An Intermediate Representation For Transforming And Optimizing The Microarchitecture Of Application Accelerators

by

Amirali Sharifian

M.Sc., Simon Fraser University, 2016
B.Sc., Isfahan University of Technology, 2014

Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy in the School of Computing Science Faculty of Applied Sciences

© Amirali Sharifian 2020 SIMON FRASER UNIVERSITY Summer 2020

Copyright in this work rests with the author. Please ensure that any reproduction or re-use is done in accordance with the relevant national copyright legislation.
Approval

Name: Amirali Sharifian
Degree: Doctor of Philosophy (Computing Science)
Title: An Intermediate Representation For Transforming And Optimizing The Microarchitecture Of Application Accelerators

Examining Committee:

Chair: Keval Vora
Assistant Professor

Arrvindh Shriraman
Senior Supervisor
Associate Professor

Nick Sumner
Supervisor
Associate Professor

Tony Nowatzki
Supervisor
Assistant Professor
Computer Science
University of California, Los Angeles

William Chow
Supervisor
Principle Hardware Engineer
Huawei Technologies

Zhenman Fang
Internal Examiner
Assistant Professor

Tor Aamodt
External Examiner
Professor
Electrical and Computer Engineering
University of British Columbia

Date Defended: July 27, 2020
Abstract

In recent years, the computing landscape has seen a shift towards specialized accelerators since the scaling of computational capacity is no longer guaranteed for every technology generation. Reconfigurable architectures like FPGAs are promising for accelerator implementation. FPGAs allow implementation of arbitrary logic functions for different classes of applications with better performance per watt over CPUs and GPUs without re-spinning the circuits like fixed-function ASICs. Unfortunately, the software programmer community has stayed away from this technology, primarily because of the abstraction gap exists between software languages and hardware design.

Hardware description languages (HDLs) are very low-level, and a hardware designer should think about the design in terms of low-level building blocks such as gates and registers. The alternative to HDLs is High-level synthesis (HLS) tools. HLS frameworks synthesize hardware from a high-level description of an algorithm in the form of untimed mathematical expressions and nested, pipeline and parallel loops in software languages. The primary limitation of HLS is that the functionality and microarchitecture are conflated together in a single language. As a result, making changes to the accelerator design requires code restructuring and microarchitecture optimizations tied by program correctness.

In this thesis we propose two new abstractions to decouple functionality from microarchitecture. The first abstraction is a hierarchical intermediate representation for describing parameterized accelerator microarchitecture, targeting reconfigurable architects. In this abstraction, we represent the accelerator as a concurrent structural graph in which components roughly correspond to microarchitecture level hardware blocks. We describe the methods we used to lower the entire application graph into a parameterized intermediate hardware abstraction, \( \mu IR \). We describe the implementation of this intermediate abstraction and an associated pass framework, \( \mu opt \). We then discuss some of the compiler optimizations that \( \mu IR \) enables, including timing, spatial, and higher-order. The final system is a compiler stack that can take a high-level program as input and translate it into optimized, synthesizable hardware design. The second abstraction is a sequence of instructions that convey the producer-consumer locality between dependent instructions. We show that this new abstraction adapts to different acceleration behaviors by varying the length and the types of fused instructions.

**Keywords:** Accelerator design, High-Level Synthesis, Configurable architecture, FPGA
Dedication

To Maman, Baba and Setareh.
Acknowledgements

First of all, I would like to thank my supervisor, Dr. Arrvindh Shriraman, for his extensive support, patience and guidance during my studies and for the opportunity to work in his exceptional research group. My sincere thanks go to my supervisory committee, Dr. William Nick Sumner and Dr. Tony Nowatzki, for providing valuable directions to improve my research outcome. I’m also thankful to Taneem Ahmed and William Chow for their extensive support during my PhD studies.

I would especially like to thank Steve Margerm for all his guidance, patience, and discipline that he taught me while working together. I want to thank Reza Hojabr, Snehashish Kumar and Apala Guha for their significant collaborations. I would like to thank the fellows of the SFU Computer Architecture lab, whom I had the pleasure of working with during my research tenure. I am also thankful to my friends Alireza, Arash, Sam, Pouria, Parmida, Ali, Shreesdish, Yunduz and all those who were my best friends and companions on this journey.

Last but not least, I am thankful to my wonderful family for their unlimited support during my studies. I owe a great debt to my sister, mother, and father without whom none of this would be possible.
# Table of Contents

**Approval**

**Abstract**

**Dedication**

**Acknowledgements**

**Table of Contents**

**List of Tables**

**List of Figures**

1 **Introduction**
   1.1 System-Design Challenges .............................................. 1
   1.2 High-Level Synthesis Frameworks ....................................... 2
   1.3 Domain-Specific Languages Frameworks .................................. 3
   1.4 Why a New Microarchitecture-Based Intermediate Representation is Needed? 5
       1.4.1 Objectives .......................................................... 5
   1.5 Approach ............................................................... 8
       1.5.1 Hardware abstraction for software defined accelerators ........ 8
       1.5.2 Hardware abstraction for fused instructions .................... 12
       1.5.3 Publications ...................................................... 13
   1.6 Dissertation Organization ............................................... 15

2 **Background And Motivation**
   2.1 Hardware Accelerators .................................................. 16
   2.2 Hardware Accelerator Design Abstractions ............................ 18
   2.3 Hardware Construction Languages (HCL) ................................ 19
   2.4 Hardware Compiler Framework (HCF) .................................. 21
   2.5 High-Level Synthesis (HLS) ............................................ 22
       2.5.1 Absence of Explicit Parallelism ................................. 23
5.2.2 Dynamic Parallelism vs Static Parallelism ........................................ 74
5.2.3 Static vs Dynamic Scheduling ...................................................... 75
5.2.4 TaskGraph vs Prior Work .......................................................... 76

5.3 TaskGraph: High-Level-Synthesizing Dynamic Parallel Accelerators ........ 79
5.3.1 Stage 1: Task Parallel Architecture ............................................. 79
5.3.2 Execution Example ................................................................. 80
5.3.3 Stage 2: Generating Task Exe Unit (TXU) ....................................... 82
5.3.4 Stage 3: Parameterized Accelerator ............................................. 83
5.3.5 Task Memory interface and Memory Model ..................................... 83
5.3.6 Compiler Front-end: Tasks from IR ............................................. 84

