex-M4F and one or two low performance ARM Cortex-M0. The M0 is designed to mainly handle peripheral interaction while the M4 does compute-intensive work. It is a shared-memory 32-bit architecture with interconnect over Advanced High-performance Bus (AHB). Texas Instruments has the OMAP5 series¹¹ containing two very powerful Cortex-A15 and two Cortex-M4. The M4 can be used for low-power offload and real-time tasks. Even though it has two M4 cores, the chip rather belongs to the high performance embedded domain through the Cortex-A15 cores, with the M4 rather resembling a co-processor. The Espressif ESP32¹² is a Tensilica Xtensa 32-bit LX6 symmetric multi processor (SMP) dual-core. The device is able to run FreeRTOS and thus task scheduling becomes possible. Besides interaction between multiple peripherals, the device is even capable of running for example a simple web-server. The Parallax Propeller¹³ is a 32-Bit hexa-core with a distributed-shared memory architecture with its own instruction set architecture (ISA). Each core has its own 2KB RAM and a round-robin arbitrated shared 64KB memory partly used as RAM and ROM. Quite uniquely, all cores can have simultaneous read/write access to the same peripheral pins and have to synchronize over mutexes. When it comes to peripherals, many microcontrollers include hardware for SPI or I²C, while the Propeller has dedicated cores for protocol handling. However, dedicating peripherals to cores and thus also dedication interrupts, the reaction time to multiple interrupts becomes smaller and also more predictable. Each core delivers 20 MIPS per core and is thus comparable with the Cortex-M0. Parallax later also released the Propeller 2 with up to 16 cores at a slightly higher clock.

2.1.2 Embedded Many-Core Architectures

Many-Core processors offer much processing power due to the high number of cores. Image processing is a application field that can very well make use of many-core architectures. Image processing is also often applied in the domain of embedded computing. Many-core architectures realized as single instruction multiple data (SIMD) processors are often proposed for low-power, high-efficiency. However, those architectures mostly focus on dividing data sets into multiple parts and applying parallel (floating-point) operations only. This approach works well for scenarios having high data parallelism to leverage, but these architectures lack in applicability to general purpose computing without data parallelism. Examples are embedded graphics processing units (GPUs) as found in today's smartphones and also other products like Hiveflex ISP2300[8], ClearSpeed CSX series[9], Intel's Myriad X¹⁴ and Teraflops/Polaris[10], Imagine Stream Processor[11] and it's commercialized variant SPI Storm 1. The Imagine Stream Processor is slightly different. Its ability to model processing pipelines through a series of cores is unique. These pipelines are realized over direct local connections instead of using rather limited

¹⁰ Product brief: https://www.nxp.com/products/processors-and-microcontrollers/arm-based-processors-and-mcus/lpc-cortex-m-mcus/lpc4300-cortex-m4-m0:MC_1403790133078#/

¹¹ Product brief: <http://www.ti.com/pdfs/wtbu/SWCT010.pdf>

¹² Datasheet: https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf

¹³ Datasheet: https://www.parallax.com/sites/default/files/downloads/P8X32A-Propeller-Datasheet-v1.4.0_0.pdf

¹⁴ Product brief: <https://www.movidius.com/myriadx>

global communication paths. Another area to which these architectures are suited well is the upcoming artificial intelligence (AI) trend, which demands high parallel floating-point performance. Compared to traditional GPUs, many hardware parts can be stripped off to get highly efficient low-power AI co-processors.

Besides SIMD approaches, multiple instruction multiple data (MIMD) designs exist, being designed as co-processors and some as standalone architectures. In MIMD, each core is able to run its own task on its own data set, independent of other cores. A hybrid SIMD, MIMD approach is the Cell processor[12]. It contains one PowerPC general purpose processor and multiple (usually eight) synergetic processing elements (SPEs) acting as co-processors. These co-processors have dual issue pipelines, one for floating-point and one for non-floating-point operations. Each co-processor is implemented as SIMD processor with multiple execution units, while all SPEs are organized as MIMD processors.

Due to high processing power demands in the high-performance computing domain, many-core architectures are often designed to achieve high processing power through a high number of powerful and feature rich cores. However, with peak performance comes high power consumption, making these processors mostly suitable for servers or high-end PCs. Examples for these processors are AMD's EPYC¹⁵ processors with up to 32 cores or Intel Xeon Platinum 8xxx¹⁶ series with up to 28 cores. AMD uses tightly coupled clusters[13] with up to eight cores. The clusters have direct connections to other clusters, while Intel uses a 2D Mesh to connect all cores. Intel's Xeon Phi started out as co-processors cards and became a standalone architecture with the Knights Landing generation. In contrast to general purpose server processors, their implemented cores have a simpler architecture to allow combinations of up to 72 processor cores. However, with so many cores, inter-core communication and accesses to global memory become a major burden with standard bus protocols. This is demonstrated by Xeon Phi's use of multiple ring buses that were later replaced by a 2D mesh. These architectures are surpassing 100W of power consumption and are out of this work's scope, even though they nicely show the newest trends of many-core processors.

The concept of Mesh interconnects or Network-on-Chips (NoCs) in general is also used for lower performance architectures among the embedded domain. A lot of these processors target networking appliances, cloud computing, image and audio processing and many others. Specifically targeting network appliances are for example Cavium Networks' Octeon CN38XX¹⁷ which contain a maximum of 16 MIPS64 cores connected via bus. NXP's T4240¹⁸ contains twelve processing cores communicating through a not further specified point-to-point network called QorIQ. Moving towards chips with hundreds of cores, the interconnect structures mostly implement variants of a 2D mesh. Broadcom's XLP900¹⁹ can be configured with 640 MIPS cores and Kalray's MPPA Manycore²⁰ contains 1024 very long instruction word (VLIW) cores, both configured as clusters, interconnected by a 2D mesh. Adapteva's Epiphany V[14] also contains 1024 very small reduced instruction set architecture (RISC) cores, interconnected by a 2D mesh. These systems are in general powerful enough to run an operating system, and have even with that many cores a mid range power consumption of 5 to 50 Watt.

Moving further into the low power domain, Toshiba[15] presented a chip with 64 VLIW cores and many special purpose accelerators interconnected through a tree-based NoC. The HyperX hx3100 processor[16] contains 100 processing elements connected through a 2D mesh. Both systems consume

¹⁵ Product brief: <https://www.amd.com/de/products/epyc-server>

¹⁶ Product brief: <https://ark.intel.com/content/www/us/en/ark/products/120496/intel-xeon-platinum-8180-processor-38-5m-cache-2-50-ghz.html>

¹⁷ Product brief: https://www.cavium.com/pdfFiles/OcteonCN38XX_CN36XX_PB-Jan29-06-web-v1.pdf

¹⁸ Product brief: <https://www.nxp.com/docs/en/fact-sheet/T4240T4160FS.pdf>

¹⁹ Product brief: <https://www.broadcom.com/products/embedded-and-networking-processors/communications/xlp900/>

²⁰ http://www.kalrayinc.com/IMG/pdf/FLYER_MPPA_MANYCORE.pdf

around 1W of power, which makes them good representatives for a low-performance and low-power embedded many-core system.

Besides commercial processors and many other 2D-mesh-like architectures with slight variations, there also exist research architectures dealing with aspects and hardware constructs off the beaten track. Apple-CORE[17] uses UTLEON3[18] processor cores bundled in clusters of four cores and clusters are connected through a NoC. The peculiarity of the design lies in the ability of the processors fast hardware-based task switching mechanism to hide communication latencies. It has hardware units to organize software concurrency among the cores instead of leaving it to the operating system (OS). The Ne-XVP[8] architecture also utilizes the strengths of Apple-CORE, such as multi-threaded cores and a hardware task scheduler. The peculiarity here is that the scheduler synchronizes data to cores based on software defined checkpoints elaborated by means of the ACOTES programming model (see Section 2.3.4). Synchronization happens via configurable *cache-to-cache tunnels*. XGRID[19] uses a scalable 2D grid of simple, low performance RISC cores forming a distributed memory system. The interesting part is the interconnect network which is very similar to current FPGA routing resources. The application is transformed into a Kahn Process Network (KPN), which is then mapped to the processors. The interconnects are configured accordingly at compile time.

Another class of processors are coarse-grain reconfigurable architectures (CGRAs)[20] which is an intermediate between fully reconfigurable FPGAs and usual processors. This means that a CGRA has some more complex structures like processing elements whose interconnect network can be dynamically configured through so called contexts, realizing a series of operations on input data. To run an application on such a processor, it is transformed into a control data flow graph (CDFG) which is then translated to multiple contexts mapped to the CGRA. The CGRA needs reconfigured upon every context switch. The similarity of CGRAs with the concepts in this work are the synchronization of processors/processing elements through arrival of data and the adaption of the hardware to the application's data flow. However, the reconfiguration in this work doesn't happen during application runtime and data flow is extracted at a coarser level.

2.1.3 Soft-core multi-/many-cores

Besides all previously shown commercial processors and research projects with configurable processor count and interconnects there also exists the class of soft-core processors. These processors are often described in hardware description language (HDL) and they can be synthesized on an FPGAs. Those processors come in handy when off-the-shelf solutions do not provide the needs or the hardware environment is rapidly changing (during development) with different requirements to the processor. There already exists a broad variety of soft-core processors for different instruction sets and bit widths. Some are one-man projects, some are university research projects and others are soft-cores provided by FPGA manufacturers. The man-power behind the projects also usually reflects the eco-system's comprehensiveness such as presence of a debugger, compiler, documentation, system-builder or available peripherals. Since FPGAs have grown considerably, multi- and even many-core systems become realizable. At the moment, most soft-core SoC kits deliver very limited support for multiple cores out-of-the-box.

One of the most popular soft-cores is the MicroBlaze[21] with many ISA compatible clones. To support multi-cores the following components exist: A common global memory, a mutex peripheral and a FIFO based bidirectional inter-core communication peripheral for distributed-memory systems called Mailbox²¹, formerly known as fast simplex link (FSL). The PicoBlaze, MicroBlaze's smaller 8 bit sibling, has no multi-core support provided. Intel (former Altera) has the NiosII, a 32 bit RISC soft-core. The NiosII uses the Qsys interconnect²², a dedicated N-to-M master-slave interconnect, for inter-core communica-

²¹ Product brief: <https://www.xilinx.com/products/intellectual-property/mailbox.html>

²² Datasheet: https://www.intel.co.jp/content/dam/altera-www/global/ja_JP/pdfs/literature/hb/qts/qsys_interconnect.pdf

tion. Access to slave components, such as peripherals, is shared by all masters and can be exclusively locked using integrated mutexes. Furthermore, communication is possible via a shared memory located in the `.data` section²³. Lattice’s LatticeMicro32²⁴ is not advertised to have dedicated multi-core peripherals or mechanisms. The only option are third-party peripherals connected to the integrated Wishbone bus. Cobham Gaisler’s (formerly Aeroflex Gaisler) Leon soft-cores can be configured as a multi-processor system²⁵. Multiple processors share a common memory and peripheral bus. Inter-core communication thus happens via the common memory.

The SpartanMC SoC kit[22] offers different variants of inter-core communication for its soft-core. Communication is possible via 1-to-1, 1-to-N and N-to-1 peripherals. Each either as FIFO-based or DMA-like variant. Additionally, shared data and/or program memory is possible.

In conclusion, soft-core vendors mostly deliver quite limited multi-core functionality. However, the user can always use one of the aforementioned cores together with a custom third-party (open-source) interconnect and trust in the interconnects compatibility to future processor releases.

2.1.4 Conclusion

As shown in Section 2.1, many multi-core systems still rely on a common central bus architecture nowadays. With additional cores, these systems are running into the memory wall [3] and thus cannot fully utilize available processing power. This is one reason why many-core designs moved towards 2D-mesh-like core-interconnects, often with distributed memory. This step shows the need for localized communication in combination with lower communication interference compared to classical bus structures. Generic 2D mesh structures may work well for processors that must have the ability to execute arbitrary applications. However, in the embedded domain, a processor might run a specific application for years. Traditionally, this application is written to leverage a target multi-core processor as much as possible. Choosing an off-the-shelf processor might limit future extensibility of the software and potentially require to target the application to a new processor. It would be promising to adapt the processor and the communication infrastructure to the application and not vice versa. FPGAs in combination with soft-core processors provide the freedom to generate an arbitrary number of cores and communication infrastructure ideally suited to the application’s communication pattern, emphasizing communication locality. Additionally, FPGAs allow continuous adaption to changing requirements. However, the price to pay for the reconfigurability is, in contrast to an application-specific integrated circuit (ASIC), a lower achievable clock speed, a lower energy efficiency [23] and a higher per unit price.

Since the full reconfigurability provided by FPGAs is not needed, one could settle for a less reconfigurable platform in the future to regain clock frequency and energy efficiency. Thus, it is thinkable to use a platform such as the proposed XGRID[19]. The cores and peripherals are fixed hardware and the interconnect network can form arbitrary point-to-point connections configured at compile time.

The proposed concept of this work could also be applied to hard processors, even though the benefit of hardware adaptability is lost. The software challenge would shift from generating the required hardware, to efficiently mapping the software to a given hardware, which is actively researched in Daedalus[24] for example.

²³ Intel tutorial: https://www.intel.com/content/dam/altera-www/global/en_US/pdfs/literature/tt/tt_nios2_multiprocessor_tutorial.pdf

²⁴ Product brief: <http://www.latticesemi.com/en/Products/DesignSoftwareAndIP/IntellectualProperty/IPCore/IPCores02/LatticeMico32.aspx>

²⁵ Vendor design reference: <https://www.gaisler.com/doc/antn/GRLIB-AN-0005.pdf>

2.2 Extracting Parallelism from Applications: Design Choices

There already exists a huge amount of approaches for parallelizing applications leveraging multi- and many-core systems. The existing solutions differ in the programming paradigm which specifies the instruments that the user has at hand for parallelization as well as the partitioning level indicating the language constructs and task granularity. The tools also differ in the target hardware architecture which allows parallelization only for specific processors or requires a shared and/or distributed memory architecture specifically for inter-core communication.

2.2.1 Programming Paradigms

There are different programming paradigms to extract parallelism from applications. The approaches mainly differ in the effort required to express parallelism and the kinds of parallelism that can be modeled. The existing approaches can be summarized into the following categories and according examples are named in Section 2.3.

Domain specific languages (DSLs): A custom language for expressing parallelism. The language models parallelism implicitly through specific constructs. Some DSLs are very close to commonly known general purpose languages and only modify/add specific aspects and thus are counted as language extensions. A new compiler is always needed to translate the language. The benefit of a DSL is that parallelism can be very well modeled and a high degree of parallelization can often be achieved. However, the user has to learn how to use a new language. It also highly depends on the experience and skill of the developer how successful the parallelization will be. In general, the user effort to rewrite an existing application in a DSL can be relatively high.

Application programming interfaces (APIs)/libraries: An API or library offers functionality (often functions) that can be called from the user code. The API only declares usable functions and the implementation has to be provided by the compiler or a target specific library. The library in general already offers an implementation which runs on one or more target platforms. The advantage of an API or library is that an existing language is used, already known by the user. The user just has to familiarize himself with the additional interfaces. Nevertheless, the user needs to rewrite existing code to use the provided interfaces. Thus, the user effort to formulate the problem is moderate when assuming that the user already knows the base language.

Language extensions: Language extensions modify or add certain aspects to existing languages. This makes them easy to use for developers already familiar the language. Some language extensions are so intrusive that they look like new languages and it becomes difficult to distinguish between language extension and DSL.

Annotations: Existing code is extended with user annotations to indicate which and sometimes how source-code parts can be parallelized. The benefit of using annotations is, that the original code remains executable since annotations can be omitted by the compiler. Thus, usually no vast reformulation of the code is needed, which typically results in little user effort. However, the user needs to identify where annotations can be applied. Depending on the amount of available pragmas more or less experience and knowledge is required.

Automatic parallelization: Automatic parallelization requires no user intervention and parallelizes applications up to a user specified or maximum possible degree. Automatic parallelization is the ideal case from the user's point of view. However, achieving a good speedup automatically is not easy since beneficial parts for parallelization have to be identified and fitting parallelization pattern applied. Also, the target hardware architecture plays an important role when selecting source-code parts to parallelize.

2.2.2 Types of Parallelism

Computer programs offer different kinds of parallelism which can be extracted and used to speed up processing. Different levels of parallelism can also be combined to achieve higher speedups. However, it highly depends on the application and also the programming style if such parallelism is exploitable. Some levels of parallelism like instruction level parallelism (ILP) are implemented in hardware through pipelines in the processors. Other types of parallelism mostly rely on compilers or parallelizing compilers to extract parallelism and map the parallelism ideally to the target hardware. While fine-grained parallelism like ILP is already well understood and leveraged, extracting parallelism on a coarser level is still lacking behind the availability of multi- and many-core hardware developments with more and more cores. In the following, the different kinds of coarse level parallelisms leveraged by current parallelizers are described:

Task-level Parallelism is similar to thread-level parallelism (TLP) but on a coarser level. While TLP is used in the processor to overcome high I/O latencies, task-level parallelism tries to distribute bigger, independent parts of a program to all available processing cores. Typically, the task granularity are whole functions, but it can also be a few statements. Task-level parallelism can very efficiently be applied on embedded systems, since the often critical communication overhead is relatively low due to the limited capabilities of embedded systems.

Data-level Parallelism is often leveraged when parallelizing loops. In many cases, loops process fractions of large arrays in one iteration. Given no (or eliminable) loop carried dependencies this is a beneficial parallelization technique. Nevertheless, data-level parallelism can also be used beyond the borders of loops.

Pipeline Parallelism is very similar to the previously described task-level parallelism. Instead of searching independent program blocks executed in parallel, different succeeding program parts may have data dependencies. The different program parts are executed on different processing cores and the dependent data is passed from core to core in a pipeline fashion. This kind of parallelism can be very efficiently be applied to embedded applications, since they are often written in a pipeline oriented fashion: collect data, multiple processing steps, output data.

2.2.3 Partitioning Level

Besides different methods how applications can be parallelized, there is also the aspect which constructs can be parallelized. The following partitioning levels are commonly used:

Instruction: A single instruction or a sequence of instructions can be selected for parallelization. Parallelization on groups of instructions allows parallelizing very small parts of the code. The keeping the inter-core communication and task creation overhead low is essential for parallelization. Parallelization is not beneficial, if the communication takes longer than the direct calculation.

Loop: Many parallelizers focus on parallelization of loops. Loops often consume the majority of computing time in some applications. Thus, it is consequent to target them for parallelization. Some tools specialize in targeting the subset of static affine nested loop programs (SANLPs) which were found to be highly parallelizable through mapping to KPNs. SANLPs are nested loops where loop conditions, boundaries and the variable index are affine functions with iterator as argument [25]. However, most applications contain more than just a loop and other parts might also require parallelization.

Function: Well written programs consist of different functions containing instruction bundles as functionally associated parts. These functions have clearly specified input and output data (neglecting

global variables). These functions often consist of big enough instruction bundles to be complex and compute intense enough to justify parallelization overhead through offloading to a new thread.

2.2.4 Memory Architecture

All parallelization tools require specific memory architectures and means for inter-core communication. The most common memory model is a global **shared memory**, to which each core has full access. A shared memory makes inter-core communication very easy since the communication data is only written to a specific memory location. The main problem of these systems is the high memory bandwidth demands which often can barely be fulfilled with many-core processors. Multiple cache levels are commonly used as countermeasure, at the cost of increased design complexity.

The complete opposite of the shared-memory model is **distributed memory**, giving each processor its own memory. In this case, inter-core communication becomes more complex and has to be realized through specific communication hardware.

A compromise between both approaches is the **shared-distributed memory**, where each processor has a local data- and instruction-memory as well as a shared-data memory that all processors can access. The shared memory can either be realized as one global memory or as several shared memories distributed to the processors address ranges, but accessible by all others. The latter is also called nonuniform memory access (NUMA). In a NUMA architecture, access time to the memory differs from local to distributed shared memory.

2.2.5 Task Scheduling

The distribution of tasks to processors can either be done dynamically at runtime or statically at compile-time. The advantage of dynamic scheduling is good distribution of the tasks to the available processors if the task's execution time is not known during compilation. The disadvantage of this method is the necessity of either an OS for task scheduling or a custom implementation with similar minimal functionality. Particularly for low-performance embedded devices running an additional task scheduling layer can be a big burden and further reduces the low processing-power. The tasks can be statically scheduled at compile time if they are known before running the application. This method is better suited for low-performance environments.

2.2.6 Conclusion and Scope

The target platforms are low-power, low-performance embedded systems which might run the same application for years. The target applications are legacy software but could also be newly written sequential software. With these requirements, the perfect combination of programming paradigms, infrastructure functionality and target platform can be selected.

Automatic parallelization is desirable as programming paradigm, since it puts no burden on the user. Annotations come with slightly higher demands on the user but could achieve higher speedups with a little manual effort. Pipeline parallelism should be leveraged to extract parallelism out of the application. Firstly, because embedded applications are often structured in this way and secondly, because task- and data-level parallelism have already been extensively researched. A distributed or shared-distributed memory model should be used to also enable parallelization for many-core systems that likely run into the memory bottleneck. Communication overhead is an extremely critical factor for successful parallelization, especially in embedded distributed-memory systems. Communication and thread creation must not cost more time than the execution of the respective code part. Functions and loops typically exhibit enough complexity to justify offloading to a different thread. Since the target platforms

typically have a static environment with constant demands on the application, the adaptivity of dynamic task-scheduling is usually not required and also not desired due to the higher required processing power.

2.3 Parallelization Tools

Parallelization methods and tools are relevant topics for many decades already. Therefore, many approaches for parallelizing software have already been researched. Listing and describing all concepts can fill a book itself. Thus, the most prominent and relevant work has been selected. Table 2.1 shows all considered tools, categorizes them by programming paradigm, usable memory-architecture, target platform, input language and more. A rating of how well the concept is applicable to embedded environments is given for each entry. The tools functionalities and concepts are described in the following. At the end of each description, the user effort for porting an existing legacy application and whether the method can be applied to embedded environments is judged.

2.3.1 Tools out of Scope

This section describes (popular) parallelization tools which are not applicable to embedded applications. Mostly, those tools target HPC environments and take resources such as an OS or libraries for granted, which are not available for embedded systems. Other tools target GPUs which have architecturally not much in common with embedded environments. The effort of porting the necessary libraries or functionality is assumed to be very high and the performance on an embedded environment questionable. Also, C as input language should be targeted, since it is still the standard programming language in most embedded environments and most likely the language in which a legacy application is written in. Such tools are only briefly described in the following together with a reason why they are not applicable for embedded systems.

The most prominent tool in this section is CUDA[32], which offers an API for parallelizing C++-code on Nvidia GPUs. Thus, the API is widely GPU architecture optimized and solely controlled by Nvidia. Even though some embedded platforms contain GPUs, this is not a widespread characteristic. Brook[27] from the year 2003 can be seen as a predecessor of CUDA. Brook itself is just a C language extension for describing streaming programs. BrookGPU and its successor Brook+ are implementations for generic GPUs using OpenCL or OpenMP. Just like CUDA they were designed for GPUs running in an x86 environment and thus are unsuitable for embedded environments. Another interesting tool is Open Hybrid Multicore Parallel Programming (OpenHMPP) which is a parallelizing compiler integrating GPUs and accelerators in general. The parallelization is indicated through annotations in the source-code. The requirements of OpenHMPP are a Unix like OS with a Pthreads library. On top of that runs the OpenHMPP runtime library to schedule and launch code snippets to be accelerated. The snippets are then either launched on the accelerator or on the host processor as a new thread, if the accelerator is not present. The different abstraction layers for managing threads are mostly too much for embedded environments. Additionally, OpenHMPP mainly focuses on accelerator integration and neglects acceleration in more homogeneous multi-core environments. No performance numbers for purely homogeneous environments are given.

Moving from GPU accelerators to purely CPU centric solutions, there is Intel Threading Building Blocks (TBB)²⁶. TBB is a C++ template library also providing additional abstraction layers for task scheduling, memory allocation and synchronization. The requirements of running TBB is a common desktop OS and an x86 compatible CPU which excludes most embedded environments. Task parallel library (TPL)[70] is a library for the .NET framework which initially was only available for the Windows OS, but also recently became available for Linux and Android and thereby for embedded environments. However, it

²⁶ https://raw.githubusercontent.com/01org/tbb/tbb_2018/doc/Release_Notes.txt

Table 2.1: Reviewed parallelization tools

Tool	prog. paradigm	partitioning level	mem.-arch.	target-arch.	task-scheduling	OS required	input language	embedded-appl.	references
Acotes	annot.	I,L,F	S,D	Cell B.E.,ISP2300,...	S	✗	C	++	[8]
AutoPar	auto.	L,F	S	any+OpenMP compiler	D	✗	C++	+	[26]
Brook	ext.	F	S&D	GPU	D	✓	brook	-	[27]
CellSs	annot.	F	S&D	Cell B.E.	D	✗	C	o	[28]
Cilk	ext.	(L), F	S	x86,SPARC	D	✓	cilk	o	[29, 30, 31]
CUDA	API	F	D	GPU (Nvidia)	D	✓	C(++)	--	[32]
Daedalus	auto.	L	D	PowerPC, MicroBlaze	S	✗	C	++	[24, 33, 34]
DMCG	annot.	L	D	x86+MPI	S	✓	C	o	[35]
EMB ²	lib.	F	S,D	any supporting MTAPI	D	✗	C(++)	+	[36]
Eldorado	auto.	I,L,F	S	ARM	S,D	✗	C	++	[37]
HMPP	annot.	F	S,D	HPC+HW Acc.	D	✓	C	--	[38]
Hypertool	annot.	F	S,D	SPARC+MPI	S	✓	C	-	[39]
Intel C++ Compiler	auto.	L	S	any+OpenMP compiler	D	✗	C(++)	o	[40]
TBB	lib.	L	S	x86	D	✓	C++	-	[41]
MAPS	auto.	I,F	S,D	TI OMAP,TCT,...	S	✗	C/CPN	++	[42, 43, 44]
MPA	annot.	I,L,F	S	ARM	S,D	✗	C	++	[45]
MTAPI	API	F	S,D	open	S,D	✗	open	++	[46]
OmpSS	annot.	I,L,F	S,D	x86,GPU (Nvidia)	D	✗	C(++),F	o	[47]
OpenCL	API	I,F	S&D	x86+GPU	D	✓	OpenCL C	+	[48]
OpenMP	annot.	I,L,F	S	any+OpenMP compiler	D	✗	C(++),F	+	[49]
OpenStream	annot.	I,F	S	x86	D	✗	C	+	[50, 51]
Par4All	auto.	L	S	OpenMP, Cuda, OpenCL	D	✗	C	+	[52]
Pluto+	auto.	L	S,D	any+OpenMP+MPI	D	✗	C(++),F	+	[53, 54, 55, 56]
parMERASA	lib.	I,F	S&D	parMERASA platform	D	✓	C	++	[57, 58]
Pthreads	API	F	S	any	D	✓	C	+	[59]
PIPS	annot.	F	S,D	x86+MPI	S	✗	C	o	[60, 61, 62]
PYRROS	DSL	F	D	nCUBE-2, iPSC 2	S	✗	DAG	-	[63]
SL	ext.	F	S	Microgrids	D	✗	SL	o	[64, 65, 66]
SUIF	auto.	L	S	ALPHA, x86	S	✗	C	o	[67, 68]
StreamIT	DSL	F	S&D	Cell B.E.,RAW	S	✗	StreamIT	++	[69]
TPL	lib.	I,L,F	S	x86+.NET	D	✓	C#	--	[70]
TFlux	annot.	I,F	S	x86,TFlux	D	✓	C	o	[71, 72]

Used abbreviations: partitioning level→ Instruction, Loop, Function; memory architecture→ Shared, Distributed, **S&D** shared-distributed; task scheduling→ Static, Dynamic; embedded applicable→ ++ designed and tested on embedded, + not designed for embedded but applicability shown, o not designed for embedded but could be applied, - not designed for embedded and embedded performance questionable, -- not reasonably applicable on embedded

was shown in [73] that the runtime environment easily consumes up to 47% of the total processing time for some benchmarks during dynamic scheduling on powerful x86 processors. Thus, the usability in an embedded environment is questionable.

What is also very popular nowadays and an often suggested method to speedup legacy code is high-level synthesis (HLS). Bailey[74] mentions that there are several flaws, often limiting automatic hardware generation for legacy code or the inferred hardware has bad performance. As stated in the user guide of today's most popular HLS tool: VivadoHLS²⁷, these are for example: pointer nesting or pointer casting,

²⁷ https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug902-vivado-high-level-synthesis.pdf

recursion, system calls like sleep/printf and dynamic memory usage. All these unsupported but not uncommon constructs make this approach often uninteresting, except if the user is willing to rewrite the code (if possible) to work around these constructs.

2.3.2 DSLs/Language Extensions

PYRROS[63] is an automatic parallelization tool for directed acyclic task graphs (DAGs) developed in 1992. As input, the user has to provide a textual description of a DAG which explicitly models tasks and inter-task communication. The language also has an interface for executing external C functions. Afterwards, PYRROS schedules the defined tasks. In the first step without processor bounds and in a second step tasks are clustered, ordered and statically mapped to a limited number of cores. As target platforms are nCUBE-2 and Intel iPSC/2 message-passing multi-core platforms are usable. To parallelize a legacy application, the user has to transfer the source-code to a DAG with the option to reuse C functions. Due to the use of message-passing and static scheduling this concept could be applied to nowadays embedded platforms even though designed for HPC in 1992. However, the used message passing interface (MPI) might be different from today's well-known standard, since the first draft of MPI was published later than PYRROS.

Cilk[29] is a C language extension developed originally at MIT in 1994. C++ support was later added and the tool commercialized under Cilk++ [29, 30, 31]. Afterwards, it was bought by Intel, integrated into the Intel C/C++ Compiler (ICC) and renamed to Cilk Plus. Cilk has keywords written ahead of function declarations to identify them as parallel procedures. Threads of this function are spawned with the spawn keyword ahead of the function call. After running the threads, the user has to explicitly declare barriers to collect the results from those threads. An interesting concept was introduced by being able to create successor threads with the spawn_next keyword which indicates the usage as a processing pipeline. After only removing the keywords, the application would be sequential again, which makes Cilk a minor language extension. Cilk++ added functionality to parallelize loops without loop-carried dependencies (a loop iteration doesn't depend on a previous iteration). *Reducer hyperobjects* were introduced to allow efficient collection of results with reducers. To run Cilk programs, the Cilk runtime library is necessary. This library launches threads, cares for communication and also implements a work-stealing scheduler to dynamically schedule tasks. Cilk was originally developed for shared-memory HPC systems like Connection Machine CM-5, but due to the low demands on the runtime library it could conceivably be applied in the embedded domain.

StreamIt[69] is a successor of Cilk, developed in 2001 at MIT as a DSL instead of a language extension. The focus of StreamIt, like the name says, lies more in parallelizing streaming applications and achieving speedups through generating processing pipelines, as they are already hinted in Cilk++. In StreamIt, a program consists of multiple data transformation functions called filters. All filters have input and output channels to connect to other filters. Multiple filters are grouped to form a processing pipeline and parallelization through splitting and joining data or feedback loops. The split and join mechanism can be configured to distribute data streams in a round-robin fashion one after another or to make a broadcast to all split filters. The filters are statically mapped each to single processors at compile-time and run on bare-metal without an OS or a runtime. Cell and the RAW[75], both shared-distributed memory systems, were used as target architectures. These processors are rather high-performance architectures if they are counted into the embedded domain at all. The main weakness of this approach is the possibility of deadlock, due to feedback loops. A deadlock detector exists but the user has to manually resolve them. The user can generate systems resulting in no speedup due to the feedback loops.

SL[64, 65, 66] is a C language extension created at University of Amsterdam in 2012. The language focuses on parallelizing whole functions as concurrent threads. The implemented constructs include

thread definitions similar to C function declarations. The shared data (so called communication channels) is also declared in the thread definition. Each access to shared data has to be performed over SL getter- and setter-functions. New threads of the previously defined tasks can be launched via SL thread-creation functions. The thread-creation function has the means to define the number of created threads, how threads are distributed to hardware resources and how threads should behave upon resource exhaustion or sharing. As target architecture, a custom multi-core architecture called Microgrid is used. Implementations based on Pthreads also exist, thus targeting a variety of architectures. Plans for using UTLEON3 as target architecture also exist. The implemented architectures point towards a usability in high to mid performance embedded environments. However, example SL applications require many language extensions, making it look more like a DSL with the implied effort for porting legacy applications.

2.3.3 APIs/Libraries

POSIX Threads (Pthreads)[59] is one of the earliest and up to today one of the most widely used APIs for multi threading C-programs. Implementations exist for nearly all desktop and many embedded operating systems and many other parallelizers make use of this library internally. The parallelization works by referencing single C-functions in the task API which will then launch the function in an extra thread. The API however covers thread management at a very low level. The user creates and joins threads through calls of according API functions. The user has interfaces for creating semaphores, mutexes and synchronization mechanisms with barriers and read/write locks. This means that the user manually takes care of data communication. The fact that an OS is required for dynamical task scheduling limits the applicability to mid- to high-performance embedded environments. The major drawbacks of Pthreads are the high user effort to manage threads on a relatively low level and the high performance demands.

OpenCL[48] is a popular C API initially developed in 2009 to make GPU computing power available to non graphical applications. The API was later also used for hardware accelerators. The nature of the API however mostly considers one central processor running the main program and then offloading application parts via OpenCL to accelerators or co-processors. The accelerator-kernel creation is similar to thread creation in pthreads, but the communication is rather complicated. This is due to different memories (host, local, private constant, global) which have to be differentiated and addressed. Special focus on efficient host and accelerator memory transfers has been laid to achieve good speedups resulting in many specific API functions. The API has already been successfully used in embedded systems [76], however mostly in combination with embedded GPUs and not in a homogeneous multi-core environment. Also, implementations targeting FPGAs have been presented [77].

Multicore Task Management API (MTAPI)[78, 46] is an API developed by the Multicore Association in 2013. The main target of MTAPI was to provide task management (runtime scheduling or task mapping) and synchronization on bare-metal as well as support for homo- and heterogeneous multi-core architectures. At the same time, it aims to focus more on embedded needs, compared to current APIs like TBB, Pthreads or Cilk. MTAPI was designed in conjunction with Multicore Resource Management API (MRAPI)²⁸ and Multicore Communications API (MCAPI)²⁹-APIs for embedded resource management and communication. Task creation and launching is very similar to Pthreads. Compared to Pthreads, a task can be implemented through multiple environments (for example one software implementation and one via a hardware accelerator). All task implementations are announced to the runtime library, which chooses where the threads will be launched. However, the scheduling can also be chosen through the implementation with different

²⁸ <https://www.multicore-association.org/workgroup/mrapi.php>

²⁹ <https://www.multicore-association.org/workgroup/mcapi.php>

schedulers or even static mapping. In conjunction with MRAPI and MCAPI, a consistent and embedded adapted environment is presented, which however lacks in broadly available bare-metal implementations. Implementations exist for example for the Freescale QorIQ family or NVIDIA Tegra K1, both shared-memory high performance embedded multi-cores. The API has to be ported to each new architecture with respect to the available hardware. Considering user-friendliness, MTAPI suffers just like Pthreads from a relatively low level task management interface with a moderate to high user effort for porting existing applications.

Embedded Multicore Building Blocks (EMB²)[36] is an open-source library that builds on top of MTAPI. The library provides parallel implementations of loops, reduction and sorting. The user gives the EMB² interface a data range to operate on and a function reference containing the desired loop iteration, or reduction method. EMB² allows parallelization, given that the library provides the needed building blocks. To the user, this comes at the cost of substituting function calls or code parts with calls to the existing library. However, vast code restructuring to fit the library is likely. The benefit of using MTAPI as interface makes programs easily portable to new architectures, given MTAPI support.

parMERASA[57, 58] was developed in an EC project in 2016. The focus of this project was a timing-analyzable parallelizer for hard real-time, embedded systems. The focus lay on parallelization and giving timing grantees in combination with a timing predictable multi-core processor. parMERASA defines four different design patterns for different kinds of parallelism: Data parallelism, pipeline parallelism, task parallelism, and periodic task parallelism. The design patterns are abstract descriptions of best practice solutions to parallelize the provided kind of parallelism. The user has to restructure the legacy C-program to fit the design patterns. Afterwards, the patterns are recognized and the whole application is transferred into an Activity and Pattern Diagram (APD). In the APD, parallelism is extracted and in a second step mapped to the target hardware. The tool also allows the construction of processing pipelines through processors as well as concurrent parallelism. At the end, the tool provides parallel C-code generated through algorithmic skeletons, for each design pattern. The algorithmic skeletons thus have to be adjusted for each target architecture. However, since the algorithmic skeletons are well-defined and understood, a worst-case execution time (WCET) analysis can be applied. Concerning user effort, the software has to be (re)structured to match the design patterns in order to enable parallelization. The authors already claim that this process should be carried out by a software developer who knows the application well. The actual effort depends on the supported design patterns and how much the code differs from the patterns. The achieved speedup of the application is moderate, considered with respect to the WCET, which mostly does not represent the common case. Also, the main drawback of the target architecture (also called parMERASA) is the (partly) common data memory which might also be the main cause of high WCETs. Applying this approach without the aspect of hard real-time systems might yield significantly better parallelization speedups.

2.3.4 Annotations

OpenMP[49] is an annotation based parallelization approach started in 1997. It is inspired by parallelization features of Cilk. Until today, it is still under active development and thus extended over the years. Today, OpenMP is a widespread standard integrated in most compilers for multi-core systems, but mainly for shared-memory systems. It can also be used for distributed-memory systems [79, 80] where often MPI is used for inter-core communication. Also, recent efforts aim to support embedded systems [81, 82]. The concept of OpenMP is to write annotations before source-code parts to be parallelized. Different annotations can be made on statements, functions and loops. The variety of annotations makes OpenMP widely applicable. However, the entry barrier is high due to the annotation possibilities. Another drawback is the necessity of manual data handling and

synchronization. The user must state which data should be shared or kept private in the parallel regions. Synchronization is necessary to handle race conditions or the execution order. To sum up, OpenMP is a widely used parallelization standard which was intended to be simple due to annotations. However, its development over the years made it the ultimate but also complex parallelization tool. The applicability to embedded systems was mainly explored with shared-memory systems. However, the achieved speedup for a image registration algorithm [82] and a beam-forming algorithm [81] showed a significant speedup degradation for parallelizations with more than 5 cores. Huang et al. [81] blames the breakdown on the lower memory performance and thereby high synchronization overhead compared to desktop computers.

CellSs [28] has a similar concept to OpenMP and was especially designed to run on the Cell Broadband Engine (Cell B.E.). In the year 2006, where OpenMP mainly addressed parallelizing loops, CellSs provided an alternative by parallelizing whole functions. Functions to be parallelized are manually annotated together with the in- and output variables. The concept is that the main function runs on the PowerPC processor element (PPE) and annotated tasks are transferred into a task graph and mapped and executed on the SPEs through the underlying runtime. The availability of only one kind of pragma makes the concept very easily applicable. Only the manual search for in- and output variables is cumbersome and error prone. It is also questionable if Cell B.E. still falls into the domain of embedded. The focus on a relatively exotic processor architecture also avoids a broad applicability of the concept. CellSs was continued by CellSs2[83] for SMP high performance desktop environments. Later on, the concept was renamed to StarSs with CellSs or GPUSs being specific architecture implementations.

Directive-Based MPI Code Generator (DMCG)[35] is a tool developed in 2008 for parallelizing MPI based systems. The authors justify their tool since most other tools (like for example OpenMP) only focused on shared-memory systems. Through the usage of MPI, distributed-memory systems are mainly targeted. DMCG expects keyword comments before loops without loop-carried dependencies. The loop is then partitioned and statically distributed to all processing cores via MPI. The necessary data dependencies are in contrast to others automatically recognized and an according communication interface is added to the generated code. The code can then be compiled to any MPI based system. From a user perspective, only one comment has to be added, which is easy. However, the user has to previously analyze if the loop has no loop-carried dependencies. This might not always be easy and limits the applicability to fewer problems. DMCG can be applied very well to low performance embedded systems (even though designed for HPC clusters), since only a lightweight MPI interface is required.

Data driven multithreading C pre-processor (DDMCP)/TFlux[71, 72] was developed as an alternative to OpenMP in 2008. The idea is to create a multi-threaded C pre-processor for handling code annotations. Based on the code annotations, parallel C-code is generated, which requires the according *DDM-CMP* runtime system and an OS. The user has to identify independent code blocks in the sequential C-program to be grouped. Inside these groups, the independent code parts are annotated as concurrent threads. Each block and thread needs a list of variables received and sent before and after code part execution. Threads are then launched whenever input-data (variables) are available to the thread. Looking at simple example programs, the necessary annotations tripled the size of the code. This represents the complexity and vast restructuring of the code required by the user. Additionally, the user needs to perform a dependency analysis for all variables. Compared to OpenMP, synchronization is done automatically. The applicability to embedded is restricted to high performance embedded, due to OS support. The concept was also only tested for x86, Cell B.E. processors and their own TFlux architecture.

MPA[45] is another alternative to OpenMP developed in 2009. Compared to OpenMP, MPA cares for data synchronization automatically. The user writes function and loop references into a separate

file, grouping the references by threads to execute them. Loops can also be partitioned and spread over multiple threads. The different threads are either dynamically scheduled with RTlib through the OS's Pthreads lib or statically mapped to processor cores without an OS. The tool also integrates a high-level simulator which can be used to evaluate parallelization approaches. From user perspective, the tool is easy to handle with a small set of annotations. The only struggle is to distribute the workloads equally to the available threads and deciding how many threads to be used. However, with more threads, an equal distribution can become difficult to be done manually. The tool was also proven to work on an embedded ARM environment.

Acotes[8] is another project developed around 2008 that identified the lack of OpenMP to automatically synchronize threads and parallelize data-stream oriented applications. Acotes was also influenced by the easy task synchronization concepts of CellSs, however focusing more on data-streaming concepts like StreamIT. The user annotates the C-program with task groups building an implicit barrier and launching tasks and task pragmas holding program parts to execute. The task group ideally organizes its contained tasks in a processing pipeline. Task sub-commands also allow the creation of task teams for concurrent task execution. However, in- and output data has to be explicitly specified for tasks and task groups. The user also needs to specify in which manner data should be distributed to task teams (array partitioning or everybody gets all). Compared to OpenMP, possible pragma annotations were reduced, making it more user-friendly. The user still needs to resolve data dependencies manually and the few available pragmas in turn have many sub-commands, making it again more complex. On source-code level, the user annotations are first translated to calls to a custom task management library. Afterwards, library implementations allow static task mapping and synchronization to different target architectures through a GNU Compiler Collection (GCC) plugin. Due to its slim interface requirements, Acotes is well suitable for embedded environments and has also been designed to run on embedded platforms like HiveFlex ISP2300 and others.

OpenStream[50, 51] evolved out of Acotes in 2013 and integrated data-stream processing concepts of Acotes as extensions into OpenMP. The user appends the in- and output variables to the OpenMP task pragma. The variables can further be annotated as data-streams with a fixed window size. This allows the tasks to operate on data windows without the need to store the full data. As a nice side effect, the variable annotations made OpenMP's user annotated task synchronization obsolete for data-streaming applications. Compared to OpenMP, the user-friendliness is increased for this application domain. The annotated application is then processed in the same way as OpenMP. The authors showed a working example with GCC on an x86 desktop computer. The authors also claim that the proposed programming model of creating a processing pipeline makes it compared to OpenMP better applicable to distributed-memory and embedded systems. They claim that the smaller communication overhead can be well implemented through small buffers. This allows a direct core communication instead of going through a global memory. However, this claim was never proven in the evaluation.

OmpSs[47] is a project that tried to integrate features of StarSs (former CellSs) into OpenMP. Namely, these are heterogeneous environments and the thread-pool model. The first aspect extends a target keyword to the task pragma allowing the execution on FPGAs or GPUs. Instead of fork-join, the new thread-pool concept takes the burden of thread scheduling from the user. Threads in the pool are thus triggered through a fulfilled data dependency and a scheduler chooses among all fitting threads. The user-friendliness is thus increased through eliminating manual synchronization. Also, the field of heterogeneous architectures is opened up to OpenMP through this approach. However, the focus of OpenMP to shared-memory architectures is rather enforced through the thread-pool approach requiring a central element with strong communication to all threads from there. Through the use of a runtime-library designed for desktop and HPC systems, this approach is only suitable for more powerful embedded systems.

2.3.5 Automatic

Stanford University Intermediate Format (SUIF)[67, 68] is maybe the first Fortran/C automatic loop parallelizer, initiated in 1994. SUIF consists of several (loop transformation) passes that increase data locality and analyze data dependencies, which is useful for parallelization. On the one hand, SUIF is able to recognize performed reduction pattern on arrays like sum, product, minimum and maximum calculation and replaces them with parallel representations. On the other hand, parallelizable loops are identified based on array-region access patterns. As target platform, shared-memory platforms like x86 and ALPHA are targeted. The compiler is able to either directly transform the program to assembler of the target architecture or convert it back to source-code with instrumented calls to a (not further specified) thread management library. The parallelization is very easily applicable as a user, even though evaluation in [67] shows applicability at only half of the tested benchmark programs. SUIF was developed for shared-memory desktop systems and in general this concept could be applicable to embedded systems.

ICC³⁰ was equipped with automatic loop parallelization around 2011. ICC is able to analyze if loops can safely be parallelized and automatically parallelizes them. However, parallelization is restricted to loops without loop carried dependencies, a fixed iteration number, without jumps in and out of the loop, without pointer aliases and without external function calls. ICC can also write a log, recording reasons why it failed parallelizing certain loops and which information could be provided for successful parallelization. The user can then annotate missing information to parallelize loops anyways. The compiler also categorizes loops based on a (not further specified) cost function that determines how beneficial parallelization would be. The user can then tune how aggressive or conservative the cost function should be applied. More loops are parallelized with higher chances of a slowdown or only loops definitely resulting in a speedup. Thus, the user can very easily parallelize some kinds of loops and with little user effort also loosen some restrictions. ICC is applicable on x86 desktop multi-cores and to the few available Intel embedded CPUs. However, these mostly target the high-performance embedded area. The Intel mid- and low-performance environments like Intel Galileo and Intel Edison are currently discontinued without successors.

Pluto(+)[54, 55]: Pluto (started 2008) and its tuned successor Pluto+[53] are affine nested loop source-to-source transformation tools. The key components of Pluto(+) are various C-source transformations of affine nested loops to allow better parallelization with OpenMP. The corresponding pragmas are automatically added after the transformation. There also exists a Pluto based tool [56] targeting distributed-memory systems through automatic insertion and synchronization with MPI. The results show good speedups that mostly outperform ICC [53]. From the user perspective, no additional effort is required for parallelization. However, affine nested loops represent only a subset of scientific problems and applicability is thus limited. Even though the tools are designed and tested for desktop and HPC environments, they can be applied to embedded through the usage of OpenMP and MPI.

Daedalus[24, 33, 34] is another tool collection for parallelizing static affine nested loops launched in 2008. Static affine nested loops in C-programs are first transferred to a process network. The generated process network is then mapped to the target architecture in a design-space exploration (DSE) process. DSE is done via high level simulation of different hardware combinations. The hardware properties are provided once for each model component. After the DSE, the custom MP-SoC can be synthesized on an FPGA. Even though the tool claims to be fully automatic, manual adaption effort is required [24] and the DSE process took a significant amount of time. However, the tool can be very well applied to even low performance embedded environments. It was proven to achieve good speed-ups [33] on MicroBlaze soft-core MP-SoCs and the concept could also be transferred to non-configurable architectures providing an appropriate description.

MAPS[42] is a widely automatic parallelizer developed at RWTH Aachen in 2008. MAPS requires a sequential C application and a target platform description. The target description holds information about execution cost of primitive operations such as multiplication and addition and the communication cost of the processors depending on the communication volume. As a first step, MAPS offers the performance profile generation of the sequential C application. This reveals hot spots and potential candidates for parallelization. MAPS suggests an application partitioning in the next step, based on the application profile and the source-code. The suggestion can be reviewed and revised in a Graphical User Interface (GUI). The evaluation showed that manual tuning is necessary to achieve the same speedups as through completely manual parallelization. As a last step, MAPS annotates tasks in source-code according to the chosen partitioning. In the initial publication, MAPS was only able to annotate so called TCT[84] tasks. In [43], MAPS is extended to also emit other programming models such as pthreads or TI OMAP specific code. This opens the approach to many more architectures. Furthermore, MAPS was extended to support heterogeneous environments [44], and the support to parallelize, schedule and map multiple applications running concurrently on one MP-SoC platform [43]. The implementation already showed good performance and speedups on different multi-core architectures in the embedded domain. Even though automatic parallelization is possible with MAPS, manual tuning is required to achieve the quality of manual parallelizations.

AutoPar[26] was developed in 2010 and it is very similar to Pluto. Compared to Pluto, AutoPar puts the focus not only on parallelizing static affine nested loops but also on functions through the OpenMP task directive. Besides this additional functionality, it is able to parallelize (only) object oriented high-level C++-code. Just like Pluto, AutoPar provides a source-to-source compilation with OpenMP pragmas and additional compiler passes to prepare the provided code for better parallelization. The tool is thus able to parallelize a broader range of applications at no user effort. AutoPar suffers restrictions on embedded environments inflicted through OpenMP. Additionally, C++ has slightly higher performance and memory requirements compared to pure C.

PIPS [60] is a C/Fortran source-to-source compiler already initiated in 1988. Until today, it is constantly extended and improved. Besides other source-code transformations, the strong points of PIPS are a good (inter-procedural) dependency analysis and an array region analysis, predicting the part of the program where an array element is to be read and written. These two methods are very handy during program parallelization. Building up on that, PIPS can create a data-dependency graph. The graph can then be analyzed through a resource aware list scheduling algorithm [62] which automatically maps the graph to the given hardware. The tool is capable of mapping statements as well as (partitioned) loops. The statement execution time is statically estimated in PIPS through an assembler transformation and a lookup table with the associated cycles per operation. PIPS is able to create OpenMP or MPI annotated source-code output. However, evaluation in [61](2016) shows that the generation of generic MPI code is in many cases not beneficial without better execution and communication cost models. To sum up, PIPS parallelizes applications just with the initial one-time effort of specifying architecture parameters. PIPS can be applied to embedded architectures due to the use of MPI and OpenMP, even though it has only been tested on desktop platforms. The use of static estimated execution time is a major flaw since this assumption doesn't hold for embedded systems with unpredictable external influences through for example peripherals.

Eldorado[37] is an automatic C parallelizer, especially developed for embedded systems in 2013. Eldorado internally uses the MPA tool for parallelization and data dependency resolving. Thus, Eldorado just cares for extracting a certain amount of tasks with an appropriate size and maps them to a given architecture with respect to communication and task creation overhead. These constraints are architecture specific and have to be manually fed to the parallelizer. Eldorado tries to find an optimal solution through integer linear programming approach. However, since it is NP-hard to solve the problem, especially for larger applications, Eldorado operates on a hierarchical

control-flow graph (CFG). This makes it possible to shrink the solution space through the levels of hierarchy but still allowing finer granularity if needed by descending in the hierarchy. From the user perspective, this tool is able to parallelize without any manual effort. Eldorado was especially developed for embedded platforms and tested with different ARM shared-memory embedded systems. However, this approach has some shortcomings. Firstly, for balancing tasks, an average execution time per statement is taken. It is puzzling how this method should work for a dynamic program flow. Secondly, Eldorado neglects peripheral interaction, which is essential for many embedded systems. Thirdly, only loops are subject to parallelization, but an embedded application consists of more than just loops.

2.3.6 Summary

Reviewing the approaches and tools brought up in the past decades, different characteristics and shortcomings become obvious:

- The effort of task creation and the communication overhead is often neglected. However, it is an important factor if the parallelized solution yields speedups or not. For embedded environments, this overhead becomes easily dominant due to the low performance compared to a HPC environment.
- Many tools try to extract as much parallelism as possible out of the application and map it to all available processor cores. Not seldom, this even results in a constant speedup with more resource usage or even a slowdown. This point is thus also related with neglecting architecture characteristics. For targeting embedded environments, specifying a desired/necessary speedup could be a solution for this problem.
- Many tools extract data- and task-parallelism. However, pipeline-parallelism is rarely extracted. Pipeline-parallelism increases throughput at the cost of latency. This is not usable in all applications but especially embedded environments with repetitive tasks are likely to benefit as long as no real time requirements exist or can still be met.
- Most solutions try to map tasks to a fixed hardware architecture. The flexibility of a configurable environment which adapts to the software as provided through soft-cores on FPGAs is barely leveraged.
- Many automatic parallelization solutions use rather inaccurate runtime approximations instead of measurements through profiling to judge complexity of software parts. Particularly embedded environments with peripheral interaction can barely be accurately approximated without user specified bounds.
- Most solutions target shared-memory architectures, since they are still dominant nowadays. Shared-distributed- or distributed-memory architectures however tend to scale better with more processors due to the bottleneck of shared memory. Also, shared-memory architectures yield in a high WCET due to the required but very pessimistic global memory access times [58]. Thus, multi-core real-time applications on shared-memory architectures are hard to verify or show low WCET speedups.
- Automatic parallelism extraction is often bound to data-parallelism in loops. Extracting loop-parallelism is very promising but for a wholesome parallelization also other parallelization possibilities should be considered.
- Many approaches target the HPC environment and use an OS with dynamic task-scheduling. These prerequisites can be taken as granted in the HPC domain, but in embedded environments dynamic

task-scheduling puts a big burden on the restricted processing power. Also, the flexibility of dynamic task-execution is not always needed for rather static and predictable embedded applications.

- All reviewed parallelization approaches are unaware of peripherals in multi-core environments. Problems like mapping tasks to specific processor cores with peripheral interaction or handling parallel access to one peripheral have to be tackled especially for embedded environments.

3 Target Platforms

As target platforms, soft-core SoCs are chosen to have a configurable hardware environment, adaptable to the needs of the parallelized software. As specific implementations SpartanMC and MicroBlaze are selected. SpartanMC is chosen since it is a project at our lab with good tool support and it can freely be changed to the requirements the software might have. MicroBlaze is chosen since it is a very popular soft-core offering multi-core and good toolkit support.

3.1 SpartanMC

SpartanMC[22] is a soft-core SoC with a data and instruction width of 18 bit. Modern FPGA's memory and arithmetic blocks are also 18 bit wide and thus the processor makes ideal use of the available resources.

SpartanMC's development was started at Professur Mikrorechner, TU Dresden and was later continued at Computer Systems Group, TU Darmstadt. The SpartanMC environment is completely open-source and downloadable from the SpartanMC website³¹.

The SoC is continuously improved, extended and already has a broad variety of peripherals and supported FPGAs. Besides the processor and peripherals, the SoC Kit also contains an adapted GCC, Binutils and GDB implementation as well as a graphical system builder and a simulator. Each SoC can thus be specifically tailored to match the application to be realized. The SoC's peripherals are attached via memory mapped IO. The SoC also supports a pseudo DMA variant. DMA peripherals don't have control over the full address range, but get assigned a fraction of the address-space. To be more accurate, the processor and the DMA peripheral share one dual-ported Block RAM (BRAM), which is mapped into the processors address-space on one port and the other port is controlled through the peripheral. This allows both parties the access the same memory region, however simultaneous access to the same address has to be resolved.

In SpartanMC multi-core environments, each core has its own code- and data-memory, while the latter has been extended in this work to be also partly shared (see Section 3.1.1.6). Peripherals are attached to a core via memory-mapped I/O and only the specific core can access the peripheral.

As interconnects, only 1-to-1 interconnects called Core-Connectors were available at the start of this work. Within the process of creating well performing core-interconnect, existing hard and software was tuned for highest possible performance and new specialized interconnects were designed.

3.1.1 Inter-Core Communication

For a simple 1-to-1 communication the Core-Connector can be used as a sending and receiving peripheral on two cores. It is a lightweight peripheral and can also be deployed for arbitrary core communication patterns. For a 1-to-N and N-to-1 communication a dispatcher and a concentrator peripheral is available. These peripherals make it easy to distribute data from one to many cores and collect data from multiple cores. The 1-to-N and N-to-1 communication pattern can also be realized with 1-to-1 interconnects, having the disadvantage of a higher hardware usage and more complex arbitration through software. Last but not least, a common global data-memory is available. Each core allocates a part of its available address range for global data. This makes core communication very easy, but with the disadvantage of handling concurrent global data accesses. The consistency of the data has to be guaranteed and simultaneous accesses might lead to processor stalls if the global memory is busy serving data to another core.

³¹ www.spartanmc.de

In the following, each communication paradigm is only briefly described since this is not the main scope of the thesis. The detailed hardware and software implementation as well as hardware usage and performance comparison can be found in [85] for the Core-Connector, Dispatcher and Concentrator and in [86] for the global memory.

3.1.1.1 Core-Connector

The Core-Connector consists of a FIFO-buffer for unidirectional communication. A simplified hardware schematic can be seen in Figure 3.1. The Core-Connector master, which is the sending module, has three registers for data exchange with the processor:

Status: Signals if the FIFO has enough free entries, as demanded by the "Message Size" register.

Message Size: Specifies the number of consecutive data words to be written into the FIFO.

Data: The data to be delivered to the receiving core.

The Core-Connector slave can read data from the FIFO. Similar to the Core-Connector master, it has three registers:

Status: Signals if the FIFO has enough used/data entries, as demanded by the "Message Size" register.

Message Size: Specifies the amount of consecutive data words that are desired to be read from the FIFO.

Data: Contains the FIFO's head element.

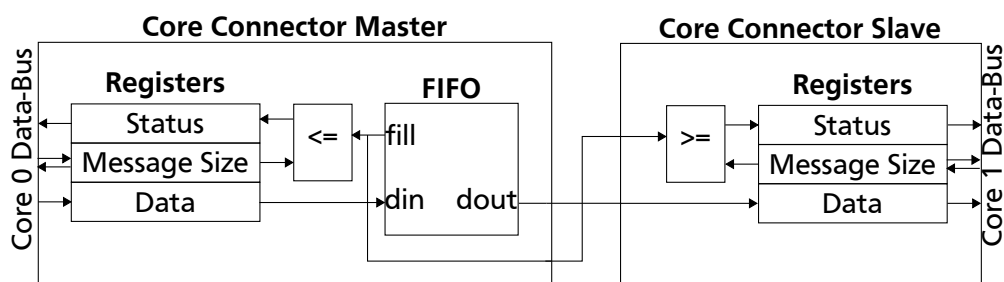


Figure 3.1: Core-Connector simplified schematic hardware design

The advantage of having a message-size register is that data can be transmitted in bursts with "Duffs Device"³² copy algorithm. Instead of checking the "Status" register for available data or free entries after each received or sent value, the available buffer space is checked once and multiple data is then read or written to/from the FIFO consecutively without intermediate checks. The data transfer is realized through multiple consecutive load and store word commands with increasing immediate offsets. The maximum burst-size is 16 for SpartanMC, since the immediate offset in SpartanMC's load and store command is four bit wide. For messages bigger than 16, the message is partitioned into multiple chunks, transmitting bursts of at most 16 words. This method allows a transmission rate of nearly two cycles per word [87], which is ideal for memory mapped IO peripherals. Jacob[85] shows that the ideal FIFO size is twice the maximum burst size for large messages, given that the receiver is idle waiting for messages. This allows the sender transmit a new burst while the previous burst is read. If the FIFO's fill state is unknown the sender has to wait until the FIFO is emptied for writing the next burst or write words one by one, which is very slow. Larger buffer sizes only avoid blocking of the sender, if the receiver is not ready.

³² <https://www.lysator.liu.se/c/duffs-device.html>

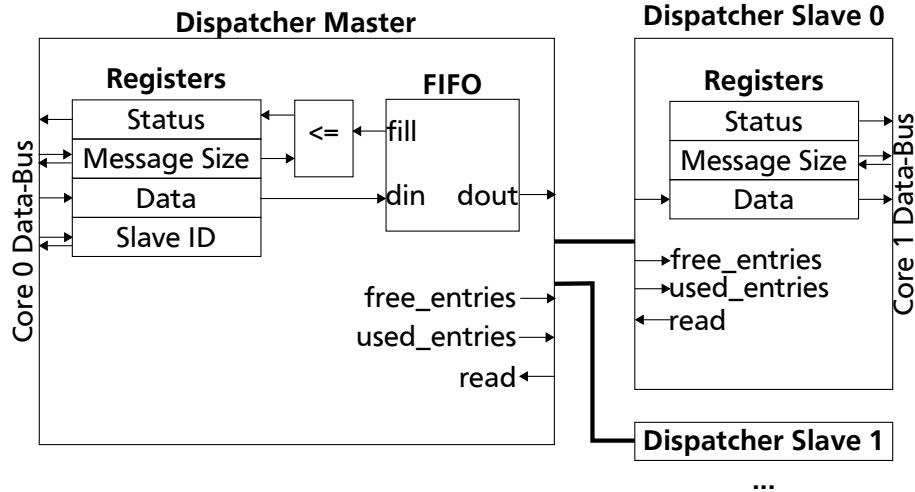


Figure 3.2: Dispatcher simplified schematic hardware design

The total size of the transmitted message cannot be read from the "Message Size" registers, and thus has to be a priori known or dynamically transmitted by a first informative transmission.

3.1.1.2 Dispatcher

The idea of the Dispatcher is that the subsystem containing the Dispatcher master generates work packages. The work packages are distributed to several connected subsystems. These receive work packages and process them in parallel. Thus, the Dispatcher generates a one-to-many topology. A simplified schematic is shown in Figure 3.2. The Dispatcher master interacts with the processor in almost the same fashion as the Core-Connector. The only exception is that the user specifies the receiver's slave-ID explicitly. Depending on the current slave-ID, only the selected slave sees the FIFO's used and free entries, while the master propagates an empty FIFO to all other slaves. Through this procedure, the Core-Connector slave can be reused for receiving.

3.1.1.3 Concentrator

The Concentrator is the inverse of the Dispatcher module. It collects processed work packages from several slaves, thus models a many-to-one connection. A simplified schematic is shown in Figure 3.3.

The Concentrator slave signals through the "Data Available" register that it wants to transmit data and waits for the "Status" register to signal that the master module is ready to listen. Afterwards, it can write its data through the "Data" register into the masters FIFO. Data sent to the master is preceded by a header, generated by the hardware, holding the message size and the slave ID to differentiate multiple messages in the FIFO from different slaves.

Arbitration can either happen through selecting the slave-ID via the "SW Arbiter" register or through a hardware round-robin arbiter. After a slave is arbitrated, receiving the message proceeds in the same way as with the Dispatcher or Core-Connector. The "Peek Data" register preserves the FIFO head entry e.g. the message size if it needs to be read several times like in non-blocking receives.

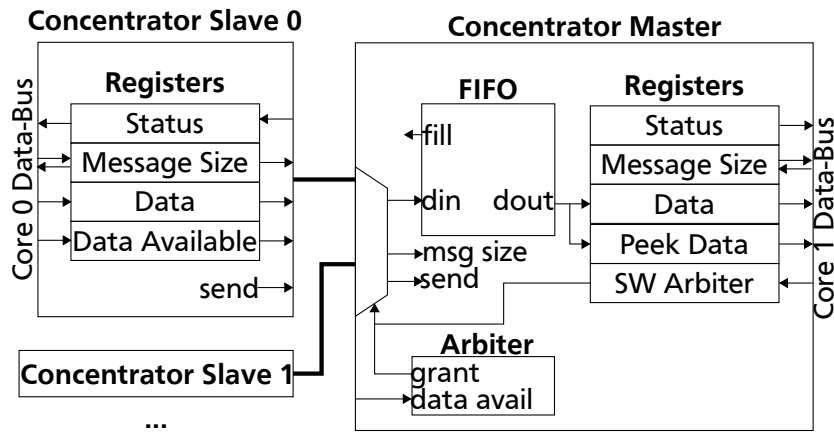


Figure 3.3: Concentrator simplified schematic hardware design

3.1.1.4 MemSwap Dual

The MemSwap Dual variant is an alternative of the FIFO-based Core-Connector to eliminate two disadvantages of the latter:

1. The maximum transmission speed is limited to two cycles per word.
2. The processor is used for data transmission to copy from local memory to the peripheral register.

The DMA based MemSwap module eliminates these disadvantages through a DMA memory section between the core and the peripheral. The MemSwap module, shown in Figure 3.4, holds two BRAMs of arbitrary size. Each BRAM is either assigned to the master or the slave module and should have enough capacity to store the largest possible message. The data to be transmitted is directly written into the peripherals DMA region. For data transmission, the assignment of the BRAM is simply swapped, which allows duplex communication. For two communication partners this can be done efficiently with the dual ported BRAM primitives. Each BRAM has two equivalent access ports (A&B) with an enabling port which lets the BRAM ignore inputs and set outputs to zero. Outputs of both BRAMs are combined with or-gates and resemble the DMA-Bus. Switching between BRAMs is initiated through the status register. If the master and slave module both set the status register, the enabling port is inverted on all BRAM ports, and thus the other BRAM becomes available through the DMA-Bus. A bit in the status register indicates to the processor that the swap was performed and new data is available.

This technique avoids memory transfers between the processors main-memory and registers. The transmission or respectively the switching between memories is a one cycle operation. Also, the processor is able to perform other tasks while waiting for the memory swap. Nevertheless, it should be noted that communication with a third partner requires memory copy operations through the processor, since SpartanMC DMA peripherals don't control the full address space.

3.1.1.5 MemSwap Multi

MemSwap Multi is an extension to the MemSwap Dual module, to support dispatcher and concentrator features. Since the Dispatcher and Concentrator module have the same drawbacks as the Core-Connector, this module eliminates those. A simplified hardware model is shown in Figure 3.5.

Compared to the MemSwap Dual module, the internal memory configuration is very similar. Each additional connection to the master module results in an additional BRAM pair, functioning the same way as the two BRAMs in the MemSwap Dual module. For the master module, an additional arbiter is used to switch between memory pairs and thereby control the current communication partner. The arbitration

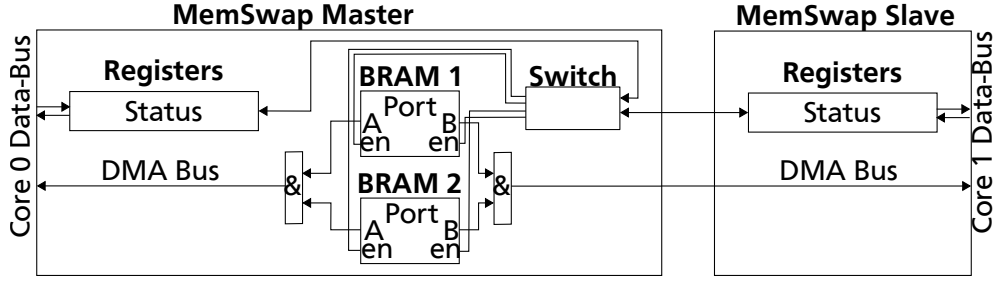


Figure 3.4: MemSwap Dual simplified schematic hardware design

can either be configured as hardware round-robin arbitration, switching partners after successful memory swap or explicitly via software through the status register. Through the bidirectional communication, this module can be used as dispatcher and concentrator.

This module has, compared to the MemSwap Dual module, an even higher demand for BRAMs. Each communication partner requires two BRAM pairs, while the master module can only write into one BRAM pair at a time. Alternatively, one could reduce the amount of BRAMs to only one BRAM per slave and one for the master module, as shown in Figure 3.6. However, this would require a crossbar switch for all slaves, since each slave needs access to any BRAM that the master can write to. Crossbars are relatively complex to realize and thus have a high hardware cost on an FPGA. Thus, it was decided to dismiss this variant even though it would save BRAMs.

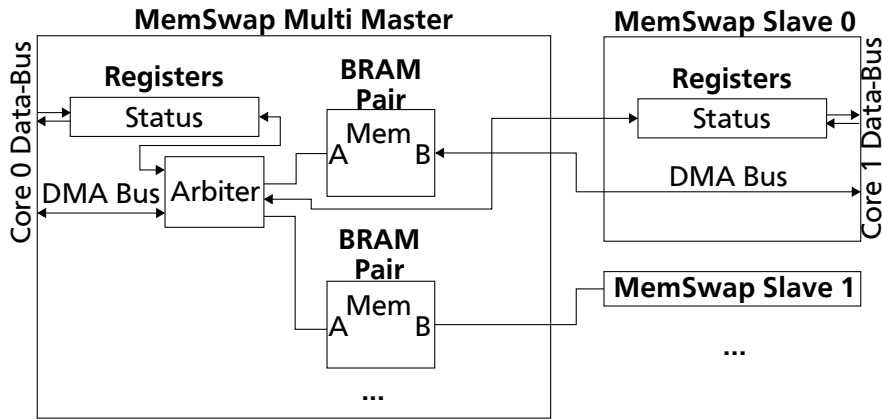


Figure 3.5: MemSwap Multi simplified schematic hardware design

3.1.1.6 Shared Memory

Herber[86] developed a shared-memory module. Just like DMA peripherals the shared memory uses a part of the processor's data section/address space. Typically, each processor also has local memory which holds the code-section and parts of the data-section. The shared memory is initializable. Each processor's memory-bus is connected to the shared memory, where an arbiter controls simultaneous accesses. Each core is optionally equipped with a one word read cache to overcome the potential BRAM bottleneck. The arbiter is implemented as a work conserving round-robin arbiter.

On the software side the processor was equipped with atomic instructions to allow the safe usage of mutexes and semaphores.

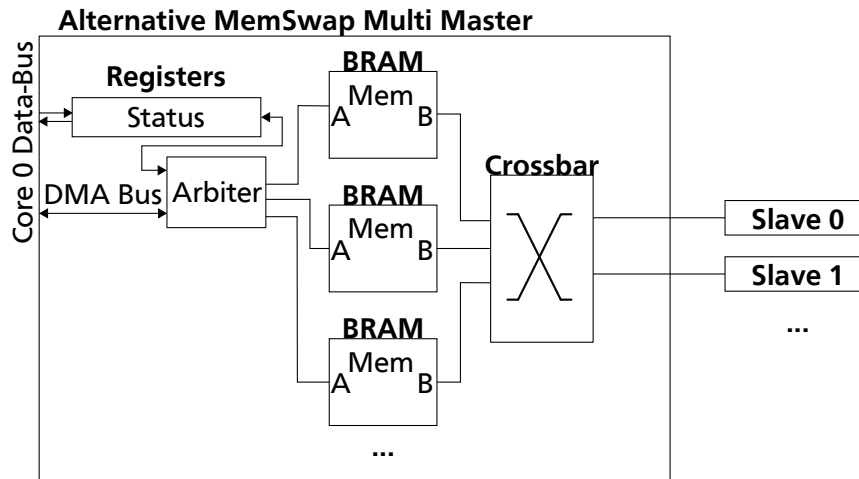


Figure 3.6: Alternative approach for MemSwap Multi with fewer BRAMs

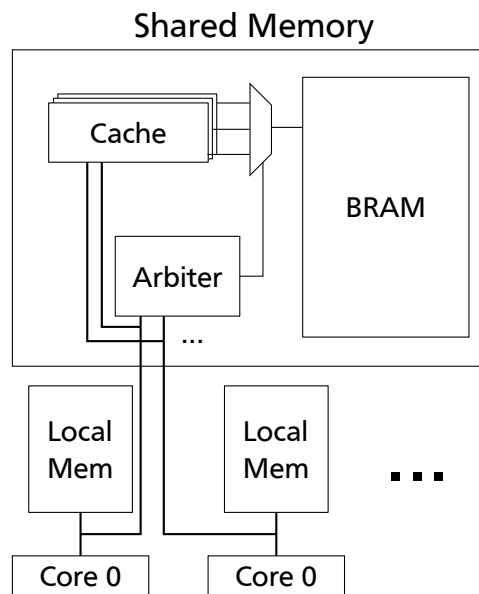


Figure 3.7: Shared Memory simplified schematic hardware design

3.1.2 Performance-Counter

When parallelizing applications, one needs to know the execution time of the different application parts and which parts of the application benefits from parallelization. An easy method is to use timers/counters on microcontrollers. This is not preferable: Firstly, especially on SpartanMC soft-core SoCs this changes the critical path and thereby the maximum frequency. Secondly, the timer control through the driver adds a variable time overhead through varying compiler optimizations which might distort the results. Performance-counters directly integrated into the processor core impose a better alternative. The implemented performance-counter [88] is custom configurable to keep hardware overhead low. To get cycle accurate measurements, the performance-counters can be started and stopped via dedicated processor instructions. Through calling the performance-counter via inline assembler, the added overhead is static and can be measured and thus automatically be corrected.

3.2 MicroBlaze

MicroBlaze is a 32 bit RISC soft-core SoC developed by Xilinx. It has a widely configurable architecture which allows it to set foot in different domains such as microcontrollers, real-time processors and application processors running Linux. The processor has for example configurable pipeline stages, caches, interrupts, hardware multiplier/divider, memory management unit, floating-point and others. Selecting from these options results in a different application performance and hardware resource usage. For this work, the MicroBlaze is used in "performance"-mode without additional options. The newest MicroBlaze version uses Xilinx Vivado as a system-builder. MicroBlaze uses the AXI-Interface for attaching extra components like peripherals to the processor. This abstraction makes it easy to add peripherals and keep the processor core at high clock rates. However, the common AXI bus also decreases the possible performance especially in terms of latency. The generated hardware system can be exported to an adapted Eclipse integrated development environment (IDE) to write the SoC's application. The IDE will use the GCC as cross-compiler and also has means to debug the application directly in the IDE.

Microblaze multi-core systems can out of the box only be configured as distributed-memory systems. A shared-memory module has also been developed, that can be accessed via AXI from different cores.

The MicroBlaze has also been integrated into SpartanMC's system-builder jConfig. The generation of a multi-core system is possible in jConfig and as well as in Vivado and jConfig also exports a preconfigured Vivado project "Block Design". Compared to Vivado, jConfig allows system generation on a higher abstraction level. Systems in Vivado are described as a schematic where different interfaces are connected with wires. This method is well suited for designing arbitrary circuits, but shows especially for multi core systems too much detail. Thus, the user might quickly loose the overview with that many cores, peripherals, interconnects and wires. jConfig is designed for multi-core SoC generation and systems are managed hierarchically in a tree structure. Details are only shown when collapsing a certain branch of the tree. Besides the better structure, jConfig is also able to import an abstract hardware description, automate most configuration and wiring.

3.2.1 Inter-Core Communication

Just like SpartanMC's core-connector, MicroBlaze has an off-the-shelf inter-core communication peripheral called Mailbox. To be able to share data among multiple processors, a global memory is designed via an AXI-BRAM-controller. For MicroBlaze, there is no direct equivalent to the SpartanMC Dispatcher and Concentrator peripherals. These communication infrastructures are modeled through multiple Mailboxes with software instead of hardware arbitration.

3.2.1.1 Mailbox

The MicroBlaze Mailbox is a FIFO-based communication peripheral. In contrast to the SpartanMC Core-Connector, the Mailbox can only be used as bidirectional communication infrastructure. As it can be seen in Figure 3.8, the Mailbox has AXI4 interfaces for two processors and each interface has its own FIFO-buffer with configurable depth. The Mailbox in AXI4-Lite mode has registers for reading and writing data. The Mailbox in AXI4-Stream mode interfaces the FIFO directly through special processor instructions and has a significantly higher throughput due to the stream interface. Thus, for the intended use case the AXI-Stream interface should be used.

The default Xilinx Mailbox drivers are optimized for comprehensiveness and robustness in terms of faulty user inputs. Since the target is to interface the Mailbox software through generated code and fast inter-core communication is a critical issue, throughput and latency should be favored over the drivers robustness and comprehensiveness. In this context a custom Mailbox driver was implemented to achieve a lower latency and a better throughput especially for large data.

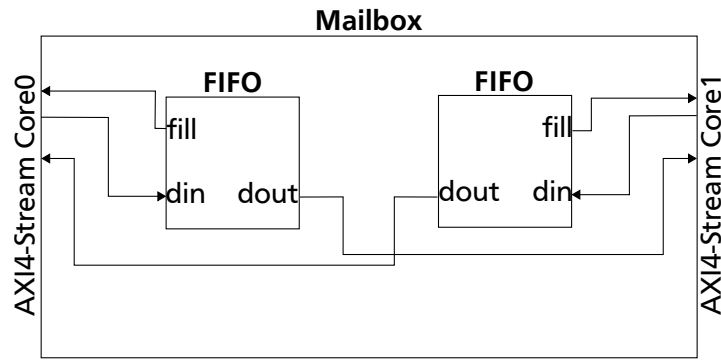


Figure 3.8: MicroBlaze Mailbox AXI-Stream simplified schematic hardware design

3.2.1.2 Shared Memory

Schladt[89] presents a shared-memory module for MicroBlaze. As shown in Figure 3.9, several processors can be attached to the module via AXI-Full interfaces attached to an AXI-Interconnect module. The memory is realized through BRAMs and interfaced through an AXI-BRAM-Controller. All elements of the shared memory are available in the Xilinx Vivado IP Integrator as building blocks that just need to be configured and wired. With increasing number of connected processors and requests, the shared memory can easily become a bottleneck. Thus, the memory accesses have to be implemented as efficiently as possible. Schladt[89] (Section 3.1.3) compares the different MicroBlaze peripheral interfaces: AXI-DP, AXI-DC and LMB in their applicability for a shared memory. The outcome is that the LMB bus is only connectable to one processor and must always respond in one clock cycle and thus cannot be used. The MicroBlaze is only capable of using data caches with the AXI-DC bus, which might relax the shared BRAM bottleneck depending on the access patterns. The throughput and latency for the AXI-DC bus supporting bursts is also better for a consecutive access pattern, however this can be different for other access patterns.

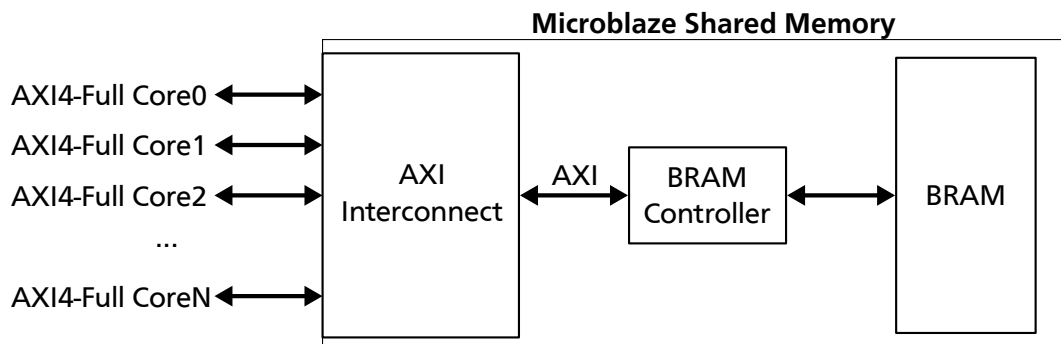


Figure 3.9: MicroBlaze shared memory simplified schematic hardware design

3.2.2 Timer - Performance Counter

The usage of MicroBlaze performance-counters has vastly been investigated by Schild[90]. MicroBlaze can be configured with the *extended Debug* option and an additional MicroBlaze Debug Module (MDM) allows the usage of event and performance-counters. This method works only for single-core systems since only one MDM is allowed for the whole system. Multiple processors can be attached to one MDM but only one processor can start and stop measurements. However, for an independent performance evaluation of multi-core systems each core would need to start and stop performance measurements independently, from within the application.

Therefore, a simple timer peripheral IP-core is used for performance evaluation, even though it is not an ideal choice. The timer can be started, stopped and read via an AXI-interface. The advantage of using a timer also resides in a significantly smaller hardware footprint.

3.3 Inter-Core Communication performance evaluation

Synthetic benchmarks are applied to get an idea of the different inter-core communication performance. With these results, an ideally fitting communication infrastructure can be chosen for a parallelized real world application and an estimate of the later overall system performance can be given.

3.3.1 1-to-1 Communication

To measure the performance of the SpartanMC Core-Connector, SpartanMC MemSwap Dual and MicroBlaze Mailbox, arrays of ascending size are transmitted. The next transmission is only started when the previous transfer completely finished (FIFO buffer is empty). The time measurement is done via general purpose input/output (GPIO) ports. Every time send/receive is started/finished, a GPIO port is set high/low. A logger module writes the current simulation time in a text file on transition change of an attached GPIO port. This method allows the measurement of the throughput as well as the latency from writing the first value to reading the first value, which would not be possible with a performance-counter since it is restricted to events on one processor. Figure 3.10 shows the send duration for different numbers of words. The measurement is started and stopped when the sending function is entered or left, respectively. The duration for sending and receiving are only marginally different for all peripherals and thus receiving is not shown here. The MemSwap module has a constant transmission time since only BRAM ports have to be switched and the 9 cycles are mainly calling the driver function. The core-connector has an initial effort for calling the driver of 45 cycles and it transmits new values each two cycles. After each burst of 16 words, a buffer space check is initiated which adds some overhead. If the buffer is large enough for the whole message, fewer checks have to be applied, which explains the gap at 32 words (the configured FIFO size). The Mailbox has an initial overhead of 76 cycles and each additional value takes on average 4.5 cycles to transmit. The Mailbox driver is tuned to be similar to the Core-Connector driver which results in a higher throughput. Concluding, the Mailbox performance is slightly worse than the Core-Connector. On the one hand this is caused by the AXI-Bus and on the other hand the lack of the Mailbox's FIFO fill count through the stream interface prohibits better performance. The fill count is required to achieve high throughput with the driver algorithm described in Section 3.1.1.1. Nevertheless, it should be noted that the Mailbox transmits 32 bit wide words, while SpartanMC is limited to 18 bit. This gives the Mailbox a transmission rate of 6.3 bit/cycle for transmitting 128 words, while the Core-Connector reaches 6 bit/cycle.

The latency of the different interconnect peripherals is:

Core-Connector: 34 cycles

Mailbox: 84 cycles

MemSwap: 1 cycle

3.3.2 1-to-N and N-to-1 Communication

Since MicroBlaze doesn't have dedicated communication peripherals for this communication type, an equivalent circuit is realized through multiple Mailboxes and software arbitration. Thus, the performance is equal to the Mailbox measurements in the previous section. As the Concentrator and the Dispatcher driver and hardware are very similar to the Core-Connector as well as the MemSwap Dual compared to the MemSwap Multi, throughput and latency measurements are nearly identical. However, when

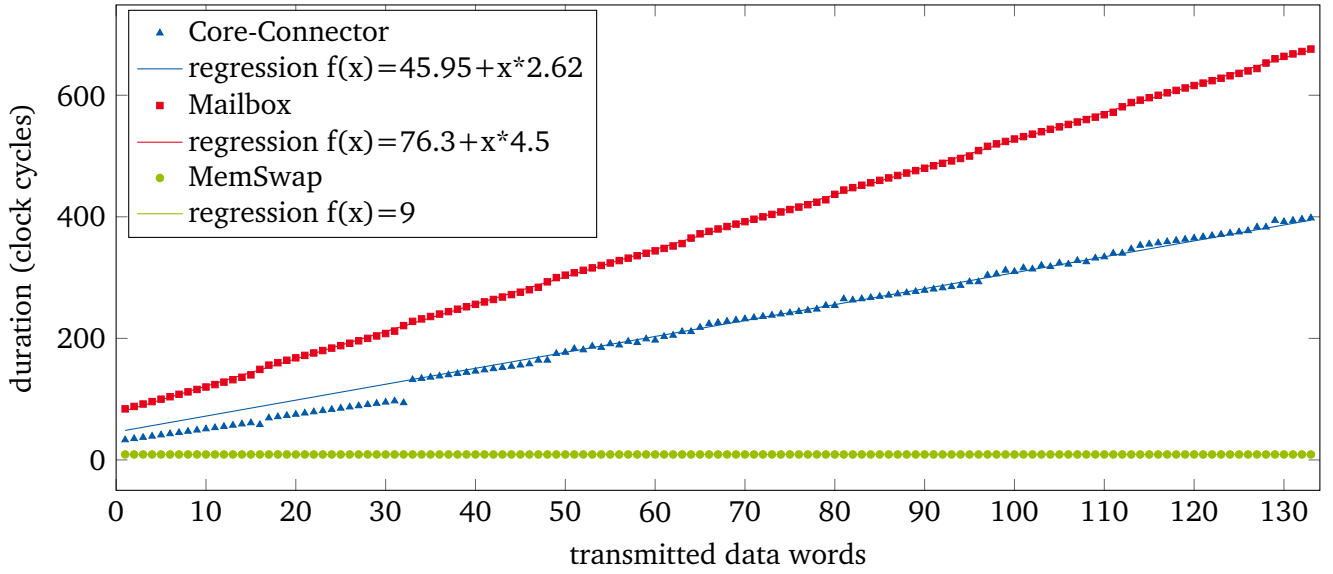


Figure 3.10: Transmission duration vs. data size for different 1-to-1 core-interconnects

multiple transmitters (N-to-1) want to communicate with the receiver at the same time, the latency increases and throughput decreases respectively. In this case the transmitter's data rate drops equivalent to the number of transmitters, while the receiver rate remains constant [85].

3.4 Global Memory

To show the performance of the global memory, the throughput for the SpartanMC and MicroBlaze global memory is measured for a varying number of simultaneous accesses. The throughput is measured with each attached core reading and writing the same array in the global memory within a loop. Thus, the generated assembler code mostly consists of load and store commands and covers the worst case for simultaneous access.

As shown in Figure 3.11a, the total throughput of the global memory slightly increases with additional cores For SpartanMC. Thus, a single core cannot utilize the maximum bandwidth of the memory. However, the throughput for each core goes down with every additional competitor. For the MicroBlaze this looks different, as shown in Figure 3.11b. For each additional core, the total throughput and the throughput per core goes down. The reason for the low performance has two reasons. One reason is that the AXI interface works very well for bursts and bursting is not possible for the applied access pattern and the other reason is that caching also does not work for the pattern. In practice this might look a lot better and the scenario describes the worst case.

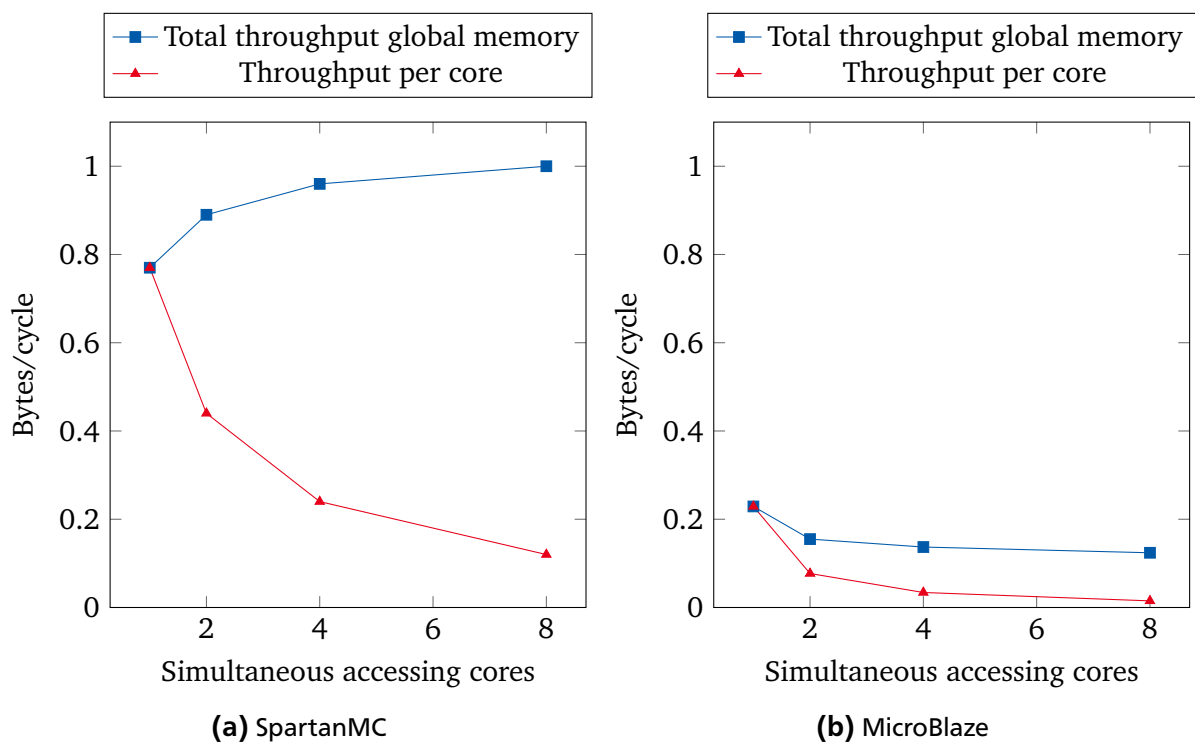


Figure 3.11: Global memory throughput

4 Used Multi-Core Architectures and Execution Concepts

Since pipeline parallelism should be extracted from the application, it is very easy and straight forward to map the extracted pipeline steps to a hardware architecture matching this structure closely. Such a structure can easily be generated through the use of soft-core SoCs on FPGAs. Three types of multi-core structures were chosen for the execution: Pipeline, Superscalar pipeline and shared global memory. The concept to use global memory is not favored in this work due to several reasons. Firstly, a shared memory is evaluated in the first tests to yield a relatively low maximum clock frequency for designs with multiple cores. Secondly, shared memory can easily become the bottleneck and performance is sometimes unpredictable. Thirdly, there already exists a variety of concepts for pipeline parallelism with shared-memory architectures [6]. In the following the required application structure for pipeline parallelism extraction is discussed and the theoretical performance of the used processing pipeline types is elaborated. The formulas have been verified with different pipeline diagrams theoretically and also measured with benchmark applications in practice.

4.1 Required Application Structure

As already stated, applications require certain characteristic to extract pipeline parallelism successfully[4, 5, 6]. The application should consist of different processing steps, where each step can calculate results for the next processing step. Later processing steps must not deliver results for a preceding step of the next iteration. Such a construct is also called loop carried dependency and would create a backwards dependency in the pipeline with idle waiting of an earlier stage. However, it is possible to have such a backwards dependency within one processing step.

It could also happen that a single processing step takes quite long but further pipeline stages cannot be extracted. In such a case replication is beneficial to create superscalar pipeline stages which will increase throughput.

Global variables are tricky since each pipeline stage contains data of different input data sets. If two stages read and write to global data, it is not certain that the later pipeline stages write before an earlier one reads data, as it would occur in the original version. Such cases could be detected and handled, but they would create undesirable backward dependencies in the pipeline. In a case where an earlier pipeline stage only writes and never reads a global variable no backwards dependencies exist and the variable can be propagated through the pipeline. Also, if only single processing steps access a global variable, it can be privatized to this step.

Different applications from PARSEC benchmark suite[91], CHSTONE[92], MediaBench[93], Dhrystone[94] have been reviewed. It was discovered that necessary structures to extract pipeline parallelism can be found in several applications, like ADPCM or (M)JPEG2000. However, a lot of applications are too large for the restricted target architecture's memory or use floating-point calculation which is also not featured in all target architectures. Also, if one imagines how embedded applications are structured, pipeline parallelism can be recognized. The structure is typically: Receive input data from a peripheral, process input data, output processed data, ...repeat forever. Even if no parallelism can be extracted during data processing, input and output often consume a considerable share of total execution time. Outsourcing input and output to an extra processing core could increase the application's throughput.

4.2 Pipeline

Figure 4.1 shows a processing pipeline with three stages. For SpartanMC, Core-Connector and MemSwap and for MicroBlaze the Mailbox can be used to transfer data from one core to another. Table 4.1 shows the expected performance from a processing pipeline type. T1, T2,... represents the different tasks, S/R

represents sending and receiving from one core to another and gray boxes indicate idle waiting of the previous task. Send and receive boxes belong to the sending as well as to the receiving task. Typically, the transferred application state (message) is larger than the FIFO size in case of FIFO interconnects. Therefore, the sending task starts to fill the buffer but can only finish sending if the receiving core reads from the FIFO. For smaller messages than the FIFO and a high sending rate of a task, the sending task only starts to wait idle when the FIFO is full of multiple transferred application state messages. For larger messages FIFO fill and empty effects at the beginning and end of a transmission are negligible and omitted in the following. For MemSwap modules, a transfer is only a multiplexer switch in between the BRAMs.

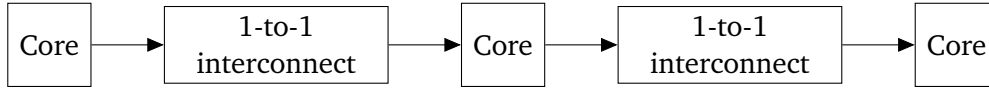
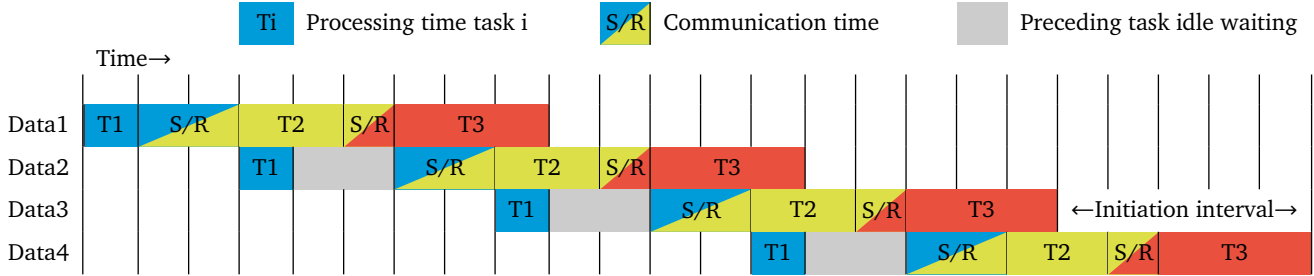


Figure 4.1: Pure pipeline, hardware configuration

Table 4.1: Pipeline execution with 1-to-1 interconnects



The throughput of the pipeline can then be calculated with multiple data input sets where task1 receives the different data sets and hands them through the pipeline. The initiation interval is then the time between data outputs from the pipeline. Thus, the termination of the last task represents the end of the data set processing. Without simulation, the initiation interval can be calculated as:

$$initiation_interval = MAX(receive_i + calculation_i + send_i)$$

where:

$$i \in [0, ..., tasks]$$

(1)

$receive_i, calculation_i, send_i$ = the receive, calculate and send time of the indexed pipeline stage.

In the example, the initiation interval is five time steps defined through task2. The throughput of the pipeline is calculated as number of data set divided by the initiation interval.

4.2.1 Pipeline Hardware Limitations

Processor pipelines cannot always be applied. Firstly, peripherals are always attached to **one** core. Cores can exchange data inputs from peripherals or outputs to peripherals, but cannot access the same peripheral interface from multiple cores. Different possible concepts are discussed in [95], to gain exclusive access to peripherals from multiple cores. However, these concepts would require extensive changes to the used soft-core SoC's peripheral implementations. Thus, the easiest and already supported method is to keep one peripheral attached to one core and communicate the necessary data to that core.

4.3 Pipeline with Replication

For replicated stages specialized 1-to-N interconnects peripherals such as Dispatcher or MemSwap Multi can be chosen or multiple 1-to-1 interconnects with the Mailbox can be used. N-to-1 specialized interconnects are the Concentrator or the MemSwap Multi. Specialized interconnects are hardware optimized and allow communication partner selection directly through software or arbitration in hardware as round-robin.

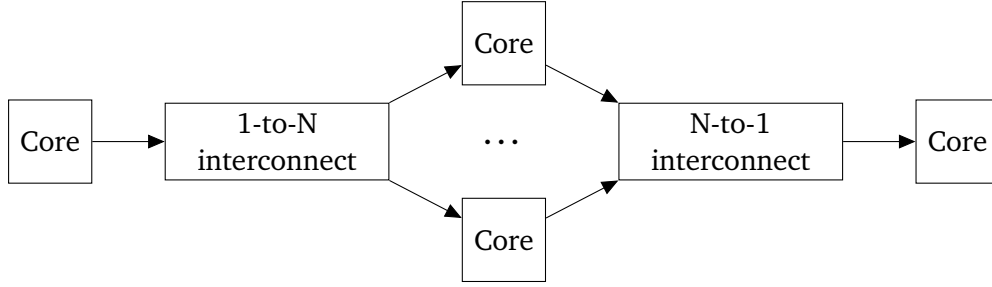


Figure 4.2: Replicated pipeline, hardware configuration

The initiation interval calculation for a superscalar pipeline stage as shown in Figure 4.2 is slightly different. In Table 4.2, task1 starts the execution and can then pass the application state to one replicated/superscalar pipeline stage. In the example the two replicated stages execute task2. The application state is then passed to the replicated pipeline stage that is currently free. After transmitting, task 1 can handle a new data set. However, since task2 takes longer to execute than task1, task1 would have to wait for task2 to finish in a non-superscalar pipeline. Since another core also executes task2 (more specifically 2.2), the next data-set can be sent to this core.

Table 4.2: Pipeline execution with 1-to-N, N-to-1 interconnects, replicated superscalar pipeline

	Time→																			
Data1	T1	S/R		T2.1		S/R	T3													
Data2			T1	S/R		T2.2		S/R	T3											
Data3					T1		S/R		T2.1		S/R	T3								
Data4							T1	S/R		T2.2		S/R	T3							
Data5									T1		S/R		T2.1		S/R	T3				
Data6												T1	S/R		T2.2		S/R	T3		

From the previous equation for calculating the initiation interval, task2 would be the critical pipeline stage requiring five time steps for receiving, calculation and sending. Through the 2x replication of task2, data-sets finish alternating every two or three time steps once the pipeline is filled. This means that the calculated initiation interval of a replicated pipeline stage with the previous formula can be divided by the replication factor. Meaning that the resulting initiation interval for the example results in 2.5 time steps. This results in the following formula:

$$initiation_interval = MAX\left(\frac{receive_i + calculation_i + send_i}{replication_factor_i}\right) \quad \text{where: } i \in [0, ..., tasks] \quad (2)$$

4.3.1 Replicated Pipeline Hardware Limitations

Replication cannot always be applied or it does not always make sense to apply it. The same limitations concerning peripherals apply as stated in Section 4.2.1. With this limitation it also does not make sense to replicate the first or the last stage since these typically receive inputs or send outputs via peripherals.

Replicated pipeline stages next to each other is another undesirable construct. Pipeline stages with different replication factors would require crossbar switches which are quite costly to implement[96]. Replication with the same replication factor could use 1-to-1 interconnects. However, Lorych[97] points out that such constructs have a higher communication and hardware overhead compared to an equivalently increased replication factor.

4.4 Shared Global Memory

As shown in Figure 4.3 global memory can be attached to any core if desired. However, usage of global memory is restricted to only read data from the global memory. Writing data to global memory easily destroys the "clocked" behavior of the pipeline stages, since each pipeline stage operates on a different data set. Writing to the global memory is possible at the user's responsibility to resolve conflicts. Since there are already enough global-memory architectures and the memory often becomes the bottleneck, global-memory should only be used as last resort. A well applicable scenario for global memory would be static data that each core reads rarely. Thus, each core would save local memory and the rare accesses do not overload the shared resource. The performance influence of global memory access can barely be modeled since it heavily depends on the number of simultaneous accesses which in turn depend on the application and the generated assembler code. From the assembler code one could find out the number of accesses, but finding out if they are simultaneous in a static program analysis is barely possible.

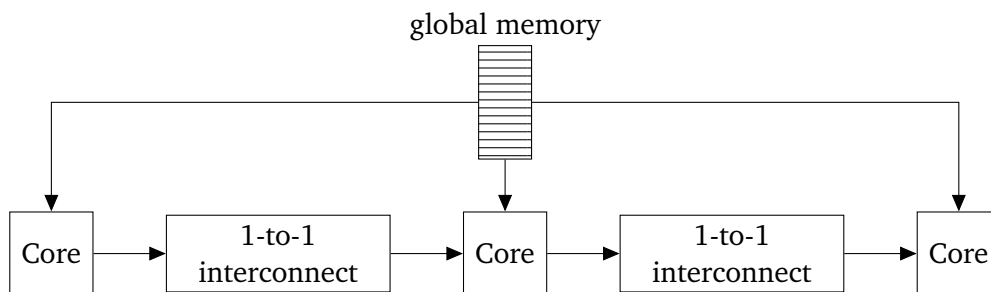


Figure 4.3: Pipeline with global memory, hardware configuration

4.5 Communication Overhead

Using FIFO interconnects, a core has to first fetch data from the inbound interconnect. In the ideal case, this is a load-store combination per data-word to copy from the FIFO to the core's local memory. Then the core can use and modify the received data. The modified data has to be again copied to the outbound FIFO if a successor core also needs the data.