5.4 TaskGraph-generated Accelerators .................................................. 85
5.4.1 Nested Parallel and Conditional Loops .......................................... 85
5.4.2 Pipeline Parallelism ................................................................. 86
5.4.3 Recursive Parallelism ............................................................... 87

5.5 Evaluation ..................................................................................... 89
5.5.1 Parallel Task Overhead ............................................................ 89
5.5.2 Resource Utilization ................................................................. 90
5.5.3 Scalability and Performance ..................................................... 91
5.5.4 Energy Consumption ............................................................... 93
5.5.5 Intel HLS vs TaskGraph .......................................................... 94

6 Chain Hardware Abstraction ............................................................... 96
6.1 Introduction .................................................................................... 96
6.1.1 Optimizing processor ............................................................ 97
6.1.2 Specialized Hardware ............................................................ 98
6.1.3 Contribution .......................................................................... 99

6.2 Background and Motivation ........................................................... 100
6.2.1 Instruction set customization .................................................... 100
6.2.2 Efficient General-purpose Processors ......................................... 102
6.2.3 Dataflow accelerators ............................................................. 103

6.3 Dataflow Graph Execution .............................................................. 104
6.3.1 Fabric Utilization and Static Power ............................................ 106
6.3.2 Reconfiguration costs ............................................................. 109
6.3.3 Data Movement Energy .......................................................... 109

6.4 Chains ......................................................................................... 110
6.4.1 Chain benefits ....................................................................... 111
6.4.2 Chain Distribution .................................................................. 116

6.5 Chainsaw architecture ................................................................. 118
6.5.1 Instruction Execution ............................................................. 119
6.5.2 Chain Scheduling and Wakeup ........................................ 120
6.5.3 Instruction Issue ...................................................... 121
6.5.4 Dynamic execution ................................................... 122
6.6 Framework and Evaluation .............................................. 123
6.6.1 Profiling ............................................................... 123
6.6.2 Chain Extraction ....................................................... 124
6.6.3 Code Generation ...................................................... 124
6.6.4 Simulation .............................................................. 124
6.6.5 Synthesis and Area Overhead ...................................... 127
6.6.6 Evaluation ............................................................. 129
6.6.7 Performance Comparison ............................................ 129
6.6.8 Energy Comparison ................................................... 130
6.6.9 Chainsaw vs. SIMD ................................................... 134

7 Framework Release .......................................................... 135
7.1 \( \mu IR \) Simulator ....................................................... 135
7.1.1 \( \mu IR \)-sim Design .................................................... 135
7.2 \( \mu IR \) Library .......................................................... 137
7.2.1 \( \mu IR \)-lib components ............................................... 138
7.2.2 \( \mu IR \) AWS ............................................................ 139
7.2.3 \( \mu IR \) RISC-V ......................................................... 139
7.3 \( \mu IR \) Generator ........................................................ 139
7.4 Chainsaw Sim ............................................................. 140

8 Summary and Future Work ................................................ 141
8.1 Summary ................................................................. 141
8.2 Future Directions and Further Challenges ........................... 143
8.2.1 How it is possible to debug \( \mu IR \)? ............................... 143
8.2.2 How else can \( \mu IR \) support more efficient computation kernels? 144
8.2.3 How else can \( \mu IR \) support better scheduling? ................ 144
8.3 Reflection ................................................................. 146
8.3.1 FPGAs Have the Wrong Abstraction ............................ 146
8.3.2 What is an FPGA? ................................................... 146
8.3.3 A Sudden Analogy ................................................... 147
8.3.4 RTL Is Not an ISA .................................................. 148
8.3.5 The Right Abstraction? ............................................. 149

Bibliography ................................................................. 151
List of Tables

Table 1.1 Table of contributions ......................................................... 8
Table 2.1 Comparing the hardware oriented Intermediate-Representations .... 32
Table 4.1 Synthesizing Baseline $\mu IR$ on Arria10 FPGA — $F$ - Floating point benchmarks [$T$]: Tensor operations. Synopsys DC Compiler. ASIC umc 28nm. ... 60
Table 4.2 Summary of $\mu opt$ passes .................................................... 62
Table 4.3 Conciseness of $\mu IR$ vs FIRRTL (All $\mu opt$) .......................... 66
Table 5.1 Comparing the features of HLS tools ....................................... 77
Table 5.2 Benchmark Properties .......................................................... 86
Table 5.3 FPGA Utilization ................................................................. 91
Table 5.4 FPGA Resources (Board: Cyclone V) ..................................... 94
Table 5.5 Intel HLS vs TaskGraph (Board: Cyclone V) ....................... 95
Table 6.1 Comparing the hardware oriented Intermediate-Representations ... 105
Table 6.2 Characteristics of CGRA execution (4x4 and 8x8). Performance (cycles), Idle Cycles (cumulative function unit idle cycles). ILP: dataflow instruction parallelism ........................................................ 108
Table 6.3 Scheduler Table ................................................................. 121
Table 6.4 Instruction fields ............................................................... 122
Table 6.5 Dynamic Execution ............................................................. 123
Table 6.6 Coverage top five traces ..................................................... 123
Table 6.7 System parameters ............................................................ 128
List of Figures

Figure 1.1 This figure provides a pictorial comparison of the organization of C-based HLS and hardware DSL frameworks. The HLS framework (part a) builds upon software compilers like LLVM IR. The front-end captures system behavior with a computation model that uses a standard language such as C, C++, SystemC or OpenCL as an input. The back-end requires the designers to use ad-hoc pragmas and types in the program to ensure that the tools produce good-quality RTL (Section 2.5). High-Level Language-Based Frameworks (part b) build upon modern programming languages such as Scala. The front-end offers highly abstract, usually object-oriented, and high-level features, such as polymorphism and automatic memory management. The back-end adds a set of low-level abstractions for control and memory and makes the algorithm specification tightly entangled with hardware optimizations (Section 2.6).............4