The advantage of using MemSwap interconnects is that communication (switching BRAMs) happens within a few cycles. The inbound data is then transparently mapped into the core's address space. The outbound interconnect BRAM lies in a different address range than the inbound interconnect BRAM. Thus, if data is received and also has to be sent to another core, a memory-copy from the inbound to the outbound interconnect's address range has to be initiated. In the ideal case, this also requires a load-store combination per word with the `mempcpy` function. The MemSwap interconnects can thus transfer

data very fast, but have a penalty for passing the data to the next core. In the worst case, one memory-copy is required with MemSwap modules, while FIFO-based interconnects require two memory-copies. Thus, transferring with MemSwap modules is twice as fast in the worst case.

4.6 Latency

Pipelining increases the throughput at the cost of latency. This section shows the theoretic influences on the latency of a pipeline. As worst case assumption, one can use the following formula for a filled pipeline and always available input data:

$$latency = initiation_interval \times N \quad \text{with: } N = \text{total pipeline stages} \quad (3)$$

Thus, the latency of the pipeline is again dictated by the longest pipeline stage duration. In an unbalanced pipeline the predecessor of the critical stage has to wait idle until the critical stage can receive new inputs and the idle waiting propagates through all predecessors. This assumption is too pessimistic for successors of the critical stage. For better understanding, Table 4.3 shows a pipeline diagram with an unbalanced five stage pipeline. The initiation interval of the pipeline is four time steps with task3. Thus, the latency with the worst case formula is $5 \times 4 \text{ time steps} = 20 \text{ time steps}$, but the latency from the simulation in Table 4.3 is only 13 time steps. What is neglected in the previous formula are firstly, that send and receive time slots are overlapping and should not be counted twice. Secondly, task4 and task5, the successors of the critical stage, do not have idle wait slots. The critical pipeline stage already slowed down the processing rate such that equally long or shorter successor stages do not have problems with the given processing rate. A better formula is shown in the following:

$$latency = index_ii \times initiation_interval - \left(\sum_{i=1}^{index_ii} MAX(send_{i-1}, receive_i) \right) + \sum_{i=index_ii+1}^N (calculation_i + MAX(send_i, receive_{i+1})) \quad (4)$$

with:

$index_ii$ = pipeline stage index of the longest stage

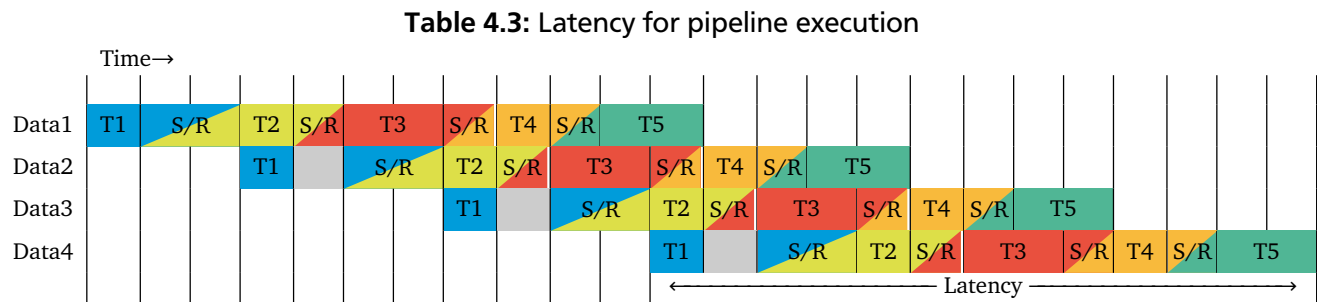
N = total pipeline stages

All predecessors of the critical pipeline stage are slowed down to the latency of the critical pipeline stage. Thus, for each preceding stage and the critical stage, the receive time has to be subtracted, since it is already included in the send time of the predecessor. Wait times of predecessors are thus automatically included. For all successors, no wait times occur, since they are either shorter or equally long as the critical stage and thus can easily keep up with the output rate of the critical stage. So, only the calculation and send time of each successor stage has to be added. It can now also be the case that, even though sending and receiving is overlapping, one is slightly shorter than the other due to buffer states or different overhead for calling the respective functions. The inaccuracy can be corrected by taking the maximum of send and receive time.

For a perfectly balanced pipeline, no wait times exist and the previous formula can be simplified to:

$$latency = \sum_{i=0}^N (calculation_i + MAX(send_i, receive_{i+1})) \quad \text{with: } N = \text{total pipeline stages} \quad (5)$$

The only additional latency overhead is the communication time per core.



5 Automatic Parallelization

The automatic parallelization is realized through several μ Streams tools. The advantage of multiple tools with intermediate files is that the user can control and influence intermediate steps since each step produces valid source-code. Also, the tools are exchangeable for other (future) target platforms.

All μ Streams tools use Cetus [98], a C source-to-source compiler, in the background to parse the source-code to be parallelized into an AST as Java representation. The AST can then be analyzed and modified through the μ Streams tools.

μ Streams is the base implementation providing the infrastructure to interact with Cetus, run optimization, modification or analysis passes. It also holds a set of passes which provide more general tasks, like for example program argument parsing. These passes are reusable for μ Streams extensions. Programs like AutoPerf, LoopOptimizer and AutoStreams are implemented as extensions to μ Streams. They widely use the μ Streams infrastructure and some passes, but also extend the functionality with their own compiler passes to fulfill the desired behavior.

The following sections first give an overview of the proposed toolflow and the input and output files. Afterwards, Cetus' capabilities and working principle are briefly explained and then the μ Streams infrastructure to run different transformation passes on top of Cetus is presented. The following sections show the working principle and some implementation details of the μ Streams tools.

5.1 Overall toolflow

Note: *Parts of this section have already been published in [99]. Self-citations are not marked in order to improve the reading flow.*

The overall toolflow for automatic parallelization is shown in Figure 5.1. The following sections give a short overview of the tools' tasks, why they are needed and how the tools interact. More detailed implementation description is given later on.

5.1.1 AutoPerf: Application Profiling

The legacy sequential source-code is firstly profiled with AutoPerf[100]. This step reveals the bottleneck of the application and hints beneficial parallelization opportunities. The code is instrumented with calls to the SoC's performance-counter or timer. Each C-statement is embraced by a call to start the performance-counter and afterwards read and reset it. The instrumented source-code and an abstract hardware configuration is provided as output. The provided design can be synthesized and run on the FPGA after automatically importing and building with the system-builder (jConfig). The user has to take care of providing an average or worst-case environment for the peripheral interaction during the measurement. A performance-profile is printed via UART or similar after the application finishes.

5.1.2 AutoStreams: Automatic Annotations

The produced performance profile and the original source-code are fed to Auto-Streams[99]. The user then specifies the desired throughput of the application. AutoStreams will parse the source-code into an AST. A CFG of the profiled source-code parts is created. A CFG node then represents a statement of the profiled application. Often, loops consume vast parts of the overall application runtime. Loops are automatically partitioned into multiple smaller loops with the LoopOptimizer, if required. This decreases the time granularity and increases number of CFG nodes.

Afterwards, AutoStreams tries to partition the application to use as little hardware as possible to achieve at least the user defined throughput. AutoStreams has the following optimization points:

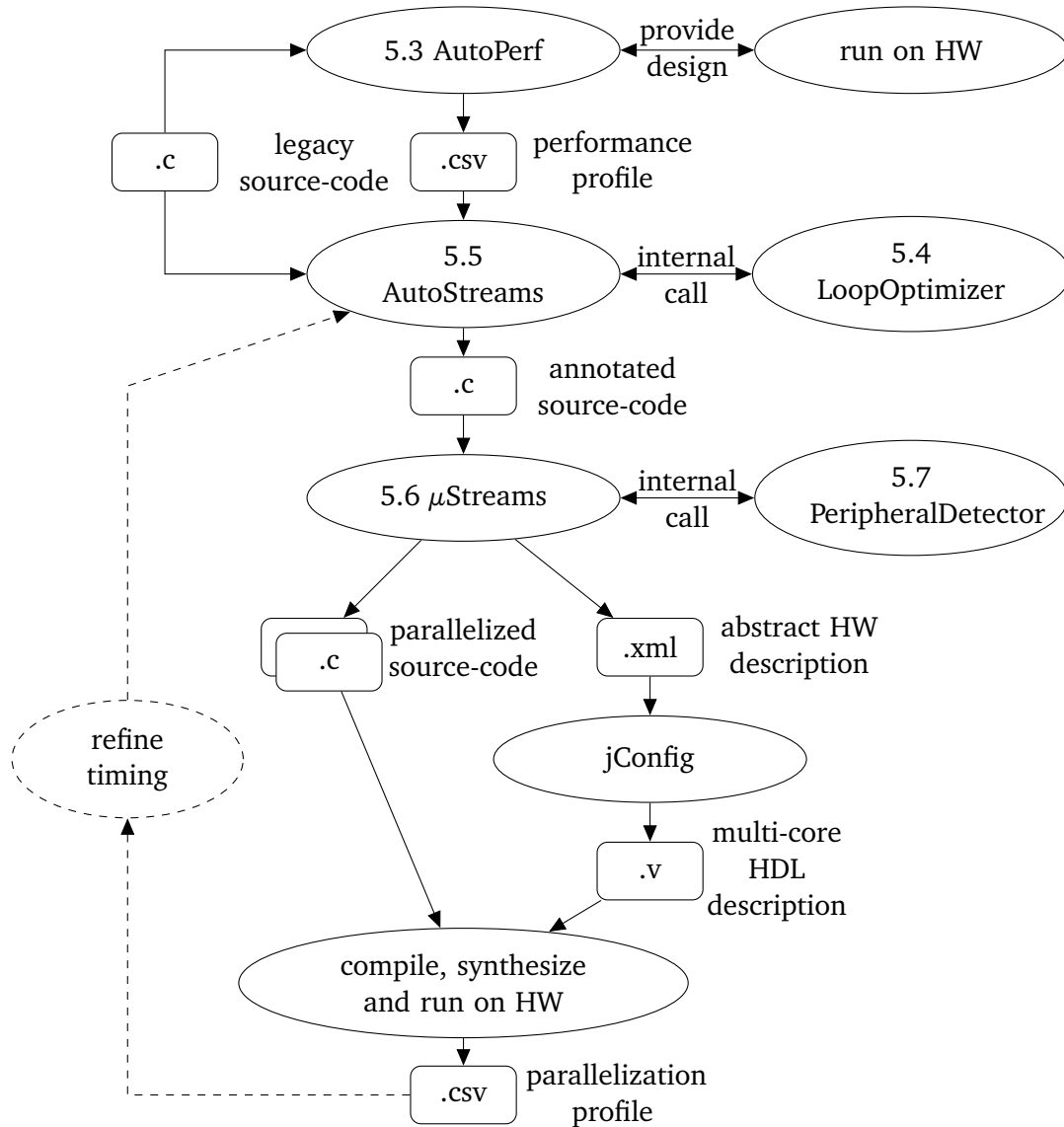


Figure 5.1: Simplified Overall Automatic Parallelization Toolflow with Tool Section Reference

Hardware: AutoStreams has to decide whether it is better to use more cores with slow communication or maybe fewer cores with fast, but hardware costlier communication interconnects. It should also be considered if some cores should be replicated in the pipeline.

Partition points: The partitioning points have to be selected firstly with respect to the maximum user defined duration per stage. As a second step as few partitions as possible should be created and each partition should have similar, at best equal runtime to get a balanced pipeline with high throughput.

Peripherals: If different code parts use the same peripheral, these parts need to be mapped to the same core.

The used partitioning is reflected in injected source-code annotations which can be reviewed and manipulated by the user. μ Streams can then be used to create a parallelized design based on the annotations provided by AutoStreams.

5.1.2.1 LoopOptimizer: Loop Optimizations

Creating balanced pipelines doesn't work well when single statements or CFG nodes, which are often loops, dominate execution time. Placing μ Streams pragmas inside loops is currently not supported, since it will create pipeline backward dependencies, undoing the benefits of the pipeline. The LoopOptimizer implements loop splitting and loop fission to provide AutoStreams with loop restructuring and more possibilities for partitioning points.

Loop fission finds independent statements inside a loop's body. Independent statements are partitioned into a separate loop while the loop condition remains identical for all loops.

Loop splitting is a method to distribute the iterations of the original loop to several loops handling a fraction of the original iterations. Thus, the iterations are partitioned while the loop's body stays the same.

5.1.3 μ Streams: Annotated Source-Code Transformation

The target of μ Streams[101] is to partition the original source-code at the pragma annotations into several chunks, forming a processing pipeline. Each pipeline stage will do a fraction of the work of the original application and pass results to the next pipeline stage proceeding in the same way. Thus, the first core is able to handle new data inputs in shorter intervals, increasing the application's throughput. A simple example can be seen in Figure 5.2. Currently, μ Streams has only one pragma: `#pragma microstreams task` with the option to specify replicate **number of replicas** to make non-dividable pipeline stages superscalar. Dependencies between the partitioned source-code parts are automatically identified and communication infrastructure is automatically created in software and hardware. μ Streams is also able to detect used peripherals at the different source-code parts, based on used APIs and variable types with the PeripheralDetector. One firmware file per core is written as C source-code at the end of modification. Additionally, an abstract hardware description (XML) is created, specifying processor cores, core-interconnects and peripherals. The user also has the option to add performance-counters to automatically measure the performance of the parallelized design. The abstract hardware description and the firmware sources can be imported into the system-builder (jConfig), automatically connecting and building the components. Afterwards, the system can be synthesized and compiled to be run on an FPGA.

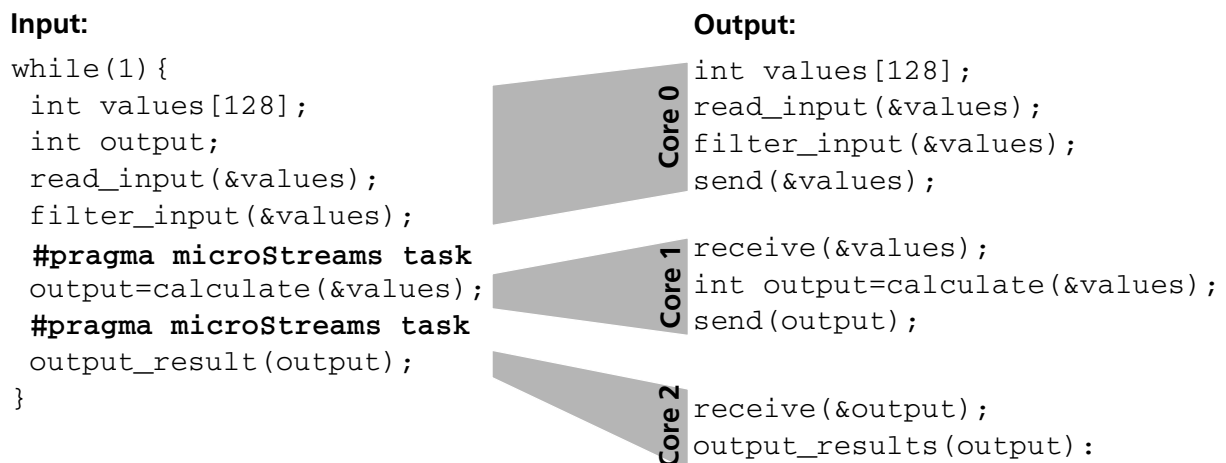


Figure 5.2: μ Streams concept: SW transformation

5.1.4 Refine Timing Constraints

The user can review the optional parallel performance profile after execution on the FPGA and match it against the requirements. In case of an unfulfilled requirement, the user can either restrict the through-

put requirements or slightly manually modify the pragma annotations, set by AutoStreams. A fitting parallelization should be reached after a few refinement iterations.

5.2 Common Software Infrastructure

All parallelization tools require Cetus for source-to-source transformations. Therefore, the following section quickly introduces the capabilities of Cetus. The following tools for parallelization are each usable as standalone or combined. Thus, a common transformation infrastructure is presented. Each tool has common as well as unique transformation steps, so called compiler passes. Section 5.2.2 presents an infrastructure to combine arbitrary compiler passes which together represent the different parallelization or profiling tools.

5.2.1 Cetus

Cetus[98] is a source-to-source compiler/transformer for ANSI-C/C89. Even though, it is possible to experimentally enable some popular C99 features like "freely placeable identifier declarations". As standalone version, Cetus offers several transformation passes like loop normalization, loop parallelization, variable privatization or data dependency testing and also a mechanism to add custom compiler passes. To transfer the C-source-files into an internal AST, ANTLR³³ (ANother Tool for Language Recognition) is used. An example of the generated AST from the source-code in Listing 5.1 can be seen in Figure 5.3. Each generated AST element implements the Traversable interface for traversing through the tree (access children and parent). Each tree element type is wrapped in its own class to access and modify the type specific elements.

Cetus was chosen because it offers the following features:

Input language C: Still the standard in the embedded domain.

Source-to-Source compilation: The user is able to check and modify the generated design very easily. Also, the user might be able to work around a bug after compilation. The compilation is also independent of the target platform compiler.

Written in Java: Java was proven to increase productivity over C++ [102], in which many other compilers are written.

As alternatives to Cetus, there would also be the possibility to use GCC or LLVM as a compiler. GCC has no possibility for source-to-source compilation, as far as I know. Additionally, GCC's plugin interface has been quite unstable throughout the versions. Thus, if the plugin is not constantly adapted it might become unusable in future releases. LLVM has source-to-source compilation features through Clang. However, source-code transformations happen as text replacements and require new parsing to an AST after each transformation. This is on the one hand very compute intense and on the other hand it is also not possible to carry out a transformation in multiple steps where one step might produce invalid syntax. Thus, Cetus was chosen over GCC and LLVM for writing a parallelizing compiler.

The main phases of Cetus are:

1. Parse command line arguments
2. Parse C-files & transfer into internal AST representation
3. Apply chosen optimization passes
4. Output (modified) C-files

³³ <http://www.antlr.org/>

Listing 5.1: Example Cetus Input Program

```
1 int i;  
2 void main() {  
3     //assign a value to i  
4     i=3;  
5 }
```

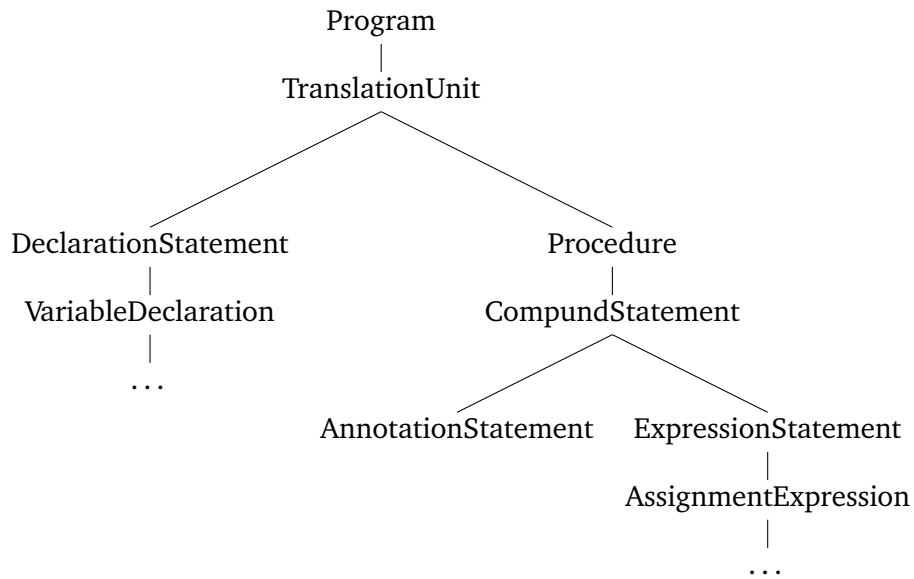


Figure 5.3: Simplified Cetus AST generated from Listing 5.1

It is also possible to use source-code to AST parsing on its own. The Cetus infrastructure for command line parsing and especially for running optimization passes has several issues:

1. Transformations are not designed to share work. Thus, execution order is arbitrary/statically defined.
2. Program model and style is outdated.
3. Custom transformation passes might require dependencies to Java libraries, one might not want per default to compile Cetus.

Solving the previous issues would require rewriting vast parts of Cetus, thus minimizing chances to benefit from Cetus updates or for fixes to be accepted in the Cetus main repository. Thus, the best alternative is to generate a modern, custom infrastructure for running compiler passes especially fitted for the needs of μ Streams. Even though Cetus uses rather outdated language features, the feature for parsing the AST and the provided API are helpful. Thus, it was decided to use Cetus and slightly update the implementation and add further functionality (mainly comfort functions and iteration through java streams) through "Util" classes in μ Streams.

Further information about the Cetus-API, papers and users-manual can be found on the Cetus website³⁴.

³⁴ <https://engineering.purdue.edu/Cetus/>

5.2.2 Common Transformation Infrastructure

As the Cetus transformation pass runner is incapable of providing the needs of the different μ Streams program variants, a new one is proposed here. Figure 5.4 shows a class diagram, depicting how different optimization passes are executed.

The central information storage component is the `Model` class, which holds all information (a List of `ModelComponents`) generated during transformation passes. For example these are all variables, dependency graphs, the traversable Cetus AST or generated tasks. Anytime a transformation pass generates or requires information it queries the "Model" object.

Each transformation pass extends the `AbstractTransformationPass` class, which in turn partly implements the `TransformationPass` interface. The `TransformationPass` interface defines methods that tell which classes (derived from `ModelComponent`) are required on the model to execute (`requiredComponents()`) or which pass has to run before (`requiredTransformationPasses()`). The interface also demands a method returning `ModelComponents` that this pass provides to the `Model`. With this information, the `processable()` method is able to tell if a pass is already executable or not. The `AbstractTransformationPass` implements this method by looking if all required transformation passes have already been executed. The transformation pass is executed, if all required `ModelComponents` are either in the `Model` or no pending transformation pass exists providing the required `ModelComponent`, since model components can also be required optionally by a pass.

To create a new compiler pass runner i.e. a new μ Streams source-code transformer, the `getTransformationPasses()` method of the abstract `TransformationPassCollector` class needs to be implemented. This returns a list of classes implementing the `TransformationPass` interface (for example `ArgumentParser` and `CetusParser` in Figure 5.4). This list of transformation passes is executed by the `TransformationPassRunner`. The `process()` method evaluates which transformation passes are processable by evaluating their respective method. In a second phase, the executable transformation passes are sorted. This is for example necessary if transformation pass A requires and provides a `ModelComponent` and pass B also requires this `ModelComponent`. Thus, pass A is an optimization pass for this `ModelComponent` and should ideally be executed before pass B.

Thus, this infrastructure provides a method to get a list of transformation passes to execute. The execution order is automatically determined by the required and provided components.

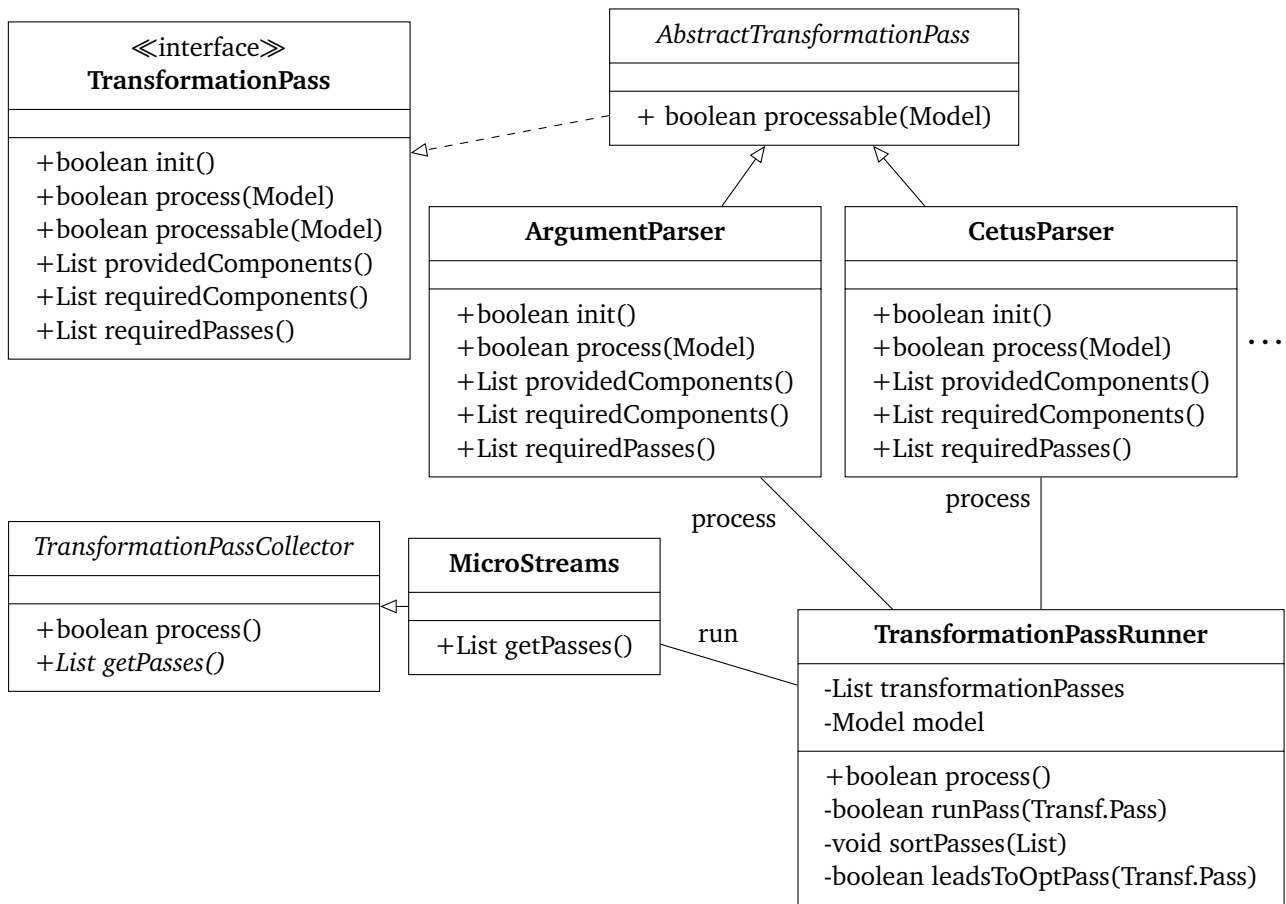


Figure 5.4: Simplified μ Streams transformation pass runner class diagram

5.3 AutoPerf

Note: Parts of this section have already been published in [100]. Self-citations are not marked in order to improve the reading flow.

AutoPerf is an application profiling tool initially developed for the SpartanMC environment by Kamp[88] during his bachelor's thesis. It is based on Cetus and the common transformation infrastructure presented in Section 5.2.2. The tool was constantly extended and now it is able to operate in the SpartanMC and the MicroBlaze environment through a wrapper. AutoPerf is able to profile functions, compound statements and loops. The idea is that AutoPerf gathers enough information to identify performance demanding application parts that benefit from parallelization.

As shown in Figure 5.5 the toolflow for AutoPerf starts with the legacy C source-code of an application. An example application is shown in Listing 5.2. The source-code is read by AutoPerf and instrumented with calls of the performance-counter (see Listing 5.3). The instrumented source-code is returned along with a description of the required hardware configuration as output. The hardware configuration is read by the system-builder (jConfig) which creates an HDL top-level description and system headers of used peripherals. The assembled hardware and the instrumented source-code can then be synthesized respectively compiled and run on an FPGA. While running the system on the FPGA the user is responsible for creating an appropriate environment in terms of input data and peripheral interaction for the measurement. Depending on the intended parallelization use case, different environments should be used. If the worst-case execution time should be revealed, an according environment revealing this scenario should be created. Often, the worst-case execution time is too pessimistic and generates unbalanced parallelizations for most usage scenarios. Therefore, an environment representing an average use case might fit better. After the profiling has been run on the FPGA, the results can be for example output via UART. The output is fed back to the PC and parsed through a small formatting tool called SerialReader, which will generate a CSV file as shown in Table 5.1. The generated CSV holds a unique identifier of the statement along with the execution time in cycles. The unique identifier consists of the file, function, surrounding compound statement, the first characters of the source-code line and the line number. Line number and file would be unique as well, but Cetus does not annotate line numbers to all AST component types.

Listing 5.2: Input source-code for profiling

```
1 void main(){
2     while(1){
3         int values[128];
4         int output;
5         read_input(&values);
6         filter_input(&values);
7         output=calculate(&values);
8         output_result(output);
9     }
10 }
```

Table 5.1: Produced performance-profile example

UNIQUE ID;	CYCLES
main.c:main:while:read_input:5;	2155
main.c:main:while:filter_input:6;	8912
main.c:main:while:output=calculate:7;	52169
main.c:main:while:output_result:8;	4652

5.3.1 Traditional Approaches

For profiling applications software and hardware based approaches are widely used [103]. The following methods are often leveraged to profile C applications on embedded systems:

Simulation executes a processor model running the application in a simulation environment. The simulation platform then is easily able to collect data about the executed instructions, in which program

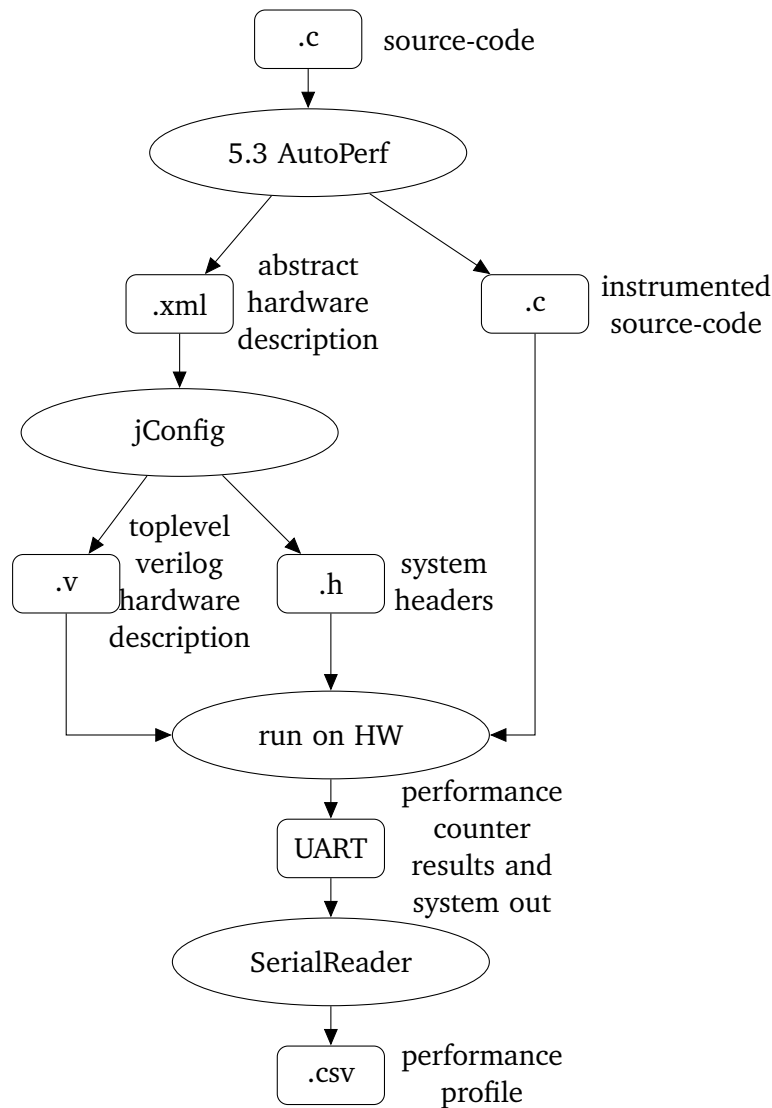


Figure 5.5: Detailed AutoPerf toolflow

part they reside and of course how long they take. Simulation based approaches have the drawback of being slow and not all embedded systems can be simulated or have a simulator. Also, peripheral IO would have to be simulated, which is not always easy.

Trace debugging adds additional tracers or debuggers in hardware, often communicating via a Joint Test Action Group IEEE 1149.1 (JTAG) interface. The crucial part here is to have a powerful enough host to process the huge amount of traced data, especially for many-core systems. With the availability of FPGAs, tracing events of processors can already be done and dynamically configured on the FPGA. Thus, the enormous amount of data can be minimized to the parts the user is interested in. The drawback of this method is that the profiler's and processor's hardware-design have to be adapted, if a new processor is to be supported.

Statistical profiling instruments code on a software basis. Counters are added for example to count how often a function is called. The code is also sampled, meaning that the application execution is regularly interrupted to collect the program-counter and infer the function currently executed. Well-known examples of such profilers are GNU gprof[104] or Intel vTune. However, Froyd[105] observed that profiling with gprof adds 10 to 260 percent execution time to the application. Especially peripheral handling might be influenced by this overhead and would not represent realistic

results. Also, statistical profiling is inaccurate depending on the sampling rate, due to its nature. Increasing the sampling rate would again add overhead.

Instrumentation adds calls to a performance-counter to the code to measure time-spans between different points in the application. Nowadays, most processors, even in the embedded domain have hardware performance-counters or timers to measure time. The intrusion overhead in the program is smaller than with trace debugging and the method is more accurate than statistical profiling. The method is observed to be not fully accurate since the instrumented code might sometimes influence the applicable compiler optimizations.

Thus, AutoPerf's method of using performance-counters or timers along with only slightly instrumented code results in a higher accuracy compared to pure software approaches. It is also faster compared to simulation. Furthermore, it offers the flexibility of being easily adaptable to other SoCs given that they have a timer or performance-counter. Mainly, only a software wrapper for starting, stopping and reading counters is required for a new environment to work with AutoPerf.

5.3.2 Implementation

Firstly, Cetus is used to read the source-files into a modifiable AST. Since Cetus would overwrite the source-files after modification and the original files should be left intact, a copy of the source-folder is created with AutoPerf appendix. Depending on the selected profiling mode via command line, one of the following profiling modes is chosen:

Compound Statement and Loop Profiling: Performance counter calls are inserted in the AST before and after statements to measure their duration. Either loops or compound statements can be profiled. By default, the main function is instrumented when in compound statement profiling mode. An example of the instrumented code from Listing 5.2 is shown in Listing 5.3. If the user wants to profile other compound statements, the `#pragma autoPerf` should be put ahead of a compound statement or function declarations. All direct children of the selected compound statements are collected. Variable declarations are ignored since they only consume a neglectable runtime. If the compound statement contains another compound statement, the contained statement is profiled as one piece and not further broken down for profiling if not told so by a pragma. Before each statement, a function call to start the performance counter is added. After each statement, function calls are added to stop the performance counter, read the measured values and store them in a structure. The structure where all results are stored has to be defined. The structure is stored globally and contains the string of the unique identifier of each statement to be profiled along with the performance measurement. The results of the measurements are output at the end of the main function. It has to be noted that multiple iterations through the profiled compound statement overwrite the results of the previous measurement. This is especially unhandy when profiling loops. For this purpose the loop profiling mode can be used.

Loop Profiling: In contrast to the compound statement mode, the results of the measurement are not printed at the end of the main function, but rather after each loop iteration and before continue and break statements. It is obvious that this method results in a highly increased application runtime especially for loops with many iterations. However, creating appropriately sized structures holding the results of many iterations and printing results only at the end can easily consume most of the valuable memory in an embedded system.

Accessing library functions of the performance counter, timer and UART requires including some headers in the modified source-code. These libraries are appended to the existing include block if not already contained.

Listing 5.3: Instrumented source-code (diff-style highlighting: green lines with + are added)

```
1 + #include<perf.h>
2 + #include<stdio.h>
3
4 + struct perf_auto_result perf_results[] = {
5 +     {.name = "main.c:main:while:read_input:5"},
6 +     {.name = "main.c:main:while:filter_input:6"},
7 +     {.name = "main.c:main:while:output=calculate:7"},
8 +     {.name = "main.c:main:while:output_result:8"},
9 + };
10
11 + FILE *stdout = (&UART_LIGHT_0_FILE);
12
13 void main(){
14     while(1){
15 +     perf_auto_init();
16 +     perf_auto_start();
17         int values[128];
18         int output;
19         read_input(&values);
20 +     perf_auto_stop(0, perf_results);
21 +     perf_auto_start();
22         filter_input(&values);
23 +     perf_auto_stop(1, perf_results);
24 +     perf_auto_start();
25         output=calculate(&values);
26 +     perf_auto_stop(2, perf_results);
27 +     perf_auto_start();
28         output_result(output);
29 +     perf_auto_stop(3, perf_results);
30     }
31 +     perf_auto_print(4, perf_results);
32 }
```

To have a unified API for controlling the SpartanMC performance counter and MicroBlaze timer in the instrumented source-code, a wrapper for MicroBlaze is written. According include directives for the wrapper are added in the source-code and the wrapper files are copied to the firmware folder. Usually one would expect a library for this, but since the MicroBlaze sources included in Vivado cannot be extended it was decided to keep them in the java project resources folder and copy them if needed.

5.3.3 Credibility of Measured Results

The measured results, acquired with the described method, are of course subject to errors. One kind of error is the function call overhead through the performance-counter function calls. However, the function call overhead is only a few cycles in both target architectures and it can statically be subtracted from the measured results, since it is constant.

A different kind of error comes from applied or not applied compiler optimizations. Since the performance counter calls hold inline assembler instructions, the GCC compiler will not apply optimizations

over inline assembler statements. On the one hand this makes the measurement accurate since instructions won't be swapped, but on the other hand compiler optimizations applied previously are not possible anymore. However, it has also been observed that isolating code parts can trigger some optimizations [106]. In general, the applied optimizations strongly depend on the source-code, and they are unpredictable without knowing the deep internals of the compiler. Since the parallelized system is also intended to hold only fractions of the original application, a (not) applied compiler optimization here might also be (not) possible in the parallelized system.

Another error resides in the possible microcontroller's interaction with the environment. Imagine a microcontroller with a peripheral constantly collecting data. The microcontroller polls the data, processes it and starts over again with polling the data. If the microcontroller takes longer for processing the data due to the additional performance profiling overhead, more data will be available for polling in the next iteration which might in turn take longer to process. In such a scenario, the user has to carefully adapt the environment (in this case the peripheral) to the changed processing speed. Alternatively, one could ignore the error, which would result in a pessimistic base for parallelization. However, the parallelized system in turn would then likely be faster than expected, which is better than vice versa.

With the previous example scenario in mind, an error can also come from synthesizing the generated hardware. Due to the added components for profiling, additional hardware has to be synthesized which might result in a lower clock frequency. Therefore, the performance results are measured in clock cycles to be independent of the achievable clock frequency.

As a countermeasure to not distort results more than necessary, the tool avoids executing different profiling types in parallel. This means that loop and compound statement/functions have to be profiled in separate runs. This eliminates the chances of loop profiling occurring during an outer compound statement profiling. The loop profiling would output a performance profile in each iteration which would heavily influence the outer compound measurement.

To eliminate false measurements when interrupts are present, the performance counter is stopped when entering an interrupt service routine (ISR) and started again when leaving an ISR. For SpartanMC, this is directly implemented in the performance-counter. Since MicroBlaze uses a simple timer for the measurement, the ISR is instrumented in software to stop and start the timer when entering and leaving the ISR function.

5.4 LoopOptimizer

Note: Parts of this section have already been published in [106]. Self-citations are not marked in order to improve the reading flow.

In many programs, loops hold a majority of the total execution time and benefit from parallelization. It has been discovered that a high degree of parallelization with μ Streams is limited by loops in many applications. These loops are atomic entities to μ Streams and thus are only mappable to one processor. It is the LoopOptimizer's task to find a way to partition those loops such that μ Streams can map loop parts to different cores. Nowadays, many tools already exist that focus on loop parallelization. A majority of these tools target HPC environments with shared-memory models or OS support. Since μ Streams uses a pipeline parallel execution, most well-known loop parallelization techniques and tools are not applicable in this domain.

The LoopOptimizer can be used as a standalone transformer where it is used as a transformation pass in the common transformation infrastructure. The workflow of this mode is visualized in Figure 5.6a. The C source-code of a program with pragma annotations at loops to be transformed is provided by the user. If a AutoPerf loop performance profile (Section 5.3) is not given, each iteration and statement is (unrealistically) assumed to have equal execution time. The pragma annotation specifies the number of loop fractions that the original loop should be transformed to. One can also choose a distribution to give for example the first partitioned loop 80% of the original loop's runtime and the second will have 20% runtime. As output, the tool delivers a transformed C source-code with the specified loops partitioned accordingly.

The LoopOptimizer can also be used as a library to any tool, operating on a Cetus AST, as visualized in Figure 5.6b. The tool delivers the loop AST to the LoopOptimizer and specifies the transformation type as well as the partition size and number. Alternatively, the maximum possible partitioning can be queried to a given loop AST. A modified AST is delivered as output, which the application can continue to work with.

The library mode is used during automatic parallelization, while the standalone mode offers this functionality for manual parallelization.

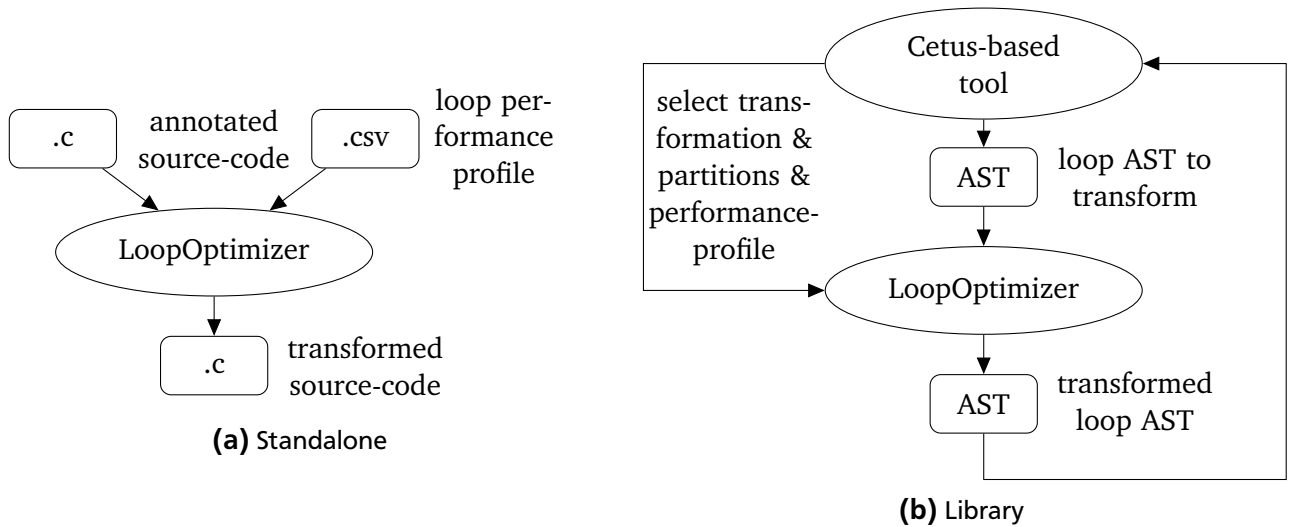


Figure 5.6: Detailed LoopOptimizer toolflow for different operation modes

5.4.1 Loop Parallelization Techniques

In many applications, loops are the most time-consuming computation steps. There exist many loop transformation techniques to improve the parallelizability of an application. Many have already been implemented in modern compilers to leverage current multi-core architectures. Techniques like loop interchange, scalar expansion, scalar renaming and index-set splitting focus on improving fine-grained parallelism, which is beneficial for VLIW or superscalar processors. Whereas techniques like privatization, loop distribution, loop tiling aim to improve coarse-grained parallelism for shared-memory parallel processors [107]. However, the aforementioned techniques are not usable to aid μ Streams with parallelization. In order to parallelize or transform an application with μ Streams, a suitable place to split the application is needed. μ Streams pragmas cannot be placed inside loops, since the created pipeline stages would also create a dependency on themselves. These large parts of non-splittable code often create an imbalanced pipeline, which increases the duration of the critical computation step. Therefore, these loops need to be partitioned into several smaller loops, each handling one part of the original loop. μ Streams pragmas can then be placed in between the partitioned loop parts and μ Streams is then able to analyze the dependencies and create a processing pipeline. All variables used in the loop's body will be transferred between the pipeline stages containing parts of the partitioned loop. All aforementioned transformation techniques aim to execute the loop's body or parts of the body simultaneously accounting for interference of data read/written by multiple entities. For μ Streams, interference is implicitly handled since each pipeline stages operates on an own data set. Two static loop transformation methods exist to allow μ Streams to operate its intended way [107]: loop splitting and loop fission.

Listing 5.4: Original loop

```
1 //possible task creation
2 for(int i=0; i<10; i++) {
3     foo();
4     bar();
5 }
6 //possible task creation
```

Listing 5.5: Fissioned loop

```
1 //possible task creation
2 for(int i=0; i<10; i++) {
3     foo();
4 }
5 //possible task creation
6 for(int i=0; i<10; i++) {
7     bar();
8 }
9 //possible task creation
```

Listing 5.6: Split loop

```
1 //possible task creation
2 int i;
3 for(i=0; i<5; i++) {
4     foo();
5     bar();
6 }
7 //possible task creation
8 for(; i<10; i++) {
9     foo();
10    bar();
11 }
12 //possible task creation
```

However, μ Streams allows further loop transformations after the program has been transferred into a pipeline structure due to its source-to-source concept. Loop splitting has already been used in [108] to increase pipeline performance for high-level synthesis with promising results.

To make a good partitioning of the loops, a detailed performance profile of the loop is helpful, but not always necessary. For loop splitting, it is relevant how much time the loop body consumes in which iteration. For loop fission, the execution time of each statement is important. AutoPerf can be used in loop profiling mode for this purpose. In the following, the two loop optimization techniques, along with their peculiarities and limitations are described.

5.4.1.1 Loop Fission

Loop Fission is a method to distribute the body of a loop over several loops, each containing a part of the original body. In practice, it is often used to improve cache hit rates on large loops [107]. Also, the generated independent program parts are very beneficial for multi-processor systems. An example can be seen in Listings 5.4 and 5.5. The critical part of this transformation is ensuring correctness: Statements that depend on other statements in the loop have to be executed in the same partial loop. Additionally, if the called functions (like `foo` and `bar` in the example) have side effects or use global variables, the order in which they are executed is also important. If it is changed by fission, the program may produce incorrect results.

Recognizing these dependencies with a static analysis, especially those hidden within functions, is hard. Without a robust interprocedural dependency analysis, uncertain statements need to be assumed as dependent, which highly limits the number of partitions for fission. In practice splitting usually delivers more partitions than fission since statements in a loop body mostly depend on each other.

A loop is first analyzed for independent code-blocks (I). These blocks are then distributed to a user-specified number of partitions (N) as follows:

$N = I$: Every block is mapped to one partition.

$N < I$: The smallest independent statement blocks are merged, until $I=N$. If profiling data is present, the smallest blocks in terms of execution time are merged. Otherwise, the smallest blocks by number of statements are merged.

$N > I$: The loop does not have enough independent parts and cannot be distributed to the desired number of partitions. A warning is given, and N is set to I.

5.4.1.2 Loop Splitting

Loop Splitting is a method to distribute the iterations of the original loop to several loops handling a fraction of the original iterations. In practice this method is usually used to improve input prefetching on scalarized loops [107]. An example can be seen in Listings 5.4 and 5.6. The critical part of this transformation is handling/recognizing the iteration variable and modifying the exit condition of each generated loop.

5.4.1.2.1 Iteration variable: The iteration variable is defined here as the variable modified in (each) iteration and controlling the exit condition of the loop. The variable can be boolean, integral (short, byte, int, long, ...), floating-point (float, double) or pointer. In this contribution, only integral types and pointers are handled. Floats are not handled, because many embedded processors do not support floating-point numbers. Boolean iteration variables either contain a trivial number of repetitions, a statically not analyzable number of repetitions or depend on other variables which are then the true iteration variables and are therefore not handled either. In the implementation, the iteration variable is guessed automatically. In for-loops, the variable occurring most often in the initial statement, exit condition, and afterthought statement is selected. In other loop types the iteration variable is the only one used in the exit condition. It is obvious that this technique does not cover all thinkable possibilities, but in practice most analyzed examples follow this pattern. Additionally, if the variable guessing fails or

is not distinct, an error is given and the user can specify the iteration variable by hand through a pragma annotation.

5.4.1.2.2 Exit condition: A loop's exit condition has to be a boolean condition, typically depending on the iteration variable. To form a boolean expression, the iteration variable is typically checked against a constant with `==`, `!=`, `<`, `<=`, `>` and `>=` operators. Another possibility is of course to check against a variable expression consisting of a function call, several other variables or concatenate different checks with boolean operators. Transformations for those cases are not automated. Splitting loops into equal parts requires knowledge of the exact iteration numbers. Modifying the exit condition can easily be done statically when a constant to check against is used. With a variable expression, the safest method is to use data from a previous profiling run. The reference value for the comparison in the exit condition of a loop can then be expressed with the following formula if the iteration variable is known.

$$exit_condition_{current_split} = init_val + \left\lfloor (end_val - init_val) \cdot \frac{t(current_split)}{t(total_splits)} \right\rfloor \quad (6)$$

With:

init_val = The iteration variable's value before entering the loop

end_val = The comparison expression part not containing the iteration variable

total_splits = The total number of loops that should be created

current_split = The current loop's index

t(x) = The partial loop's execution time from the first iteration up to *x*

5.4.1.3 Special Conditions & SoC Peculiarities

5.4.1.3.1 Break Statement: It can be the case that split loops contain a break statement. If the break is executed in one loop partition, all further partitions must not be executed. The solution is to add a boolean variable which is set when a break statement is triggered. All transformed loops are conditionally executed based on the boolean variable. The variable's value can then be transferred to the following cores to abort execution. An example can be seen in Listings 5.7 and 5.8.

Listing 5.7: Break loop

```
1 for(; i < 10 ; i++){
2     if(i==rand())
3         break;
4 }
```

Listing 5.8: Break loop transformed

```
1 int br=false;
2 for(i = 0; i < 5 && !br; i++)
3     if(i==rand()){
4         br=true;
5         break;
6     }
7 for(; i < 10 && !br; i++)
8     if(i==rand()){
9         br=true;
10        break;
11    }
```

5.4.1.3.2 Peripheral Usage: One peripheral cannot be used on several cores simultaneously, since a peripheral is currently only attachable to single processor core (more detailed reasoning is later given in Section 5.7.4). Thus, a peripheral should not be used in more than one loop. If this case is detected, an error message informs the user to manually take action and resolve the problem if possible.

5.5 AutoStreams

Note: Parts of this section have already been published in [99, 109]. Self-citations are not marked in order to improve the reading flow.

AutoStreams is the tool filling the gap between the performance profile and setting the pragma annotations for μ Streams. Without this tool, the user would have to analyze the performance profile and set μ Streams pragmas by hand. However, it is exhausting to create an optimal parallelization by hand, since there are many factors to consider and optimize.

The tool flow of AutoStreams (shown in Figure 5.7) starts with a legacy source-code to be parallelized. With AutoPerf, a performance-profile can already be provided. The performance-profile and the legacy source-code is then given to AutoStreams. The user specifies the maximum time that the application should take to process new input data or the number of pipeline stages to be created. AutoStreams finds a possible processing pipeline considering the user requirements. Internally, the LoopOptimizer is called to partition loops restricting further parallelization. AutoStreams outputs annotations into the legacy source-code that in turn is parallelized with μ Streams.

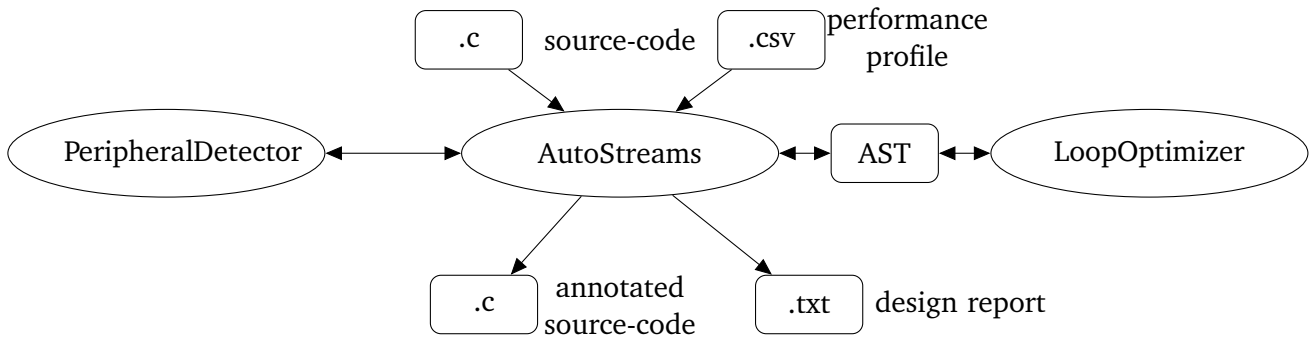


Figure 5.7: Detailed AutoStreams toolflow

5.5.1 Optimization Points

For manual pragma setting, it is often complicated to optimize different aspects at the same time and thus AutoStreams can deliver a multi objective optimization of the following factors:

Communication Overhead: Depending on the pragma positions, more or less program-state information has to be passed to the next processor/pipeline stage. Setting pragmas a few statements later might increase the processing time of the pipeline stage, but could dramatically decrease the number of variable dependencies and thus communication time. To balance reasons for choosing one or another, also the time required for communicating a certain amount of data has to be known or estimated. Finding dependencies of all variables manually is tedious especially on larger programs. Thus, a function to estimate the communication overhead based on the amount of exchanged data is derived from the characteristics of the communication infrastructure. When using the SpartanMC environment besides FIFO-based interconnects, also DMA-based interconnects can be used. These interconnects have very low communication time cost. However, when variables are received that also must be passed to the next pipeline stage a memory-copy between the two direct memory access (DMA) regions is required.

Hardware Overhead Consideration: During the search for processing pipelines fulfilling the user requirements, not only one, but often many solutions are possible. AutoStreams should prioritize solutions with the same performance, requiring less hardware than others. Each communication hardware

module and the processor have been synthesized for different FPGA families to estimate the hardware cost. The modules have been synthesized multiple times with different seeds to average out synthesis variations. Synthesis uses a search heuristic and different seeds deliver different starting points which might result in different proposed solutions. The hardware numbers are stored in AutoStreams for the different FPGA families. AutoStreams takes these measurement points for hardware usage estimation, by calculating a linear regression with the measurements for each hardware component and different configurations. The linear regressions are combined in an analytical model, where different hardware components can be queried and an estimated FPGA resource usage is returned.

Automatic Loop Partitioning: Since loops often consume much processing time, they should automatically be partitioned to get balanced pipeline stages during parallelization with μ Streams.

Trade off: Pipeline vs Superscalar: Making pipeline stages superscalar with replicate pragmas avoids the necessity of additional pipeline stages adding communication overhead. However, it is not always possible to make pipeline stages superscalar, for example if peripherals are used. Figuring out when to use superscalar pipelines goes hand in hand with estimating communication and hardware overhead and thus should be done automatically.

5.5.2 Implementation

The legacy source-code is transformed into an AST with Cetus in order to transform and analyze the source-code. Firstly, the performance profile is read and the execution time is associated with the according AST statements. Thus, the duration of each statement is known. By default, the main function is profiled and transformed, but also any other function called only once is possible. One exception is that the loop can be inside one (endless) while loop, as it is often the case for embedded applications.

5.5.2.1 Control-Flow-Graph Generation

A CFG is generated from the AST in the next step. This is necessary to create a pipeline structure from the program flow and to determine the used variables in the different source-code parts. Initially, loops appear as a single CFG node. Possible loop partitioning points can be requested from the LoopOptimizer, and they replace the original loop's CFG node with multiple virtual CFG nodes. Since only the execution time of the full loop is known, a proportional amount of time is annotated to each partitioned virtual CFG node. The actual loop transformation in the source-code is done only once in the end. When using loop splitting, the LoopOptimizer returns a possible split-point after each iteration. However, especially loops with many iterations would result in many virtual nodes and many possible split-points to be evaluated for AutoStreams. For high numbers of nodes, the evaluation process can take extremely long since so many possibilities exist. Thus, it was decided not to create one virtual node for every iteration, but to create bigger bundles. By default, one virtual node holds a number of iterations that take as long as 5% of the total runtime of the application, but the user can also specify a desired percentage. The fewer iterations a virtual node has, the better pipelines can be balanced, but the longer it takes to evaluate all possible solutions. The default value worked well for all tested benchmarks and solutions. The number of virtual nodes is high enough to create balanced pipelines and low enough to evaluate all solutions within a minute. In practice, it was observed that the estimated duration of an iteration often differs in the parallelized system due to different compiler optimizations and thus a highly accurate split point selection would not necessarily result in a better solution quality.

5.5.2.2 Communication and Hardware Cost Estimation

To decide on a code partitioning, the time costs for communication and the hardware costs for processor cores and different communication interconnects are required. The different core-interconnects' transfer speeds have been measured with different communication volumes as shown in Figure 3.10. AutoStreams calculates linear regression functions to estimate communication cost for larger communication volume for each interconnect separately. The hardware cost is also inter- and extrapolated through linear regression. Each core and core-interconnect has been synthesized multiple times with different configurations, like memory size, number of endpoints for N-to-1 interconnects or varying DMA-size for DMA-interconnects. For each synthesized configuration, the numbers of used look up tables (LUTs), registers, BRAMs and digital signal processing blocks (DSPs) are imported into AutoStreams. These values are the supporting points for the linear regression. The procedure has to be done for each FPGA family, since synthesis results and also hardware costs can differ. Currently, AutoStreams already contains measurements for Artix-7 and Spartan-6 FPGAs. With the supporting points, AutoStreams calculates a linear regression and inter- and extrapolates other configurations. Components like MemSwap Multi have two dimensions of freedom: endpoints and DMA memory size. It is essential to have a multiple measurement points distributed over the two-dimensional space to achieve good estimation results.

5.5.2.3 Design Space Exploration: Search Method

It is possible for all analyzed benchmarks to search for an ideal solution with respect to the chosen granularity. Two steps are used to search for a solution with the branch-and-bound method. Firstly, a non-optimal solution, fulfilling the users requirements, is created through a search heuristic. Secondly, a branch-and-bound method is used to explore the design space through a decision tree. The numerous possible solutions are limited through the non-optimal one. Every time a solutions is worse or knowingly cannot get any better than the non-optimal solution, further search in this direction is stopped. In the SpartanMC environment these steps are repeated with FIFO-based and DMA-based interconnects.

The search heuristic to find one possible solution for bounding the design space is described in the following. The heuristic is used in different ways depending on the user specification for a maximum processing time (initiation interval) or a desired number of pipeline stages:

Initiation interval: The search heuristic puts as many CFG nodes in a pipeline stage as the node's execution times and communication time stays below the specified limit. A new pipeline stage is started if an additional node's execution and communication time would exceed the given limit. This search method does not deliver an optimal solution since it is not able to look ahead and see if adding a node would decrease communication overhead, such that the pipeline stage fulfills the requirements again.

Pipeline Stages: The optimal maximum execution time per stage (initiation interval) is calculated by dividing the total application runtime through the number of specified stages. With the optimal maximum execution time per stage, the previous search heuristic can again be applied. However, it is unlikely that the heuristic finds the specified ideal solution. Thus, the search is repeated with relaxed timing constraints until a solution with the specified number of pipeline stages is found.

The procedure of finding the optimal solution through the branch-and-bound method is shown in Figure 5.8. The algorithm starts with the first statement in the CFG as the first pipeline stage. Afterwards, the next statement of the CFG is added to the current pipeline stage. As a second version, the new statement is added as a new pipeline stage. This process is repeated until all CFG nodes have been handled. It is checked if the generated pipeline stage fulfills the requirements in terms of user defined pipeline stages and/or pipeline stage duration after each step. It is also checked as upper bound if the current pipeline is better than the previously generated non-optimal solution. The quality of a valid solution is firstly defined by the hardware cost and secondly by the execution time of the longest pipeline stage. As

lower bound, it is checked if the remaining CFG nodes accumulated execution time can yield in less or equal amount of pipeline stages and shorter execution times per stage compared to the upper bound. The pipeline configuration is dismissed if the described bounds are exceeded. The usage of these bounds ensures that the pipeline stages are not too short, nor too long and the number of used pipeline stages stays small.

5.5.2.3.1 Hardware Comparison Metric: Looking in a synthesis tool's hardware report, the main numbers for hardware cost of an FPGA are: LUTs, registers, BRAMs and DSPs. The FPGA can realize various hardware designs out of these components. The number of all components is limited. The number of used components for a design specifies complexity and cost of a circuit. However, each of these components (specifically LUTs) have different configuration options, which is left out here for simplicity. To compare the complexity of a design, a cost metric has to be chosen. At first, there was the idea to compare the component with the highest utilization and thus the rarity of the component. This solution is not practical, since the rarest component might change during optimization, the optimization direction will also change. That means, depending on the starting point, results might differ and these are barely traceable to the developer or the user. As a second solution, the average utilization of all components could be compared. However, due to the characteristics of soft-core processors to have a comparatively high BRAMs utilization for memory, there would be no difference to directly comparing BRAMs. However, BRAM and also registers do not tell much about the complexity of the design. DSPs also seem unfitting for comparison since only the processor core uses DSPs. Thus, only the number of cores matter and core-interconnects would not be compared. The last component left are LUTs, which are actually used for comparison since they reflect the design complexity and every component uses them. Nevertheless, the user is able to restrict the maximum usage of one of the previous mentioned components via the options and thereby influence the optimization direction.

5.5.2.4 Design Space Exploration: Solution Selection

After all pipelines have been generated, the pipelines are sorted by their solution quality. The following filters and sort criteria are applied to all generated pipelines:

1. **Multiple peripheral usage:** The PeripheralDetector library is leveraged to determine used peripherals in the different pipeline stages. As argued in Section 5.7 a parallelization with a peripheral access spread over multiple cores most likely does not reflect the proper or intended behavior. If this filter leaves no pipelines in the list, the user can deactivate this filter and solve the problem manually afterwards.
2. **Prefer small Hardware:** If all configurations meet the desired pipeline stage duration requirements, the pipeline with the lowest estimated hardware resource usage is preferred. If the user specified an FPGA device or maximum allowed hardware resources, the tool also ensures that the

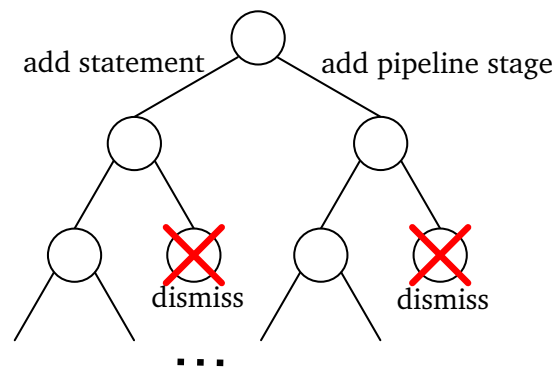


Figure 5.8: Search tree for design space exploration

design is able to fit the specified hardware bounds. In case it doesn't fit, the design with the smallest initiation interval is given, meeting the available resources.

3. **Prefer fast pipelines:** Among systems with the same hardware, AutoStreams prefers systems whose longest pipeline stage has the shortest runtime (smallest initiation interval).

5.5.2.5 Reports

After a pipeline has been selected, the task pragmas are set into the source-code by modifying the AST as specified in the selected pipeline configuration. The annotated source-code is written back to the file system.

A parallelization report can also be created to sum up the characteristics of the selected pipeline:

- Used pipeline stages
- Cycles of the longest pipeline stage
- Estimated speedup compared to the performance profile of the sequential version
- Estimated hardware usage
- A list of pipeline stages:
 - Start/end source-code segment
 - Estimated calculation cycles
 - Estimated communication overhead with communicated variables (send and receive)
 - Found peripherals with the PeripheralDetector
 - Stage replication count

The user can now start μ Streams to parallelize the annotated source-code.

5.6 μ Streams

Note: Parts of this section have already been published in [101]. Self-citations are not marked in order to improve the reading flow.

The main idea behind μ Streams is to transform the sequential source-code of a legacy C-Program into a multi-core pipeline. The toolflow is shown in Figure 5.9 and Figure 5.2 shows how pragmas can be added in the source-code, indicating pipeline stages. The example results in a three stage pipeline, since the source-code itself is already one task. Each pipeline stage contains parts of the original program along with a communication infrastructure between the pipeline stages. As different pipeline stages access the same variables, they need to be transferred. It is important that each stage has its own copy of the data, since each stage might modify the data. The generated code is shown in a simplified version in Figure 5.2.

According to the software design, an abstract hardware XML-Description is automatically generated. The description holds used components such as processors and communication interconnects along with information about how they should be connected to construct a processing pipeline. Performance-counters can optionally be added to evaluate the communication and processing time of each stage. Additionally, μ Streams is able to detect used peripherals based on the source-code with the PeripheralDetector. The generated abstract hardware description can be imported into jConfig, an SoC system-builder, which generates a synthesizable hardware description. This process allows hardware generation in a default configuration with little to no user intervention.

For a more detailed explanation, this section is divided into a paragraph describing the usable pragmas and unsupported C constructs, followed by a description of the necessary transformation steps to parallelize an application.

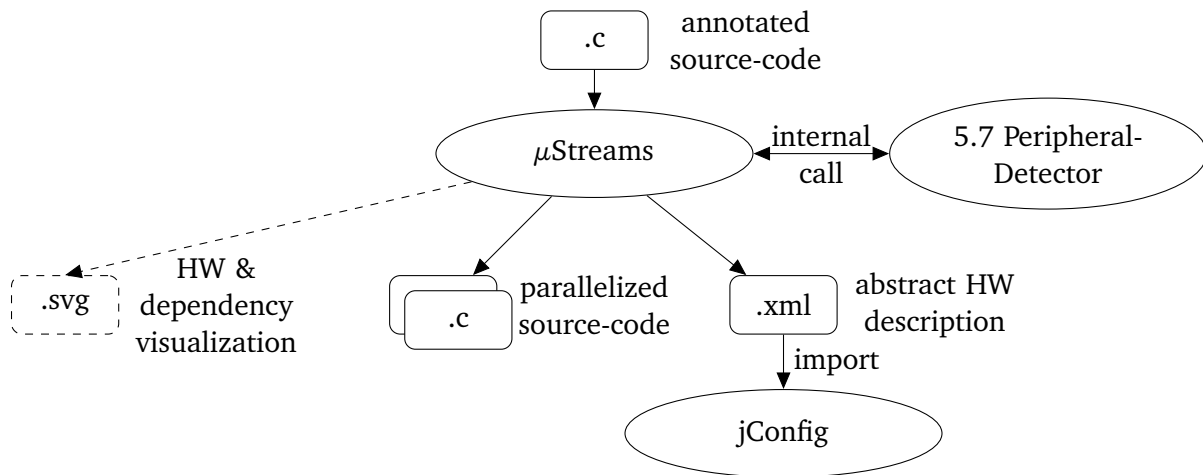


Figure 5.9: Detailed μ Streams toolflow (dashed=optional)

5.6.1 Usable Pragmas

As it can be seen in Listing 5.9, μ Streams task pragmas can be placed at the following positions:

- Ahead of any statement, including loops
- Ahead of a function call or function definition.
- Inside a function if this function is called only once.

Each task pragma adds a task/pipeline stage to the generated system. Pragmas ahead of function definitions add a new task for each function call appearing in the source-code.

Task pragmas cannot be placed at the following positions, otherwise μ Streams shows an error message:

- Outside the main.c file, except for pragmas before function definitions.
- Inside a function called multiple times.
- Inside loops, except for loops that don't have further executable statements after their execution (line 32 in Listing 5.9).
- Ahead of recursive functions

Currently only the main.c is transformed by μ Streams, other C-files are copied as is. If pragmas inside other C-Files should be used, the FTL-Template system, writing the generated C-Files, has to be replaced. The best alternative would be to directly do transformations in the Cetus AST. Compared to the FTL-Template system, this method has a huge source-code overhead and makes modifications more complex for μ Streams developers. However, most reviewed applications had the main processing steps declared in the main.c file. Other applications had a mostly empty main.c file with only one function call to another c-file's function. The contents of the other c-file could easily be copied to the main.c file, thus the constraint of pragmas only in the main.c file can easily be worked around. For pragmas inside loops/recursive calls, it is hard (sometimes impossible) to statically analyze exactly how often the code inside is executed. Thus, it can't be determined how many tasks have to be created.

Listing 5.9: Usable μ Streams pragmas

```
1 void read_input(int values[128]){
2     for(int i=0; i<128; i++){
3         values[i] = PERIPHERAL.
4             readNext;
5     }
6 }
7 void filter_input(int values[128]){
8     for(int i=0; i<128; i++){
9         if(values[i] > 200)
10            values[i] = 200;
11    }
12    #pragma microstreams task
13    for(int i=0; i<128; i++){
14        if(values[i] < 10)
15            values[i] = 10;
16    }
17 }
18
19 int calculate(int values[128]){
20     int result=0;
21     for(int i=0; i<128; i++){
22         result+=values[i];
23     }
24 }
25
26 #pragma microstreams task
27 void output_result(int output){
28     printf("Output: %d", output);
29 }
30
31 void main(){
32     while(1){
33         int values[128];
34         int output;
35         read_input(&values);
36         #pragma microstreams task
37         filter_input(&values);
38         #pragma microstreams task
39         replicate(2)
40         output=calculate(&values);
41         output_result(output);
42     }
```

Beside the pipelined execution model, it is also possible to make pipeline stages superscalar with the replicate syntax as seen in line 38 of Listing 5.9. The previous pipeline stage alternates between the replicated pipeline stages for each data set passing through the pipeline. This is especially helpful for pipeline stages whose source-code cannot be further pipelined, but impose the longest running stage.

5.6.2 Unsupported Constructs

Due to the freedom of the C-language there are some language constructs and programming techniques which will result in a corrupt parallelized design. Even though these constructs might work, many are considered bad practice. Unsupported constructs and why they are or cannot be considered are listed in the following:

Static variables are variables having the `static` keyword at their declaration. Effectively, these are only valid in their declaration scope but treated like global variables. Defined inside a function, they keep their value among different calls of the function. Considering a scenario where the static variable is declared in one pipeline stage and then modified in a later pipeline stage, the value of the static variable would have to be passed back to the declaring pipeline stage before it is executed the next time. Thus, the processing pipeline would have to be stalled and forwarding communication infrastructures would be needed. Since this would drastically degrade the pipeline performance, this is not implemented. An error is shown if a static variable is used in more than one task. The user can ignore this error and manually handle it after parallelization.

Writing global variables is not supported, even though reading is. The main difficulty is that the variable can be written by multiple pipeline stages. The first pipeline stage holds the first variable modification v' , the second stage holds the second modification v'' and so on. Having a single common memory, the different pipeline stages write and read the modifications $v', v'' \dots$ simultaneously, even though in the non parallelized version the modifications would be written and read one after another. Thus, the global variables would be in an inconsistent state. A solution would be to have a big global memory that holds the global variable for each pipeline stage and transfer the variables to the next stage when all pipeline stages finished execution. Implementing this mechanism would result in a big hardware overhead, especially considering a giant crossbar for all pipeline stages. However, the described mechanism is very similar to core-interconnects transferring the variable states from one stage to the next. Thus, if possible the global variable is privatized and transferred like other variables. An error is shown if privatization is not possible.

Pointer modifications are sometimes hardly comprehensible in a static analysis. For example if the pointer address is modified to a different variable address. The other variable could be modified without being noticed in static program analysis. The only option to track the addresses and memory changes is during program execution. The user has to track and avoid this behavior. However, such pointer modifications are anyways considered bad practice. Nevertheless, pointers whose address is only assigned once impose no problem.

Pragmas inside loops are not possible since they would create a backward dependency. A solution is to partition loops with the LoopOptimizer (Section 5.4) and then put pragmas between the partitioned loops.

ISRs are not directly supported. μ Streams is not able to infer to which core different service routines should be applied. Also, applying ISRs to pipeline stages might lead to an inconsistent pipeline stage duration if the ISR is triggered. One option would be to copy the ISR to a pipeline stage whose execution time including the ISR execution is below the execution time of the critical pipeline stage. However, this is not always possible, especially in a well-balanced pipeline and ISRs could be triggered very often slowing down the pipeline stage. The chosen solution is an ISR core, excluded from the pipeline, handling all ISRs. This is not always possible, since ISRs might have

dependencies to data in other program parts. Thus, μ Streams gives a warning if ISRs with data dependencies are detected in the source-code and the user can copy the ISR functions to the desired core.

`goto` statements are not supported if the jump label is located in another task. An error is given if such a case is detected. However, `goto` statements are anyways considered bad style and should be avoided.

Task annotated functions are functions which have a task pragma before the function definition. The restriction is that this function must be called only once. Usually, a new task per function call has to be created. However, function calls inside loops or recursive functions cannot always be statically counted. Thus, μ Streams gives an error message. The user has the possibility to (re)move the function task pragma. Also, function calls to task annotated functions should ideally be followed by another task pragma. Otherwise, the function task might have a backward dependency to the previous task. In this case, an error is shown but the user is able to ignore the error and resolve the problem manually after parallelization.

5.6.3 Implementation

5.6.3.1 Parsing Source-Files

μ Streams uses Cetus to parse the annotated source-files into an AST. The input files can optionally be preprocessed via an external preprocessor (GNU GCC) which will resolve all preprocessor directives, like defines or includes. A fitting preprocessor is automatically chosen, based on the selected target architecture. Running the preprocessor command could fail, often due to missing header files, like for example `peripherals.h`. These headers are created by the system-builder, containing hardware addresses of peripherals. Thus, this include-file is a leftover of a previous hardware system and will be invalid with the parallelized hardware system. Therefore, such files are mimicked by empty dummies, and corresponding errors are ignored.

5.6.3.2 Source-Code Partitioning

All μ Streams pragma (`#pragma microStreams task`) occurrences are searched on the created AST. Different actions are taken depending on the succeeding elements of the pragma:

Function definition: For each according function call occurrence in the program a new task containing the function call is created, if the number of calls can statically be analyzed.

Statement inside a function: A new task is created containing a copy of the traversable(s). Traversables are added to the task until:

- a new task pragma is found
- the scope that the pragma is declared in ends
- a function call leading to a function definition with a task pragma is found

Each task pragma can be equipped with an optional replicate annotation. All tasks are searched for a replicate annotation. Depending on the amount of specified replica, clones of the current task are created.

5.6.3.3 Tracking Source-File and Function Dependencies

A function-call-graph is needed to know which task requires which function. A function-call-graph is generated for each task's traversable. All found function calls are stored in the task's data structure. In the end, distinct source-code folders are generated for each task. Only necessary functions and declarations are copied to the main.c-file. Functions originally included via the `#include` directive are copied as is to the new source-code folder.

5.6.3.4 Creating a Processing Pipeline

To transform the tasks into a processing pipeline the relationship between all tasks is analyzed. Each task's traversable is searched for nested tasks. All tasks are then organized as a tree and each task stores its parent and child tasks.

5.6.3.4.1 Variable Access Analysis: After having created all tasks, it is highly likely that one task uses a variable that is also used inside another task. Thus, this variable would create a dependency between both tasks. Before analyzing the dependency, it has to be known which task uses which variable and if it is read and/or written. Thus, all used variables, accesses to these variable inside each task's traversable and called functions are analyzed and saved for each task. If variable accesses occur in a nested task or a variable declaration is global, it is flagged as such. Also, variable accesses in nested tasks are marked as untouched. Furthermore, a write-access to the variable is added if the variable is declared and initialized inside this task. The variables are also marked as read or write if the user specified required or provided variables via pragma extensions.

In a second step, each found variable access is classified as read, write, read&write. This is done by ascending the AST from the current variable and searching for access patterns. For example a write access is assigned if a parent in the AST is an assignment expression (`foo=bar+1`) and the variable appears on the left side of the assignment. An appearance on the right side of the assignment returns a read. For a unary expression (`var++ ...`) read&write access is set. The usage as an argument of a function is a read access, except if the argument is a pointer, then it is a read&write access. Setting the access to read&write on pointer types is very pessimistic but necessary since a static analysis of the pointed value, especially if it is modified, is hard and sometimes impossible. If the AST tree reveals different types of accesses for a variable in a statement, they are finally combined.

5.6.3.4.2 Variable Privatization: If a global variable has been declared, but it is used in a single scope by multiple tasks, the variable is transformed to a local variable. Variable privatization of global and local variables is later implicitly done when generating the parallelized source-files. Since the global memory is likely to become the performance bottleneck it is avoided whenever possible.

5.6.3.4.3 Control-Flow Graph: With the current information, it is only possible to tell if a statement in the program is located before or after another statement in the source-code but not if the statement might also be executed before or after this statement. Therefore, a CFG for each task and all functions is necessary. The CFG later allows a prediction of where a variable was previously and will later be used in the program flow.

The CFG is composed of different nodes containing one statement of the tasks/functions traversable. The next node containing the next statement (usually the next line in the source-code) is appended to the current node via an edge. Some nodes containing for example the head of a loop (respectively foot for do while loops) have two next nodes, depending on the loop condition evaluating as true or false. Through occurrences of loops, the CFG will also contain loops. Also, the generated nodes sometimes hold additional information of the contained statements, like if this node is a graph start or end node or transition to a new task is done. Task nodes will be created if a task pragma is found inside the traversable of the current task. The nested task's control-flow will not appear in the parents CFG.

Since the CFG should be used to predict the variable dependencies in different tasks, it's straight forward to put the extracted variable access information also into the CFG nodes.

5.6.3.4.4 Creating Task Dependencies: Until now, it is only known if a task is nested in another task. However, the order of multiple nested tasks is not known yet, but needed to successfully create a pipeline. The generated CFG delivers all necessary information through the order of the graph nodes and the variables read/written in the nodes according statements.

The CFGs of the tasks are traversed for each variable that the task uses. If a variable is accessed in another task, a task dependency is created. For read variables, only dependencies to previous read accesses in the CFG are created. For written variables only read accesses to succeeding elements in the CFG create dependencies respectively. Write after write accesses and read after read accesses are not considered, since they either overwrite data unconditionally or do not change the data. An exception to this are arrays and structures, since both have multiple fields which in this case do not appear as distinct accesses. It is also not possible to analyze the accessed array offset statically if it depends on another variable. This means that any write after write access on arrays and structures in another task is also modeled as a task dependency.

5.6.3.4.5 Pipeline Formation: Now, task dependencies are modeled with respect to variable usage. However, the dependencies are not yet ensured to have the desired pipeline structure. The dependencies could even form loops which would create counterproductive parallelized systems that are slower than a single-core implementation. Thus, the dependencies have to be transformed into a pipeline structure and scenarios where a parallelization is not applicable have to be detected.

Listing 5.10: Example code to visualize task pipeline creation

```
1 foo(int *x){
2     (*x)++;
3     #pragma microStreams task
4     bar(&x);
5 }
6
7 main(){
8     int x = 5;
9     int y = 2;
10    #pragma microStreams task
11    foo(&x);
12    #pragma microStreams task
13    bla(&x, y);
14 }
```

Figure 5.10a shows the created task dependencies from the source-code in Listing 5.10. Each task has a dependency to the parent task they are nested in, even though the parent task might not further touch the variable and only forwards it (dependency variable x task $2 \rightarrow 1$, $1 \rightarrow 0$, $0 \rightarrow 3$). This would mean that task 0 in the example needs to wait idle for task 2 to finish, which effectively makes launching task 1 and 2 counterproductive. Task 0 could have done the same job in its idle time. These forward only dependencies are detected by looking in the CFG if the intermediate nodes between task nodes only hold annotation statements (comments or task pragmas). In the example in Figure 5.10a the task dependency $2 \rightarrow 1$ and $1 \rightarrow 0$ of variable x would be replaced by a dependency from task 2 to task 0. In a second iteration the task dependency $2 \rightarrow 0$ and $0 \rightarrow 3$ is converted to a $2 \rightarrow 3$ dependency. The process is repeated until no further shortcuts are found.

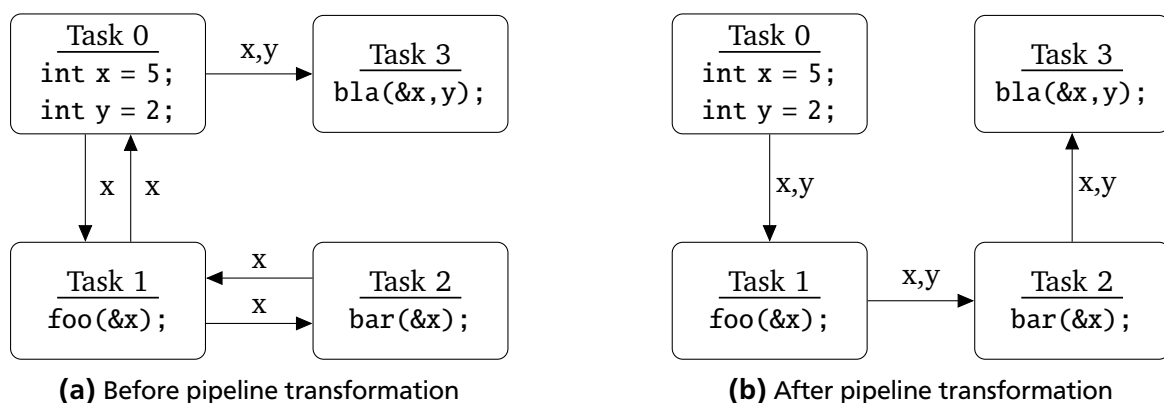


Figure 5.10: Task dependency created from Listing 5.10

Now that the dependencies of variable `x` have been transformed into a pipeline, variable `y` imposes a different kind of dependency. Considering synchronization between tasks happens more or less synchronous and task 0 receives/generates new data sets each time, such that the new data set is called d and the data transferred to the task 1 becomes d' , and task 2 receives d'' and so on. In the example, task 0 would be able to transmit `y` directly to task 3 and task 2 would transmit `x`. This would however result in a false output, since task 3 would get d from task 0 and d''' from task 2. Thus, the variables have to be tunneled through the pipeline until they reach their destination task to maintain timing of the different data sets.

5.6.3.5 Setting up Inter-Core Communication

Now that the task dependencies are transformed into a pipeline with optimized predecessor and successor tasks, appropriate core-interconnects based on the communication pattern have to be chosen. This is either a Core-Connector, a MemSwapDual or a Mailbox for one predecessor and Concentrator/Dispatcher, MemSwap Multi or Multiple Mailboxes for multiple predecessors/successors. Also, if a task contains globally read variables by multiple cores, a global memory is instantiated. Mailboxes are chosen when the target system is MicroBlaze. Core-Connector, Dispatcher and Concentrator are chosen when the target system is SpartanMC and no DMA interconnects should be used. MemSwap modules are chosen for SpartanMC targets with DMA interconnects enabled. The procedure of assigning the connections is simply by iterating through all tasks and requesting their predecessor tasks. All transmitted variables are attached to each connection.

5.6.3.5.1 Transferred Variable Alignment: All used core-interconnects transmit data in words and the core-interconnect drivers work very efficiently with load and store instructions on word-width. However, it could happen that a variable with a smaller data-type (in the following example a `char`) than the word-size has to be transmitted. By default, the compiler doesn't naturally align the `char` to word addresses. Thus, when using the drivers with a `char`, it could happen that the `char` is located in the upper half-word in the sender's firmware. The `char` is transferred into the core-interconnect's word-wide register's upper half word by the driver. The driver writes the core-interconnect's register contents also in word-width, if the receiver reads the `char`, a pointer to the `char` is handed to the driver. The assigned receiver's `char` value is not the sender's `char` value, but the memory content below the sender's `char`, if now the receiver's `char` declaration is located in the lower half-word by the driver.

Another problem resulting from there is, that the receiver also overwrites the other half-word in the memory next to the received `char` due to the word wide load/store in the driver. This is a potential source of error. The solutions on this problem are:

Align everything on word bounds: This potentially wastes memory.

Put send/received variables in a struct: All variable accesses will need to be replaced by the struct access. It has to be paid attention, that the structures size ends at a word boundary.

Chars become Integers: Operations counting for example on a char overflow would not work anymore.

It was decided to wrap all send and received values smaller than a word into a struct, since this method has compared to the others no obvious disadvantages.

5.6.3.5.2 DMA-Interconnect Variable Preparation: The DMA core-interconnects have BRAMs mapped into the address-range of the processor. The DMA interconnects are leveraged in μ Streams such that all transferred variables reside in the interconnect's DMA section. During the computation phase, these variables can be read and modified. After the computation is done, the DMA interconnect gets the signal for transferring the DMA section. To tell the compiler that the respective variable should reside in the according interconnect's DMA section, the compiler directive `__attribute__((section(".dma.*peripheral_name*")))` is appended to the variables. However, the compiler has the freedom to choose the variable's memory locations inside this section. This is not necessarily the declaration order. During memory switching, the location of each variable is implicitly assumed. Thus, these variables must be declared in a struct in the interconnect's DMA section, to fix the location and order of the variables. This procedure allows transferring large variables to the next core within just a few cycles. If a core receives variables, modifies them and then has to forward them again to the next core, a memory copy from one receiving interconnect's DMA space to the sending interconnect's DMA space has to be executed.

Since all sent/received variables now reside in a struct, their original declaration is deleted and all accesses to the variable are replaced by the access to the variable inside the struct. At first, it has been assumed that the additional wrapping in a structure would cost performance through less efficient code. Therefore, a minimal test with a structure of 16 integers which are summed up was created, it became obvious that accessing the elements through the structure produces fewer instructions thus more efficient code. The compiler was able to use offsets for the load instruction, which was not used in the other case. With these results, all used variables in the IIR benchmark single-core variant were packed in a structure and the execution time was measured. This revealed that the variant with all variables in structures produced around 3% faster code. However, this behavior might change in newer GCC versions or other test applications (tested with SpartanMC GCC version: 7.1.0).

5.6.3.6 Used Hardware Detection and Instantiation

The peripheral detection described in Section 5.7 allows inferring used peripherals through analyzing a Cetus AST. The peripheral detection analyzes each tasks traversable separately, returning the used peripherals of each code section. The found peripherals are attached to each task for later hardware generation. After the analysis, a sanity check analyzes that each peripheral is only used in one processor core. If the same peripheral is used in multiple cores, the program most likely will not work as expected by the user (a more detailed explanation of this limitation is given later in Section 5.7.4). An error is given which in turn can be explicitly ignored by the user to manually handle the problem later.

If the user has selected a parallelization with performance evaluation, additional peripherals and hardware configurations are required in order to analyze different execution phases of the parallelized program:

Receive application-state: The processor receives the used variables from its predecessor in the pipeline

Process: Run the application with the received data.

Send application-state: The processor sends the used variables that are needed in later application parts to the next processor in the pipeline.

Listing 5.11: Simplified abstract XML hardware description

```
1 <hardware>
2   <subsystem>
3     <core name="spartanmc_0" type="spartanmc" firmware="core0" globalMem="true" perfCounter="true"/>
4     <peripherals>
5       <peripheral name="spartanmc_0_core_connector_master_0" type="core_connector_master"/>
6       <peripheral name="spartanmc_1_uart_0" type="uart"/>
7     </peripherals>
8   </subsystem>
9   <subsystem>
10    <core name="spartanmc_1" type="spartanmc" firmware="core1" globalMem="true" perfCounter="true"/>
11    <peripherals>
12      <peripheral name="spartanmc_1_core_connector_slave_0" type="core_connector_slave"/>
13    </peripherals>
14  </subsystem>
15  <wiring>
16    <connection>
17      <peripheral name="spartanmc_0_core_connector_master_0"/>
18      <peripheral name="spartanmc_1_core_connector_slave_0"/>
19    </connection>
20  </wiring>
21 </hardware>
```

To make these measurements, each processor's performance-counter has to be activated. In case of the MicroBlaze timer peripherals are added to each core. The generated code will start and stop the performance-counter/timer before and after each execution phase. To present the measured results to the user, they are collected on one processor and printed via UART. Thus, a UART peripheral is added to the first processor in the pipeline. The result collection is done through a global memory attached to each core, where each core stores the measurement results. The result collection could also be done through a concentrator, but the global memory was chosen since it's slightly easier to handle in software.

Since most information about the generated multi-core system is already present, the idea is to generate an abstract system description which can then be imported by a system-builder. A simplified hardware-XML example is given in Listing 5.11. The description holds the generated cores (subsystems), along with the used peripherals found through peripheral detection. Additionally, the chosen inter-core communication is added to each subsystem. Properties in the XML core entry tell whether performance counters for a performance evaluation shall be enabled and if the core should be connected to a common global memory. The wiring entries reflect the communication relation between the different cores. Thus, which core-interconnects shall be connected. The system-builder jConfig has an interpreter plugin to read this file and instantiate an according configuration. jConfig has automation routines for many common peripherals that configure and wire the peripherals based on the environment. In ideal case, the user will just review the configuration and confirm it to generate and build the Verilog top-level design for the designed system. However, the automation routines cover default values for common cases, the user is required to manually modify for more exotic settings.

5.6.3.7 Software Generation

Finally, new firmware source-files for each created task are written. This includes writing a new main.c for every task in a new source-code directory. Only needed headers and C-files like previously collected (Section 5.6.3.3) are copied to a newly created firmware directory for each core. The natural way of generating the C-files would be to use Cetus which is the case for all non main.c files. However, it is evident that generating a new C-File in the required structure has a lot of programming overhead in

Listing 5.12: Simplified main.c Freemarker task template

```
1  #include "main.h"
2  <#list include_files as include> ${include.print()} </#list>
3
4  #define CORE_ID ${core_id}
5
6  <#list task.localFunctions as function> ${function.definition}; </#list>
7
8  <#if containsISRs> ${printISRs} </#if>
9
10 void main() {
11     <#if isPerformanceEvaluated> initPerfEval(__GLOBAL_MEM_HEAP_END); </#if>
12
13     while (TRUE) {
14         //rec/send variables
15         <#list task.getUsedLocalVariables() as variable> ${variable.getDeclarationWithAssignment()}; </#list>
16         <#list task.getBody() as body>
17             <#if body.type == BodyType.CODE>
18                 ${body.getCode()}
19             <#elseif body.type == BodyType.CONNECTION_PERIPHERAL>
20                 <#include " ${body.type}_template.ftl" >
21             ...
22             </#if>
23         }
24     }
```

Cetus. Generating even simple things like for example a function call requires much effort since Cetus requires that all AST components are properly linked. After the software generation, the AST is not touched anymore by any other transformation pass. Thus, a thorough build AST is not that important anymore. It was decided to use Apache FreeMarker³⁵, a template engine, for generating parallelized source-code. This results in a template main.c file which will be filled with necessary information from the Java objects. The advantage of this method is that static code contained in all generated main-files can just be written down and the structure of the program is widely visible to the developer. A simplified template for the main.c file is shown in Listing 5.12. FreeMarker commands are initiated with `<#cmd>` and ended with `</#cmd>` references to java objects outside FreeMarker commands are initiated with `${object.getString()}`. Thus, functions can easily be created through calling the `toString` method on the Cetus AST traversables or includes by generating appropriate strings within Java.

Another important part is the endless while loop's body creation. In Listing 5.12 this is only hinted in line 16 through `task.getBody()` and thus retrieving a list of typed bodies that are treated differently. The generation of these bodies is done through the so called *BodyBuilder*. The *BodyBuilder* gets a task with all previously collected information sorts out the necessary components that must go into the template by type. Additionally, the bodies also have some content that is either a string to be directly pasted into the template (Line 18 Listing 5.12) or further properties that will be for example queried by other templates (Line 20 Listing 5.12).

5.6.3.8 Parallelization Visualization

The visualization was mainly designed to reflect the generated pipeline structure along with the exchanged variables between the tasks for debugging. Otherwise, the user will have to open all generated

³⁵ <https://freemarker.apache.org/>

firmwares to compare send and received variables and the hardware-XML to see hardware and communication structure. An example output is given in Figure 5.11.

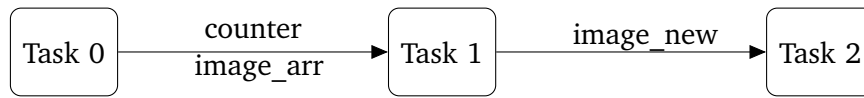


Figure 5.11: Generated pipeline structure and communication

5.7 PeripheralDetector

Note: Parts of this section have already been published in [95]. Self-citations are not marked in order to improve the reading flow.

The purpose of the peripheral detector is the inference of a hardware configuration from source-code. Peripheral detection becomes possible since hardware is usually interfaced through a software API which can be categorized. Through the usage of soft-cores on FPGAs, peripheral components are not fixed, but can be instantiated as required. It is possible that the user writes an application and a fitting hardware infrastructure is generated automatically, which had to be done before manually. Since not all configuration information can be drawn from the application, a good system-builder is needed that connects and configures components intelligently with default configurations.

Besides peripheral inference, the tool is also usable to detect in which part of the source-code a peripheral is used. This allows the parallelizer to conclude how an application with peripheral usage can be parallelized. Possible peripheral usage from multiple concurrent cores has to be handled.

In contrast to the other presented tools, the PeripheralDetector is not a standalone tool and thus can only be used as a library for example in a transformation pass in μ Streams. The peripheral detector could only be used as a standalone tool when using μ Streams without pragma annotations.

5.7.1 Workflow

To demonstrate the different usage methods of peripherals in SpartanMC (and many other SoC-Kits), a small code example is provided in Listing 5.13. The example shows the usage of interrupts, a USB interface as DMA peripheral and UART, SPI master and I2C master as memory-mapped-peripherals. The sample application covers all types of peripherals that are possible in this environment. The example also shows the different types of how peripherals can be interfaced: Comfortably through a driver function call (line 10/12) or directly through the defined custom peripheral structure on hardware registers (line 8).

Listing 5.13: Usage of differnt Peripherals in a SpartanMC C-application

```
1  extern void heavyProcessingTask(int* data);
2
3  FILE *stdout = &UART_LIGHT_0_FILE;
4
5  void main() {
6      while(1){
7          interrupt_enable();
8          USB11_0_DMA->data01[0] = 16;
9          int data[1000];
10         while(i2c_master_readn(I_SQUARE_C, 1,1000,&data));
11         heavyProcessingTask(&data);
12         spi_master_write(SER_PERI_IF_0, &data);
13     }
14 }
15
16 //ISR0 interrupt
17 void isr00(void) {
18     printf("HELLO");
19 }
```

The peripheral detection could work on a Cetus generated AST of an application. The provided AST can either be a full C-program or a part of the program i.e. a branch of the AST, such as a procedure or a basic block.

The proposed workflow can be seen in Figure 5.12. To detect peripherals, some knowledge of all available peripherals is needed. This information can be found in the peripheral module XML-descriptions. For each component in the system-builder jConfig, a XML hardware description must exist. This module description includes besides other information:

- Module name
- HDL sources
- Parameters (for configuring HDL sources)
- HDL input/output ports
- C-header to interface peripheral
- C-struct name

Especially module-name, C-header and C-struct can be used for peripheral detection. In jConfig the peripheral address is aliased with the module name by default in the generated C-headers. This alias can comfortably be used in the application. However, the peripheral name and thereby the alias of the peripheral can manually be changed by the user. Furthermore, the C-struct holds the struct type that is used for the alias definition. The C-header holds all function declarations that can be used to interface the driver of the module/peripheral. This information can be used for inferring peripherals from the AST.

If a peripheral has been detected in the given AST, a Java object will be created for each peripheral. These peripheral objects can then be used in the program, calling the PeripheralDetector.

The peripheral detector has originally been developed for SpartanMC, but it is also usable for MicroBlaze. The jConfig system-builder has XML module descriptions for many MicroBlaze peripherals and the processor itself. The systems can be configured inside jConfig which will generate a Vivado project and a respective system configuration through the Vivado TCL interface.

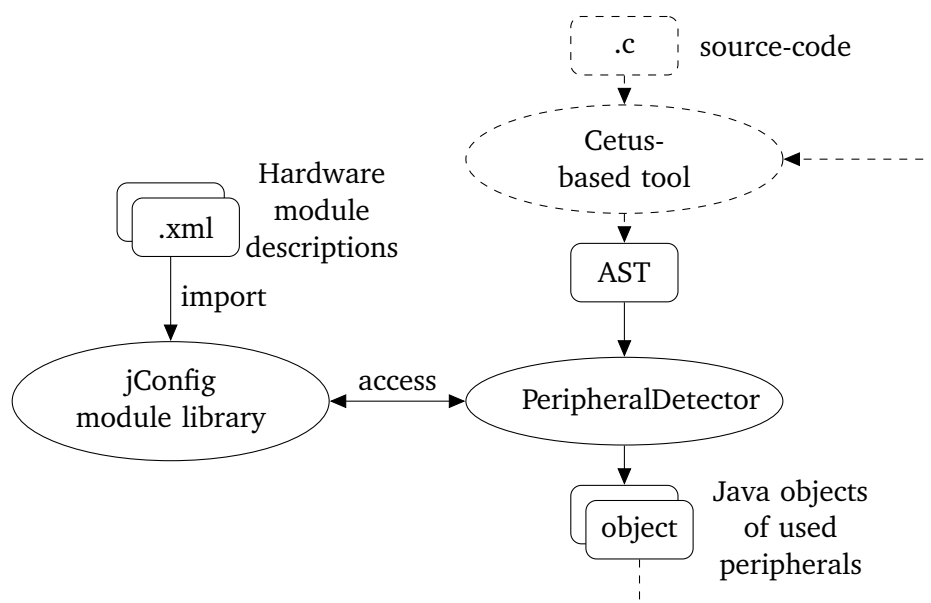


Figure 5.12: Detailed Peripheral-Detector toolflow (dashed=optional)

5.7.2 Implementation

5.7.2.1 Approaches for Peripheral Detection

The peripheral detection can be done in three ways:

Matching names of constants: As can be seen in the example (Listing 5.13), one has the possibility to access the peripheral directly through a constant pointer/alias to the according memory-mapped-peripheral address. The constants' names most likely contain the name of the peripheral and thus it is possible to distinguish between different peripherals. However, since the constants' names can be arbitrarily changed by the user, this method is only reliable if the constants' name contain a name of a peripheral unit.

Analyzing includes: This is also a very easy technique to identify which peripherals are used. Unfortunately, sometimes includes are specified but never used in the code or several peripherals share the same include file. Thus, this method is not used here.

Matching API function calls: Most peripheral units have a driver API which gives easy access to its functionality. With a mapping between API functions and peripherals, a peripheral can be distinguished by looking at the API calls. This method is more reliable compared to constant's name analysis since, if an API function is called it is very unlikely that the corresponding peripheral will remain unused.

5.7.2.2 Detection Algorithm

As previously described, two methods for peripheral detection have been chosen: API function call and constants matching.

5.7.2.2.1 API Function Call Matching:

- At the beginning the XML descriptions of all peripheral modules are collected and loaded into a library to easily access all components.
- In the next step all function calls in the program are searched in the AST provided by Cetus. A function call that is not resolvable in any C-header-file in the project directory is assumed to be a system function call.
- Now, all functions declared in the peripheral's header files or respectively in the API of the peripheral are fetched. For this purpose all header files from all peripheral libraries are parsed in a new instance of Cetus. All unresolvable function calls will now be matched with the system library in the new Cetus instance. The matching of course considers the function name, argument number and types.

5.7.2.2.2 Constants Matching:

- In the first step the AST provided by Cetus is parsed. All peripherals are usually accessed through constants, and they are replaced by the memory address through the preprocessor during compilation. The constants are defined in an automatically generated header-file from the system-builder. The constants are generated by the names of the peripherals to the upper case. Since the system-builder will run after the peripheral detection, the constants for the peripherals are not yet defined. Thus, all undefined constants can be collected from Cetus. As an optimization, all peripheral constants/names already found by API function call matching are neglected.

-
- For each undefined constant found, first a check is performed if the constant follows the naming conventions for peripherals (all uppercase letters). Then the peripheral names from the library are searched, to determine if the constant contains the peripheral name.
 - After the previous step either one or more peripherals can be found. If for example the constant is “UART_LIGHT_0” and there is a module named *uart* and one named *uart_light* in the library, there would be two matches and a decision which one to prefer is needed.
 - If there are more than one peripheral modules matching, the peripheral name having more successive matching characters in common with the constant will be preferred.
 - In the last step the matching peripheral is added to a list of found peripherals.

5.7.3 Sources of False Detection

In the following scenarios a (correct) peripheral detection is not possible and an appropriate warning message will be printed:

Non inferable constant name: If a constant is only directly used via the peripheral’s structure and the constant is named such that it does not match any predefined peripheral name, a detection is not possible. A warning for this scenario will be created in every case, since all defined constants except for the peripheral constants must be resolvable in this step. The user can either modify the constant’s name to properly match against a peripheral name or manually add a peripheral in the system-builder. Alternatively, if the defined constant is not intended as a peripheral a proper `#define` should be created in the source-code.

Defining variables from a constant: If the constant’s address is assigned to a variable, the peripheral detector sees the variable as an alias to that constant. The variable can then be modified in the following code to point to a different address, which can lead to a false peripheral detection. A warning for this scenario can be given if the aliasing variable is modified in the following code. The modified variable however could or could not be a valid alias. Even though no modification happens on the variable itself, it could be changed through direct memory modification. Thus, a certain detection of this scenario is not possible. However, such code only appears if the programmer violates peripheral usage conventions through the freedom that the C-Language gives. Such techniques are as well commonly considered as bad style and not recommended.