Figure 1.2 Block diagram of the $\mu$IR compiler framework. The framework compiles high-level languages, such as C/C++ and Cilk, and DSL languages such as TensorFlow, from LLVM intermediate representation (IR) to a task-level representation. Using Task Compiler, task IR is optimized, transformed, and lowered to a lower level intermediate representation for hardware accelerator designs called $\mu$IR structural graphs. This lower level IR is compiled by a $\mu$IR compiler and is generated as an efficient hardware accelerator design.........................9

Figure 2.1 A general FPGA architecture ..................................................17
Figure 2.2 Finite Impulse Response (FIR) Digital Filter ..........................20
Figure 2.3 A chisel implementation of FIR with similar Verilog primitives [62] ........20
Figure 2.4 While Chisel provides similar base primitives as synthesizable Verilog, the power of Chisel comes from the ability to create generators, such as in FIR filter that is defined by the list of coefficients and use and re-use them across designs..........................21
Figure 2.5 HCF can create a Chisel-to-ASIC-Verilog compiler or Verilog-to-FPGA-Verilog compiler and share internal transformations. [120] ..............22
Figure 2.6 FIR filter in HLS [215] ..........................................................24
Figure 4.2 Dot product implementation in C .......................... 53
Figure 4.3 Baseline implementation of Dot product in µIR .................. 53
Figure 4.4 Dot product with dual buffer local memory ...................... 53
Figure 4.5 In this moment, the Tile 3 is been loaded from DRAM to the local buffer in Stage 2, while in Stage 1 the value from Tile 2 is accumulating. .... 54
Figure 4.6 Unrolled version of dot product by factor 2 ....................... 55
Figure 4.7 Inner parallelization with factor 4 ............................... 55
Figure 4.8 µIR compiler framework compilation steps and abstraction .... 56
Figure 4.9 µIR vs HLS. Normalized performance. HLS = 1. < 1: µIR is better. > 1 HLS is better. ............................................. 61
Figure 4.10 Illustration of Auto-Pipelining and Op-Fusion pass. ............. 62
Figure 4.11 Execution time improvement due to Op-Fusion — Baseline IR = 1 ... 63
Figure 4.12 Reduction with in execution time by increasing number of parallel execution units. .............................................. 64
Figure 4.13 Implementation of 2MM with Tensor ops .......................... 64
Figure 4.14 Multiplier unit for Tensor2D. C_{2\times2} = A_{2\times2} \times B_{2\times2} .......... 65
Figure 4.15 Performance improvement due to tensor ops — Baseline IR = 1 .... 65
Figure 4.16 Effect of cache banking (1–4 Banks) — Baseline IR = 1 ............ 67
Figure 4.17 Effect of stacking multiple µopt optimizations .................... 68
Figure 4.18 Optimized µIR vs ARM A9 1Ghz. ARM = 1. > 1: µIR is better. < 1 ARM is better. Note: ARM does not support Cilk. .............................. 69
Figure 5.1 Pseudocode for PARSEC’s [18] Dedup based on Cilk-P .......... 71
Figure 5.2 Parallel accelerator generated by TaskGraph for dynamic pipeline parallelization of PARSEC’s Dedup ................................. 72
Figure 5.3 Generating parallel for loop. HLS and static scheduling vs. TaskGraph and dynamic scheduling ................................. 74
Figure 5.4 Limitations of static scheduling ........................................ 75
Figure 5.5 Static Schedule: a)no pipeline, b)pipeline .......................... 75
Figure 5.6 Dynamically schedule .................................................. 76
Figure 5.7 Overview the TaskGraph framework. The input to the framework is a concurrent program written in Cilk. TaskGraph is organized in three stages. The input to Stage 1, is program dependence graph from LLVM IR. In Stage 2, the input is the generated task level architecture. At this stage the framework, optimizes the task level architecture, generates TXUs and connect them to memory hierarchy. Finally, at Stage 3, the framework sets the proper parameters and generate RTL design. .............................. 78
Figure 5.8 TaskGraph generated microarchitecture in µIR. .................... 80
Figure 5.9 Execution flow of Nested-loop accelerator generated by TaskGraph ... 81
Figure 5.10 Task-2’s Task Execution Unit (TXU). ........................................... 82
Figure 5.11 Multiple tasks simultaneously outstanding on TXU. ....................... 83
Figure 5.12 Data Box. Interfaces with memory operations in logic box and transfers operations to/from a cache or scratchpad. ....................................................... 84
Figure 5.13 TaskGraph pass for extracting tasks from Tapir ............................... 85
Figure 5.14 Stencil example using μIR and TaskGraph ................................. 87
Figure 5.15 Accelerator for recursive mergesort. ........................................... 88
Figure 5.16 (a) Test Code, (b) Parallel Task Tiling .......................................... 89
Figure 5.17 Performance Scaling with Tiles .................................................... 90
Figure 5.18 ALM Utilization by Sub-block ...................................................... 91
Figure 5.19 Performance Scalability. ............................................................. 92
Figure 5.20 Performance. TaskGraph vs Intel i7. ........................................... 93
Figure 5.21 Performance/Watt: TaskGraph vs Intel i7. .................................. 93

Figure 6.1 Chainsaw overview. Our compiler constructs control-free superblocks [16, 81] to eliminate branches from the offload region. The compiler then fuses sequences of instructions in the dataflow graph to construct chains (C1, C2, C3 etc.) and statically schedules them on the Chainsaw multi-lane architecture at chain granularity. The unaccelerated program regions continue to run on the OOO. Compared to prior work that fused subgraphs [75, 73, 44] chains only fuse only sequences of operations and do not require specialized compound function units. ................................................................. 99
Figure 6.2 Array of FUs [75]. ................................................................. 104
Figure 6.3 DFG example. ................................................................. 105
Figure 6.4 Execution on a spatial fabric. ...................................................... 106
Figure 6.5 Execution on a von neumann architecture. .................................. 106
Figure 6.6 Chains built using Dilworth decomposition followed by cycle removal. Chain schedule on two FUs; the height of a chain is proportional to the number of ops in it. Fusing operations to chains reduce inter-chain register writes (21 to 4). Chains can exploit ILP with only two FUs (latency: 16 cycles for DFG). ................................................. 110
Figure 6.7 Strategy MaxILP breaks chains at live-ins and live-outs leading to shorter chains and a higher chain count. More inter-chain data movement but critical path reduces to 13 ops, identical to the unchained DFG. Inter-chain register writes increased to 9. .......................................................... 112
Figure 6.8 Strategy MaxSize merges chains greedily to reduce data movement. Register writes decreased to 4. Critical path length increased to 14 ops. .... 112
Figure 6.9  The chained graph for a frequently executed region in gzip. The DFG has varying levels of ILP which is unsuitable for spatial fabrics, and has several long chains. 114

Figure 6.10 Relative proportion of inter-chain and intra-chain dependencies. Fusing ops into chains localizes communication. 114

Figure 6.11 Computation coverage by chains. Histogram showing the percentage of ops subsumed by chains of different lengths. 50–80% of operations are subsumed by chains of length 3 or more. 115

Figure 6.12 The amount of ILP mined from the dataflow graph by the MaxILP and the MaxSize algorithms. MaxILP has average ILP that is equal to the unchained dataflow graph’s ILP. MaxSize shows notable ILP loss in some applications. 116

Figure 6.13 # of chains generated by the MaxILP algorithm and the MaxSize algorithms. MaxSize typically creates fewer chains enabling. 117

Figure 6.14 Average chain lengths generated by MaxSize algorithm. 117

Figure 6.15 Block diagram of the Chainsaw accelerator. It is composed of lanes. Each lane has an INT and FPU with forwarding registers, two input/output registers (INs and OUTs), a live-in register bank, and an instruction buffer. 119

Figure 6.16 Example DFG execution on Chainsaw. 120

Figure 6.17 Chain’s instructions example. 121

Figure 6.18 Single lane’s pipeline stages. 122

Figure 6.19 Instruction fields: op=opcode, IN0/1: Operand consumes IN0 or IN1 live in value, WR: does instruction produce live-out, FWD: Forward value to subsequent instruction, L/R Op: operand order (is the IN the left or right operand of the instruction), the other operand is the forwarded value. OUT0/1=write output to register OUT0 or OUT1. 122

Figure 6.20 Chainsaw Simulator overall modules 125

Figure 6.21 Performance of various architectures normalized to the IDEAL performance. We don’t plot CGRA8 on this plot since in all applications the CGRA8 attains the performance of IDEAL. Higher is better. 130

Figure 6.22 Dynamic Energy. Data normalized to function unit energy i.e., > 1 indicates the overhead of the different architectures. The number on top of the bars for each workload indicates the OOO (includes the decode and backend costs; excludes TLBs and Caches). CGRA8 and CHAINSAW8 include all the components; CGRA overhead dominated by network energy. 131
Figure 6.23 Communication Energy Breakdown (*Chainsaw*8). Localized computation in CHAINSAW8 reduces communication costs significantly compared to dataflow architectures. The dominant energy component in both CHAINSAW8 and CGRA8 is the network transfers required between producer-consumer operations.

Figure 6.24 Static Power. CGRA8 and CHAINSAW8 normalized to CGRA8. IDLE: the static power expended while waiting for scheduled operations to be ready to run. FREE indicates the static power due to over-provisioning resources compared to the available ILP i.e., the PE or Lane does not have any instruction scheduled to execute.

Figure 7.1 A *µIR* python driver example for a simple test case.

Figure 8.1 Limitations of static scheduling

Figure 8.2 Example of nonspeculative versus speculative execution. In nonspeculative execution, the compiler needs to serialize the loop iterations and cannot pipeline the loop because the loop termination condition is dependent on data calculated on each iteration. In contrast, in speculative execution, the loop can be pipelined independently from existing data dependency.
Chapter 1

Introduction

The slowing of technology scaling, the availability of large amounts of data generated from social applications, sensor applications, business automation systems, interactive multi-media systems, and breakthroughs in algorithm design have inspired accelerator architecture research. In a general-purpose microprocessor, the overhead of instruction processing is much higher than the actual operations performed by each instruction. This overhead includes the necessary steps to fetch and decode the instructions, provide required operands for the instructions, and perform the necessary bookkeeping to ensure correctness when multiple instructions are executed in the microprocessor. Conversely, application-specific hardware is faster and lower in power consumption than general-purpose processors because it eliminates most of the overhead of a general-purpose processor [43, 82]. Although fixed-function accelerators are more energy-efficient than software running on general-purpose processors (GPPs), they are not suitable for applications that change frequently. As an alternative to fixed-function accelerators, reconfigurable architectures like field-programmable gate arrays (FPGAs) and coarse-grain reconfigurable architectures (CGRAs) have received renewed interest from academic researchers and industry practitioners alike, primarily due to their potential performance and energy efficiency benefits compared to GPPs. For instance, Amazon EC2 [6] and Huawei have made FPGAs available to the public through the cloud. Microsoft and Baidu [163, 144] are exploiting FPGAs for accelerating data center services, and Intel has announced products like in-package Xeon-FPGA systems [80] and FPGA-accelerated storage systems [92]. Similarly, several recent research prototypes [76, 149, 201, 150, 160, 34] have explored various kinds of CGRAs at different granularity. The growing use of such reconfigurable architectures has made them attractive to programmers now more than ever before.

1.1 System-Design Challenges

A hardware accelerator is a special-purpose hardware device optimized to perform a particular function as part of a general-purpose computational system. To achieve better performance per watt compared to GPPs, accelerators build custom data pipelines, exploit fine- and coarse-grained parallelism and define custom memory hierarchies. FPGA-based hardware accelerators [41, 103,
117, 165] can be used to provide both performance and flexibility by trading part of the performance gain for the reconfigurability of the implementation substrate. However, to design a new efficient hardware accelerator on an FPGA, a designer needs to consider the following challenges: 1) They should account for timing between pipelined signals, 2) Partition data between local scratchpad memories and off-chip memory to improve data reuse and exploit available bandwidth on the chip, and 3) They should account for physically limited computational and memory resources available on the chip.