Duplicate recognition: A constant is used in a peripheral API call but the constant is matched by its name against a different peripheral. In such a case the function call detection takes precedence. An unwanted name clash with an existing peripheral is more likely than the correct usage of a peripheral’s API with a pointer to a different peripheral type. The warning in this scenario can always be certainly given. The user is prompted to change the constant’s name for the sake of a more readable source-code.

5.7.3.1 Detection Accuracy

Several legacy projects have been used to test the accuracy of the detection algorithm. Most legacy projects used two to four different peripherals with UART being most frequently used. Table 5.2 shows that all detected peripherals were correct. Even though, it is possible to construct scenarios where a peripheral detection is not possible, results show that this rarely happens.

Table 5.2: Detection accuracy with different applications

legacy application	detected	peripherals used types
Hello World	1/1	UART
Firewall	2/2	Ethernet, UART
VP8 Decoder	4/4	USB, UART, DVI, DDR
MJPEG2000	2/2	Timer, UART
MD5 hash	3/3	2x Timer, UART
SHA256 hash	2/2	UART, Heartbeat
Proximity	4/4	UART, ISR, SPI, I2C

5.7.4 Automatic Peripheral Detection on Multi-Core Systems

Since the peripheral detection can be used in μ Streams, it is possible to automatically detect peripherals and parallelize the source-code in one process. To accomplish this, the application source-code will first be split up based on the annotations. μ Streams provides access to the AST of each task. So instead of parsing the whole input code, the parsing will be restricted on a task basis and peripherals are attached to each task. Each task is mapped to one processor core in μ Streams. If a peripheral is not exclusively used by one core, there are different policies which could be pursued:

Map all tasks sharing a peripheral to the same core: This policy would break parallelization of tasks on many levels. First of all, the increase in performance achieved through the constructed pipeline is likely to be degraded since the slowest pipeline stage dictates the operating speed of the whole pipeline. Secondly, only adjacent tasks in the pipeline could be merged otherwise the pipeline can not work properly. Thirdly, in the worst case all tasks would be mapped to one core and thus the parallelization would be lapsed.

Multiplex peripherals: For a few peripherals this policy might be a valid approach. Yet, not all peripherals would deliver the intended behavior when accessed from multiple cores as when accessed by one core.

Rearrange the source-code: Since the before mentioned methods have major drawbacks or do not work in all cases, fitting the hardware to the software is not automatically solvable. The only possible method is to restructure the source-code or the parallelization. The following possibilities exist to resolve the issues:

- Use several instances of one peripheral type to regain exclusive access per task.
- Rearrange all peripheral accesses to be within one task through statement reordering or different parallelization choices. Necessary data could be communicated between the cores.

A case that needs special handling is the peripheral usage in an ISR. Since the ISR is called from outside of the program flow, the used peripheral inside it cannot directly be associated with an application part reached through the program flow of the main function. In such a case, the peripheral is associated with the function call to enable the ISR, since this call will appear in the program flow.

6 Evaluation

The automatic parallelization is evaluated in the following. Firstly, the used applications and their characteristics are described, along with generated application profiles. Secondly, the automatic parallelization is inspected and the different optimization steps, as well as parallelization possibilities enabled one by one. This gives detailed insights into optimization decisions by AutoStreams. In the next step, two real-world systems with actual peripheral interaction are evaluated to see performance as it would be in a real system. Afterwards, the quality of the solution provided by AutoStreams is evaluated through the achieved prediction accuracy and an automatically found solution is compared with a manual parallelization. The influence of multiple cores on the achievable clock frequency and application latency is also evaluated. Last but not least, a comparison with related approaches is exercised and best practices are proposed with the conclusions from the evaluation.

6.1 Test Applications

In total, four different applications are selected for parallelization: ADPCM, MJPEG2000, IIR Filter and a Firewall. These four applications cover a range of complexity as well as data sizes. The handled data size defines the communication overhead and the according computation complexity if parallelization is beneficial.

The chosen benchmarks like ADPCM, JPEG compression and digital filters are contained in many benchmark suits like: CHSTONE[92], MiBench[110], Powerstone[111] and others.

However, the aforementioned benchmark suites contain way more benchmarks than the selected ones. Other benchmarks from these suites were not chosen due to various reasons:

Complexity: For example the PARSEC benchmark suite[91] is especially developed to be parallelized for multi-core processors. The presented benchmarks including their data sets are often too large for low-performance embedded systems. The necessary memory cannot be provided. Another counter example are low complexity benchmarks like "bitcount" or "qsort" in MiBench. If an application only has few instructions, coarse-grained pipeline parallelization might not be beneficial. Retrieving the necessary data might take much longer in an embedded system compared to the computation. It is also barely thinkable that an embedded system just executes bit counting. These trivial applications would rather be implemented as one of many intermediate steps in a real system.

Pipeline parallelism: The application has to exhibit pipeline parallelism. Thus, different loosely coupled processing steps one after another. Some applications directly exhibit this kind of structure and for some it is hidden and cascaded due to the program structure. The application would need restructuring or AutoStreams would need better mechanisms to restructure the application or detect pipeline parallelism in these scenarios.

32 bit focus: Many reviewed applications implicitly assume 32 bit architectures due to their data structures and bit shifts with overflow. The 32 bit focus is fine for MicroBlaze, but not for the 18 bit SpartanMC processor and many other embedded eight and 16 bit architectures. Thus, it is necessary to port these applications to make them agnostic to target bit-width.

Thus, it has been decided to stay with a small but representative number of benchmarks and analyze those in more detail than it would be possible with numerous benchmarks.

6.1.1 ADPCM

Note: Parts of this section have already been published in [106]. Self-citations are not marked in order to improve the reading flow.

ADPCM is a compression approach used in many places like ITU audio codec G.726 or for signal compression in wireless sensing applications. The encoding procedure is used here, as it consumes more processing power.

ADPCM is based on differential pulse code modulation, where only the difference between consecutive values is transmitted (together with one initial absolute value). Due to the continuous nature of most signals, this leads to a reduced variance of the transmitted values and thus to smaller codes (given that differences are efficiently encoded, e.g. with Huffman encoding).

ADPCM increases the efficiency of this encoding process by adding a prediction mechanism. Extrapolating the past samples, a prediction for the next value is made. Instead of transmitting the difference between samples, now the difference between the prediction and the sample is transmitted. This difference is also called prediction error $d_i = x_i - \tilde{x}_i$. The prediction is calculated with a prediction filter of order M with the coefficients $a_i, i \in 0 \dots M-1$ and the previous samples x_{i-M}, \dots, x_{i-1} :

$$\tilde{x}_i = \sum_{k=0}^{M-1} a_k \cdot x_{i-M+k}$$

The decoder needs to know the first M samples and the filter coefficient. Thus, they are transmitted directly.

A block of samples is passed to encoding and for each block, the filter coefficients are calculated in advance. To find the optimal filter, coefficients of the autocorrelation r_k of the current block are calculated. These values are used to build a system of linear equations, whose solution results in filter coefficients that minimize the variance of the prediction error sequence (d_1, \dots, d_N) [112].

After computation of the prediction parameters and transmitting the first M samples directly, the remaining samples can be encoded by computing a prediction for each sample, computing the prediction error as the difference between the sample and the prediction and then finding a clever binary representation for the prediction error as described in the following. The prediction error sequence is mapped from signed to unsigned integer via code spreading.

These unsigned integer values are coded with Golomb-Rice-Coding which is very effective for input streams in which small values are more probable than large values. This is the case for the prediction error sequence for input streams like audio signals which can be predicted effectively.

6.1.2 MJPEG2000

Note: Parts of this section have already been published in [101]. Self-citations are not marked in order to improve the reading flow.

To implement a simple Motion JPEG 2000 encoder³⁶, each single picture of the raw image stream is compressed with the JPEG 2000 encoder and appended to the compressed image stream. In Motion JPEG2000 no inter-frame coding exists.

As a code base we used the Honeywell Versatility Stressmark[113], which implements a JPEG 2000 encoder. The following steps clarify the computational effort and data access on the image during encoding (referenced image tiles can be seen in Figure 6.1):

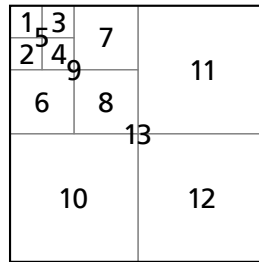


Figure 6.1: Image tiles as processed by the JPEG 2000 encoder

Discrete wavelet transformation (DWT): The wavelet transformation is used to shift relevant image information (low frequencies) to the upper most left part of the image, while less relevant image data (high frequencies) are moved to the right and down. This is done in several recursive steps. First tile 13 is convoluted with the wavelet. This means that the tile is analyzed row by row and low frequency parts are stored in the left half of the row and high frequency parts in the right one. This procedure is repeated on the resulting image for each column and tile 9 and 5.

Quantization: After the DWT, the resulting values are quantized. First, all values of tile 1 are searched for maximum and minimum values. Based on these values, individual quantization steps are chosen and the whole tile is scanned again, binning the values accordingly. This is repeated with tiles 2-4, 6-8 and 10-12. After these steps, tiles 10-12 are zero.

Run-length encoding: All equal and consecutive values of tile 1 are run-length encoded. The process is repeated for tiles 2-4 and 6-8.

Entropy encoding: For further compression, tiles 1-4 and 6-8 are entropy encoded, giving each run-length encoded value an individual bit pattern varying in length.

6.1.3 IIR Butterworth Filter

IIR Butterworth Filter is a digital Filter implementation based on the IIR example from Atmel Advanced Software Framework³⁷. Compared to FIR filters, IIR filters require less processing power/operations to reach equivalent accuracy.

This example implementation describes a Butterworth IIR high-pass filter of 5th order. The filter expects 48+5 samples of a signal in the time domain as input. The input and the previous output of the filter are both convolved with a static function (e.g. array) representing the desired filter characteristics. The output of the two convolutions are then accumulated to form the new 48 output values.

6.1.4 Firewall

In [86, 114, 89] a network firewall is implemented as another showcase for a possible parallelization. The goal is to filter packets between an internal trusted and an external untrusted network. The firewall works on the network and transport layer of the OSI-Model. The implementation contains a stateless and a stateful filter, which will be explained in the following.

³⁷ [http://www.microchip.com/avr-support/advanced-software-framework-\(asf\)](http://www.microchip.com/avr-support/advanced-software-framework-(asf))

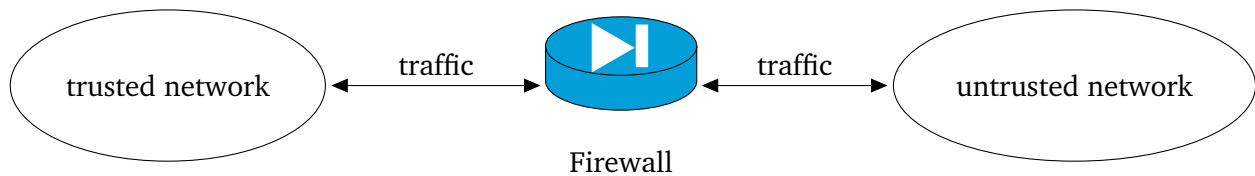


Figure 6.2: Firewall zones

6.1.4.1 Stateless Filter

The stateless filter is implemented as a simple list containing either elements to block or to allow. The following elements can be specified in the filter rules and multiple elements can be combined to form more precise rules:

- Packet types TCP/UDP/ICMP
- Network ports
- IPv4 & IPv6 source and/or destination addresses or address ranges

The implementation compares the received packets header to each entry on the filter list via linear search and discards or forwards the packet accordingly. Whenever a rule matches, the search is aborted and the according action is executed. A default action is executed as specified (Allow/Deny) if no rule matches. With slight optimizations, this is the same behavior as carried out by PF [115], the OpenBSD packet filter and many other firewall implementations.

6.1.4.2 Stateful Filter

As TCP is a connection oriented protocol, each connection to the untrusted zone through the firewall needs a response channel. The open response port is noted in the TCP header and randomly chosen by the connection initiator. To increase security, all incoming TCP packets from the untrusted zone are usually blocked if not explicitly allowed. A stateful firewall keeps track of all initiated TCP connections to the untrusted zone. The opened response ports are managed in a dynamic filter table by the firewall. Closing the TCP connection or a timeout will delete the filter table's entry.

6.2 Application Profiles

Note: Parts of this section have already been published in [99]. Self-citations are not marked in order to improve the reading flow.

A performance-profile of each benchmark is required to parallelize the applications with the method proposed in Figure 5.1. Therefore, AutoPerf is started with the respective firmware source-code folder as arguments. The different computation steps of the benchmarks are all carried out step by step in their *main* functions, which is profiled by default.

The AutoPerf generated abstract hardware description and the instrumented firmware is automatically loaded into the system-builder. Afterwards, the top-level hardware design is generated by the system-builder. This process worked for all benchmarks except the firewall which has a rather complex hardware environment in contrast to the other benchmarks. The user thus has to review the generated design and configure for example the MAC address and connections to the Ethernet chip located on an FPGA Mezzanine Card (FMC) extension board. Alternatively, it is also possible to open a hardware design where the firmware was previously developed on and simply enable the processor's performance-counter.

The hardware is then synthesized and the software compiled and run on the FPGA. Alternatively, the system can also be simulated through HDL simulators if no external interfaces are required or simulation models exist for the respective interfaces.

The performance-profile is then printed via UART and captured by the SerialReader to transfer the profile into a CSV-file. This file can then be handed to AutoStreams to select an appropriate task partitioning.

An embedded system usually has some kind of in- and output, usually a peripheral device with a certain protocol. Since this work aims towards parallelizing the whole embedded application peripheral in- and output is also considered. For the current implementation, a copy-loop for all input data should model data in- and output including a short UART message giving feedback that processing finished. This models peripheral interaction but still allows easier evaluation without providing manual peripheral input each time. The effects of the additional overhead using real peripherals is discussed further in Section 6.5. However, this simplified view neglects environment effects and allows easier detailed evaluation.

The measured performance-profiles can be seen in Table 6.1. The different processing steps are numbered and have representative names. The duration of each processing step is given in cycles since it has little dependence on the achieved clock frequency per FPGA and can easily be scaled. The initiation interval is the sum of all processing steps. This is also the duration after which the application is able to accept new input data. This time is intended to be reduced during parallelization. Thus, speedups take this time as reference.

6.2.1 Benchmark Characteristics

Looking at the benchmark performance-profiles, the steps consuming most processing time are loops like ADPCM steps 1&7, MJPEG2000 steps 1&2 and IIR step 1. The structure of these loops allows processing with the LoopOptimizer. Thus, these steps can be broken down to smaller pieces and don't necessarily impose the critical step of the processing pipeline. The filter loops in the firewall benchmark also consume the most processing time. These loops are cascaded in several functions intermixed with other function calls and thus, AutoStreams will not be able to detect and partition these loops. AutoStreams would have to be extended to accept performance-profiles of multiple functions, and each function would have to be annotated so that AutoPerf profiles this function in a separate run. Even though this might be a useful extension, it increases the burden of the user and is not required for the benchmarks to be parallelized successfully anyways.

After analyzing the benchmarks' data structure, a rough prediction on the communication effort can be made. The computation effort in contrast to the communication overhead defines how efficiently parallelization can be realized.

ADPCM has a 4096 element *char* array as input data, which is used in all parts of the benchmark. Additionally, an 1510 elements output data *char* array is used from step 6 on. Around 10 integers and integer arrays with around 10 to 20 elements are used to store intermediate values of the computation steps.

MJPEG2000 contains a 128x128 sized uncompressed image as input data, resulting in an 16384 element array. The element type is 18-bit wide integer for SpartanMC and 16-bit short for MicroBlaze. The input array is transformed in steps 1-6. Steps 7-9 each generate an 16384 element array with intermediate results that will only be used by directly succeeding steps. The benchmark also supports larger images, but the size of the arrays would exceed SpartanMC's address space.

IIR has an input integer array of 48+5 elements containing fixed point values between zero and one. The input is only used in step 1. The output data is an array of the same size, even though only 48

Table 6.1: Benchmark processing step runtimes in cycles for SpartanMC and MicroBlaze**(a)** ADPCM: In units of 10^3 cycles (rounded)

Processing step	SpartanMC	MicroBlaze
0: receive input	29	36
1: read adaptive input	624	582
2: auto correlation	7	4
3: extract eqn. system	1	1
4: solve eqn. system	139	76
5: back-substitution	9	4
6: write coefficients	1	1
7: compression loop	1060	754
8: write results	11	14
initiation interval (Σ)	1880	1472

(b) MJPEG2000: In units of 10^3 cycles (rounded)

Processing step	SpartanMC	MicroBlaze
0: read input	164	147
Fwd. wavelet full image	1: row loop	376
	2: col. loop	360
Fwd. wavelet top left quarter	3: row loop	90
	4: col. loop	93
Fwd. wavelet top left eighth	5: row loop	23
	6: col. loop	23
7: quantization	176	267
8: run-length encoding	63	63
9: entropy encoding	118	111
10: print output	8	6
initiation interval (Σ)	1494	1385

(c) Butterworth IIR Filter: In units of cycles

Processing step	SpartanMC	MicroBlaze
0: read input	429	494
1: filter loop	12966	8801
2: move output	19	40
3: write results	1243	1181
initiation interval (Σ)	12609	10516

(d) Firewall: In units of cycles

Processing step	SpartanMC	MicroBlaze
1: Receive eth. frame	2803	2094
2: Static filtering	1685	1663
3: Dynamic filtering	18194	16521
4: Send eth. frame	2815	1942
initiation interval (Σ)	25497	22220

values are used as output. The other elements are required to efficiently perform the convolution during processing. The output is used in all benchmark steps.

Firewall has varying input data length ranging from 64 to 1518 bytes Ethernet packets stored efficiently in a ring buffer. The data to be analyzed is however only the IP, ICMP, TCP or UDP header of the packet, not the payload which varies in size. The header information is required in each processing step. The payload of the packet is written while the packet is received and might or might not be required in the last step, dependent on the filter's decision to forward or block the packet. To generate the performance-profile, a TCP IPv4 packet of 736 Bytes is send through the firewall configured with 50 static filter rules and 1000 open TCP connections as dynamic filter rules. This should represent an average internet packet in a small office configuration like described in Section 6.1.4.

6.3 Possible Parallelization & Performance Gain

In the following sections, AutoStreams is run without optimization options and then step by step enabling different implemented optimizations to highlight their benefit, resulting in increased speedups or different parallelization methods. Each benchmark is parallelized and tested with a 2x, 4x, 8x and 12x speedup requirement. Thus, the most time-consuming task in the generated pipeline should be 2x, 4x, ... as fast as the total single-core execution time. Speedups are used here to unify results throughout the different benchmarks and to have comprehensible numbers, even though AutoStreams expects a requirement in a time unit. All parallelized designs target the Xilinx Artix-7 XC7A200T FPGA. The detailed evaluation is measured in cycles to make results comparable and independent of achieved clock frequencies on different FPGAs. It is usually a quite long process of trial and error and multiple synthesis runs to find the maximum frequency for an FPGA design. The achievable frequency for the multi-core designs is also assumed to have a small but not dominating influence. Thus, it is not carried out in this evaluation. The influence of different inter-core-communication peripherals and core numbers on the maximum frequency is later evaluated in Section 6.7.

The evaluation is done by starting the pipeline with a single data input. Each core measures the cycles for receiving and sending communication overhead, as well as the execution cycles of the assigned task. After each core finished processing, the results are stored in a global memory and the pipeline's first core outputs the measured results (for example via UART).

The firewall benchmark is only evaluated in the full-system evaluation in Section 6.5 and not during the detailed pipeline stage evaluation. The firewall is the only system using global memory for communication and managing filter rules. Thus, processing only a single packet would not reveal global-memory contention. However, evaluating multiple packets would distort measurements for the input and output core due to active waiting in an endless while loop.

6.3.1 Parallelization without Optimizations

The first parallelization evaluation uses AutoStreams without any optimizations. Only 1-to-1 FIFO based core-interconnects are available and loop splitting, DMA based interconnects and core replication with 1-to-N and N-to-1 interconnects is prohibited. Thus, only processing pipelines e.g. chains of processing cores without replication are generated.

Figures 6.3 to 6.5 show the resulting performance for the different benchmarks. For each resulting core, the cycles spent receiving and sending data through interconnects and the cycles for processing are shown. Solid bars indicate the actual measured results after executing the parallelized design. The dashed bars indicate AutoStreams' estimation based on the chosen partition points, the performance profile and the chosen communication peripheral's analytical send/receive time model. The red bar

indicates the users timing requirement to achieve the different speedups. The black bar indicates the actual achieved speedup. If the black bar is above the red bar, the user's timing requirements are not fulfilled.

The SpartanMC ADPCM benchmark with a 2x speedup requirement (Figure 6.3a) can not fulfill the requirements. Only a speedup of 1.78 is possible, while AutoStreams estimates a speedup of 1.76. AutoStreams recognizes that the requirement is not achievable without optimizations, gives a warning and returns the best possible configuration it can find. Looking at the ADPCM performance-profile (Table 6.1) reveals that step 7 already takes 1060×10^3 cycles, which is more than the 2x speedup requirement of 940×10^3 cycles. This prohibits successful parallelizations without further optimizations. The figure also shows that the workload for core 2 is tiny and one could argue that adding the workload to the previous core (core 1) would not make much of a difference. However, since AutoStreams could not find any solution satisfying the requirements, selecting the solution with the smallest hardware overhead is skipped and the one with the highest possible throughput is chosen. The only bound for this solution are available FPGA resources. The parallelized design for MicroBlaze shows the same characteristics as the SpartanMC design, only that the achieved speedup with 1.93 matches AutoStream's estimated speedup and comes closer to the requirement. Also noticeable is that the communication overhead is tiny in contrast to the task workload, usually making the application ideal for parallelization.

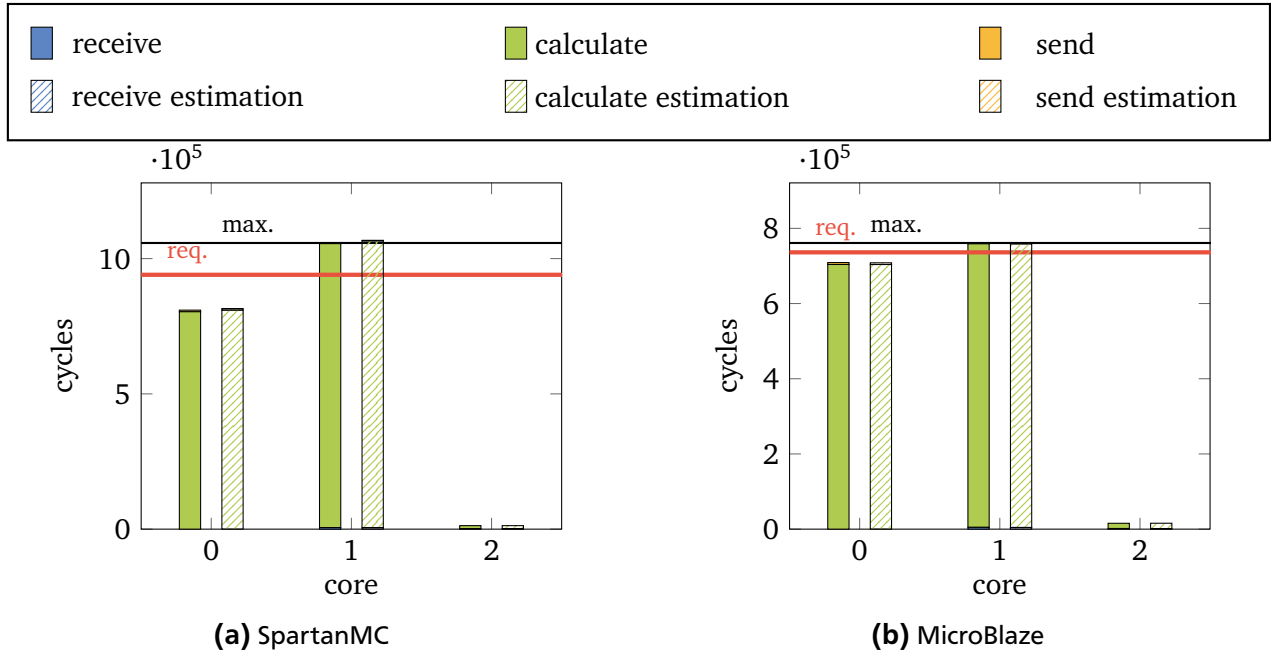


Figure 6.3: ADPCM 2x speedup requirement , no optimizations

For the MJPEG benchmark, a 2x speedup requirement can flawlessly be achieved (see Figures 6.4a and 6.4b). The three core design achieves a speedup of 2.69 for SpartanMC and 2.64 for MicroBlaze. In contrast to the ADPCM benchmark the communication overhead is now noticeable but not (yet) dominating, even though an integer array with 16k elements has to be transferred from core to core. Also, the workloads for each core are quite balanced due to number of intermediate calculation steps that can be well distributed among the cores. A balanced task duration over pipeline is desirable to have a smaller core count and thus fewer required hardware resources. However, the distribution for the 4x speedup is not that balanced anymore, as it can be seen in Figures 6.4c and 6.4d. Through the increased number of required cores fewer distribution possibilities exist. The SpartanMC design achieves only a speedup of 3.32 and the Microblaze design only a speedup of 2.8. For the SpartanMC design, a successful parallelization is prevented by the yet unsplitable processing step 1 (see performance-profile in Table 6.1). Yet, the MicroBlaze design could achieve a higher speedup with an additional core and is

not yet limited by unsplitable processing steps. Adding another core to the pipeline would require more BRAMs than there are available on the used Artix-7 FPGA. AutoStreams recognizes this automatically through the analytical hardware model and thus does not suggest such a configuration. In contrast to the other benchmarks, MJPEG requires 64 BRAMs per core instead of 8, due to the large image that has to be stored on each core.

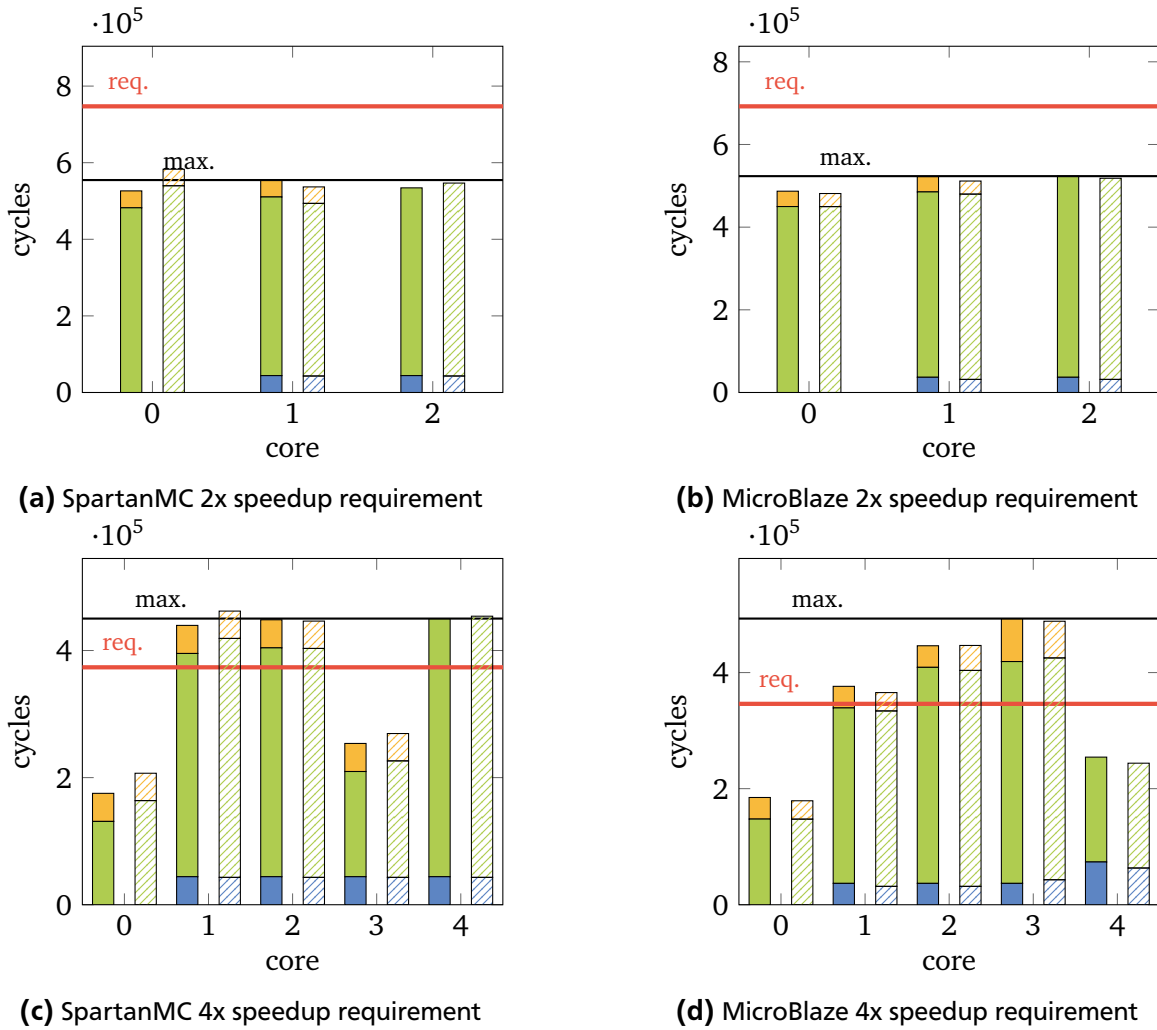


Figure 6.4: MJPEG 2x & 4x speedup requirement , no optimizations

From the performance-profile of IIR benchmark in Table 6.1 it is already quite clear that a parallelization without optimizations would not be very beneficial. The benchmark mainly consists of one big processing loop and only short processing steps before and after that can potentially be distributed to other cores. Therefore, the parallelization with SpartanMC revealed only a speedup of 1.12 and with MicroBlaze 1.05. Nevertheless, the MicroBlaze parallelization with only two cores is surprising and one would rather expect a three core design to achieve a better speedup, as seen with SpartanMC. Looking deeper, AutoStreams correctly identified that the required communication overhead for the extra core would be larger than the required processing time to accomplish this task. Thus, the processing steps were combined and communication avoided, a step barely identifiable during manual parallelization. This lead to a faster pipeline and also a reduced hardware consumption.

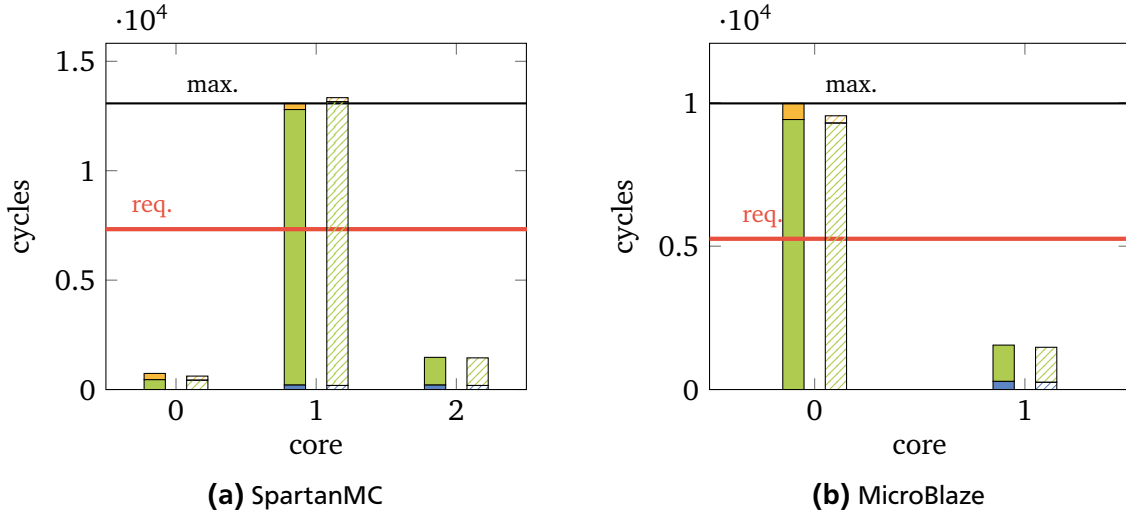


Figure 6.5: IIR 2x speedup requirement , no optimizations

6.3.1.1 Estimation Accuracy

The MJPEG 2x speedup requirement in Figure 6.4a and also other measurements reveal a discrepancy between the estimated and the actual processing cycles. The assumption is that the source lies in the GCC's compiler optimizations after parallelization. Since the source-code is split into several smaller pieces during source-to-source transformation and parallelization, it is vastly restructured. For compiling to the target architecture, the GCC³⁸ is used afterwards. Due to the restructuring, now different compiler optimizations are applied compared to the single-core variant where AutoStreams' estimation is based on.

To verify this statement, the IIR benchmark is chosen since it is the benchmark with the lowest complexity, where understanding the produced assembler code is still feasible. The SpartanMC IIR 2x speedup example has an error of 4.9% in the estimation of processing step 0 mapped to core 0. To support the claim of the dependency on compiler optimizations, the IIR single-core variant is profiled again with GCC compiler optimization `-O0` (no compiler optimizations) instead of `-O2`. AutoStreams is again executed with the new performance profile and the parallelized code also compiled without GCC optimizations. Now, the AutoStreams estimation is 100% accurate to the cycle. However, this is of course not a practical usable option and only used for problem identification. To track the root of the error with compiler optimizations, GCC's generated optimized assembler code of the profiled single-core variant and the parallelized version are compared. As shown in Listings 6.1 and 6.2 the parallelized-variant uses the branch delay slot, while the profiled single-core variant has a NOP in the branch-delay slot, yielding less optimal code.

With these findings, the SpartanMC MJPEG2000 example is also built with GCC optimizations turned off. However, there the optimized parallel-profile did not match AutoStreams' estimations for core 1 while the others estimations are cycle accurate. Again, looking into the generated unoptimized code, it became obvious that the parallelized source-code variant uses different techniques to access image data. It uses load offsets `l18 i0,OFFSET(i1)` while the profiled single-core variant leaves the offset 0 and requires an additional command to implicitly calculate the offset in register `i1`. The first assumption for the difference is a different scope of the used variables after parallelization which however could not be confirmed. Thus, the SpartanMC GCC seems to have a problem with determinism in the generated code.

³⁸ The used GCC versions are 7.1.0 for SpartanMC and 7.2.0 for MicroBlaze

Listing 6.1: Generated assembler code, IIR benchmark processing step 0, parallelized variant

```
...
3c8:  mov    i1, i0
3cc:  add    i1, i0
3d0:  add    i1, g2
...
3e4:  bnez   i2, 0x3cc
3e8:  mov    i1, i0
...
```

Listing 6.2: Generated assembler code, IIR benchmark processing step 0, single-core variant

```
...
5ec:  mov    i1, i0
5f0:  add    i1, i0
5f4:  add    i1, g2
...
608:  bnez   i2, 0x5ec
60c:  nop
...
```

It is also observed that the estimation for the MicroBlaze variants only showed estimation errors of maximum 3.5% and an average error of 0.5% with mostly under-estimation. The SpartanMC variants showed an estimation error of maximum 25% and on average 8% with over- and under-estimation.

As conclusion, the MicroBlaze GCC implementation seems to deliver more consistent optimization results compared to SpartanMC's GCC implementation. The SpartanMC GCC backend seems to have room for detecting possible optimizations.

For estimating the communication overhead, an estimation function is used, created from testing the transfer duration for different data sizes. The observed estimation error for SpartanMC FIFO based interconnects is at maximum at 45% for very small transmissions like in the IIR benchmark, where a few cycles difference result in a huge value in percent. For all other benchmarks, the observed error is always below 10% with constant underestimation. For MicroBlaze, the maximum observed error is 55% for small transmissions, while larger transmissions show an error below 20% with constant underestimation. The source of error also lies in the different applied compiler optimizations. The transmission time benchmark's assembler-code, which served as basis to the estimation function is well optimized. In the benchmarks, some compiler optimizations like LTO (link time optimizations) or IVOPTS (high-level loop induction variable optimizations) are crashing and are thus not applicable. Disabling these optimization passes for compiling the transmission time benchmark also revealed a lower performance, fitting the observed error. The observed error number in general could be better, but the observed constant under-estimation is very good, since it prevents AutoStreams claiming that there is no solution found due to too large communication overhead, when there actually exists one.

6.3.2 Parallelization with Replication

Using Dispatcher, Concentrator or multiple mailboxes as interconnect, pipeline stages can be replicated becoming superscalar. The replicated cores then work as a team on the same task. New input data is assigned to an idle core of the team and thus, the input-data acceptance-rate is increased. As limitation, cores with peripheral interaction cannot be replicated. This applies specifically to the first and the last core in the pipeline, since they have to handle data in- and output via peripherals.

AutoStreams now has the option to either add another pipeline stage or replicate this pipeline stage. The benefit of replication is that replicated cores speedup one or multiple assigned benchmark processing steps exactly by the replication factor. Partitioning the processing steps to multiple stages requires at best an identical duration of the processing steps. This is necessary to ideally leverage the pipeline, but it also adds a communication overhead per pipeline stage. The required interconnect hardware-overhead for replication in contrast to the regular pipeline is slightly smaller for the SpartanMC. For the Microblaze,

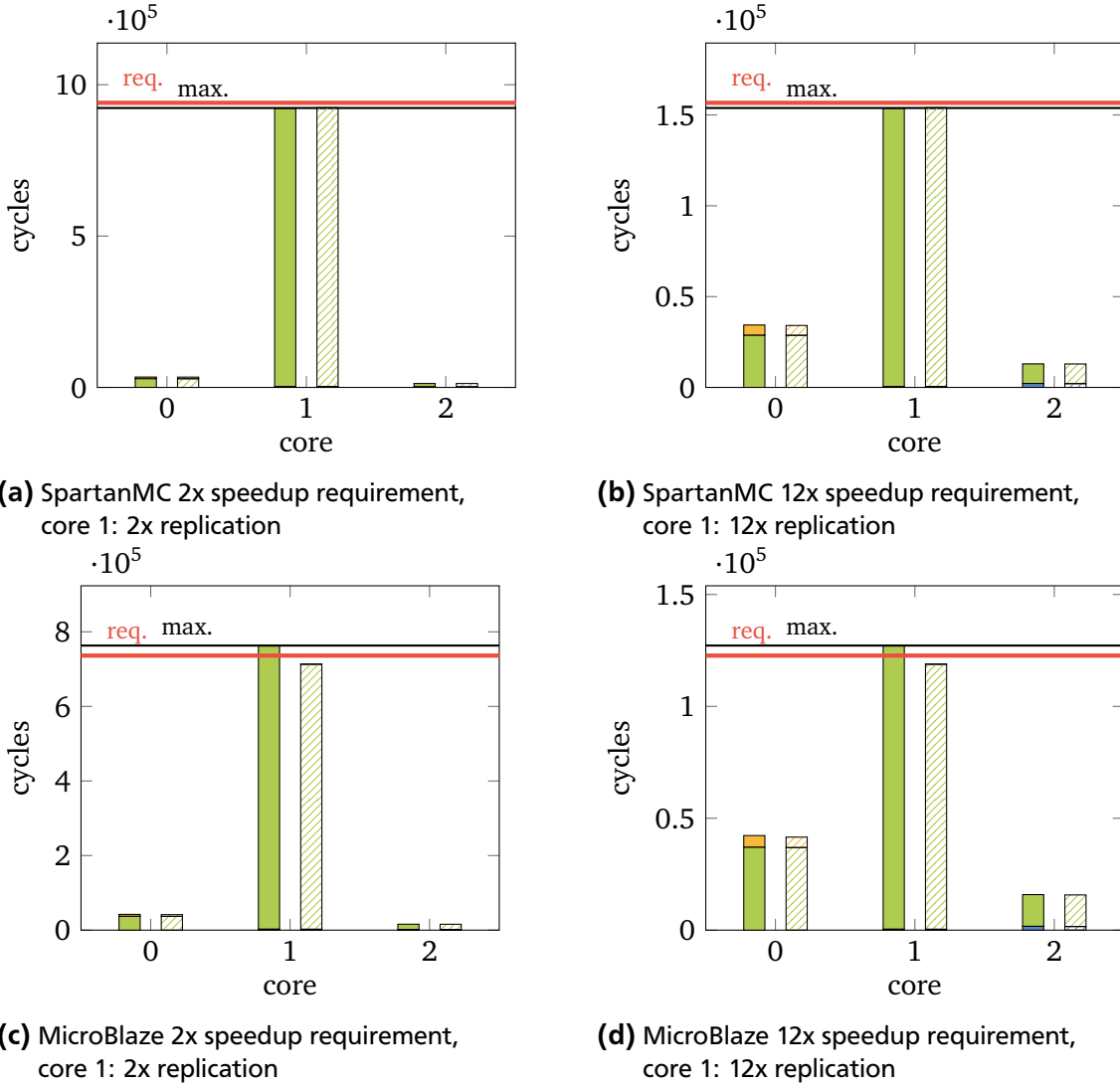


Figure 6.6: ADPCM with replication

an N times replication requires $2 \times N$ Mailboxes while a pipeline with an equivalent core number requires only $N + 2$ Mailboxes.

Just like before, the different benchmarks are parallelized through AutoStreams with replication enabled and 2x, 4x, 8x and 12x speedup requirement.

The ADPCM benchmark in Figure 6.6 is very well parallelizable with replication³⁹. For SpartanMC, the requirements are always met from 2x to 12x speedup requirement. Also, higher speedups until 54x would be possible until core 0 becomes the critical pipeline stage. For lower speedups, core 0 and 2 basically just handle in- and output. The reason for this is simply the lack of sufficiently small processing steps after the first and before the last processing step, as it can be seen in performance-profile in Table 6.1. Evaluating ADPCM on MicroBlaze reveals that the execution time of core 1, holding the critical pipeline stage, is always underestimated. Thus, AutoStreams estimates a fulfilled requirement while the measurement is slightly below the requirement: 1.93x for a 2x requirement or 11.58x for a 12x requirement. The requirement can still be fulfilled by simply slightly tightening the requirement, such that AutoStreams promotes an additional replication core.

³⁹ The replicated cores' initiation intervals (cycles) are already scaled according to the replication factor

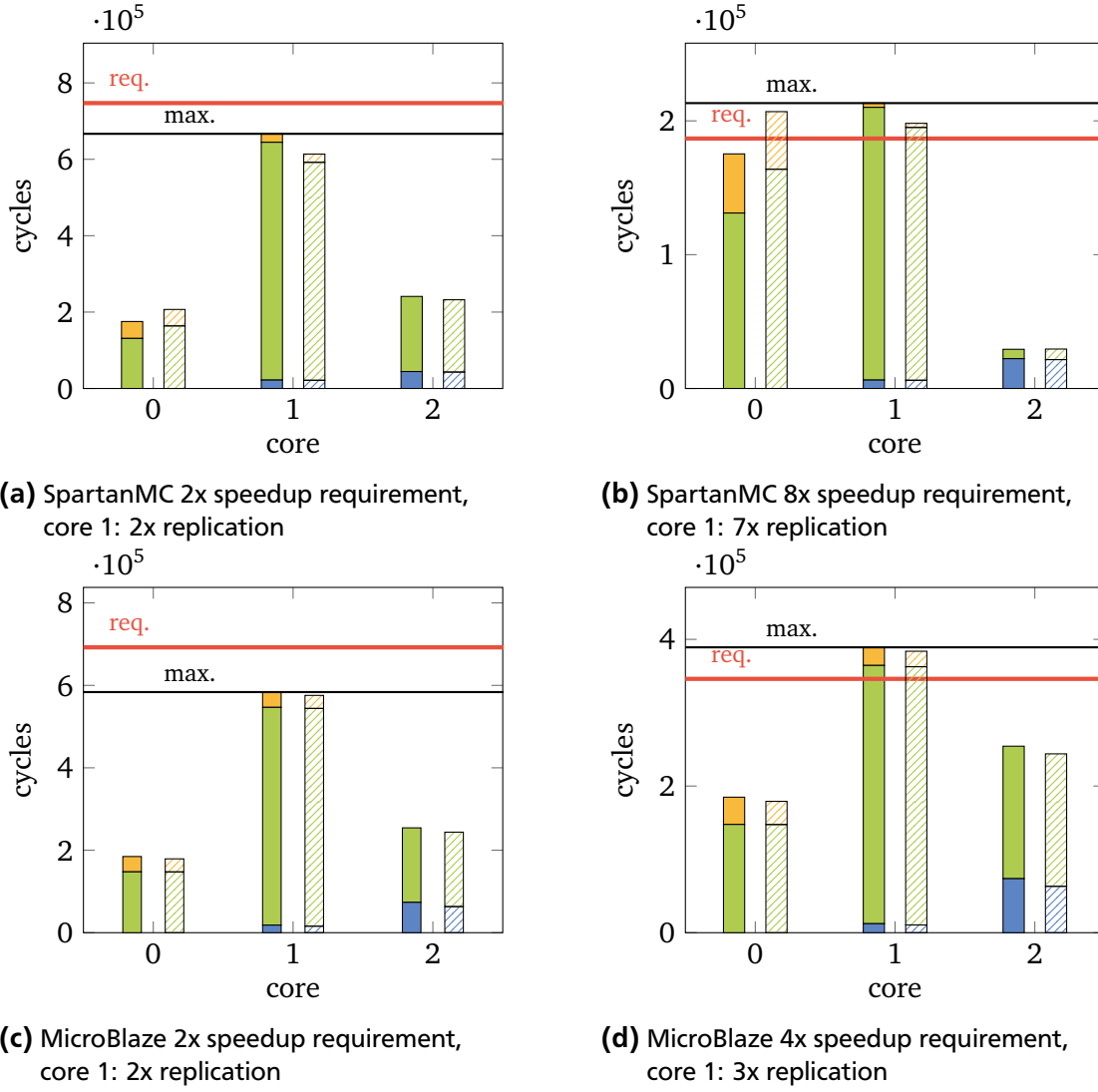


Figure 6.7: MJPEG with replication

Figures 6.7a and 6.7b show the results for parallelizing MJPEG2000 with SpartanMC. A parallelization up to 4x is automatically successfully possible. For the 8x requirement, AutoStreams estimates core 0 to be the critical pipeline stage, while through misprediction actually core 1 is the critical pipeline stage resulting in a speedup of 7.01x. Tightening the requirements would not work since AutoStreams sees no option to speedup processing step 0 on core 0 which it thinks is the critical pipeline stage. A successful speedup would still be possible when AutoStreams' generated μ Streams task replicate pragma is manually modified from 7x replication to 8x replication before executing μ Streams. The parallelization for MicroBlaze gives a speedup of 3.56x with a requirement of 4x (see Figures 6.7c and 6.7d). Since each MicroBlaze needs so much memory (BRAMs) for executing the benchmark, more cores do not fit on the used FPGA. AutoStreams automatically detects this hardware boundary and does not suggest higher replication factors that would result in hardware overuse.

Parallelizing the IIR benchmark with SpartanMC succeeds up to an 8x requirement (see Figures 6.8a and 6.8b). For the 12x requirement, AutoStreams estimates core 2 to be the limiting pipeline step through misprediction, while actually the 10 times replicated core 1 is the limiting stage resulting in a 11.22x speedup. Manually increasing the replication factor in the pragma would lead to a fulfilled requirement. For MicroBlaze a replication up to 7 times resulting in a speedup of 6.78 is possible with

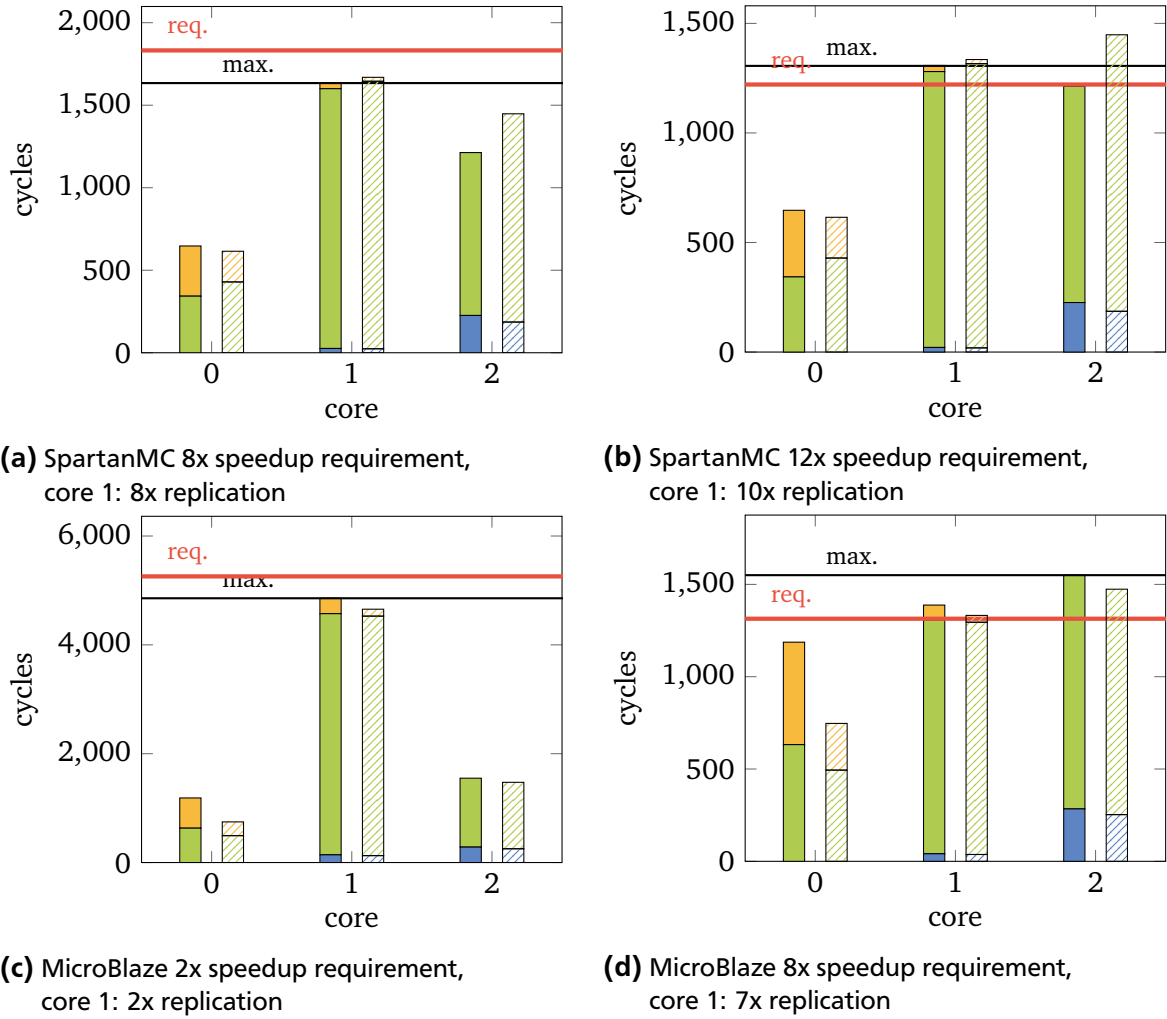


Figure 6.8: IIR with replication

an 8x requirement. There again, the last core, which cannot be replicated since it handles the output, becomes the critical pipeline stage.