These challenges in designing FPGA accelerators combined with the demand for new applications and the advance of technology, lead to the productivity gap of designing hardware accelerators and make programmability a key limiting factor in widespread adoption of FPGAs [198]. State-of-the-art in programming FPGAs uses a combination of vendor-specialized IP blocks, hand-tuned hardware modules written in low-level RTL, and logic to connect the accelerator with off-chip components such as DRAM. Hardware description languages (HDLs) like Verilog and VHDL are designed for explicit hardware specifications, placing the burden on the user to solve all of the complexities of implementing their algorithm in hardware essentially. One of the commonly-accepted solutions for closing the productivity gap, as proposed by semiconductor roadmaps, is to raise the level of abstraction in the design process [198].

### 1.2 High-Level Synthesis Frameworks

High-level synthesis (HLS) techniques [216, 218, 4, 93] have been proposed to raise the level of abstraction compared to traditional hardware design techniques like Hardware Definition Languages (HDLs). An HLS tool allows the designer to: 1) capture various types of parallelism in the input application, 2) build the design space of various micro-architectures that exploits the parallelism in terms of untimed, nested loops, and 3) explore the design space to find the micro-architecture with the lowest silicon area and/or power consumption.

Pure C-to-gates HLS techniques rely on capturing parallelism between fine-grained operations of sequential code by constructing the control and dataflow graph (CDFG) of the computation kernels. They use scheduling algorithms [97, 118, 154] to extract parallelism between the operations that are provably independent in the CDFG. However, the absence of explicit parallelism primitives in C-based HLS frameworks results in two significant limitations: first, it leads the compiler to make a conservative decision where the CDFG contains dynamism. Second, the HLS compiler is limited to capturing coarse-grain parallelism. Since the highest-performance hardware exploits both fine-grained and coarse-grained parallelism, for many applications, the Quality of Results (QoR) of these HLS tools is often lower than that of the manual design process using low-level hardware-description languages (HDL).

To address these limitations, HLS tools extend C-based frameworks, with an ad-hoc, often underspecified mix of software and hardware abstractions. This, in turn, results in difficulties in HLS compiler implementations. For instance, while SDAccel [216] can convert nested loops into
hardware state machines, the language’s compiler does not allow pipelining of loops at arbitrary nesting levels [218].

When using high-level synthesis tools, programmers must keep in mind that, despite the software programming abstractions, they must employ hardware optimization techniques when using HLS tools [136]. In C-to-gates HLS tools that rely on hardware pragmas for hardware-specific information, the addition, changing, or removing of various pragmas can become more of a trial and error process. Such limitations restrict the user’s ability to explore more complex design spaces. For most domain experts, combining these factors makes it challenging to write code that produces fully optimized designs.

1.3 Domain-Specific Languages Frameworks

Domain-Specific Languages (DSL) help to specialize software programming model [153, 1] by presenting the user a very high level of abstraction, generally at the cost of reducing the number of possible operations and data structure types. This focus, in turn, allows domain-specific types and operations to be isolated as higher-level operators with semantic information, which the compiler is aware of, enabling optimizations that would otherwise be difficult or impossible.

The key benefit of domain-specific languages for hardware accelerators lies in this high-level of abstraction, which is typically almost entirely removed from any target architecture. This gives the DSL compiler freedom in how it can implement the desired operations. However, this degree of freedom implies vast design spaces, which can be challenging to navigate. One of the approaches that existing hardware DSL compilers use to search in the design space is using a kernel-based method. The DSL compiler performs domain-specific operations, then lowers the resulting abstraction directly to a hardware implementation using pre-existing hand-written kernels in an HDL, or implementations are written in a high-level synthesis language [71]. In these kernel-wise techniques, the compilation path tends to miss key cross-operation optimizations like fusion and tiling, resulting in excessive transfers to and from main memory. While DSLs like Darkroom [87] in the image processing domain have demonstrated a complete compiler-driven approach for generating hardware, this approach has not yet been generalized across domains. DHDL [111] attempts to introduce a representation of hardware using parameterized templates in Scala that captures locality and parallelism information and compiles the representation into FPGAs and CGRAs. Spatial [109] extends DHDL by adding low-level control and memory abstractions, such as Map, Reduce, Stream-In/Out and DRAM and expects the designer to use the new abstraction to implement an accelerator.

In Figure 1.1, we provide a pictorial comparison of High-Level Synthesis stack frameworks and High-Level Language-Based stack frameworks (Hardware DSLs). In both frameworks, the framework starts with a high-level description of the hardware that captures the behavior and structure. The front-end compiler lowers the input description to an intermediate representation (IR), optimizes it, and the back-end compiler generates structural RTL.
Figure 1.1: This figure provides a pictorial comparison of the organization of C-based HLS and hardware DSL frameworks. The HLS framework (part a) builds upon software compilers like LLVM IR. The front-end captures system behavior with a computation model that uses a standard language such as C, C++, SystemC or OpenCL as an input. The back-end requires the designers to use ad-hoc pragmas and types in the program to ensure that the tools produce good-quality RTL (Section 2.5). High-Level Language–Based Frameworks (part b) build upon modern programming languages such as Scala. The front-end offers highly abstract, usually object-oriented, and high-level features, such as polymorphism and automatic memory management. The back-end adds a set of low-level abstractions for control and memory and makes the algorithm specification tightly entangles with hardware optimizations (Section 2.6).
1.4 Why a New Microarchitecture-Based Intermediate Representation is Needed?

Our insight in this dissertation is that the key limitation in prior works is the desire to use a single representation and monolithic framework to transform and optimize the functionality, concurrency, locality, and microarchitecture. In the next section, we provide our objectives concerning why a new intermediator representation that exposes the microarchitecture-level concurrent execution model of hardware is necessary for enabling aggressive optimizations of hardware and design space exploration. The section also outlines the objectives we achieve toward this ambition.

1.4.1 Objectives

Current HLS approaches use IRs present within the software compiler (e.g., LLVM IR). Such compiler IRs tend to be instruction-oriented, following the Von Neumann model, and only specify execution flow. However, accelerator designs are concurrent and dataflow-based. They require the use of spatial abstraction, which is structural and implicitly concurrent and accounts for a finite amount of hardware resources. This structural specification is a graph-based representation that specifies microarchitecture blocks at a specific abstraction and their connections. We describe our goals with defining a new microarchitectural abstraction for hardware accelerators with five objectives:

**Objective 1 — Decoupling hardware from software intermediate representation:** Hardware and software development are both fundamentally iterative. Both require successive transformations on a program representation with a model of how they will affect some desired metrics in the generated output. It may be tempting to view all transformations within the context of a software compiler’s IR designed for program optimization, but our thesis is that these transformations require fundamentally different frameworks with different intermediate representations and data structures. For example, a compiler’s function call inlining transformation may expose call-site information exploitable by a constant propagation transformation, which may further enable an opportunity for dead code elimination. For hardware, one transformation may be to concurrency-tile a loop across multiple hardware units, which exposes a memory bandwidth problem that can be resolved with memory localization to separate memory accesses to logical regions, followed by memory banking to further improve throughput at the cost of hardware. Take loop unrolling as another example, which has the effect of breaking loop induction-variable dependencies in software. A hardware-generation framework could interpret the output of the unrolled loop as simply parallel hardware instances of instructions. Essentially, what the framework would be doing here is assuming an execution model based on some canonical transformation from software compiler IR (e.g., LLVM IR) to RTL-level IR.
We make two important observations at this point: 1) There are abstractions unique to hardware generation that are challenging to represent within the compiler IR, and 2) For generating optimal accelerators, it is useful to have both software IR and microarchitecture IR.

Objective 2 — A framework to optimize at the microarchitectural level: When decoupling hardware from software intermediate representation we can draw a line between hardware and software optimizations and provide a compiler framework that only focuses on microarchitectural optimizations while the overall framework can benefit from aggressive hardware and software optimizations. While it is controversial to completely divide these two groups of optimizations, we show that there are many useful transformations that we believe are challenging to express within the software compiler’s IR.

Optimization 1 — Orchestration of Communication and Concurrency: While a software IR expresses what should be done, it does not convey how to schedule the various concurrent operations. For example, a traditional compiler IR lacks a principled way to express the pipelining of concurrent loops with dependencies that are satisfied through explicit communication. Another example is the dependency between operations; in compiler IRs, the dependency is implicit, either through register names or memory locations. In dataflow-based accelerator hardware all dependencies have to be explicitly specified.

Optimization 2 — Pipelining and Contention: The creation of an optimized accelerator, necessitates the incorporation of pipeline depth and timing. For example, if it is known that the critical path through a loop takes 3 pipeline stages instead of 10 stages, this can affect the rate of loop body invocation and how much contention there is on a particular shared computational or memory resource accessed by the loop body, which in turn can affect further transformations in replication to mitigate. A traditional program IR does not express execution pipelining.

Optimization 3 — Memories: Compiler IRs tend to view memory as a single globally-shared address space and, at best, a set of coarse-grained locales (e.g., CUDA). However, hardware implementations can have any number of address-spaces with any kind of sharing (or not) between them. These address spaces can choose from different physical FIFO implementations, registers, SRAMs and caches, based on the access pattern.

Optimization 4 — Resource Management: Software compiler IRs tend to assume a machine model that has unbounded resources. Hardware, however, is finite and is a first-class design constraint when implementing a microarchitecture. Computer architects have to carefully budget for the resources (cycles, power, or area) and schedule the program operations using finite resources. Another important consideration is the ratio between resources for compute and storage. For instance, the width of SIMD execution units and, consequently, the power and area dedicated to compute vs the data streaming buffers is an important design consideration for hardware.

1 Using threads as the concurrency abstraction is possible, but is generally far-too heavyweight for many hardware designs.
Objective 3 — A compiler that compiles software representation (*What*) to hardware representation (*How*): Despite the benefits of having a hardware microarchitecture IR, we also recognize the need for a compiler IR. We recognize that the plurality of the optimizations that operate on the functionality of the program for optimizing software (e.g. common sub-expression elimination) are equally useful for optimizing hardware. This is sensible as simplifying the program or breaking certain dependencies are generally useful. However, the reverse seems to be less accurate, how the operations are carried out in parallel, and at a cycle level, intuitively has no bearing on traditional compiler optimizations. Our goal is to ensure that unlike traditional HLS, we do not hoist hardware optimizations on a compiler IR (not meant for it), and thereby make programs more complicated by intermingling structure (used to specify hardware) and execution (used to specify algorithm/program).

Objective 4 — A new abstraction that captures dynamic parallelism: Existing HLS tools that are built on top of a compiler IR, restricted to a sequential program-dependence-graph. These tools primarily target applications that can be statically scheduled; it is challenging, and imposable in some cases for them to extract parallelism in the presence of control and memory dependencies that are undecidable at compile time or the parallelism that evolves as the program runs, either due to control flow [125], or run time nondeterminism [39, 42]. We recognize that to support programs with irregular fine-grained parallelism expressed implicitly within the program, we need a new abstraction – at the software and at the hardware layers to capture this type of parallelism. From the software compiler side, there are related works [177] that realized this limitation with existing compiler IRs, and to overcome this limitation, they extended existing compiler IRs to enable them to capture other types of parallelism such as fork-join parallelism at the compiler IR level.

Our goal is to define a new abstraction at the microarchitecture level that captures high-level behavior from the software and makes an explicit structural representation of it in the new abstraction. We intend to show that with this new abstraction, the new abstraction has flexibility for realizing nested, heterogeneous, recursive, irregular or regular concurrency behavior patterns.

Objective 5 — A new abstraction that captures dependent instructions: It is unclear whether the gains in efficiency of hardware accelerators arise from recognizing that a less sophisticated processor for the acceleratable regions provides better performance per watt efficiency or the new microarchitecture. For instance, in dataflow-based accelerators, we observed that reducing the energy required for moving values between dependent operations mapped across function units imposes high dynamic power, and is a challenge. There are dataflow architectures that seek to improve utilization by mapping multiple operations temporally to the same function unit; however, doing so would require a complex packet-based network [174] that also results in more power consumption. In contrast, in Von-Neumann execution model, because the processor temporally maps multiple dependent operations to the same function unit, the data movement is minimized. In contrast, the data movement becomes substantial when it comes to the program’s regions that
contain irregularity and the instruction parallelism below of the available peak parallelism [63] in the hardware. In this case, the dataflow-based accelerators encounter challenges with fabric utilization and static power.