In conclusion, replication increases the possible speedups of all benchmarks. Fast inter-core communication is required to achieve higher speedups for benchmarks with much communication overhead. For example the MJPEG2000 8x parallelization in Figure 6.7b would be possible with lower communication overhead on core 0 and a higher replication factor would then automatically be chosen since now core 1 is estimated to be the critical pipeline stage.

It is also obvious, that especially for the smaller speedup requirements, the pipeline is not well-balanced and the first and last cores have almost nothing to do. For example core 0 with processing step 0, in the ADPCM 2x speedup example shown in Figure 6.6a, is not able to accommodate processing step 1 as-well since this is a too compute intensive loop to still hold timing requirements. If the loop could be partitioned and a fraction distributed to core 0, the pipeline would be more balanced and the speedup would increase with the same number of cores.

6.3.3 Parallelization with DMA Interconnects

The influence of faster but more hardware intense DMA core-interconnects is evaluated in this section, since in the previous section some requirements are not achievable due to communication overhead.

Table 6.3: SpartanMC core and interconnect hardware cost on Artix-7 XC7A200T FPGA
(EP=end-points/connections, each core in default configuration with 10 BRAMs memory)

Module	LUTs	Registers	18 bit BRAMs	DSPs
SpartanMC core	899	202	10	1
Core-Connector + 2 cores	1910	458	20	2
MemSwap Dual + 2 cores	2247	141	24	2
Dispatcher 2EP + 3 cores	2875	695	30	3
Dispatcher 8EP + 9 cores	8512	2109	90	9
Concentrator 2EP + 3 cores	2838	691	30	3
Concentrator 8EP + 9 cores	8436	1995	90	9
MemSwap Multi 2EP + 3 cores	3348	610	38	3
MemSwap Multi 8EP + 9 cores	10975	1885	122	9

Firstly, the different interconnect characteristics are recalled to understand AutoStreams' trade-off between DMA-based interconnects' hardware cost and the speedup. The FIFO-based interconnects' transmission time grows linearly with the data size to be transmitted. The DMA-based interconnects have a constant transmission time, but the required memory size must accommodate all transferred data.

In contrast to FIFO-based interconnects, the DMA-based interconnects use BRAMs, but also some LUTs and registers. A short overview of the used hardware for the different interconnects is given in Table 6.3. In the following, there is just a focus on the used LUTs, since this is also used in AutoStreams for hardware cost comparison. Comparing the FIFO-based Core-Connector with the DMA-based MemSwap Dual interconnect reveals that the MemSwap module requires in total 337 more LUTs.

The same applies for 1-to-N and N-to-1 interconnects used during replication. An 1-to-N connection with two endpoints realized with a MemSwap Multi interconnect consumes 473 more LUTs than an equivalent configuration with a Dispatcher interconnect. A configuration with eight endpoints consumes 2463 more LUTs, when realized with MemSwap Multi interconnects. In this case, the additional LUTs are more than two additional SpartanMC cores would cost.

In conclusion, DMA-based interconnects always consume more LUTs than their FIFO-based counter parts. Replication with the same number of total cores uses more hardware than a FIFO-based pipeline without replication, but less than a DMA-based pipeline without replication.

Once again the benchmarks are executed with 2x, 4x, 8x and 12x speedup requirement. The previous design points: "no optimizations" from Section 6.3.1 and "replication" from Section 6.3.2 are again evaluated with DMA-based interconnects enabled. For each design point it is observed if DMA-based interconnects are automatically suggested by AutoStreams. If FIFO-based interconnects are suggested, DMA-based interconnects are forced to observe the achievable speedup with DMA interconnects. The results for the designs "no optimization" are shown in Table 6.4 and "replication" in Table 6.5. The highlighted cell indicates AutoStreams' suggestion.

The "no optimization" ADPCM benchmark 2x speedup requirement is still not achievable. DMA-interconnects are automatically suggested, since they slightly increase the achieved speedup from 1.78 to 1.8 but still don't fulfill the requirements. The same applies for the IIR 2x requirement and MJPEG 4x requirement. For MJPEG 2x benchmark, FIFO-based interconnects are suggested even though DMA-based would be slightly faster. Since the required speedup is fulfilled anyways, the FIFO-based solution is selected to minimize the hardware cost.

The replicated ADPCM benchmark shows that FIFO-based interconnects are always chosen since requirements can always be fulfilled and thus hardware saved. The DMA-variant is surprisingly always slower

Table 6.4: Achieved speedups and AutoStreams DMA design choice (highlighted) for previous designs "no optimizations", only DMA peripherals enabled

Speedup requirement	2x		4x	
	FIFO	DMA	FIFO	DMA
ADPCM	1.78	1.80	-	-
MJPEG	2.69	2.87	3.32	3.68
IIR	1.12	1.14	-	-

Table 6.5: Achieved speedups and AutoStreams DMA design choice (highlighted) for previous replicated designs and DMA peripherals enabled

Speedup requirement	2x		4x		8x		12x	
	FIFO	DMA	FIFO	DMA	FIFO	DMA	FIFO	DMA
ADPCM	2.04	1.94	4.07	3.88	8.15	7.77	12.22	11.65
MJPEG	2.24	2.95	4.48	3.97	7.01	8.0	-	-
IIR	2.24	2.29	4.49	4.57	8.97	9.14	11.22	11.39

than the FIFO variant. Nevertheless, AutoStreams' estimation predicts equal performance to FIFO variant. Comparing the parallelized measurement with the estimation reveals less optimized code when compiling for the DMA variants and thus a prediction error of 5%.

For the replicated MJPEG 2x and 4x requirements, AutoStreams selects the FIFO based solution to save hardware. The 8x speedup requirement with FIFO-based interconnects cannot be fulfilled due to communication overhead on core 0 (see Figure 6.9a). Through the reduced communication overhead, processing step 9 (see Table 6.1) can be placed on core 2 instead of the replicated core 1. Thus, the communication overhead as well as the workload on the critical pipeline stage can be reduced, which results in a fulfilled timing requirement with an 8x speedup (see Figure 6.9b).

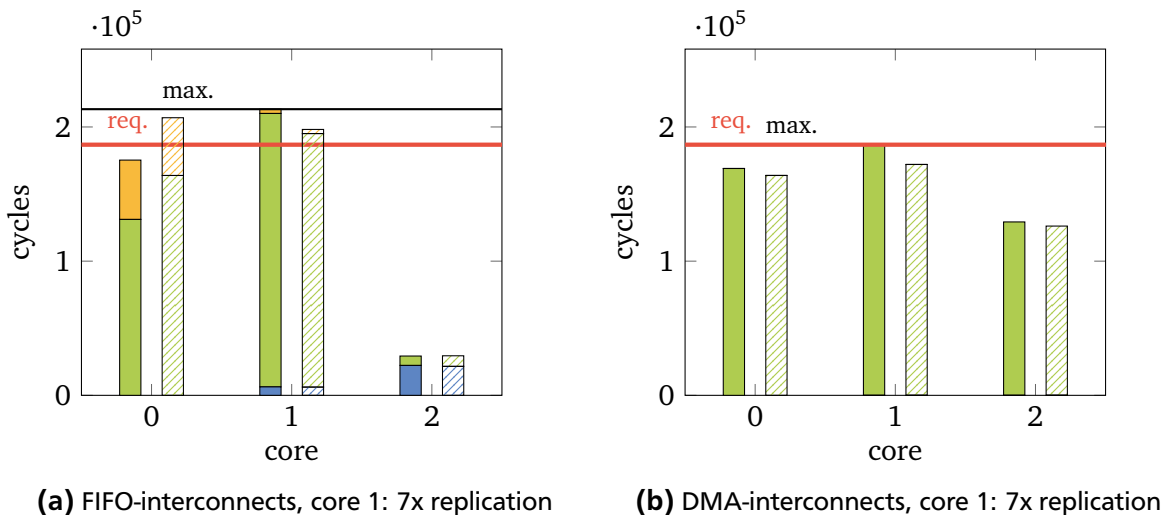
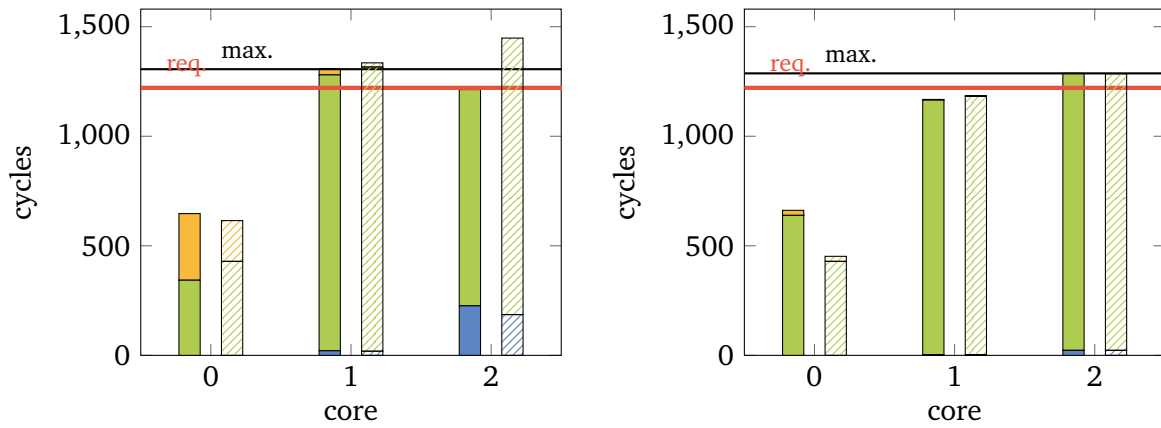


Figure 6.9: SpartanMC MJPEG replication with and without DMA-interconnects, 8x speedup requirement

For the replicated IIR benchmark, FIFO interconnects are selected up to 8x speedup requirement since the requirements can always be fulfilled. In the 12x requirement as shown in Figure 6.8b, the last stage is estimated to be the slowest stage and using DMA interconnects would increase the speedup. The replicated core 1 would become the critical pipeline stage without the communication overhead and



(a) FIFO-interconnects, core 1: 10x replication

(b) DMA-interconnects, core 1: 11x replication

Figure 6.10: SpartanMC IIR replication with and without DMA-interconnects, 12x speedup requirement

thus AutoStreams decides to increase the replication factor by one. Even though AutoStreams warned that the requirement could not be fulfilled, the speedup is increased to 11.39 with DMA interconnects.

All transferred variables have to be put in a struct for using DMA-interconnects, which is placed into the DMA address range. AutoStreams' estimation accuracy is also evaluated, since the variable transformations to struct accesses are a heavier source-code transformation compared to the use with FIFO interconnects. The observed average error for estimations with DMA interconnects is 7.3% and with FIFO interconnects 8%.

6.3.4 Parallelization with LoopOptimizer

It was already shown in the previous sections that it is often hard to create balanced pipelines due to the granularity of the processing steps. It was observed in the benchmarks, that mostly single loops dominate the execution time of the whole application. Thus, this step evaluates how well AutoStreams' implemented loop partitioning technique works and how it improves parallelization.

AutoStreams is executed with 2x to 12x speedup requirement with loop partitioning enabled, but replication disabled. Replication is disabled, since loop partitioning shows its strengths best for long pipelines and replication in the benchmarks only uses three stage pipelines with the middle stage replicated. There are also scenarios where replication is not possible, for example if a processing step uses peripherals.

Parallelizing the ADPCM application with SpartanMC fulfills the requirement up to a 2x speedup (see Figures 6.11a to 6.11c). For a 4x speedup requirement and higher, the requirement has to be restricted in a second parallelization to meet the requirements. For the 12x requirement DMA interconnects are automatically selected. Two processing cores can be saved, due to the reduced communication overhead. The requirements are not met due to processing step 1 (see Table 6.1). The processing step's loop is in the 2x parallelization mapped to core 0 and for further parallelizations partitioned and distributed over the first cores. Cores containing parts of this loop are highly underestimated by AutoStreams in most parallelizations. After looking into the assembler code, it became obvious that there are many possible loop optimizations omitted. This effect is increased through multiple nested loops. Since core 4 in the 12x speedup requirement only has a very short runtime, there is also the assumption that some iterations of the loop have different complexity. To verify this statement, the code is inspected and the loop is profiled with AutoPerf. However, the assumption could not be verified and each iteration has the same runtime, thus only compiler optimizations cause the difference. Other partitioned loops can be optimized very well even after partitioning and come very close to the expected duration. The parallelization with MicroBlaze is also not showing that high estimation errors, which also indicates missing optimization

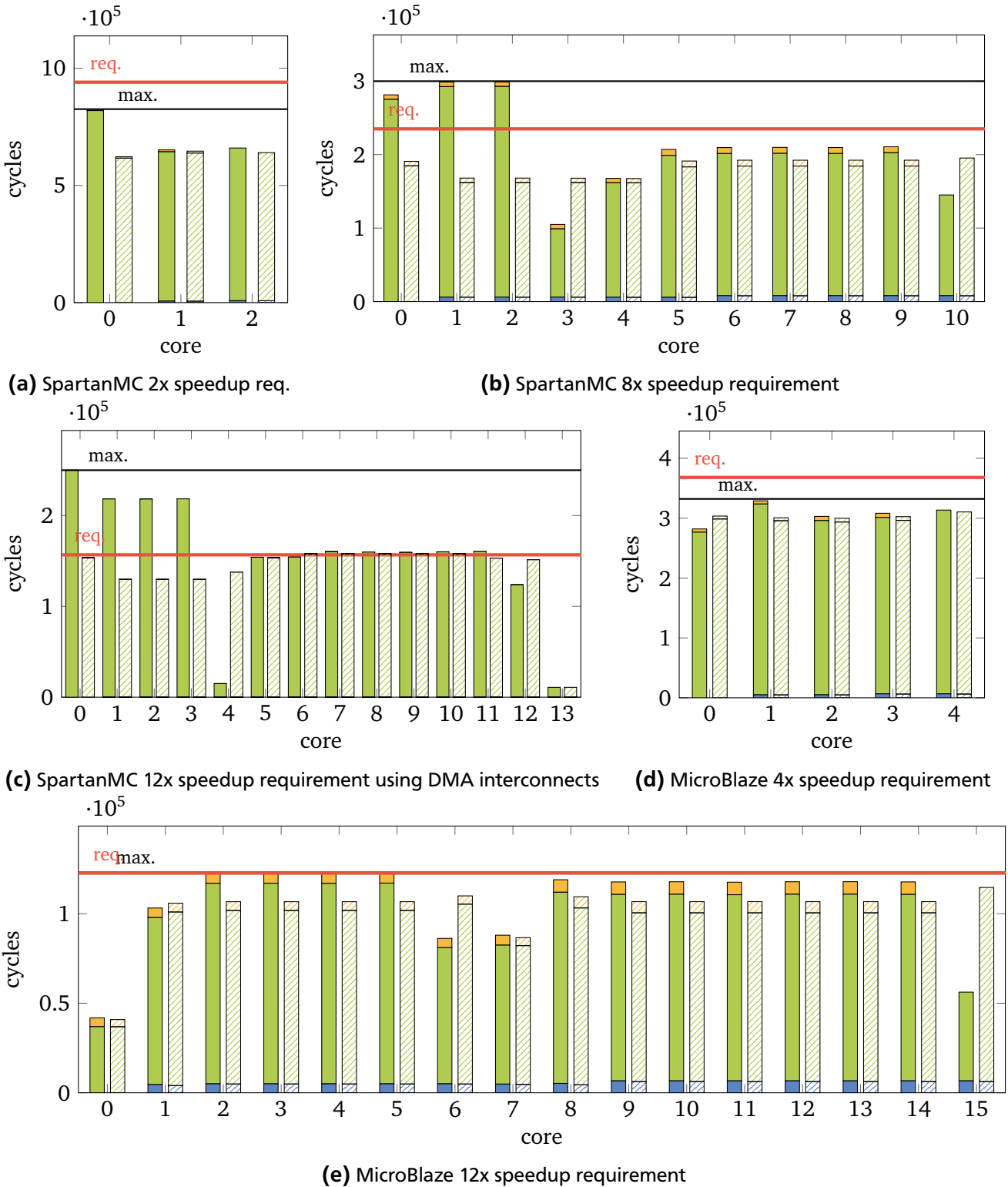


Figure 6.11: ADPCM with loop optimization

with the SpartanMC GCC compiler and more consistent behavior of the MicroBlaze GCC. All speedup requirements for ADPCM with MicroBlaze were fulfilled, even though for 12x requirement only tightly (see Figures 6.11d and 6.11e).

A speedup of up to 4x can be achieved for parallelizing the MJPEG2000 benchmark with SpartanMC (see Figures 6.12a and 6.12b). Since the MJPEG2000 application requires much memory, the BRAMs

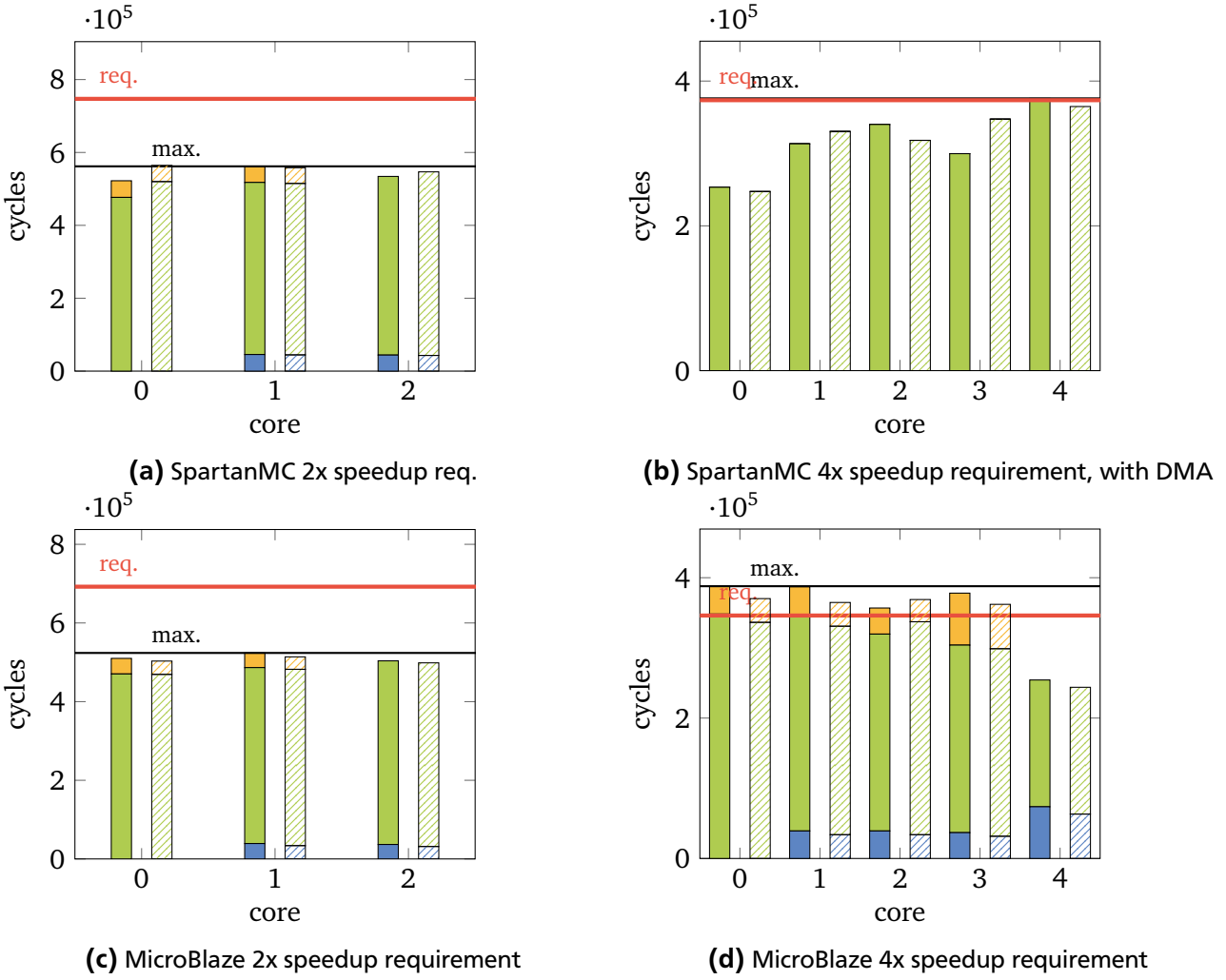


Figure 6.12: MJPEG with loop optimization

become the critical resource and not many cores can fit on the used FPGA. 13 cores are possible with FIFO-interconnects and 8 cores are possible with BRAM-consuming DMA-interconnects. However, a valid solution would require 12 cores with DMA interconnects or 16 cores with FIFO interconnects. DMA interconnects are suggested by AutoStreams for the 4x speedup requirement, since this saves one processing core in contrast to the usage of FIFO interconnects. However, due to the increased hardware consumption of DMA interconnects, AutoStreams estimates to save 690 LUTs and the measurement after synthesis revealed that actually 423 saved LUTs. Thus, the decision is correct even though estimation is slightly inaccurate.

Using MicroBlaze for parallelization reveals that a 4x speedup is not possible anymore (see Figures 6.12c and 6.12d). Communication becomes more dominant and more processing cores cannot be used since no more BRAMs are available on the FPGA. AutoStreams gives a warning and returns the best possible solution with a speedup of 3.57.

For parallelizing the IIR application with SpartanMC, DMA-interconnects are automatically chosen from the 4x requirement on, since communication would otherwise become very dominant as it can be seen for MicroBlaze in Figure 6.13. For the 8x speedup requirement, only five loop iterations are executed per core. Through slightly false execution time estimation only a speedup of 7.76 can be achieved. Parallelizing the 8x requirement only with FIFO interconnects would result in a 24 instead of a 12 core design. The 12x requirement achieves a speedup of 11x and each core executes only two iterations.

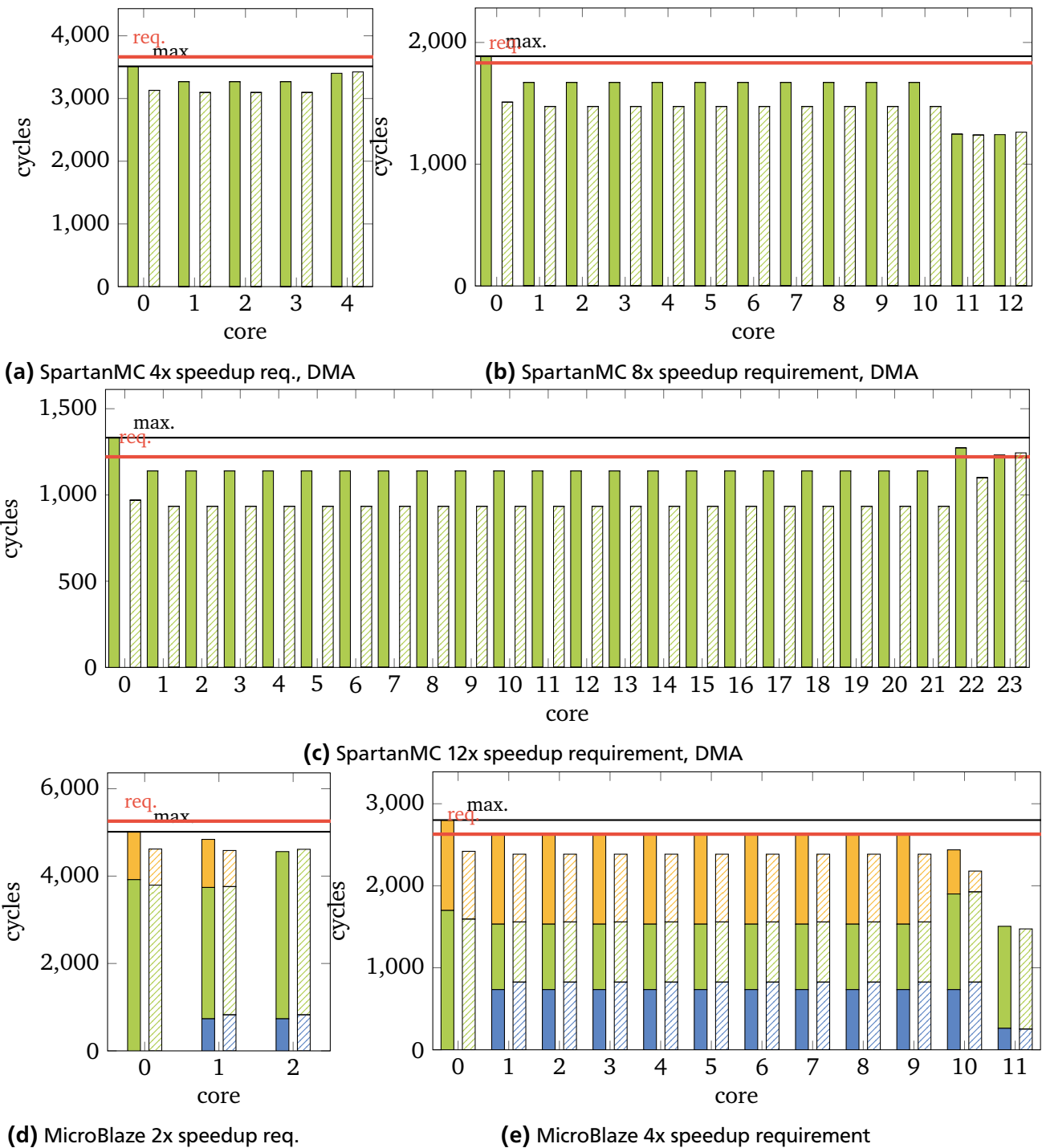


Figure 6.13: IIR with loop optimization

Higher speedups are not possible since the first and last processing step which cannot be parallelized or partitioned become the limiting step.

With MicroBlaze the 4x requirement cannot be reached, and slightly restricting the timing requirement would work to just fulfill the 4x requirement. Further parallelization is not possible since the communication time dominates too much. Even for the 4x requirement, communication consumes about 66% of the core's processing capabilities.

Concluding the loop optimizations, it is a well applicable technique to increase the granularity of critical processing steps since these are often loops. With the loop splitting technique, well-balanced pipelines can be generated. In practice the GCC's applicable optimizations after transformation are much more limited and therefore a slightly higher discrepancy between AutoStreams' estimation and the actual performance must be accepted. However, the positive effects outweigh the negatives, since the negative effects can usually be corrected through stricter timing requirements or manual modification of the μ Streams pragmas.

6.3.4.1 LoopOptimizer using Loop Fission Instead of Splitting

For the previous loop optimization benchmarks only loop splitting is used. However, the LoopOptimizer also has the possibility to perform loop fission. The IIR benchmark is used to generate a two core system and compare both methods. As it can be seen in Figure 6.14, loop fission results in the smaller communication overhead. This fact lies in the nature of loop fission, which separates independent statements of a loops body into multiple loops with the same iteration count. Since statements are independent, also the dependencies and thereby communication is minimized.

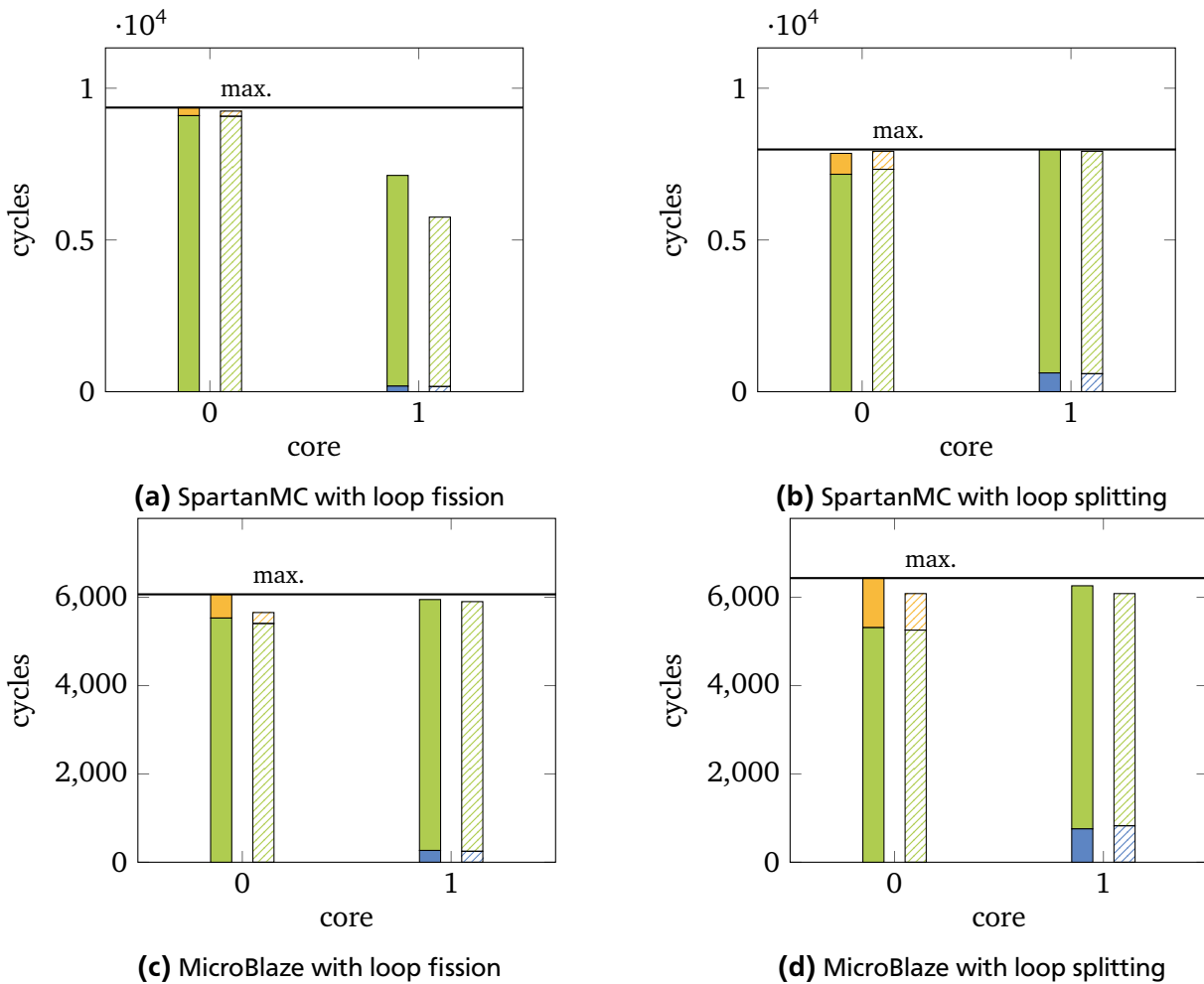


Figure 6.14: IIR 2x speedup requirement, loop splitting VS loop fission

The downside of fission is firstly that it is seldom applicable. Fission is only applicable with IIR from all used benchmarks. I assume the nature lies in programmer's thinking structures, to solve transformation steps one after another. Also, separating independent transformation steps in different loops usually increases the understandability of the code. In practice, it is seldom possible to extract a high number of

independent parts with fission. At least, there are usually more loop iterations that can be leveraged with loop splitting than independent body statements. This can also be observed with the SpartanMC example in Figures 6.14a and 6.14b. With loop fission, the two extractable independent code parts have different execution time and thus result in an unbalanced pipeline which is slower than a pipeline generated with loop splitting.

6.4 AutoStreams Estimation Accuracy

AutoStreams estimates required hard- and software during parallelization. The estimation quality thereby also influences the quality of the parallelized system in terms of pipeline balancing and minimal hardware selection. Sources of software estimation errors have already been discussed in Section 6.3.1.1. Thus, the following sections should give a quantitative overview of all measured systems to observe maximum and average errors.

6.4.1 Hardware Estimation

The hardware estimation accuracy is especially relevant if AutoStreams needs to make a choice between different possible hardware configurations. For example, is it better to use replication or an additional pipeline stage? Choosing a non optimal solution in such a scenario might not be too bad, as long as speedup requirements are fulfilled. The estimation is also required to estimate if a given design fits in an FPGA or user specified hardware limits. In such a scenario it would be bad to reject a good design point due to hardware overuse, while it would actually be synthesizable. Thus, it would be good to either have 100% accurate estimators or rather slightly optimistic estimators to avoid false negative rejection.

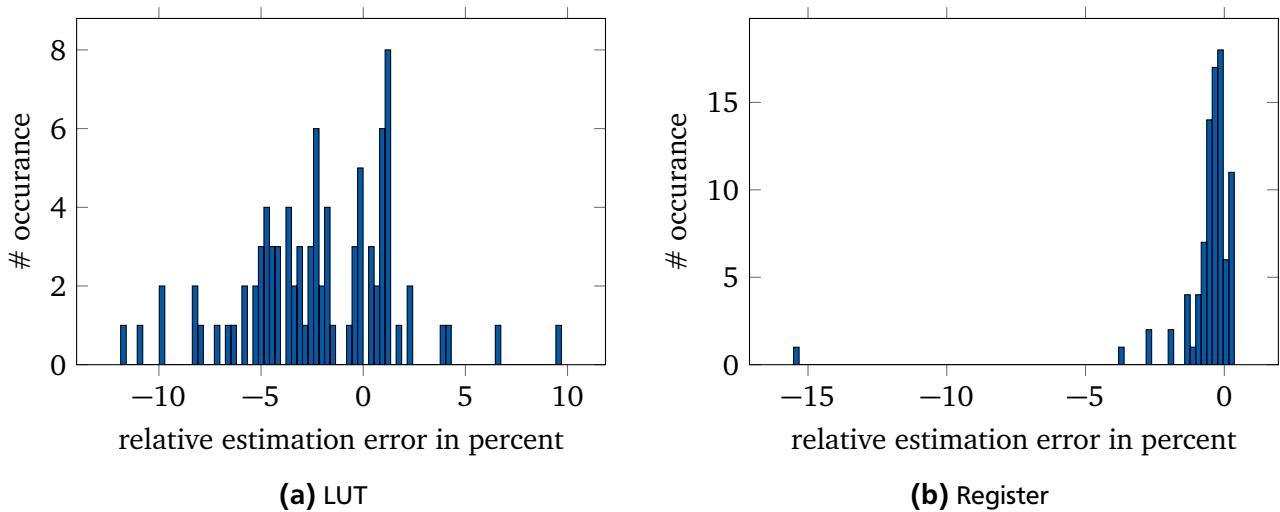


Figure 6.15: SpartanMC hardware estimation error

Figures 6.15 and 6.16 show a histogram of the relative estimation error in percent for LUTs and registers. The data for the diagrams is collected by synthesizing design point sweeps for the ADPCM, IIR and MJPEG2000 benchmark. Processor pipelines with two up to 25 SpartanMC and MicroBlaze cores and all possible interconnects are synthesized. The same procedure is repeated for a replicated three stage pipeline with intermediate core replicated from two, up to a factor of 10.

The synthesis target is the Artix-7 XC7A200T with a target frequency of 66MHz. This frequency is chosen since it is achievable for almost all designs.

For synthesizing the SpartanMC systems, Xilinx ISE 14.6 is used due to better SpartanMC tool support and for Microblaze systems, Xilinx Vivado 2018.2.1 is used. All default synthesis options were used for

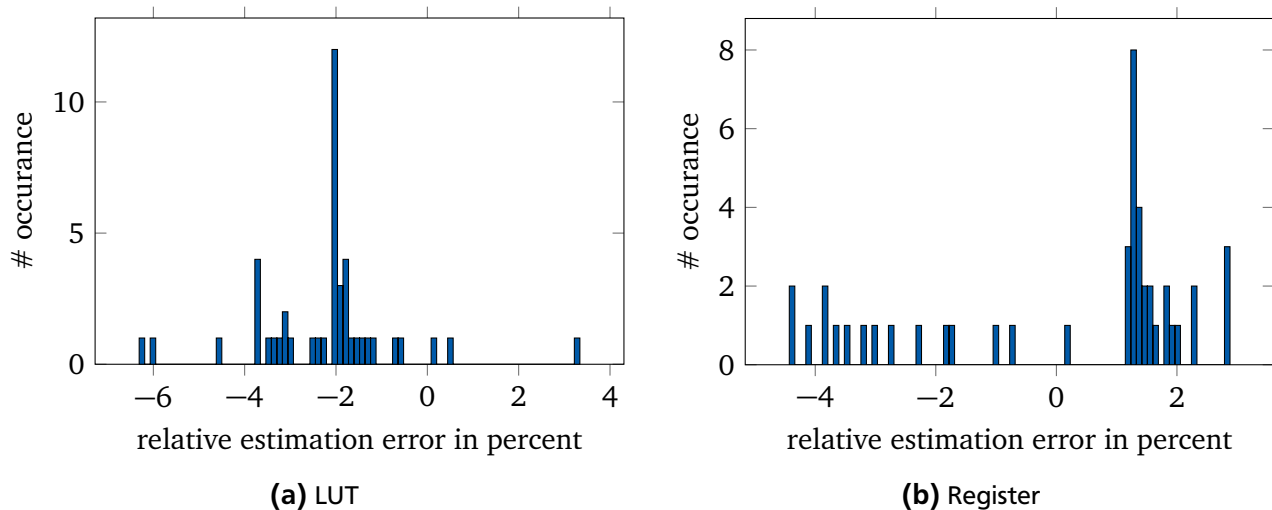


Figure 6.16: MicroBlaze hardware estimation error

both tools, only the optimization level for ISE synthesis and mapping effort is set to high. Officially, Xilinx suggests to use Vivado for 7 series FPGAs, but according to Xilinx FAQs⁴⁰ ISE supports the first released 7 series models (including XCA200T), but not all models.

As it can be seen in Figure 6.15 the achieved absolute value of the relative LUT estimation error for SpartanMC is mostly below 7% with a tendency to slight underestimation. The observed maximum absolute error is at 12%. The estimation error is assumed to come from an unfortunate random seed of the synthesis placement process. To verify this assumption, the random seed has been explicitly set via the `-t` option to a different random value. After the synthesis of the design with the former 12% error, the new random seed results in a design with much fewer LUTs and thus with a relative estimation error only 2%. The estimation accuracy for registers is at absolute maximum 16%, but mostly below 1% with a tendency for underestimation. The maximum register error comes from a design with very high timing pressure, since the synthesis struggles to achieve the demanded design frequency. The synthesis has options to reduce the fan-out of a register by duplicating it and thus reducing the register's load and therewith improve the speed of the circuit. Another technique, called *retiming*, is used to move registers along combinatorial paths to improve timing. This might also lead to more registers. After looking into the generated design with the unexpected high register usage, it became visible that these techniques must have been applied. One module holds the majority of the critical path and it has an extremely high register usage. Expected are 45 registers, but actually 285 registers are used by the module. The estimation error is below 2% for another synthesis run with reduced target frequency.

The LUT estimation for MicroBlaze designs in Figure 6.16a has a maximum absolute error of 6% but is mostly below 3%. The register estimation in Figure 6.16b has similar error rates. In contrast to the LUT estimation, AutoStreams has a tendency to slightly overestimate the registers.

The LUT estimation accuracy has to be taken with caution. The synthesis report counts the 7-series 6-input LUTs, configurable as two 5-input LUTs, each as one LUT independent of the actual width. However, the synthesis might vary implementing a function with 5- or 6-input LUTs. The decision for one LUT type might also depend on the timing pressure. Thus, for the same design, the number of LUTs in the report may vary, leaving the reference value also with a certain variance.

The estimation accuracy for BRAMs and DSPs shows no error. DSPs can easily be estimated since each core has a fixed number, thus only the core count is required. The same applies partly for BRAMs, the configured memory size is directly reflected in BRAM numbers. Each core's memory size in the

⁴⁰ <https://www.xilinx.com/support/answers/62332.html>

generated many-core design is the same as the one of the initial single-core design. This method is a quite pessimistic approach but is proven to work well for all benchmarks⁴¹. DMA-based interconnects also require BRAMs. The used BRAMs are again directly related with the used DMA address range and the necessary range is calculated by AutoStreams based on the communication variables' sizes. Thus, it can accurately be calculated.

It is also observable that usually BRAMs are the limiting factor for the number of cores synthesizable on an FPGA. For the Artix-7 XCA200T, 25 SpartanMC cores with default memory size of ten 18-bit wide BRAMs (including two BRAMs for the register file) already uses 250 BRAMs. This is already roughly 35% of the available BRAMs, while only 20% LUTs, 3% registers and 4% DSPs are used. The BRAMs are also the limiting factor for MicroBlaze multi-core designs. Other FPGAs like Spartan-6 family have a similar ratio of BRAMs compared to other components.

As a result, the bounds of parallelization in terms of FPGA components can be exactly calculated, since BRAMs are accurately estimated.

6.4.2 Application Runtime Estimation

Reasons for estimation errors have already been discussed in detail in Section 6.3.1.1 and Section 6.3.4. The estimation error is identified there to come from not applicable compiler optimizations after parallelization. Figures 6.17 and 6.18 show the estimation errors for all previously measured benchmarks in Section 6.3. The errors are presented as relative error in percent between measurement and estimation for calculation, receive and send time in cycles and per core.

The calculation relative estimation error in percent for SpartanMC in Figure 6.17a shows overestimation as well as underestimation, while underestimation slightly dominates. The maximum observed error is at -52% for the IIR 12x speedup requirement with loop optimizations (see Figure 6.13c, page 113). The IIR source-code, which is mainly one loop, is split into 21 smaller loops with only two iterations and each split loop has an estimation error of -52%. Since each loop is estimated with only 933 cycles, a few cycles difference already have a huge impact on the percental error.

The estimation error for MicroBlaze systems, shown in Figure 6.18a, is not that high which might be due to the better compiler optimizations. The relative estimation error is mostly between -10% and 0% with extreme values from -16% to 20%.

The receive estimation errors for SpartanMC in Figure 6.17b are below 10% and for MicroBlaze in Figure 6.18b below 15% on absolute. Send estimation errors for SpartanMC in Figure 6.17c are in the worst case -36% and for MicroBlaze in Figure 6.18c even -57%. The maximum errors for both systems primarily occur in scenarios with a low communication overhead. Thus, if communication only takes a few hundred instead of several thousand cycles, a few cycles estimation error result already in a high percental error. Interestingly, the AutoStreams estimator has support points at these exact points. The estimator is based on a testbench with measurements for different transferred data sizes. After looking into test's assembler-code, it became obvious that the testbench yields well optimized code. However, the parallelized design receive function's assembler-code often has room for improvement. As already stated in Section 6.3.1.1, the cause lies in not applied compiler optimizations. This also explains the tendency for underestimation.

What is also significant are the peaks in the receive and send estimation error histograms. The peaks for SpartanMC at 0% error represent all estimations for DMA interconnects. Since the time for switching

⁴¹ For a better memory utilization one would need a tool to calculate the maximum memory usage of source-core including the used stack, which is problematic for recursive functions. Alternatively, one needs to reduce the memory size of each core and observe when errors appear during program execution. For many-core systems, this is manually quite exhausting and not certain to deliver a guaranteed memory bound.

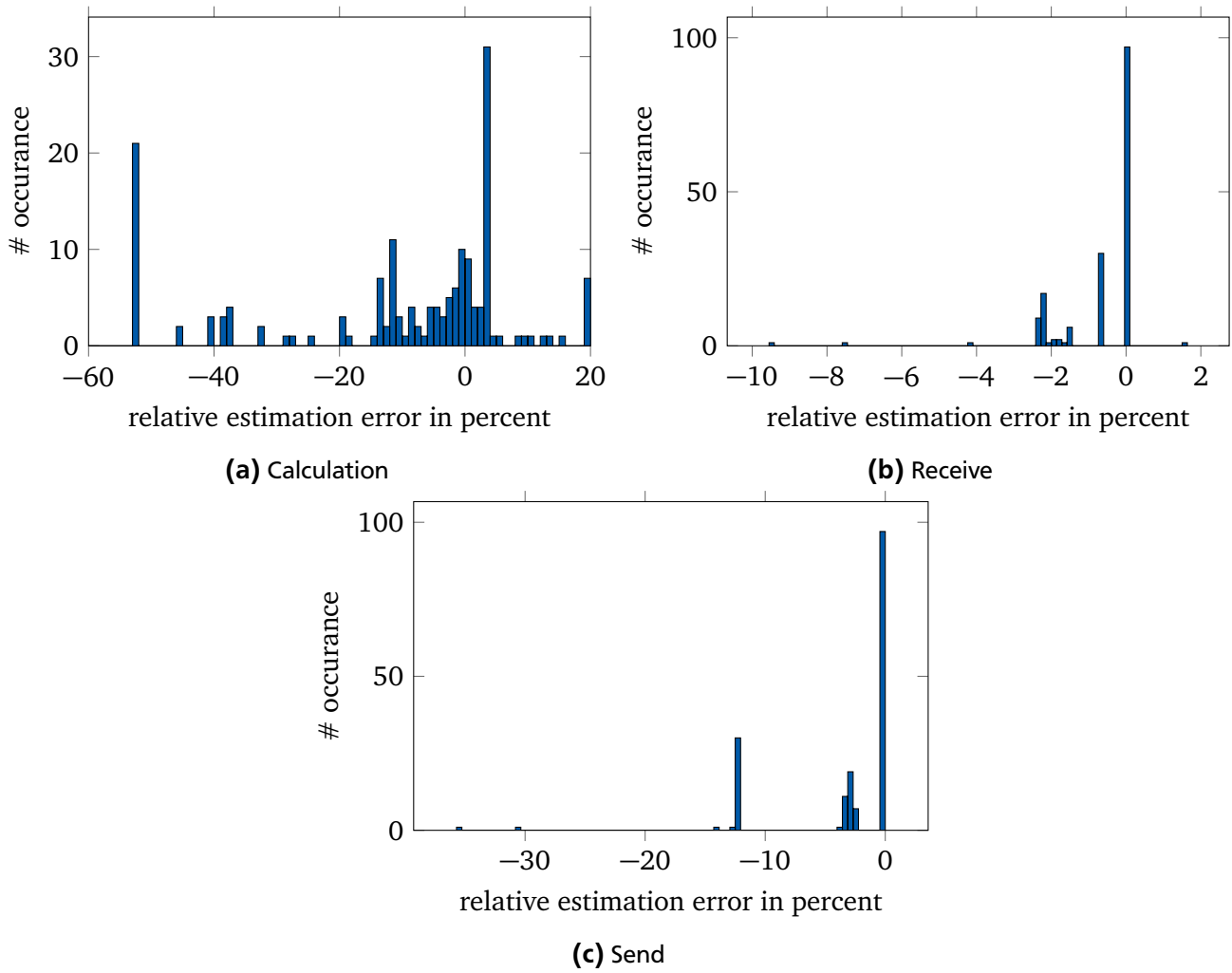


Figure 6.17: SpartanMC cycles estimation error of different parallelized software parts

the memory is fixed it is well predictable. The other peaks represent FIFO interconnects. For all systems, loops have to be partitioned for successful parallelization. These loop partitions are often of equal or similar size and they are mapped to one core each, such that the programs look very similar for these cores. This also results in identical applied compiler optimizations. Thus, the same prediction error also occurs for these cores, which explains the polarized peaks.

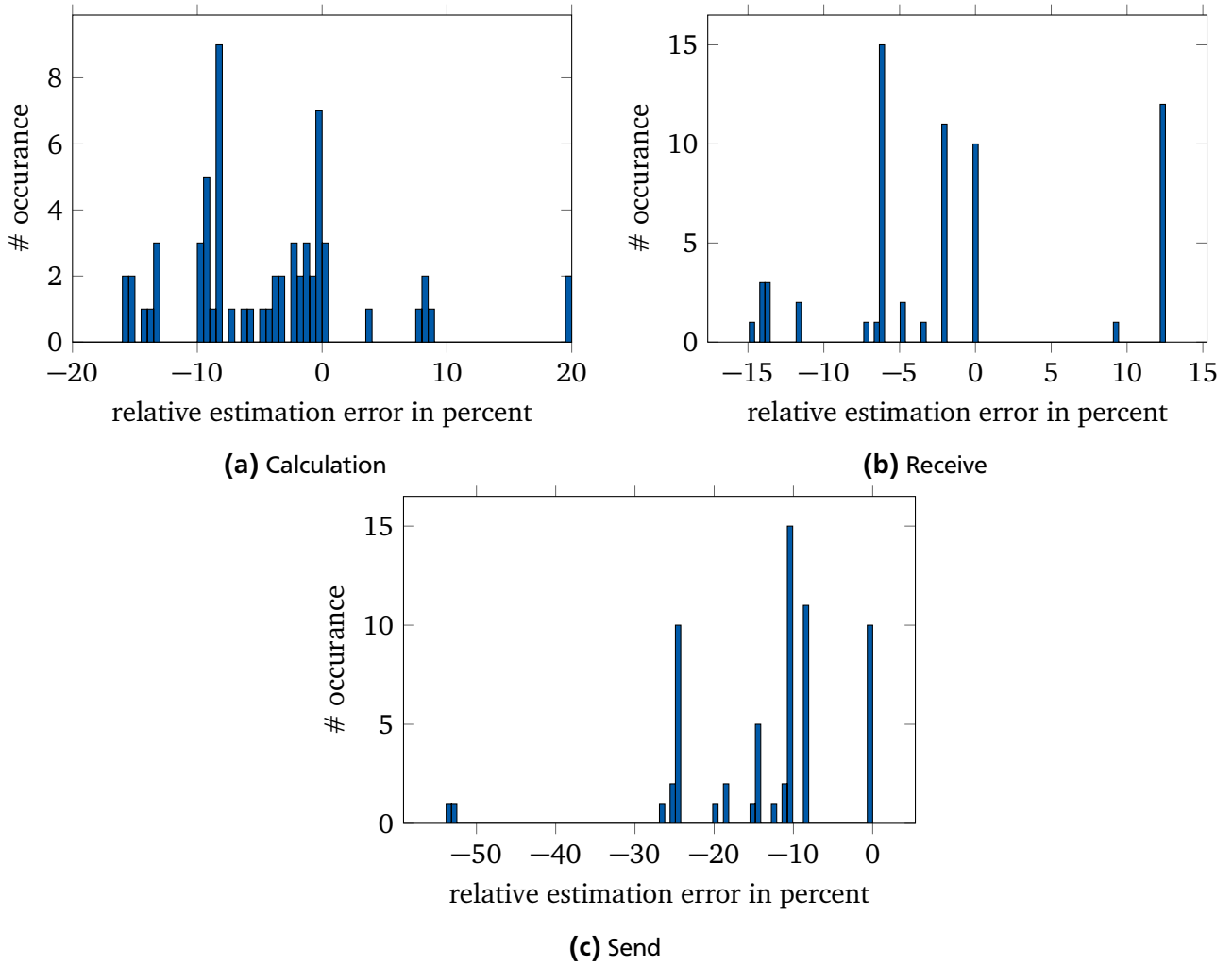


Figure 6.18: MicroBlaze cycles estimation error of different parallelized software parts

6.5 Parallelization with Peripheral In-&Output

In this section, two benchmarks are parallelized with real peripheral in- and output. The performance is measured under these real-world conditions and it is proven that the concept is practically applicable.