Our goal in addressing this problem is twofold: first, to define a new program abstraction that best fits these specific regions of the code, irregular dependent instructions, that still have the potential for acceleration and recognize this new abstraction at compile time. Second, propose a simplified Von-Neumann style accelerator that can execute this abstraction by reusing the functional units and less static power, and reducing the data movement with less dynamic power.

1.5 Approach

The contributions of this thesis are in the form of three published papers. These contributions are given in two parts, where each part is directed by objectives introduced in Section 1.4. The list of papers included in this thesis is given in Table 1.1. In the following, two main parts of this thesis are introduced.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Paper</th>
<th>Conference</th>
<th>Objective</th>
</tr>
</thead>
<tbody>
<tr>
<td>Part One</td>
<td>μIR – An intermediate representation for transforming and optimizing the microarchitecture of application accelerators [182]</td>
<td>MICRO2019</td>
<td>Objective 1 &amp; 2 &amp; 3</td>
</tr>
<tr>
<td></td>
<td>TAPAS: Generating Parallel Accelerators from Parallel Program [131]</td>
<td>MICRO2018</td>
<td>Objective 3 &amp; 4</td>
</tr>
<tr>
<td>Part Two</td>
<td>CHAINSAW: Von-Neumann Accelerators to Leverage Fused Instruction Chains [183]</td>
<td>MICRO2016</td>
<td>Objective 5</td>
</tr>
</tbody>
</table>

Table 1.1: Table of contributions

1.5.1 Hardware abstraction for software defined accelerators

We propose a generalized, intermediate representation for describing an accelerator microarchitecture, μIR and an associated pass framework, μopt. μIR represents the accelerator as a concurrent structural graph in which the components roughly correspond to microarchitecture level hardware blocks (e.g., function units, network, memory banks). As shown in Figure 1.2, this framework introduces two intermediate levels of abstraction where key optimizeds and tradeoff decisions can be made. There are two important benefits with this framework: It decouples microarchitecture optimizations from algorithm/program optimizations. It also decouples microarchitecture optimizations from the RTL generation. Designers express their ideas as a set of iterative transformations of the μIR graph that successively refine the accelerator architecture.

The μIR stack relies on a common intermediate representation composed of LLVM IR with parallel extension [177]. This extension 1.2(a) provides the front-end language bindings that trans-
Figure 1.2: Block diagram of the $\mu$IR compiler framework. The framework compiles high-level languages, such as C/C++ and Cilk, and DSL languages such as TensorFlow, from LLVM intermediate representation (IR) to a task-level representation. Using Task Compiler, task task IR is optimized, transformed, and lowered to a lower level intermediate representation for hardware accelerator designs called $\mu$IR structural graphs. This lower level IR is compiled by a $\mu$IR compiler and is generated as an efficient hardware accelerator design.
late Cilk/OpenMP and TensorFlow programs [178] to a parallel version of LLVM IR and it serves two purposes: 1) raising the level of abstraction for the programmer, starting from a high-level language, and 2) provides richer semantic information to the backend compiler about data parallelism.

Compilation in this framework is composed of two phases, Figure 1.2(b, c). In the first phase, a high-level compiler extracts loops as a set of high-level tasks, loop extraction, and forms a task graph, Figure 1.2(c), that represents loops as a group of nested tasks. In the second phase, each task in the task IR is then automatically lowered into a second IR level, $\mu$IR structural graph, Figure 1.2(e), which is a hardware-specific intermediate representation. We identify several vital abstractions required to create a new high-level synthesis language from the ground up and use these abstractions to build $\mu$IR. $\mu$IR is then optimized and lowered further by the hardware accelerator design compiler, Figure 1.2(f), referred to $\mu$opt as the $\mu$IR compiler. $\mu$opt is a toolchain that realizes architecture ideas as iterative transformations of the accelerator microarchitecture graph.

We demonstrate that $\mu$opt is capable of applying three broad classes of optimizations: 1) **Timing**, statically assigns operations to hardware units and changes pipelining, 2) **Spatial**, which replicates nodes representing hardware structures in the graph to improve throughput and reduce contention, 3) **Higher-Order Ops** enable a designer to introduce operators on composite data types such as Tensors (and vectors) to increase computational intensity. During these transformations $\mu$opt tunes the parameters of $\mu$IR components to optimize the generated RTL (e.g., operator bit-width, channel width).

When targeting FPGAs, we show that $\mu$IR generates optimized, synthesizable code along with C++ code and Python wrapper, which can be used on a host CPU to manage initialization and execution of the accelerator on the target FPGA. $\mu$IR currently supports Xilinx Ultrascale+ VU9P FPGAs on Amazon’s EC2 F1 Instances, Ultrascale+ ZCU102 SoCs, Arria10 and Altera DE1 SoC boards. $\mu$IR also supports the RISC-V RoCC accelerator interface.

**Contributions:**

- **Objective 1.** We decouple hardware from the software intermediate representation of software by proposing $\mu$IR, a new structural intermediate representation for designing software defined accelerators. $\mu$IR is essentially a software data structure that canonicalizes the description of microarchitectures and describes an execution model that is better matched to hardware, than the HLS compiler IR. This IR permits a richer set of microarchitecture optimizations and promotes a clear, quantitative understanding of the performance implications. In Chapter 3; first, we discuss the requirements of a new intermediate representation for hardware accelerators and then we provide details on $\mu$IR abstractions and their hardware implementation.

- **Objective 2.** We created $\mu$opt, a compiler framework that manipulates $\mu$IR. $\mu$opt decouples microarchitecture optimizations from the lower RTL and helps designers realize optimiza-
tions as an iterative pass of the $\mu IR$. $\mu opt$ optimization passes can be written as a set of transformations on $\mu IR$ and they can be applied to different accelerators and stacked for a specific accelerator. We implemented three classes of optimizations, Timing, Spatial, and Higher-Order Ops on five different components: compute units, concurrency control, memory network, scratchpads, and caches. In Chapter 4.5, we provide details on $\mu IR$’s transformations and compare the $\mu IR$ optimized accelerator with other existing HLS frameworks.