6.5.1 Firewall

As a real-world example, different parallelization variants of the firewall are evaluated in the following. The firewall is configured according to the assumptions described in Section 6.1.4. And the performance-profile from Section 6.2.1 is taken. AutoStreams is used with the measured performance-profile from Table 6.1 and requirements from 2x to 10x speedup.

6.5.1.1 Generated Hardware

AutoStreams does not have much degree of freedom for parallelizing the firewall in contrast to other benchmarks. Firstly, the filter loop is cascaded in multiple function calls, which cannot be partitioned with the LoopOptimizer. Even if it would be possible, loop partitioning is mostly helpful for creating processing pipelines and (long) pipelines are not desirable for a firewall. If an early pipeline filter stage finds a match in the filter tables, the packet would have to be handed through the pipeline and cores

located at the end of the pipeline would often idle through a previous rule match. Also, this would drastically increase the packet's latency. Thus, the only reasonable parallelization for AutoStreams is to put the Ethernet frame receiving, sending and filtering on an extra core. The filter cores are replicated to increase the performance. After 8x replication for SpartanMC and 9x replication for MicroBlaze, the function to receive Ethernet frames theoretically becomes the critical pipeline step that cannot be parallelized due to peripheral interaction.

TCP connections for dynamical filter-rules need to be centrally managed and synchronized among all filter cores. Each of the replicated cores needs a list of the currently open TCP connections along with the possibility to add a new TCP connection or discard a closed one. Thus, the easiest solution is a global memory for the filter tables.

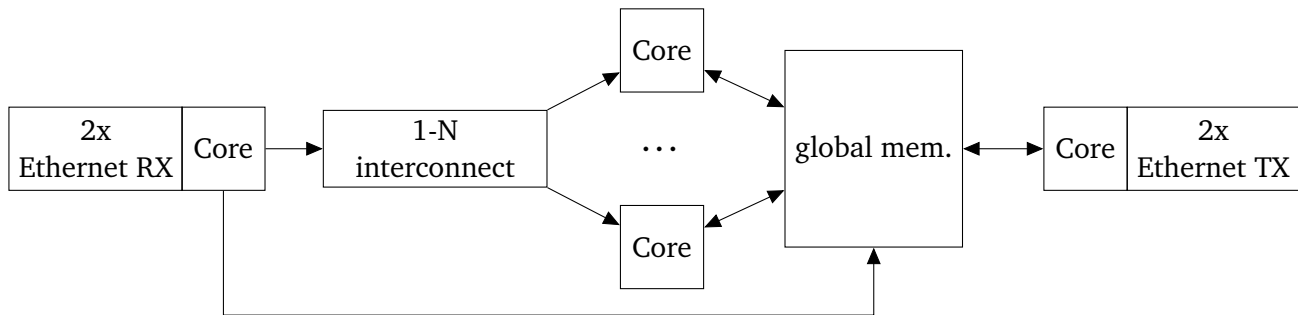


Figure 6.19: Firewall hardware design

AutoStreams' suggested hardware design is displayed in Figure 6.19. The core receiving the Ethernet packets puts them into the global memory and forwards the index with 1-N interconnects to the filter cores. Each filter core reads the packet header at the received index and matches it against the filter rules also residing in the global memory. On a match, the packet is either discarded from the memory or a *forward allowed* flag is set on that packet. The Ethernet send core waits for the *forward allowed* flags in the global memory and then copies the packets to the Ethernet hardware.

The achievable frequency of the generated design is widely independent of the core count. The critical path goes through the Ethernet peripheral or the global memory depending on the synthesis run. The resulting frequencies are 60MHz for SpartanMC and 125MHz for MicroBlaze.

6.5.1.2 Performance Evaluation

The generated designs for 2x,3x, up to 9x speedup are evaluated with a traffic-pattern that should fit to a realistic application scenario.

Since a firewall's performance heavily depends on the number of filter rules and the packet types, a realistic setting has to be created for the number of static and dynamic filter rules and the mixture of packet types. A small office with around 20 computers is used as evaluation scenario. Thus, the Rechnersysteme institution's firewall also matches this scenario and it currently holds 50 static TCP and UDP filter rules. This parameter is used to configure the static filter rules. The number of open connections per computer is measured with netstat under normal computer usage with internet surfing and reveals around 5 to 50 open TCP connections depending on the usage. Thus, the open TCP connections are set to 1000 open connections for 20 computers.

The mixture of packet and protocol types has been researched by Murray[116, 117]. It was found that the network traffic consists of 89.55% TCP and 9.91% UDP packets. Furthermore, the packet size on average has been measured with 736 Bytes. However, 90% of the observed packets were either below

100 Bytes or more than 1300 Bytes. Google publishes daily statistics⁴² for the ratio of IPv6 and IPv4 calls to their servers. On 21th of February 2019 this has been 22.45% IPv6 traffic.

To generate a worst-case scenario, the traffic and filters were configured to have no filter matches forcing the firewall to traverse all rules.

A packet generator Python script has been written and configured with the above numbers to produce a realistic packet mix. The script generates 500000 packets and sends them at full speed to one 100Mbit network port of the firewall. The script is executed twice at the same time to also fully utilize the firewall's second network interface and simulate bidirectional traffic. The network traffic is measured with the Linux tool *vnetstat* on the computer attached to the firewall network interfaces. *vnetstat* allows to capture the packet send and receive rate in Mbit/s and packets/s. The maximum achieved throughput rate for the firewall input is measured with 193 Mbit/s and 32845 packets/s, which is close to the optimum of 200 Mbit/s. Since the used processors are not too powerful, they struggle with forwarding and filtering at this rate. Thus, there are (high) chances for packet drops due to network interface's full ring-buffer. The measured receive throughput with *vnetstat* are thus the throughput rates that the firewall is capable to handle. The measured throughput in Mbit/s at the receive interface of the PC is mainly a metric for the performance of the firewall's network interfaces. Thus, how fast the used processor can serve the network interface. The throughput in packets per second rather shows the filter capabilities of the firewall, since every packet, independent of the size has to be matched against the filter rules. The measured throughput rates for the different suggested AutoStreams designs are shown in Figures 6.20 and 6.21. The figures show measured throughput in Mbit/s and packets/s for the different parallelized requirements with 2x, 3x, ... speedup summed up for both firewall network interfaces. The 1x speedup measurement point indicates the single-core firewall implementation. Hatched bars indicate the packets/s or Mbit/s throughput needed to fulfill the desired requirement. The dashed line shows the maximum possible throughput defined by the network interfaces and the used packet mixture. The speedup is set as processing time requirement with respect to the performance-profile, but detailed time measurements are not feasible since the profiler would distort continuously running application with multiple iterations. Thus, it has been decided to measure the actual achieved speedup in packets/s and Mbit/s. The timing requirement should relate to these values and these are anyways the important numbers to judge a firewall's performance.

As shown in Figure 6.20, the achieved network throughput in terms of Mbit/s is only at around 1 Mbit/s for SpartanMC and 3 Mbit/s for the MicroBlaze single-core design. The reason of the relatively low single-core throughput rate is due to network interface's full ring-buffer. Since the network interface buffers are implemented as a ring-buffer, the chances for smaller packets to get a free spot are higher. The buffer runs full and the single-core design cannot keep up with fetching packets from the interface since it also has to filter and forward packets. The 2x requirement for MicroBlaze just fulfills the requirement, while the SpartanMC system highly exceeds the requirements. The explanation lies in the different performing global memory implementations in both systems. The network input and output are no longer critical steps, since each network interface has a dedicated core which only copies network packets to or from the global memory. However, the global memory in the MicroBlaze system has a higher latency and lower throughput compared to the previously used local memory. For the SpartanMC, global and local memory have identical characteristics which explains the higher speedup gain. The 4x and 5x speedup for SpartanMC and 5x and 6x speedup requirement can be fulfilled with the same hardware requirement. That is totally six cores with four times replication for SpartanMC and seven cores with five times replication for MicroBlaze respectively. As it can be seen for higher speedups a saturation sets in. The saturation is due to the full utilization of the global memory, which cannot keep up with all the requests of that many filter cores. After the 8x speedup design for SpartanMC or 9x speedup for MicroBlaze, AutoStreams does not suggest higher replication factors. The network interface performance

⁴² Google IPv6 statistics <https://www.google.com/intl/en/ipv6/statistics.html>

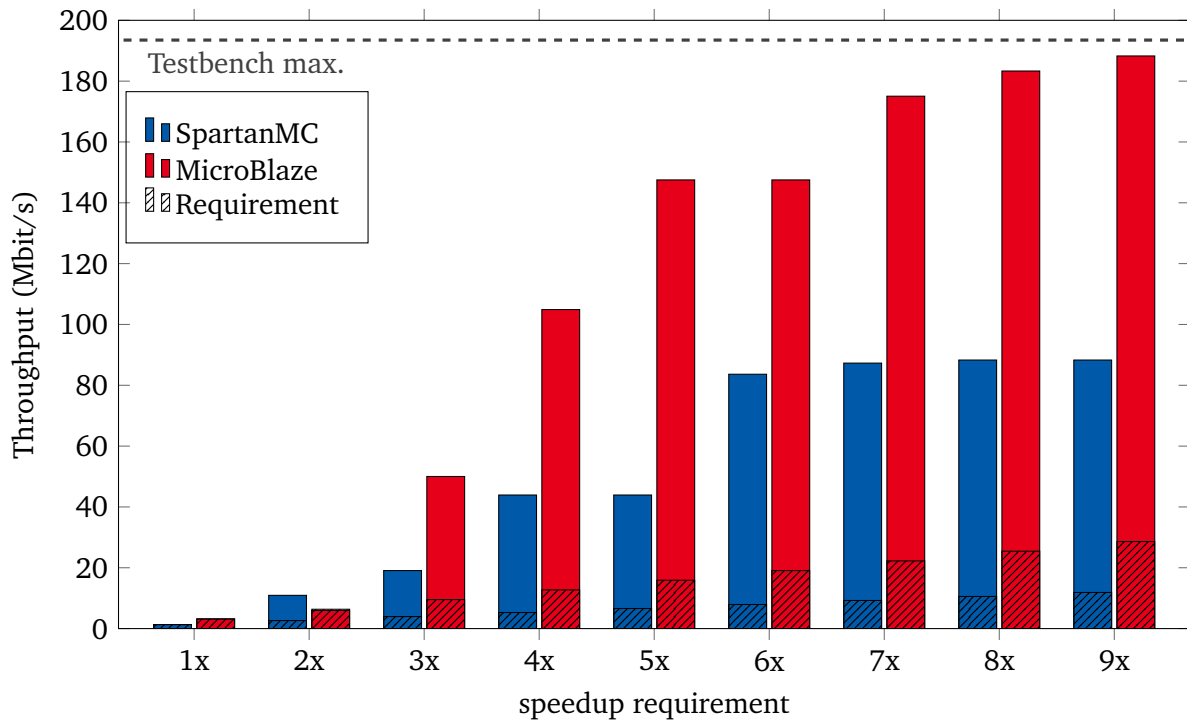


Figure 6.20: Network throughput in Mbit/s for different system configurations

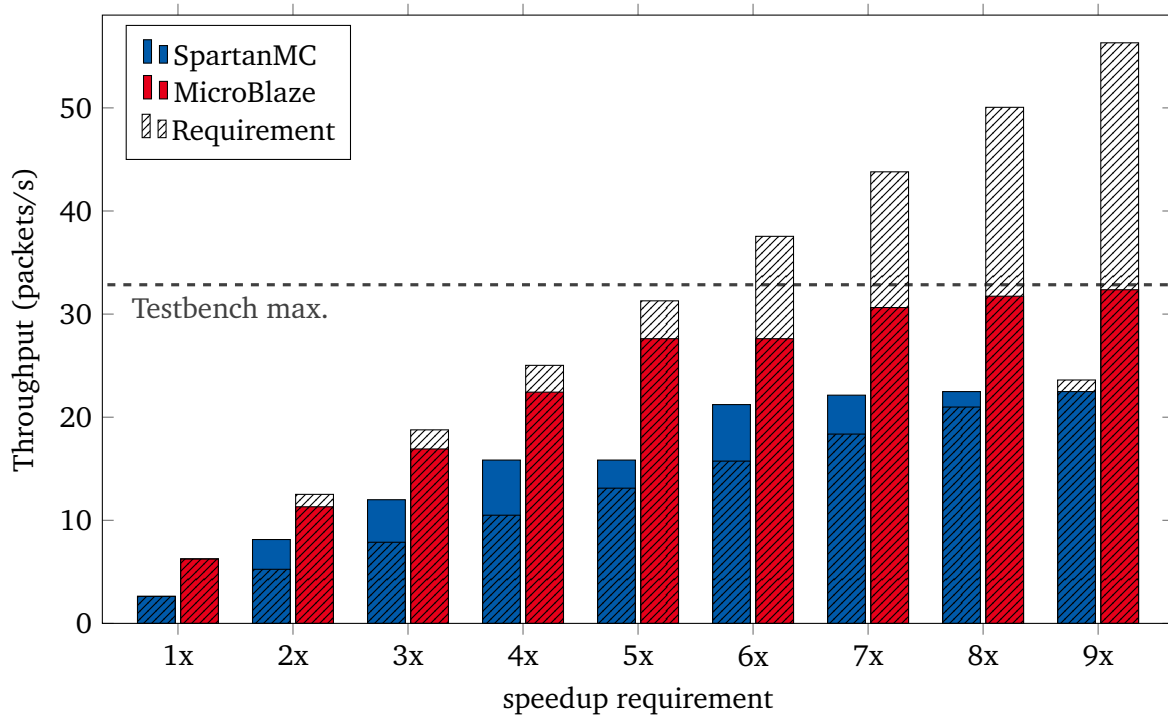


Figure 6.21: Network packet throughput for different system configurations

is now the limiting factor and the I/O cores are fully utilized but cannot provide more packets to utilize the filter cores at higher replication factors. It is also interesting to see that the MicroBlaze is able to handle higher throughput compared to SpartanMC. The reason here is most probably a 2.5 times higher clock frequency and the 32-bit instead of 18-bit architecture. This also results in a 2.1 times higher peak data rate. The maximum reached data rate for SpartanMC is 88 Mbit/s and for MicroBlaze 188 Mbit/s.

Overall, the speedup requirements for the data rates were well achievable, due to the bad performance from the high packet drop rates in the single-core design.

The behavior for the measured packet rate in Figure 6.21 is much more consistent with the expectations since the packet drop rate for larger packets is not relevant from this perspective. Thus, the achieved 15x speedup for data rate results in a 2.7x speedup for the packet rate with the MicroBlaze 3x speedup requirement. Each additional core for the designs with low speedup requirements results in a 60% to 70% higher packet filter rate. However, the packet rates for higher speedup requirements also show the previously described saturation effect of the global memory. The speedup requirements for SpartanMC can be fulfilled up to the 8x requirement and afterwards the I/O cores become the limit preventing further scaling and AutoStreams already warns that the requirement cannot be fulfilled. The requirements for MicroBlaze are never fulfilled. From the 5x requirement on, the demanded packet rate lies above the maximum possible data of the network interface with the used packet mixture. Even the lower requirements cannot be achieved. The explanation for this are two factors that AutoStreams does not consider. Firstly, the performance-profile is generated with the faster MicroBlaze local memory for the filter tables and not the slightly slower global memory implementation. Secondly, AutoStreams has no estimation model for contention of multiple cores over the global memory, which also might be hard to realize. Until the global memory saturates, a successful parallelization could be achieved through restricted requirements. Summed up, the MicroBlaze is able to almost fully handle the maximum provided data stream with 32362 packets/s, while the SpartanMC can only handle 22483 packets/s (about 70% of the provided packet rate) before the network interface cores limit.

In conclusion, even though the speedup does not always behave as expected due to peculiarities of the Ethernet hardware and unmodeled global-memory performance in AutoStreams, speedups are still achievable automatically.

6.5.1.3 Estimation Accuracy

The estimation accuracy for the generated systems is collected in Table 6.6. The speedup estimation error is collected for the estimated speedup in Mbit/s and packets/s. AutoStreams cannot estimate increased MBit/s or packets/s directly, but a speedup for processing time. Thus, the estimated speedup for processing time is transferred into estimated Mbit/s and packets/s for the multi-core designs, based on the single-core measurements. The hardware estimation error is displayed in terms of LUT and Register relative estimation error in percent. The DSP and BRAM error is not shown, since these values were accurately predicted.

Target processor	Speedup relative estimation error				Hardware relative estimation error			
	Mbit/s		Packets/s		LUT		Registers	
	average	max.	average	max	average	max	average	max
SpartanMC	566%	887%	12%	26%	1%	6%	8%	11%
MicroBlaze	491%	659%	31%	47%	1%	1%	3%	3%

Table 6.6: Estimation accuracy as relative estimation error in percent

The prediction for the Mbit/s is always heavily below the measured performance. So the actual achieved speedup in Mbit/s is better than expected, since the single-core design performs badly at high input data rates (see Section 6.5.1.2). When the input data rate for the single-core design is lowered, it is observed that the firewall's throughput data rate goes up, since the drop probability for larger packets decreases. However, in practice this is not an option, since a firewall should not limit the network interface data rate. Interpolating the speedups from this throughput data rate gives data rates close to the observed data rates for the multi-core designs.

The error for the packets/s for the SpartanMC systems is 12% on average of the absolute values and 26% at maximum. The achieved speedup is always underestimated by AutoStreams, so the real speedup is higher than expected. The error decreases for higher speedups due global memory saturation, which is not modeled in AutoStreams. The packets/s estimation error for MicroBlaze is 31% on average of the absolute values and 47% at maximum. The average estimation is so high because MicroBlaze's low performance global memory is not considered in AutoStreams. The maximum error is observed for high speedup requirements, where global memory saturation sets in.

In the first place the estimation errors for hardware usage were way off the track. Looking closer into the synthesis hardware report revealed that the Ethernet interface's hardware peripheral holds considerable amounts of resources. After adding the resource usage for the Ethernet peripheral to AutoStreams' hardware cost table, the results match the previously observed error ranges in Section 6.4.1. Not all available peripheral hardware costs are modeled in AutoStreams, since there are so many and each peripheral might have different configurations all yielding different hardware costs. The LUT estimation error for SpartanMC is 6% at maximum and 1% on average of the absolute values, for MicroBlaze the error is slightly better with maximum and average of 1%. The register estimation error is at maximum 11% and 8% on average of the absolute values with constant underestimation. The error for MicroBlaze systems is better with a rounded 3% error at maximum and average. The presented estimation error matched with measurements in Section 6.4.1, where parallelized systems without additional peripherals (except core-interconnects and Universal Asynchronous Receiver Transmitter (UART)) are evaluated.

6.5.1.4 Performance Comparison with Related Work

Currently, there exists no related work leveraging many-core soft-core SoCs for a firewall. At the one hand there are commercial firewall solutions using server hardware and desktop processors which are widespread. These systems are able to leverage two 10 Gbit interfaces with default firewall applications like pfsense⁴³. On the other hand there are specialized FPGA solutions with custom data paths. The presented systems[118, 119, 120] are able to handle from 1Gbit/s up to 80Gbit/s for larger Virtex-5 FPGAs depending on the quality and intention of the published solution. These systems are built for high performance and are barely comparable with the used microprocessors. Rather comparable are low-cost general purpose embedded boards with ARM processors, like the Raspberry Pi or Cubieboard. Nabi[121] evaluated the achievable network filter performance on these boards with *iptables* and 1000 filter rules. The Raspberry Pi achieved a throughput of 15 Mbit/s and the Cubieboard 23 Mbit/s. The presented parallelized firewall achieves maximum rates of 88Mbit/s for SpartanMC and 188Mbit/s for MicroBlaze at an equivalent number of filter rules. Thus, the presented firewall has a higher data rate compared to general purpose embedded boards, but lower performance than the aforementioned general purpose server systems or application specific FPGA designs. However, it must be kept in mind that the specialized FPGA designs require much manual tweaking and packet handling in hardware which was not the case for the presented automatically parallelized firewall.

6.5.2 ADPCM with IO

The ADPCM benchmark is now equipped with real input and output peripherals to show the real live applicability of the parallelizer and also the transferability of the measured results from Section 6.3. The ADPCM, MJPEG2000 and IIR benchmark were previously run with input data initially loaded into the memory and the output data also only remained in the memory after processing. This method allows an easy evaluation of the design, since the peripherals require no external data drivers. The ADPCM benchmark is selected to show the applicability of the concept with peripheral IO. ADPCM is

⁴³ <https://docs.netgate.com/pfsense/en/latest/book/hardware/hardware-sizing-guidance.html#table-pfsense-hardware>

picked since it operates on medium-sized input data sets (4096 bytes) and performed very well during parallelization. USB and SPI are chosen as embedded microcontroller interfaces which also deliver the required throughput for a parallelized application. An Ethernet peripheral would also be an option, but it is already used for the Firewall example. SpartanMC is chosen as target platform since the provided MicroBlaze USB peripheral requires an additional ULPI interface chip which is not available on the used Nexys-Video FPGA board. The SpartanMC has a USB1.1 peripheral to directly interface the USB wires. USB is used as input interface and SPI as output interface. Thus, the ADPCM compressed audio could be for example stored on an SD-card, which has an SPI interface.

Table 6.7: SpartanMC ADPCM performance-profile with peripheral IO: In units of 10^3 cycles (rounded)

Processing step	simulated IO	peripheral IO
0: receive input	29	388
1: read adaptive input	624	624
2: auto correlation	7	7
3: extract eqn. system	1	1
4: solve eqn. system	139	139
5: back-substitution	9	9
6: write coefficients	1	1
7: compression loop	1060	1060
8: write results	11	254
initiation interval (Σ)	1880	2483

A new application profile for ADPCM is generated in Table 6.7 to see the influences of the peripheral interaction. However, since the algorithm stays the same, only the duration for input and output changes. In the previous examples, IO is modeled with a memory copy. The memory copy function is implemented very efficiently and also has no control overhead, thus imposes an ideal case. With real peripherals there is a higher control overhead and also data preprocessing is necessary for peripheral interaction as shown in Table 6.7. The results show that peripheral IO holds a notable part of the applications runtime. In numbers, this is 25% of the total application runtime just for peripheral interaction.

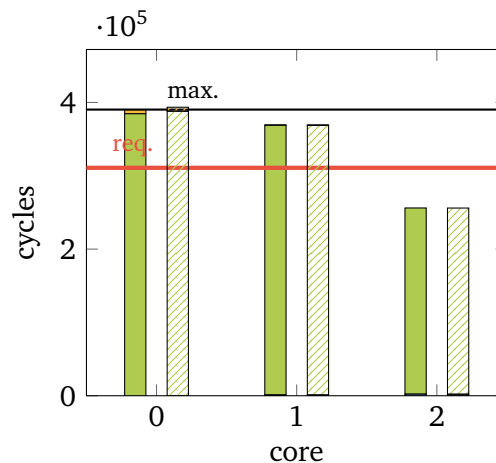


Figure 6.22: Duration per SpartanMC core with ADPCM 8x speedup requirement, core 1: 5x replication

A parallelization is started with an 8x speedup requirement and the new performance-profile. AutoStreams suggests a three stage pipeline with the intermediate core five times replicated and a 6.3x speedup. AutoStreams recognizes that a higher replication or more cores gives no benefit, since reading input data becomes the critical step and cannot be parallelized due to peripheral interaction.

Figure 6.22 shows the runtime per core. It is noticeable that the measured speedups are lower compared to a system without peripheral IO and peripheral interaction becomes critical. Also, the communication overhead becomes nearly negligible in the example since the calculation effort is much higher than the communication overhead. Peripherals with better performance (USB2/3 or Quad SPI) are required to achieve higher speedups. Unfortunately, these are not implemented in the SpartanMC SoC-Kit.

In conclusion, AutoStreams is also well applicable for real systems with peripheral IO. Successful parallelization also depends on the peripheral's performance. The other benchmarks could also be parallelized with peripherals since the only change is the duration of input and output depending on the data size.

6.6 Manual vs. Automatic Parallelization

To evaluate the quality of the solutions provided by AutoStreams, it is tried to generate a hand optimized solution which is better than the automatically parallelized one. As test case, a relatively bad automatically parallelized solution is taken which does not fulfill the requirements and has a relatively unbalanced pipeline. A bad automatically parallelized design as comparison represents an easier opponent to beat with manual parallelization. A not ideally automatically parallelized example is the ADPCM 12x speedup requirement with DMA and loop optimizations as shown in Figure 6.11c (page 111).

The process for finding a manual solution is described in the following. It is only tried to generate an ideally balanced pipeline during manual parallelization. The communication overhead and a decision to use FIFO or DMA-based interconnects is not manually evaluated. This decision makes finding a manual solution a lot easier, since finding data dependencies is extremely exhausting and error prone when done by hand.

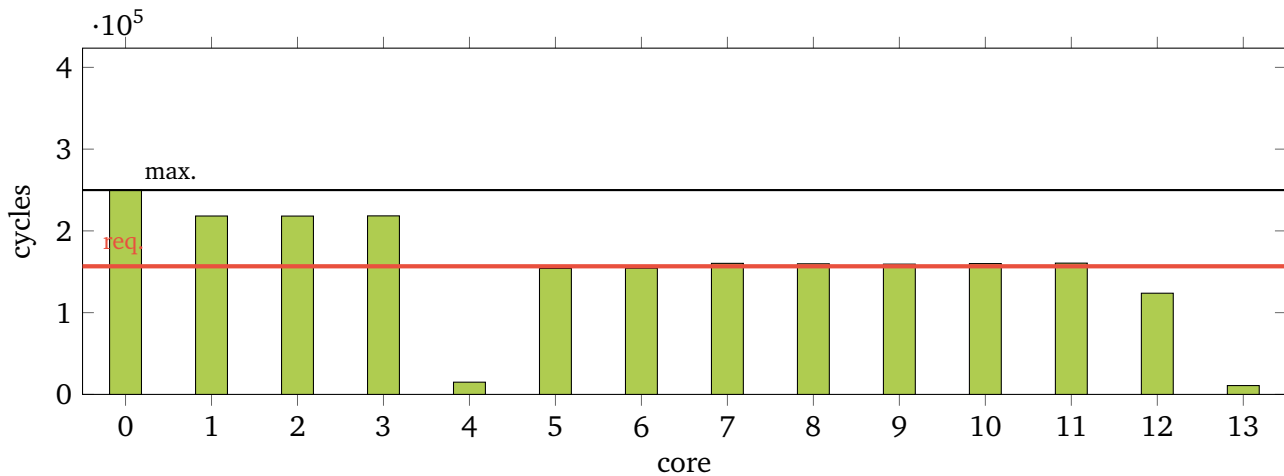


Figure 6.23: ADPCM 12x speedup requirement with DMA and loop optimizations, manually parallelized, first try

At first, the ADPCM application-profile in Table 6.1 (page 99) is analyzed. It is decided to go with a 13-core design since it is unlikely that 12 cores will fulfill the requirement. As a first try, a similar application partitioning as suggested by AutoStreams is taken. Also, the partitioned loops have equal partitioning points. This is the natural selection when calculating with equal runtimes for each loop iteration.

The first performance-profile in Figure 6.23 looks similar to the one from the automatically parallelized design after running μ Streams and synthesizing the design. Core one to three have a significantly higher runtime than expected and core four a significantly shorter runtime. These cores hold partitions of application step one from Table 6.1. Also, the loop partitions of the application step seven from Table 6.1, mapped to cores 6 to 12, were not always meeting the requirement.

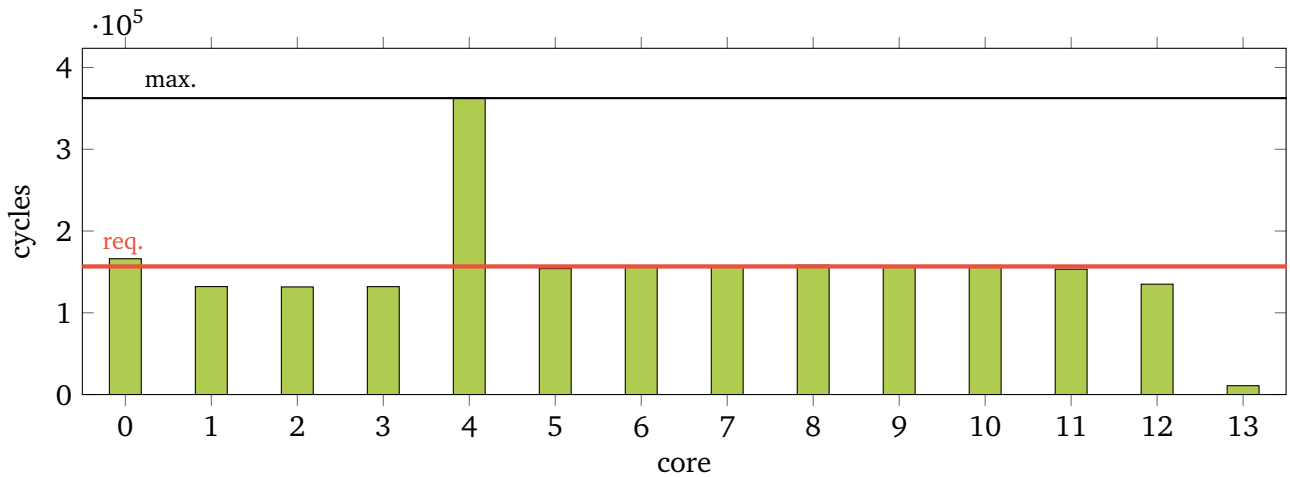


Figure 6.24: ADPCM 12x speedup requirement with DMA and loop optimizations, manually parallelized, second try

In the next iteration, the relative difference of each core with the requirement is calculated and the iterations adapted accordingly. Fortunately, BRAM content can be updated in the bit-file (FPGA configuration file) if only the software is changed, which makes a new synthesis (roughly 20 minutes) unnecessary. The measured parallelized performance profile in Figure 6.24 shows that now core four, which previously had a very short runtime, is highly overshooting the requirement. Thus, the linear scaling of runtime to iterations did not work for the partitioned loop of application step one of Table 6.1. Even though, it works well for the partitioned loop of the application profile step seven of Table 6.1. This is caused by optimizations that could not be applied by the SpartanMC GCC compiler. The generated ADPCM 12x speedup requirement MicroBlaze system has similar partitioning points, but does not show the same behavior.

In the next step, the iterations of each loop partition are changed successively in small steps as well as the smaller processing steps (2-6) of Table 6.1 are shifted back and forth. After 16 iterations of carefully approaching runtimes of each core slightly below the requirement, a result with a well-balanced pipeline below the requirement is found (see Figure 6.25). This process took in total three hours of manual work including one synthesis run, manual modification of loop iterations, shifting μ Streams task pragmas, running μ Streams, compiling the application, updating the BRAM contents and programming the FPGA in each iteration.

The speedup is increased from 7.5x for the automatically generated design to 12x through the manual tuning. Nevertheless, it should be noted that the chosen automatically generated design is one of the worst performing parallelizations suggested by AutoStreams. Other automatically generated designs have no or slight room for improvement through manual parallelization. Thus, manual tuning is neither necessary, nor possible. Communication overhead calculation also is neglected for manual parallelization, due the usage of DMA-interconnects. Also, the distinguishing between more cores with slower FIFO-interconnects or fewer cores with faster DMA-interconnects is not done here. The lack of these steps makes finding a manual solution a lot easier. The user doesn't need to calculate data dependencies and hardware costs for different scenarios. Particularly the automatic communication overhead analysis resulted in slightly better designs for the MJPEG2000 (not shown here) compared to previously hand optimized systems due to partitioning point choices with less communication overhead.

Thus, finding good manual parallelizations is extremely time-consuming. Even though, sometimes it results in slightly better parallelizations. However, more often it results in equal or worse designs since communication or hardware overhead of different possible configurations is not considered.

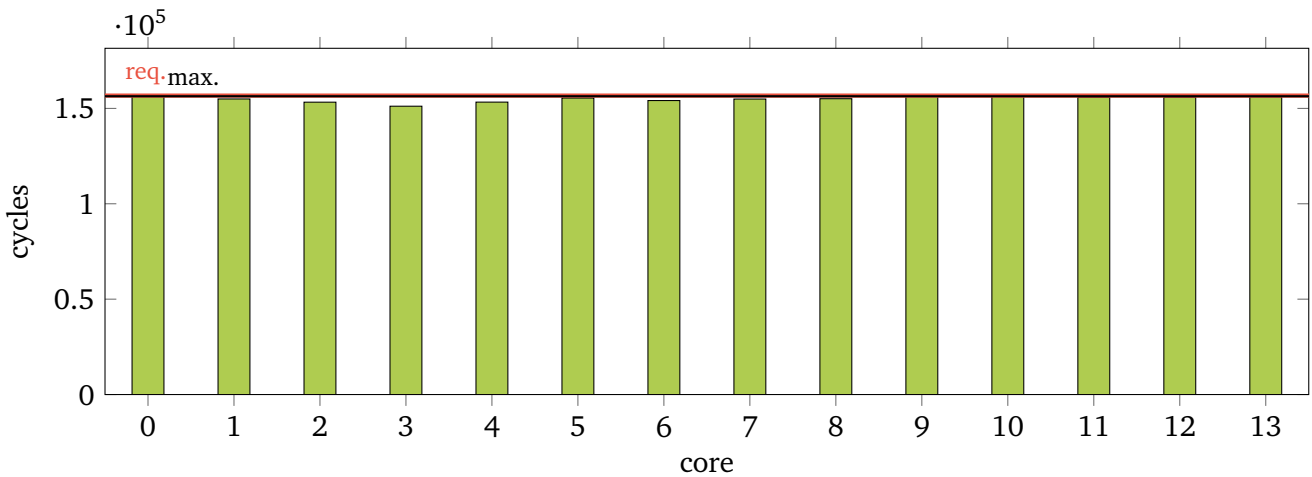


Figure 6.25: ADPCM 12x speedup requirement with DMA and loop optimizations, manually parallelized after 16 tries

6.7 Maximum Frequency Multi-Core Designs

Different proposed hardware designs can result in different achievable maximum frequencies, since FPGAs are used as target platform. Thus, through the increased hardware demand of a multi-core system the synthesis tool might not be able to place the hardware as optimally as it can be done with a single-core design. The possible reduced frequency of the multi-core system could then result in a lower speedup than expected. To evaluate the influence of varying core numbers and core-interconnect types, the generated systems for the previous benchmarks in Section 6.3 are synthesized for maximum frequency.

The Artix-7 XC7A200T is used as FPGA target platform. For synthesizing SpartanMC designs, Xilinx ISE 14.6 is used due to better SpartanMC tool support and Xilinx Vivado 2018.2.1 is used for MicroBlaze systems. The tools are run with default presets, only the ISE synthesis and mapping effort is set from normal to high.

Each design point has been synthesized multiple times with different clock frequencies to find the maximum possible frequency. The search algorithm is implemented as binary search. The algorithm first synthesizes with a relatively low frequency where the design will run for sure, afterwards at a high frequency where the design is unlikely to run. Then, an intermediate frequency is chosen. Depending on successful synthesis with the intermediate frequency, the search is continued in the lower half frequency range on failure or in the upper half on success. This process is continued until the FPGAs clock manager (DCM) does not offer smaller frequency steps. The maximum frequency where synthesis does not have timing violations is returned in the end. However, since the synthesis includes random decisions it can be that the best found solution and thereby the maximum frequency varies.

The ADPCM/IIR multi-core designs with low core numbers typically achieve 90% to 95% of the single-core design's frequency, as it can be seen in Figures 6.26 and 6.27. The synthesized designs with the highest core numbers achieve around 70% to 80% the frequency of the single-core design. Whereas the 70% achieved frequency seems to be due to synthesis variations, since designs with a higher core number achieve again higher frequencies. To verify this statement, the frequency evaluation was started again for the low peak designs with the option to try five synthesis runs with different seeds before declaring a frequency not working. This option is not enabled by default, since frequency evaluation takes then extremely long due to 30 up to 50 performed synthesis runs. However, after the exhaustive frequency evaluation, the low peak design points had successful synthesized designs with higher frequencies in the expected regions.

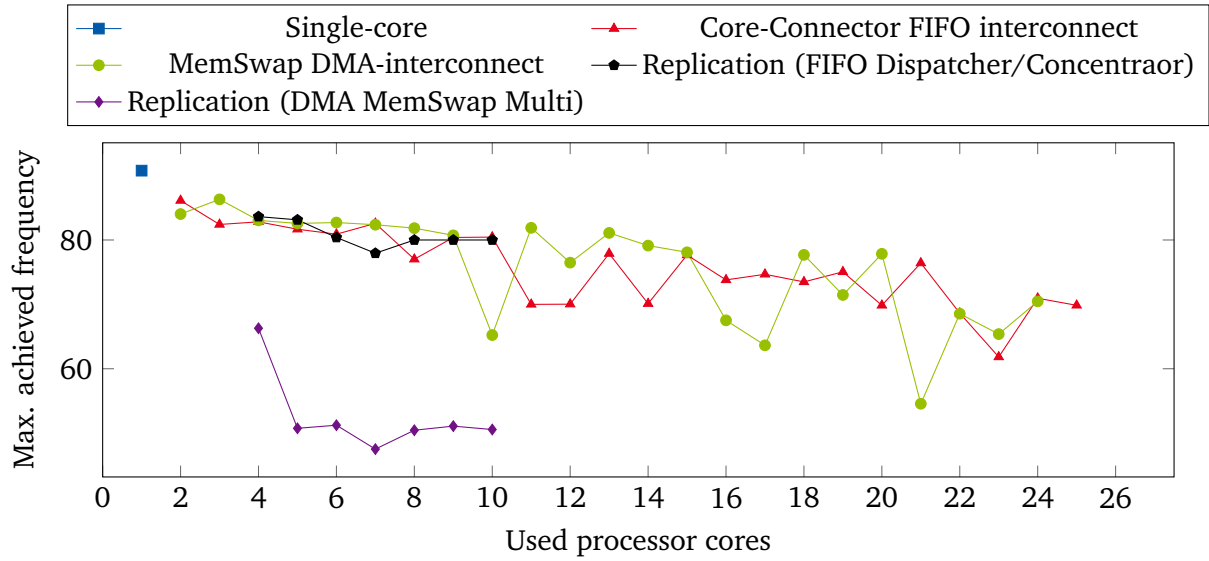


Figure 6.26: ADPCM and IIR maximum achievable frequency evaluation over multiple connected SpartanMC cores and interconnect types

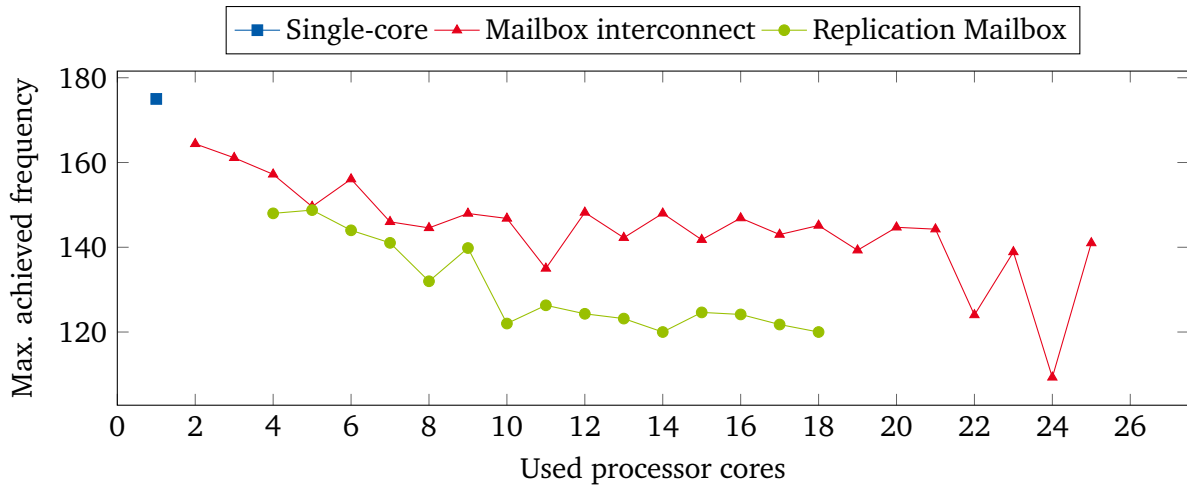


Figure 6.27: ADPCM and IIR maximum achievable frequency evaluation over multiple connected MicroBlaze cores and interconnect types

The used interconnect type does not play a significant role for the maximum frequency, except for replicated designs with MemSwap Multi DMA-based interconnects. The reason for the low achievable frequency lies in the pattern of the design and the higher BRAM usage. Fast routes in between the components work best if the cores and interconnects are packed together as closely as possible. However, there are not enough BRAMs available close by, so further away BRAMs have to be used resulting in longer and slower routes. This does not apply for other interconnects, since they either don't use BRAMs and other components are plenty locally available or the MemSwap interconnects (using BRAMs) form a long pipeline, where only the direct neighbors have to be close together. The same also applies to the replicated MicroBlaze systems that perform around 10% worse than MicroBlaze systems as multi-core pipelines.

The synthesized MJPEG designs in Figures 6.28 and 6.29 show that even the largest designs with maximum BRAM usage achieve 70% to 80% of the single-core frequency, just like the largest ADPCM/IIR designs. For the same core number, the MJPEG designs achieve only around 75% of the ADPCM/IIR

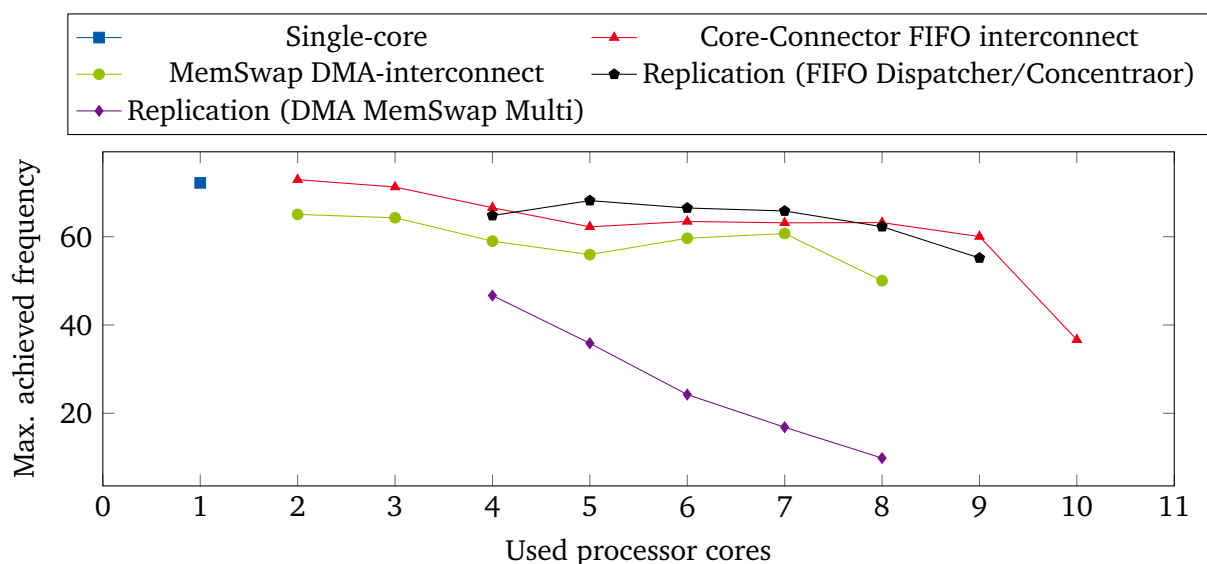


Figure 6.28: MJPEG2000 maximum achievable frequency evaluation over multiple connected SpartanMC cores and interconnect types

design frequencies. Due to the higher required local memory and thereby the increased BRAM usage, the BRAMs further away have to be accessed which results in long routes. Significant is the frequency drop for the ten core SpartanMC design. This might be caused by the design using most of the available BRAMs requiring long routes to reach far away BRAMs. However, other designs with similar BRAM usage do not have such a significant frequency drop. Also, the replicated MicroBlaze designs show a significant behavior. A design with a higher core count achieves a higher frequency, which is counter-intuitive. Thus, these designs were investigated further and the top critical paths were reviewed. Also, these designs were again synthesized multiple times with different seeds and the critical paths again inspected. Two factors for the low performance are identified during this process. The significant designs use almost all the FPGA's BRAMs which are equally distributed over the FPGA. The critical path always goes from the processor core to some BRAM used as memory and the path consists 80% of routing delay. Also, all processor cores are always placed closely together. Thus, the first observation is, that the designs that have cores placed closer to the center of the FPGA achieve a higher frequency. Furthermore, cores use mostly BRAMs as memory close to the position of the core on the FPGA. The second observation is, that some cores use most BRAM close to the core and some BRAMs are located extremely far away. Such that the connection to the BRAM needs to be routed from one side of the FPGA to the other. It seems as if the cores could swap some BRAMs, long paths could be avoided. The significant designs with low frequencies both showed two described effects. With new synthesis runs and other seeds higher possible frequencies were observed during this process.

Synthesized designs for the firewall are not shown since the critical path goes through the Ethernet peripherals and the influences from multiple cores are not observable.

6.7.1 Speedup vs. Performance Loss through Lower Frequency

Achievable speedups in contrast to the performance loss through lower frequencies for multi-core designs are mostly not significant. One exception are replicated systems with DMA-interconnects. Achievable speedups of up to around 12x in terms of clock cycles would result in a speedup of 8.4x in wallclock through the 30% lower frequency of the multi-core design. Thus, it is not that impressive anymore, but still a very positive balance.

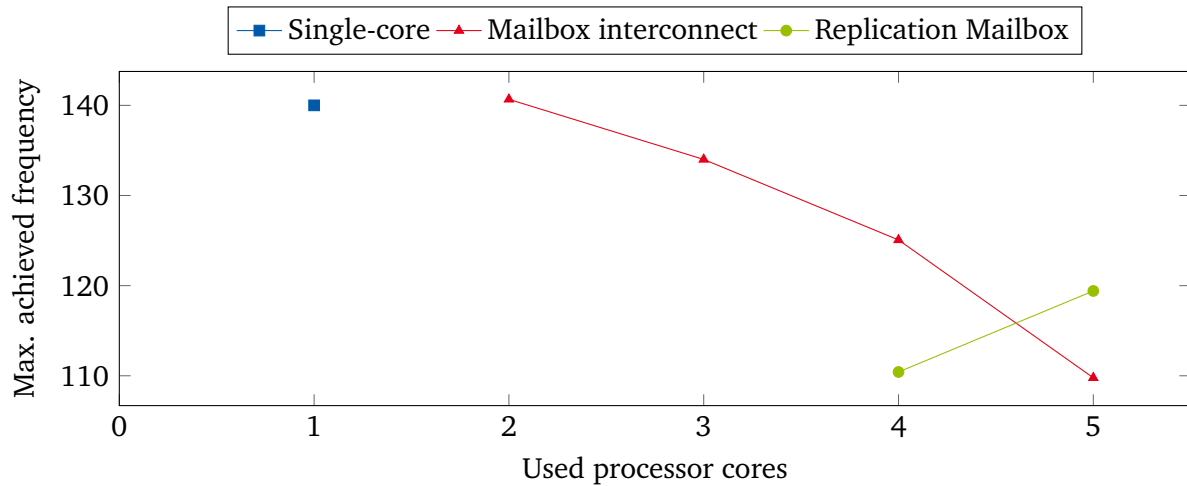


Figure 6.29: MJPEG2000 maximum achievable frequency evaluation over multiple connected MicroBlaze cores and interconnect types

It is also observable that replication with DMA-based interconnects results in quite low frequencies, much lower compared to an equivalent FIFO-based interconnect. Thus, the increased performance through the faster interconnects mostly vanishes. For example the replicated SpartanMC MJPEG design in Table 6.5 achieves a speedup of 8x with DMA-based replication and 7.01x with FIFO-based replication in terms of clock cycles. However, DMA-based replication achieves only 15% and FIFO-based replication 75% of the single-core design's frequency. Thus, the resulting speedup in wall-clock time is only 1.2x with DMA-based replication, but 5.2x with FIFO-based replication. Thus, using DMA-based MemSwap Multi modules for replication is mostly not advisable. To maintain higher frequencies, the FIFO-based Dispatcher and Concentrator modules should be favored, even though their slower transmission speed.

It would be useful to integrate an achievable frequency estimator into AutoStreams. Interconnects could be chosen based on expected frequencies and also the performance loss through a lower frequency could be compensated automatically by AutoStreams. However, these results only count for the used Artix-7 FPGA device. For other FPGA families the results could look different, thus the findings in this sections have limited transferability and are not universally valid.

6.8 Latency in the Generated Pipelines

The influences on latency are evaluated here since pipelines are known to increase the throughput at the cost of latency. For some embedded systems latency is an important number, since they need to react on inputs within a certain time frame.

Different parallelized systems from Section 6.3.4 are used to retrieve latency numbers. These systems were chosen since they have more pipeline stages than other evaluated systems and latency mainly depends on the characteristics of the different stages.

As it can be seen in Figures 6.30 and 6.31, systems with fewer cores mostly result in a lower latency. Fewer pipeline stages result in fewer data transfers and thereby a lower latency. The IIR benchmarks have a comparably high latency. On the one hand IIR systems have a relatively large communication overhead compared the processing time. On the other hand, the 12 core system also requires many data transfers which also increases the latency. However, it can also be observed at the MicroBlaze ADPCM 12 core and 16 core systems that more cores result in lower latency, which seems contradictory. Thus, it also plays a role for the latency, at which position in the pipeline the critical stage is located, according to Equation (4) page 55. With respect to this equation, it is beneficial to have a well-balanced pipeline

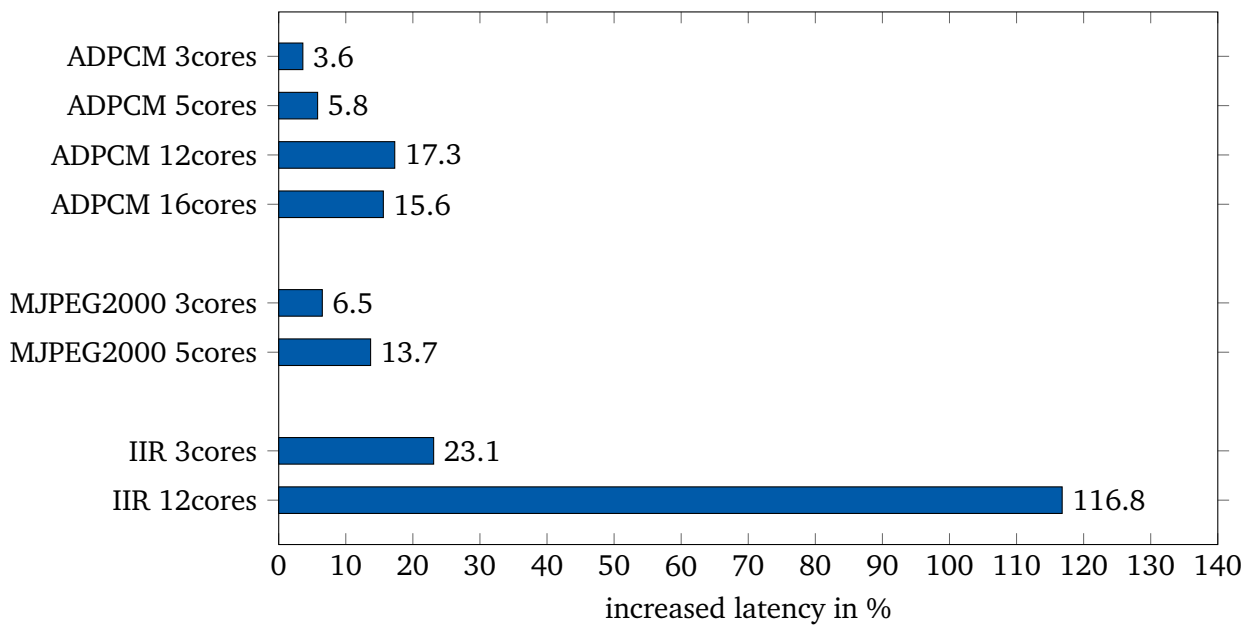


Figure 6.30: Latency increase compared to the sequential variant with MicroBlaze

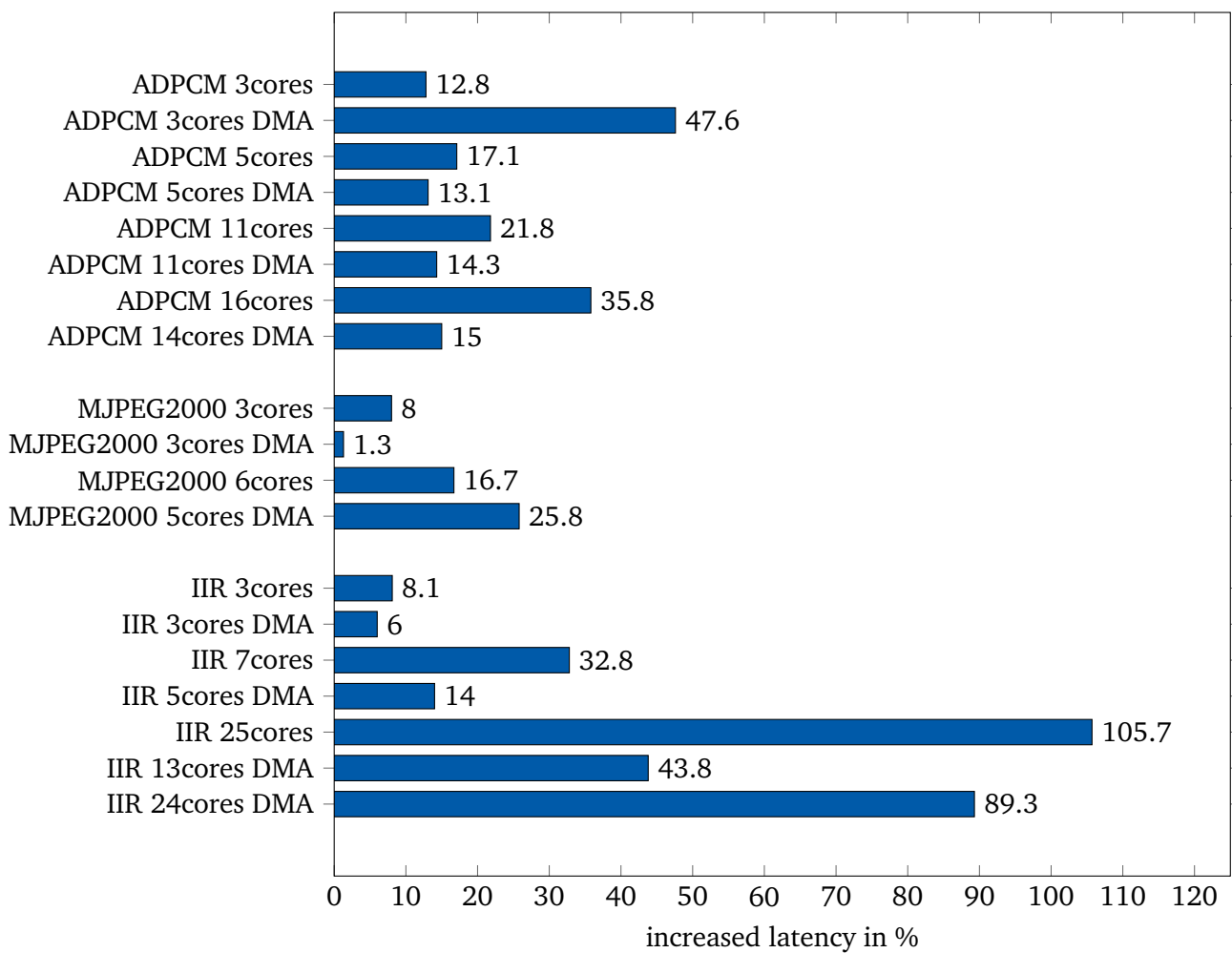


Figure 6.31: Latency increase compared to the sequential variant with SpartanMC

and the critical pipeline stage as early as possible to achieve a low latency. For example the SpartanMC ADPCM 3 core DMA system has the second core as the critical pipeline stage and the first core only has 3% of the second core's duration. The first core has to wait idle for a long time until the second core can receive data from the first core, thus a 47.6% higher latency is caused.

In conclusion, latency is doubled with the worst benchmark. Mostly, only a 10% to 20% increased latency is observed. The impact on the latency are not as dramatic as expected. In conclusion, the throughput speedup grows at much higher rates compared to the latency increase. It depends on the purpose and requirements of the embedded system if the increased latency is still acceptable.

6.9 Dynamic Verification: System Tests

Not only the shown benchmarks are used to verify the correctness of the implemented tools. Over 100 small test applications have been written to test different aspects. These tests are implemented as system tests, taking around three minutes to execute, to ensure consistent behavior after implementation modifications. The tests include handling of different C constructs and their correct transformation as well as tests for the generated multi-core architectures. The starting point for the tests is typically a minimal C-program with around 10 lines to cover the construct to be tested. A pre-parallelized reference (software and hardware description) is also contained as golden sample to check against. Besides tests checking the correct parallelization, there also exist tests to verify behavior with unsupported constructs and error handling.

6.10 Comparison with Related Work

It is essential to take the same benchmark for a realistic comparison to show the performance of the presented tools compared to related work. The only benchmark that is often found in related work is ADPCM. The tool MAPS[42] from Cheng et al. and Eldorado[37] from Cordes both used the ADPCM as benchmark.

MAPS is a semi-automatic C parallelizer for embedded systems. Sequential C applications can be profiled and parallelized. The tool automatically gives parallelization hints and the user can revise the suggestions in a GUI. MAPS also accepts applications described as C for Process Networks (CPNs), a custom KPN representation, for easier parallelism detection. The tool creates transformed parallel C source-code for different target platforms. The generated tasks are statically mapped to processing cores. A detailed description is given in Section 2.3.5.

Eldorado[37] is an automatic C parallelizer, also especially developed for embedded systems. Eldorado uses internally the MPA[45] tool, to extract parallelism and dependencies. It also cares for the extraction of properly sized tasks with respect to communication and task creation overhead.

Eldorado is evaluated on an ARM11 quad-core processor and MAPS uses a custom RISC multi-core design[84] with a full crossbar interconnect. There exists an octa-core implementation on silicon of this MP-SoC. However, an according cycle-accurate simulator with a configurable core number is used to evaluate the generated MAPS designs.

MAPS achieves a maximum speedup of 1.28 with three generated threads. The parallelization took 30 minutes. Eldorado is able to achieve a 2x speedup and the automatic parallelization took less than a minute. In contrast to the other tools, AutoStreams is able to extract a speedup of 12x and higher at a reasonable hardware overhead. Just like Eldorado, parallelization with AutoStreams takes less than a minute, not including synthesis time.

To get an idea of speedups achieved by other **automatic** parallelization tools for other benchmarks, two other tools besides Eldorado and MAPS are picked: AutoPar[26], PIPS[62]. Other automatic parallelization tools in Section 2.3.5 are omitted, since they only focus on loop and not full system parallelization.

AutoPar achieved a nearly linear scaling speedup up to eight threads with 8x speedup for three out of four benchmarks on a system with two quad-core server CPUs (2x Intel X5460). One benchmark only results in a 2x speedup.

Speedups with PIPS are shown in [62], with two dual socket, hexa-core server systems. Two out of the presented four benchmarks show speedups of 4x to 5x with eight cores, while the other benchmarks are only able to leverage six cores and achieve speedups of 2x to 3x.

The authors of MAPS also publish speedups for a JPEG encoder in [42, 43]. With MAPS' automatic parallelization, a speedup of 3.6x is achievable with 16 cores. A speedup of 5.5x is possible with the same number of cores after some manual tweaking. With the application reformulation into a KPN representation, a speedup of 4.1x is automatically extractable and a speedup of 9.5x is reached after manual tweaking.

Eldorado's speedup results are shown in [37] on a simulated quad-core architecture. Seven benchmarks are presented, where one application can be parallelized with a speedup of 3.7x. The average achieved speedup is at 2.7x.

In conclusion, parallelizers often use server multi-cores, shared-memory systems or simulators to evaluate their results. Simulators have questionable transferability of the results to real systems and existing architecture's scalability is often limited by a fixed number of cores. Using FPGAs as evaluation platform has the flexibility of a simulator to vary interconnect types and core numbers, but on the other hand the design runs on real hardware for transferability. AutoStreams has shown speedups of 12x and more. The MJPEG2000 benchmark yields the worst results with only a speedup of 4x. On average, AutoStreams achieves higher total and average speedups compared to other tools. However, it should be noted that all related tools target for optimizing latency and thereby also achieve a throughput speedup. AutoStreams increases the throughput through pipelining at the cost of the latency. Compared to the other tools, AutoStreams is not applicable to increase the latency through parallelization. However, when it comes to an increased throughput, AutoStreams can well play its advantages.

6.11 Best Practice Proposals

This section summarizes the different lessons learned for parallelized hardware designs and software partitioning methods.

In terms of minimal hardware, a pure pipeline is better than replication with the same number of cores. Using FIFO interconnects always requires less resources compared to DMA interconnects. Also, replication with FIFO interconnects requires fewer resources than a pipeline with DMA interconnects.

FPGA designs with shared components on multiple processors result in relatively low frequencies. Thus, global memories or DMA interconnects with replication are not desirable.

From the software side, loop splitting makes it possible to balance pipeline stages, since often loops dominate the application runtime. A pure pipeline works well for applications with splittable loops or high numbers of processing steps. Replication reduces communication overhead and larger source-code parts can usually be mapped to the replicated cores supporting better GCC optimizations. Replication also has a lower latency from input to output due to the reduced communication overhead. From software side, DMA interconnects produce parallelizations with higher throughput (neglecting maximum frequency).

Each core holds smaller source-code fractions for pipelines without replication. This would usually result in an overall lower memory footprint compared to replication. However, this aspect is not optimized in this work and has potential for improvement.

7 Conclusion & Future Work

Multi- and many-core embedded systems offer great processing power even in the low-power low-performance domain. However, software developers struggle to leverage the offered processing power. New concepts and tools are required to extract the necessary parallelism. Most of the offered tools are focused on extracting data-parallelism in the context of desktop or HPC systems. Only few tools exist which focus on low-performance embedded systems. These tools seldom extract pipeline parallelism, even though it is well extractable from many embedded applications. In related work, the pipeline structure is mapped to general purpose static multi-core architectures, not optimal for the extracted parallelism. The parallelism extraction often only focuses on loop parallelization without a view on the whole embedded system with peripheral interaction. Soft-core SoCs offer opportunities to configure a many-core environment ideally fitting the structure of the extracted pipeline parallelism.

In this work, a set of tools is presented which support the developer with the full spectrum required for a successful parallelization. Most of the tools require very little to no user interaction. The tool support ranges from automatic application profiling to discover bottlenecks, over automatic parallelization, automatic many-core design generation up to an automated performance evaluation of the parallelized application. The automatic parallelization is the key contribution of this work. It offers the specification of a minimal required processing speed to the user and the parallelizer extracts the necessary parallelism automatically. At the same time, the hardware is kept minimal with respect to number of processing cores and core-interconnect types. The goal to extract only necessary parallelism is realized through the automatic parallelizers design space exploration in Section 5.5.2.3 by evaluating all possible parallelizations and selecting a solution with minimal hardware, just fulfilling the user's timing requirement. The hardware's minimalism and choices of the design space exploration are reflected and checked for various design points in Section 6.3. The goal to consider parallelization restrictions with regard to peripheral interaction is realized through the implementation of a peripheral detector in Section 5.7. This section shows how peripherals are detectable in the source-code and also tells the resulting restrictions for parallelization. The most important restriction for MP-SoCs is that one peripheral can only be dedicated to one core. This restriction is implemented in the automatic parallelizer. The consideration of peripheral interaction and the used concept of pipeline parallelism make the approach suitable for the initially stated goal of full-system parallelization. Section 6.5 also shows the successful automatic full system parallelization with two real world systems having peripheral input and output. Peripheral interaction is often neglected in related work. It is shown in this work that peripheral interaction can hold large parts (25%) of the total application's processing time and should thus also be considered for parallelization.

The usage of freely configurable soft-core processors is proven to adapt very well to the requirements of the parallelized application. The selected soft-core processors fulfill the initially stated requirement of low-performance embedded systems as target platform. Extracted pipelines are directly transferable to a multi-core pipeline as stated in Section 4 and proven with many parallelized designs during evaluation. The soft-core's interconnect performance is evaluated in Section 3.3.1 and Section 3.4. These sections also propose new interconnect types specifically designed for pipeline parallelism and show how existing interconnects are tweaked to peak performance, such that higher parallelization gains become possible. It is shown in Section 6.3 that soft-cores are well performing as distributed-memory systems despite large communication overheads in many cases. It is also shown in this section that a high amount of parallelism is extractable: For example a 12x speedup is possible for one application on a moderate 14-core system. These numbers are out of reach for related approaches which mostly use shared-memory architectures. These facts also answer the initial question of the applicability of pipe-parallelism as well as the usage of distributed-memory systems for this concept. It is also shown in Section 6.5.1 that soft-core shared-memory systems run into a saturation for speedup gain earlier than it is observed with distributed-memory soft-core designs through communication overhead in Section 6.3.

Thus, an optimized many-core system can automatically be generated with soft-core processors for each parallelized application. The necessity to only specify the source-code and minimum processing speed offers the freedom of a configurable hardware architecture to software developers that are not experienced in hardware design. The fact that no automatically parallelized hardware design had to be touched or tuned during evaluation in Section 6.3 shows that automatic hardware creation works great.

Furthermore, different optimizations are proposed in order to extract more pipeline parallelism from applications. The constant usage of source-to-source transformation leaves the concept open to new architectures as well as further optimizations such as specific accelerators. Also, other kinds of parallelism could still be extracted. The user also has the chance to review the parallelized design in each step and modify it, if desired. Generally unsupported constructs can still be realized with a few tweaks in the generated source-code, but without the need to modify the parallelizer itself.

During the evaluation, the presented approach delivers high throughput speedups of 8x to 12x and in some cases even higher. Compared to related approaches, which use static hardware or extract data parallelism, much higher speedup gains are possible. It is shown in Section 6.10 that related approaches are able to parallelize the ADPCM application with a speedup of 1.2x to 2x, while the tool presented in this work achieves speedups of up to 12x for the same application. The automatic parallelization is able to optimize software and hardware with multiple objectives. Even experienced users struggle to reconcile the different options and possibilities. It is shown in Section 6.6 that hand parallelized solutions only beat the first shot of the automatic parallelization, but cost several refinement iterations and hours of work. However, these refinements are also possible after automatic parallelization due to the source-to-source concept. Often, a manual parallelization is not necessary, since the automatic parallelization already delivers a well optimized system fulfilling the users requirements. The presented automatic parallelization tool is also able to predict the performance of the parallel design very accurate with an estimation error below 10% in most cases, as shown in Section 6.4.2.

The influence on the application latency is shown in Section 6.8 and is evaluated to be increased through pipelining by usually only around 10% to 20% and 120% in the worst scenarios. Thus, the initial question for effects on the latency can be answered with: Yes, it slightly effects the latency, but the massive increased throughput outweigh the increased latency for throughput focused systems.

The clock frequency degradation caused by multiple soft-cores on one FPGA is another concern stated at the beginning of this work. The frequency degradation between single-core and multi-core designs is shown to exist in Section 6.7. In this section the influence on the frequency of different interconnect types and number of used cores is evaluated. It is found, that the frequency degrades only slightly with more cores. The chosen interconnect types and the used memory per core have a bigger influence. It is shown that the usage of MemSwap Multi module and a large memory size per core require long routing paths over the FPGA which results in low achievable frequencies. For example designs with the MemSwap Multi interconnect achieve 20% up to 50% of the single-core design's clock speed. All other interconnects reach, depending on the core count, 75% up to 95% of the single-core design's clock speed. Thus, the MemSwap Multi interconnect should be avoided and equivalent interconnects with a lower transmission rate but higher clock frequencies should be favored. Even though there is a slight frequency degradation for these interconnects, still remarkable speedups are achievable.

In conclusion, the formulated questions are answered, the initial concerns were either baseless or solved and the defined goals are fully achieved. To the best of my knowledge, no tool exists with the same scope covering full system parallelization of embedded systems with configurable multi-core architectures. However, there is still room for improvement. Also, new problems and aspects have arisen during this work which could be subject to future work:

Automatic refinement: For a further automation of the proposed toolflow, it would be possible to automatically read back the parallel performance-profile with AutoStreams, evaluate it and adapt the

proposed parallelization in a second iteration if requirements are not fulfilled. Currently, the user has to take appropriate actions but it is thinkable to automate this step.

Consistent GCC optimizations: The SpartanMC GCC compiler is observed to deliver inconsistent results for executed assembler optimizations. The MicroBlaze GCC shows more consistent assembler optimizations in the sequential and parallelized designs. This yields a better performance predictability and well-balanced pipelines. Some missing optimizations in SpartanMC GCC have already been pointed out with examples.

Global memory support: Since this contribution mainly lays focus on distributed-memory systems, shared-global memory handling is not extensively supported. There is currently room for improvement to support synchronous global memory accesses. It is thinkable to automate synchronization of critical memory sections with mutexes and semaphores. However, it is questionable whether automatically inferable accesses would result in too pessimistic constructs. Maybe, user annotations could relax some restrictions with a bit manual effort and application knowledge. If the global memory was integrated better, it would make sense to implement a performance estimator for a parallelized system with global memory just like the estimators for the core-interconnects. Work in the field of global memory and WCET estimators often predict extremely pessimistic results and show that it is not easy to predict performance for shared-memory multi-core architectures[57]. Since it is already well possible to predict execution time for pipelined distributed-memory systems, a WCET analysis of such a system might deliver less pessimistic worst-case speedups.

Memory size optimization: Currently, the memory of each core in the generated multi-core design is designed with the same memory size as the single-core system. This assumption is very pessimistic, since each core only holds a fraction of the single-core application. At the moment, a student thesis is in progress to analyze maximum used memory for code and data sections as well as required stack size from GCC intermediate files. The necessary memory size for each core could be optimized with such a tool and the overall resources, especially precious BRAMs, could be saved.

MicroBlaze DMA interconnects: In contrast to SpartanMC, MicroBlaze has no equivalent to the Mem-Swap DMA interconnects to allow fast data exchange. This DMA memory is directly integrated into the processor's memory range and has the same latency and throughput as the usual local memory. The default peripheral interface to realize such a solution is AXI for MicroBlaze. In contrast to the local memory interface (LMB), AXI has a higher latency and lower throughput. Another option would be to use the MicroBlaze local memory interface. Any additional hardware on this bus influences the critical path of the processor. It would be unclear for any approach if this concept yields the same gain as for SpartanMC.

Statement reordering: Currently, the automatic parallelization uses the source-code as it is to extract a pipeline. It is thinkable that some code fragments could be relocated after a dependency analysis. This might sometimes help to reduce communication overhead and result in a more balanced pipeline.

DSE for larger applications: Even though parallelization is possible within a minute for all used benchmarks, significantly larger programs might run into performance issues with the branch-and-bound algorithm. It might be necessary to implement a search heuristic like simulated annealing or a genetic algorithm to find good solutions during design space exploration in the future. However, this step would sacrifice accuracy for search speed. Currently, the upper bound algorithm still outputs a valid solution even if exhaustive search takes too long.

Frequency estimation: It is desirable to have a maximum possible frequency estimator for the different multi-core designs integrated in AutoStreams. Thus, multi-core designs resulting in a higher frequencies could be preferred during design space exploration in AutoStreams. Since embedded

systems also contain peripherals, measurements for all peripherals at different configurations might be required to get accurate estimations.

Energy consumption: The energy consumption of a multi-core design is completely neglected during design space exploration. It would make sense to have measurements or estimators for this, since power consumption is sometimes a critical factor for embedded systems.

Accelerators & other parallelism: The focus of all presented tools lies on source-to-source transformations to enable later modifications, among other reasons. It would be interesting to see if the parallelized design can be further accelerated by partial application execution in external accelerators. These accelerators could be nicely implemented on unused FPGA area. Also, opportunities for further parallelization exploiting data-parallelism could be investigated.

Other multi-core architectures: FPGAs have the drawback to be quite limited in frequency compared to an ASIC. Even though, they are an excellent vehicle for investigating new digital designs and testing. However, fine-grained reconfigurability is not required for the presented concept. It could be beneficial to use a multi-core design with fixed processors but configurable interconnect network at compile time like presented with XGRID[19]. Such an approach would result in higher clock frequencies and lower power consumption if implemented as an ASIC.

References

- [1] Andrs Vajda. *Programming Many-Core Chips*. Springer Publishing Company, Incorporated, 2011.
- [2] J. Svennebring et al. *Embedded Multicore: An Introduction*. Tech. rep. 2009.
- [3] S. Pllana and F. Xhafa. *Programming Multicore and Many-core Computing Systems*. Wiley Series on Parallel and Distributed Computing. Wiley, 2017.
- [4] William Thies, Vikram Chandrasekhar, and Saman Amarasinghe. “A Practical Approach to Exploiting Coarse-Grained Pipeline Parallelism in C Programs”. In: *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 40. Washington, DC, USA: IEEE Computer Society, 2007, pp. 356–369.
- [5] Georgios Tournavitis and Björn Franke. “Semi-automatic Extraction and Exploitation of Hierarchical Pipeline Parallelism Using Profiling Information”. In: *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*. PACT ’10. Vienna, Austria: ACM, 2010, pp. 377–388.
- [6] Daniel Cordes et al. “Automatic Extraction of Pipeline Parallelism for Embedded Heterogeneous Multi-core Platforms”. In: *Proceedings of the 2013 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. CASES ’13. Montreal, Quebec, Canada: IEEE Press, 2013, 4:1–4:10.
- [7] ARM. *big.LITTLE Technology: The Future of Mobile; Making very high performance available in a mobile envelope without sacrificing energy efficiency*. Tech. rep. 2013.
- [8] Harm Munk et al. “ACOTES Project: Advanced Compiler Technologies for Embedded Streaming”. In: *International Journal of Parallel Programming* (2010).
- [9] Clearspeed. *CSX Processor Architecture*. Tech. rep. 2007.
- [10] S. R. Vangal et al. “An 80-Tile Sub-100-W TeraFLOPS Processor in 65-nm CMOS”. In: *IEEE Journal of Solid-State Circuits* 43.1 (Jan. 2008), pp. 29–41.
- [11] U. J. Kapasi et al. “The Imagine Stream Processor”. In: *Proceedings. IEEE International Conference on Computer Design: VLSI in Computers and Processors*. Sept. 2002, pp. 282–288.
- [12] J. A. Kahle et al. “Introduction to the Cell multiprocessor”. In: *IBM Journal of Research and Development* 49.4.5 (July 2005), pp. 589–604.
- [13] TIRIAS-Research. *AMD Optimizes EPYC Memory with NUMA*. Tech. rep.
- [14] Andreas Olofsson. *Epiphany-V: A 1024 processor 64-bit RISC System-On-Chip*. Tech. rep. Oct. 2016.
- [15] H. Xu et al. “A low power many-core SoC with two 32-core clusters connected by tree based NoC for multimedia applications”. In: *Symposium on VLSI Circuits (VLSIC)*. June 2012, pp. 150–151.
- [16] J. Irza, M. Doerr, and M. Solka. “A third generation many-core processor for secure embedded computing systems”. In: *IEEE Conference on High Performance Extreme Computing*. Sept. 2012, pp. 1–3.
- [17] R. Poss et al. “Apple-CORE: Harnessing general-purpose many-cores with hardware concurrency management”. In: *Microprocessors and Microsystems* 37.8, Part C (2013). Special Issue on European Projects in Embedded System Design: EPESD2012, pp. 1090–1101.
- [18] Martin Daněš et al. *UTLEON3: Exploring fine-grain multi-threading in FPGAs*. Springer, Nov. 2011, pp. 1–219.

-
- [19] V. Gunes and T. Givargis. “XGRID: A Scalable Many-Core Embedded Processor”. In: *IEEE 17th International Conference on High Performance Computing and Communications, IEEE 7th International Symposium on Cyberspace Safety and Security, and IEEE 12th International Conference on Embedded Software and Systems*. Aug. 2015, pp. 1143–1146.
- [20] G. Theodoridis, D. Soudris, and S. Vassiliadis. “A Survey of Coarse-Grain Reconfigurable Architectures and Cad Tools”. In: *Fine- and Coarse-Grain Reconfigurable Computing*. Dordrecht: Springer Netherlands, 2007, pp. 89–149.
- [21] Joel Seely, Srikanth Erusalagandi, and Jayson Bethurem. *The MicroBlaze Soft Processor: Flexibility and Performance for Cost-Sensitive Embedded Designs*. Tech. rep. 2017.
- [22] Gerald Hempel and Christian Hochberger. “A resource optimized Processor Core for FPGA based SoCs”. In: *10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD)*. Aug. 2007, pp. 51–58.
- [23] David Castells-Rufas, Albert Saá-Garriga, and Jordi Carrabina. “Energy Efficiency of Many-Soft-Core Processors”. In: *CoRR abs/1601.07133* (2016).
- [24] M. Thompson et al. “A framework for rapid system-level exploration, synthesis, and programming of multimedia MP-SoCs”. In: *5th IEEE/ACM/IFIP International Conference on Hardware-/Software Codesign and System Synthesis (CODES+ISSS)*. Sept. 2007, pp. 9–14.
- [25] E. F. Deprettere et al. “Affine Nested Loop Programs and their Binary Parameterized Dataflow Graph Counterparts”. In: *IEEE 17th International Conference on Application-specific Systems, Architectures and Processors (ASAP’06)*. Sept. 2006, pp. 186–190.
- [26] Chunhua Liao et al. “Semantic-Aware Automatic Parallelization of Modern Applications Using High-Level Abstractions”. In: *International Journal of Parallel Programming* (2010).
- [27] Ian Buck et al. “Brook for GPUs: Stream Computing on Graphics Hardware”. In: *ACM Trans. Graph.* 23.3 (Aug. 2004), pp. 777–786.
- [28] Pieter Bellens et al. “CellSs: a Programming Model for the Cell BE Architecture”. In: *ACM/IEEE Conference on Supercomputing*. 2006, p. 86.
- [29] Robert D. Blumofe et al. “Cilk: An Efficient Multithreaded Runtime System”. In: *SIGPLAN Not.* 30.8 (Aug. 1995), pp. 207–216.
- [30] C. E. Leiserson. “The Cilk++ concurrency platform”. In: *2009 46th ACM/IEEE Design Automation Conference*. July 2009, pp. 522–527.
- [31] Matteo Frigo et al. “Reducers and Other Cilk++ Hyperobjects”. In: *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*. SPAA. Calgary, AB, Canada: ACM, 2009, pp. 79–90.
- [32] John Nickolls et al. “Scalable Parallel Programming with CUDA”. In: *Queue* 6.2 (Mar. 2008), pp. 40–53.
- [33] H. Nikolov et al. “Daedalus: Toward composable multimedia MP-SoC design”. In: *45th ACM/IEEE Design Automation Conference*. June 2008, pp. 574–579.
- [34] H. Nikolov, T. Stefanov, and E. Deprettere. “Systematic and Automated Multiprocessor System Design, Programming, and Implementation”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27.3 (Mar. 2008), pp. 542–555.
- [35] Chao-Tung Yang and Kuan-Chou Lai. “A Directive-based MPI Code Generator for Linux PC Clusters”. In: *J. Supercomputer* 50.2 (Nov. 2009), pp. 177–207.
- [36] Tobias Schüle. “Embedded Multicore Building Blocks - Parallel Programming Made Easy”. In: *Embedded World Conference*. 2015.

-
- [37] D. Cordes, P. Marwedel, and A. Mallik. “Automatic parallelization of embedded software using hierarchical task graphs and integer linear programming”. In: *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. 2010.
- [38] Roger Ferrer et al. “Optimizing the Exploitation of Multicore Processors and GPUs with OpenMP and OpenCL”. In: *Languages and Compilers for Parallel Computing: 23rd International Workshop*. Springer Berlin Heidelberg, 2011, pp. 215–229.
- [39] M. Y. Wu and D. D. Gajski. “Hypertool: a programming aid for message-passing systems”. In: *IEEE Transactions on Parallel and Distributed Systems* 1.3 (July 1990), pp. 330–343.
- [40] Levent Akyil et al. *Intel Guide for Developing Multithreaded Application*. Tech. rep. Intel, 2011.
- [41] Alexey Kukanov and Michael J. Voss. “The Foundations for Scalable Multicore Software in Intel Threading Building Blocks”. In: 2007.
- [42] J. Ceng et al. “MAPS: An integrated framework for MPSoC application parallelization”. In: *2008 45th ACM/IEEE Design Automation Conference*. June 2008, pp. 754–759.
- [43] R. Leupers and J. Castrillon. “MPSoC programming using the MAPS compiler”. In: *2010 15th Asia and South Pacific Design Automation Conference (ASP-DAC)*. Jan. 2010, pp. 897–902.
- [44] Weihua Sheng et al. “A Compiler Infrastructure for Embedded Heterogeneous MPSoCs”. In: *International Workshop on Programming Models and Applications for Multicores and Manycores*. PMAM. Shenzhen, Guangdong, China: ACM, 2013, pp. 1–10.
- [45] Rogier Baert et al. “Exploring Parallelizations of Applications for MPSoC Platforms Using MPA”. In: *Proceedings of the Conference on Design, Automation and Test in Europe*. DATE. Nice, France: European Design and Automation Association, 2009, pp. 1148–1153.
- [46] P. Sun, S. Chandrasekaran, and B. Chapman. “OpenMP-MCA: Leveraging Multiprocessor Embedded Systems Using Industry Standards”. In: *IEEE International Parallel and Distributed Processing Symposium Workshop*. May 2015, pp. 679–688.
- [47] Eduard Ayguadé et al. “Extending OpenMP to Survive the Heterogeneous Multi-Core Era”. In: *International Journal of Parallel Programming* 38.5 (Oct. 2010), pp. 440–459.
- [48] J. E. Stone, D. Gohara, and G. Shi. “OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems”. In: *Computing in Science Engineering* 12.3 (May 2010), pp. 66–73.
- [49] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP*. Mit University Press Group Ltd, 2007.
- [50] Antoniu Pop and Albert Cohen. “OpenStream: Expressiveness and Data-flow Compilation of OpenMP Streaming Programs”. In: *ACM Trans. Archit. Code Optim.* 9.4 (Jan. 2013), 53:1–53:25.
- [51] Antoniu Pop and Albert Cohen. “A Stream-computing Extension to OpenMP”. In: *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*. HiPEAC. Heraklion, Greece: ACM, 2011, pp. 5–14.
- [52] Mehdi Amini et al. “Par4All: From Convex Array Regions to Heterogeneous Computing”. In: (May 2012).
- [53] Uday Bondhugula, Aravind Acharya, and Albert Cohen. “The Pluto+ Algorithm: A Practical Approach for Parallelization and Locality Optimization of Affine Loop Nests”. In: *ACM Trans. Program. Lang. Syst.* 38.3 (Apr. 2016), 12:1–12:32.
- [54] Uday Bondhugula et al. “Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model”. In: *International Conference on Compiler Construction (ETAPS CC)*. Apr. 2008.

-
- [55] Uday Bondhugula et al. “A Practical Automatic Polyhedral Program Optimization System”. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. June 2008.
- [56] Uday Bondhugula. “Automatic Distributed-Memory Parallelization and Code Generation using the Polyhedral Framework”. In: 2012.
- [57] Martin Frieb et al. “A Parallelization Approach for Hard Real-Time Systems and Its Application on Two Industrial Programs”. In: *International Journal of Parallel Programming* 44.6 (Dec. 2016), pp. 1296–1336.
- [58] Theo Ungerer et al. “Parallelizing Industrial Hard Real-Time Applications for the parMERASA Multicore”. In: *ACM Trans. Embed. Comput. Syst.* 15.3 (May 2016), 53:1–53:27.
- [59] David R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley Professional, 1997.
- [60] Ronan Keryell et al. “Pips: a Workbench for Building Interprocedural Parallelizers, Compilers and Optimizers Technical paper”. In: (Apr. 1994).
- [61] N. Lossing, C. Ancourt, and F. Irigoien. “Automatic Code Generation of Distributed Parallel Tasks”. In: *IEEE Intl Conference on Computational Science and Engineering (CSE) and IEEE Intl Conference on Embedded and Ubiquitous Computing (EUC) and 15th Intl Symposium on Distributed Computing and Applications for Business Engineering (DCABES)*. Aug. 2016, pp. 234–241.
- [62] Dounia Khaldi, Pierre Jouvelot, and Corinne Ancourt. “Parallelizing with BDSC, a resource-constrained scheduling algorithm for shared and distributed memory systems”. In: *Parallel Computing* 41 (2015), pp. 66–89.
- [63] Tao Yang and Apostolos Gerasoulis. “PYRROS: Static Task Scheduling and Code Generation for Message Passing Multiprocessors”. In: *Proceedings of the 6th International Conference on Supercomputing. ICS '92*. Washington, D. C., USA: ACM, 1992, pp. 428–437.
- [64] Raphael Poss. “SL: a ‘quick and dirty’ but working intermediate language for SVP systems”. In: *CoRR* (2012).
- [65] R. Poss et al. “Apple-CORE: Microgrids of SVP Cores – Flexible, General-Purpose, Fine-Grained Hardware Concurrency Management”. In: (Sept. 2012), pp. 501–508.
- [66] R. Poss et al. “Apple-CORE: Harnessing General-purpose Many-cores with Hardware Concurrency Management”. In: *Microprocess. Microsyst.* (Nov. 2013), pp. 1090–1101.
- [67] M. W. Hall et al. “Maximizing multiprocessor performance with the SUIF compiler”. In: *Computer* 29.12 (Dec. 1996), pp. 84–89.
- [68] Sungdo Moon, Byoungro So, and M. W. Hall. “Evaluating automatic parallelization in SUIF”. In: *IEEE Transactions on Parallel and Distributed Systems* 11.1 (Jan. 2000), pp. 36–49.
- [69] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. “StreamIt: A Language for Streaming Applications”. In: *Proceedings of the 11th International Conference on Compiler Construction*. 2002, pp. 179–196.
- [70] Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. “The Design of a Task Parallel Library”. In: *SIGPLAN Not.* 44.10 (Oct. 2009), pp. 227–242.
- [71] K. Stavrou et al. “TFlux: A Portable Platform for Data-Driven Multithreading on Commodity Multicore Systems”. In: *37th International Conference on Parallel Processing*. Sept. 2008, pp. 25–34.
- [72] Pedro Trancoso, Kyriakos Stavrou, and Paraskevas Evripidou. “DDMCP : The Data-Driven Multithreading C PreProcessor”. In: 2007.

-
- [73] G. Contreras and M. Martonosi. “Characterizing and improving the performance of Intel Threading Building Blocks”. In: *IEEE International Symposium on Workload Characterization*. Sept. 2008, pp. 57–66.
- [74] Donald G. Bailey. “The Advantages and Limitations of High Level Synthesis for FPGA Based Image Processing”. In: *Proceedings of the 9th International Conference on Distributed Smart Cameras*. ICDSC ’15. Seville, Spain: ACM, 2015, pp. 134–139.
- [75] E. Waingold et al. “Baring it all to software: Raw machines”. In: *Computer* 30.9 (Sept. 1997), pp. 86–93.
- [76] J. Leskela, J. Nikula, and M. Salmela. “OpenCL embedded profile prototype in mobile device”. In: *IEEE Workshop on Signal Processing Systems*. Oct. 2009, pp. 279–284.
- [77] J. Rohde, M. Martinez-Peiro, and R. Gadea-Girones. “SOCAO: Source-to-Source OpenCL Compiler for Intel-Altera FPGAs”. In: *FSP 2017; Fourth International Workshop on FPGAs for Software Programmers*. Sept. 2017, pp. 1–7.
- [78] Markus Levy Urs Gleim. *MTAPI: Parallel Programming for Embedded Multicore Systems*. Tech. rep.
- [79] Costa JJ et al. “Running OpenMP applications efficiently on an everything-shared SDSM”. In: *18th International Parallel and Distributed Processing Symposium*. Apr. 2004, pp. 35–.
- [80] A. Basumallik, S. J. Min, and R. Eigenmann. “Programming Distributed Memory Systems Using OpenMP”. In: *IEEE International Parallel and Distributed Processing Symposium*. Mar. 2007, pp. 1–8.
- [81] L. Huang et al. “Parallelizing ultrasound image processing using OpenMP on multicore embedded systems”. In: *IEEE Global High Tech Congress on Electronics*. Nov. 2012, pp. 131–138.
- [82] R. Prokesch. “Evaluation of parallelization of an image processing algorithm for an embedded multicore platform using manual parallelization and the OpenMP parallel framework”. In: *39th Annual Conference of the IEEE Industrial Electronics Society (IECON)*. Nov. 2013, pp. 2256–2260.
- [83] J. M. Perez, R. M. Badia, and J. Labarta. “A dependency-aware task-based programming environment for multi-core architectures”. In: *IEEE International Conference on Cluster Computing*. Sept. 2008, pp. 142–151.
- [84] Mohammad Zalfany Urfianto et al. “A Multiprocessor SoC Architecture with Efficient Communication Infrastructure and Advanced Compiler Support for Easy Application Development”. In: *IEICE Transactions* 91-A (2008), pp. 1185–1196.
- [85] Frederic Jacob. “Automatische Aufteilung und Parallelisierung von Embedded Software mit μ Streams”. Bachelor’s thesis. TU Darmstadt, FG Rechnersysteme, 2017.
- [86] Sebastian Herber. “Entwurf und Implementierung eines Kohärenz Mechanismus für globale Variablen in SpartanMC Many-Core Systemen”. Master’s thesis. TU Darmstadt, FG Rechnersysteme, 2017.
- [87] Markus Noll. “Implementierung einer Dispatcher- und Konzentratorschaltung für den SpartanMC”. Bachelor’s thesis. TU Darmstadt, FG Rechnersysteme, 2015.
- [88] Laurenz Kamp. “Erweiterung des SpartanMC um einen Performance-Counter”. Bachelor’s thesis. TU Darmstadt, FG Rechnersysteme, 2016.
- [89] Tobias Schladt. “Portierung der Many-Core-fähigen SpartanMC-Firewall auf Xilinx MicroBlaze”. Master’s thesis. TU Darmstadt, FG Rechnersysteme, 2018.
- [90] Lukas Schild. “Integration des Xilinx MicroBlaze in μ Streams”. Bachelor’s thesis. TU Darmstadt, FG Rechnersysteme, 2018.
- [91] Christian Bienia. “Benchmarking Modern Multiprocessors”. PhD thesis. Princeton University, Jan. 2011.

-
- [92] Yuko Hara et al. “Proposal and Quantitative Analysis of the CHStone Benchmark Program Suite for Practical C-based High-level Synthesis”. In: *JIP* 17 (Jan. 2009), pp. 242–254.
- [93] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. “MediaBench: a tool for evaluating and synthesizing multimedia and communications systems”. In: *Proceedings of 30th Annual International Symposium on Microarchitecture*. Dec. 1997, pp. 330–335.
- [94] Reinhold P. Weicker. “Dhrystone: A Synthetic Systems Programming Benchmark”. In: *Commun. ACM* 27.10 (Oct. 1984), pp. 1013–1030.
- [95] Kris Heid, Ramon Wirsch, and Christian Hochberger. “Automated Inference of SoC Configuration through Firmware Source Code Analysis”. In: *FSP 2016; Third International Workshop on FPGAs for Software Programmers*. Aug. 2016, pp. 1–9.
- [96] D. Bafumba-Lokilo, Y. Savaria, and J. David. “Generic crossbar network on chip for FPGA MP-SoCs”. In: *2008 Joint 6th International IEEE Northeast Workshop on Circuits and Systems and TAISA Conference*. June 2008, pp. 269–272.
- [97] Dominik Lorych. “Usage of Replication Pipelines with Soft-Cores and μ Streams”. Project seminar. TU Darmstadt, FG Rechnersysteme, 2018.
- [98] C. Dave et al. “Cetus: A Source-to-Source Compiler Infrastructure for Multicores”. In: *Computer* 42.12 (Dec. 2009), pp. 36–42.
- [99] Kris Heid and Christian Hochberger. “AutoStreams: Fully Automatic parallelization of Legacy Embedded Applications with Soft-Core MPSoCs”. In: *International Conference on Reconfigurable Computing and FPGAs (ReConFig)*. Dec. 2018, pp. 1–8.
- [100] Kris Heid, Jakob Wenzel, and Christian Hochberger. “Fast DSE for Automated Parallelization of Embedded Legacy Applications”. In: *Applied Reconfigurable Computing. Architectures, Tools, and Applications*. Springer International Publishing, 2018, pp. 471–484.
- [101] Kris Heid, Jan Weber, and Christian Hochberger. “ μ Streams: A Tool for Automated Streaming Pipeline Generation on Soft-core Processors”. In: *International Conference on FPGA Reconfiguration for General-Purpose Computing (FPGA4GPC)*. May 2016, pp. 25–30.
- [102] Geoffrey Phipps. “Comparing Observed Bug and Productivity Rates for Java and C++”. In: *Softw. Pract. Exper.* 29.4 (Apr. 1999), pp. 345–358.
- [103] Rajendra Patel and Arvind Rajawat. “A Survey of Embedded Software Profiling Methodologies”. In: *International Journal of Embedded Systems and Applications (IJESA)* (2011).
- [104] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. “Gprof: A Call Graph Execution Profiler”. In: *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*. SIGPLAN ’82. Boston, Massachusetts, USA: ACM, 1982, pp. 120–126.
- [105] Nathan Froyd, John Mellor-Crummey, and Robert J. Fowler. “Low-overhead call path profiling of unmodified, optimized code”. In: Jan. 2005, pp. 81–90.
- [106] Kris Heid, Jakob Wenzel, and Christian Hochberger. “Improved Parallelization of Legacy Embedded Software on Soft-Core MPSoCs through Automatic Loop Transformations”. In: *Fifth International Workshop on FPGAs for Software Programmers (FSP)*. Aug. 2018, pp. 1–8.
- [107] John Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc., 2002.
- [108] J. Liu, J. Wickerson, and G. A. Constantinides. “Loop Splitting for Efficient Pipelining in High-Level Synthesis”. In: *FCCM 2016*. 2016, pp. 72–79.
- [109] Kris Heid and Christian Hochberger. “Generating Optimized FPGA-Based MPSoCs to Parallelize Legacy Embedded Software with Customizable Throughput”. In: *Workshop Parallel -Algorithmen, -Rechnerstrukturen und -Systemsoftware (PARS)*. 2019.

-
- [110] M. R. Guthaus et al. “MiBench: A free, commercially representative embedded benchmark suite”. In: *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*. Dec. 2001, pp. 3–14.
 - [111] Jeff Scott et al. *Designing the Low-Power M*CORE Architecture*. Tech. rep. 1998.
 - [112] Khalid Sayood. *Introduction to Data Compression*. Morgan Kaufmann Publishers Inc., 2000.
 - [113] Honeywell Technology Center. *Versatility Stressmark*. Benchmark Specification Document Version 0.8 CDRL A001. Honeywell Technology Center, 1997.
 - [114] Steven Leduc. “Many-Core Firewall Application for SpartanMC”. Master’s thesis. TU Darmstadt, FG Rechnersysteme, 2016.
 - [115] Daniel Hartmeier. “Design and Performance of the OpenBSD Stateful Packet Filter (Pf)”. In: *USENIX Annual Technical Conference*. USENIX Association, 2002, pp. 171–180.
 - [116] David Murray and Terry Koziniec. “The State of Enterprise Network Traffic in 2012”. In: *18th Asia-Pacific Conference on Communications (APCC)*. 2012.
 - [117] David Murray et al. “An Analysis of Changing Enterprise Network Traffic Characteristics”. In: *23rd Asia-Pacific Conference on Communications (APCC)*. 2017.
 - [118] Shunhao Lin et al. “A Design of the Ethernet Firewall Based on FPGA”. In: *10th International Congress on Image and Signal Processing, BioMedical Engineering and Informatics*. 2017.
 - [119] Gajanan S. Jedhe, Arun Ramamoorthy, and Kuruvilla Varghese. “A Scalable High Throughput Firewall in FPGA”. In: *16th International Symposium on Field-Programmable Custom Computing Machines*. 2008.
 - [120] Sven Hager, Björn Scheuermann, and Frank Winkler. “MPFC: Massively Parallel Firewall Circuits”. In: *39th Annual IEEE Conference on Local Computer Networks*. 2014.
 - [121] Zubair Nabi. “A 35 Dollar Firewall for the Developing World”. In: *Cornell University Computing Research Repository* (2014).

Supervised Students' Theses

- [85] Frederic Jacob. "Automatische Aufteilung und Parallelisierung von Embedded Software mit μ Streams". Bachelor's thesis. TU Darmstadt, FG Rechnersysteme, 2017.
- [86] Sebastian Herber. "Entwurf und Implementierung eines Kohärenz Mechanismus für globale Variablen in SpartanMC Many-Core Systemen". Master's thesis. TU Darmstadt, FG Rechnersysteme, 2017.
- [87] Markus Noll. "Implementierung einer Dispatcher- und Konzentratorschaltung für den SpartanMC". Bachelor's thesis. TU Darmstadt, FG Rechnersysteme, 2015.
- [88] Laurenz Kamp. "Erweiterung des SpartanMC um einen Performance-Counter". Bachelor's thesis. TU Darmstadt, FG Rechnersysteme, 2016.
- [89] Tobias Schladt. "Portierung der Many-Core-fähigen SpartanMC-Firewall auf Xilinx MicroBlaze". Master's thesis. TU Darmstadt, FG Rechnersysteme, 2018.
- [90] Lukas Schild. "Integration des Xilinx MicroBlaze in μ Streams". Bachelor's thesis. TU Darmstadt, FG Rechnersysteme, 2018.
- [97] Dominik Lorych. "Usage of Replication Pipelines with Soft-Cores and μ Streams". Project seminar. TU Darmstadt, FG Rechnersysteme, 2018.
- [114] Steven Leduc. "Many-Core Firewall Application for SpartanMC". Master's thesis. TU Darmstadt, FG Rechnersysteme, 2016.

Own Publications

- [95] Kris Heid, Ramon Wirsch, and Christian Hochberger. “Automated Inference of SoC Configuration through Firmware Source Code Analysis”. In: *FSP 2016; Third International Workshop on FPGAs for Software Programmers*. Aug. 2016, pp. 1–9.
- [99] Kris Heid and Christian Hochberger. “AutoStreams: Fully Automatic parallelization of Legacy Embedded Applications with Soft-Core MPSoCs”. In: *International Conference on Reconfigurable Computing and FPGAs (ReConFig)*. Dec. 2018, pp. 1–8.
- [100] Kris Heid, Jakob Wenzel, and Christian Hochberger. “Fast DSE for Automated Parallelization of Embedded Legacy Applications”. In: *Applied Reconfigurable Computing. Architectures, Tools, and Applications*. Springer International Publishing, 2018, pp. 471–484.
- [101] Kris Heid, Jan Weber, and Christian Hochberger. “ μ Streams: A Tool for Automated Streaming Pipeline Generation on Soft-core Processors”. In: *International Conference on FPGA Reconfiguration for General-Purpose Computing (FPGA4GPC)*. May 2016, pp. 25–30.
- [106] Kris Heid, Jakob Wenzel, and Christian Hochberger. “Improved Parallelization of Legacy Embedded Software on Soft-Core MPSoCs through Automatic Loop Transformations”. In: *Fifth International Workshop on FPGAs for Software Programmers (FSP)*. Aug. 2018, pp. 1–8.
- [109] Kris Heid and Christian Hochberger. “Generating Optimized FPGA-Based MPSoCs to Parallelize Legacy Embedded Software with Customizable Throughput”. In: *Workshop Parallel -Algorithmen, -Rechnerstrukturen und -Systemsoftware (PARS)*. 2019.