- **Objective 3.** We created a synthesizer that translates behavioral specification (software) to structural specification ($\mu IR$). The input to the synthesizer is an unmodified software program specifying the functionality and behavior required of the accelerator. Currently, we have passed C++, Cilk, Halide and Tensorflow programs to the synthesizer. Our flow automatically translates the program specification to a $\mu IR$ structural specification using LLVM IR as an intermediary step. The $\mu IR$ graph is a hierarchical specification of the whole accelerator block-level pipeline. More details on $\mu IR$ abstractions used in this stage are described in Chapter 3.

- **Objective 4.** We made $\mu IR$ as a hierarchical structural graph based on a parallel compiler intermediate-representation, TaskGraph. We demonstrate that the abstraction of TaskGraph permits the spawn functionality to be implemented similar to unconditional branch, except that the parent and child tasks are allowed to run concurrently. TaskGraph is built on Tapir-LLVM, a compiler that introduces parallelism markers in the compiler’s intermediate code. TaskGraph leverages these parallelism markers to auto extract the tasks that will be synthesized into execution units. TaskGraph includes support for arbitrarily nested parallelism and irregular task parallelism. It is language-agnostic and has been tested using Cilk, Cilk-P, and OpenMP. We have developed a library of hardware components for spawning and synchronizing tasks, buffering tasks, and inter-task communication. We demonstrate that using TaskGraph, we can compose these structural components to generate high-performance parallel accelerators. In Chapter 5, we introduce $\mu IR$’s hierarchical structure and how to explain the details on how $\mu IR$ can capture dynamic parallelism from high-level language.

**Frameworks:**

- **Muir-sim:** Muir-sim is a new simulation environment that improves software and hardware integration, and simulation accuracy compares to functional simulation. This framework integrates the hardware development process into the software stack from the beginning, allowing features to be incrementally implemented and evaluated as workloads evolve. Under this environment, the hardware description is the actual specification. This reduces the burden of maintaining consistency between the specification written usually in a higher language such as C/C++ and the actual hardware design described in a language such as Verilog. Moving to $\mu IR$-sim allows us to have a more fluid hardware-software specification, and invite more contributions to modify different layers of the stack. Moreover, this inte-
igration provides more accurate performance feedback, i.e. clock cycles, compared to the traditional functional model of a hardware accelerator. This is because \( \mu IR \)-sim is based on an open-source hardware simulator called (Verilator), which compiles Verilog designs down to C++ classes for cycle-accurate simulation.

https://github.com/sfu-arch/muir-sim

- **Muir-lib**: Muir-lib is a library of hardware components for auto-generating highly configurable parallel dataflow accelerator. \( \mu IR \) provides the implementation of the following hardware units: 1) a set of highly configurable and parameterizable computation nodes, 2) a set of control units to support arbitrary control path, 3) a collection of configurable Memory structures like Cache, Scratchpad memory, and 4) a set of standard flexible junctions and interfaces to connect different pieces of the design.

https://github.com/sfu-arch/muir-lib

- **Muir-Generator**: \( \mu IR \)-Generator is a tool to generate hardware accelerators from software programs. MuIR-Generator uses \( \mu IR \) as an intermediate representation (IR) to design hardware accelerators. Currently, the MuIR-Generator supports C/C++ and Cilk programs.

https://github.com/sfu-arch/muir-lib

- **Muir-shell**: We provided different shells with software drivers for \( \mu IR \) accelerator to connect to a different framework. Currently, we support Xilinx Ultrascale+ VU9P FPGAs on Amazon’s EC2 F1 Instances, Ultrascale+ ZCU102 SoCs, and Arria10 and Altera DE1 SoC boards. \( \mu IR \) also supports the RISC-V RoCC accelerator interface. We released these shells and software drivers under the following repository:

https://github.com/sfu-arch/muir-shells

- **Muir-Tutorial**: We provided a detailed tutorial on \( \mu IR \) framework, including how to use \( \mu IR \) and build hardware accelerators and how to extend \( \mu IR \) and include new abstractions to the compiler and backend for research purpose. We described all the engineering efforts that been done on both hardware and compiler parts of the \( \mu IR \) project. We also included a comprehensive tutorial on how to use \( \mu IR \) with AWS and Rocket-Chip. We believe this tutorial can be used for educational purposes and define various research and course projects based on the framework. We discuss some of the research ideas on the \( \mu IR \) framework in Chapter 8. We released this tutorial under the following repository:

https://github.com/sfu-arch/muir-tutorial

### 1.5.2 Hardware abstraction for fused instructions

*We propose Chainsaw, a Von Neumann-style accelerator, for executing fused instructions Chains.* The key contribution is that chains are decoupled from functional unit design and are discovered at compile-time, thereby eliminating the tension between fused instruction efficiency and generality, application coverage and hardware design cost. A chain is a set of instructions that exhibits a
strictly sequential dependence pattern, i.e., each instruction in the chain strictly communicates only with the next instruction in the sequence. Chains are a generalization of the widely used fused multiply-and-accumulate instruction (a chain of an add and a multiply operation) or paired μops [18]. Converse to SIMD or VLIW instructions, which express parallelism, chains express the lack thereof. Chapter 6 provides details on chain abstraction, and our new microarchitecture to execute the chains efficiently.

Contributions:

- **Objective 5.** We present a new instruction abstraction, *Chains*, that localizes communication between dependent instructions to minimize energy consumption. Chains do not require any custom function units. We develop a fully working prototype compiler based on LLVM to extract chains. We analyze chain formation algorithms for maximizing chain lengths (MaxSize) and ILP (MaxILP) and study the tradeoffs between increasing ILP and fusing operations to localize communication. We design the *Chainsaw* accelerator and evaluate its efficiency compared to a reconfigurable dataflow fabric (CGRA); we demonstrate the efficiency of *Chainsaw* for applications that do not possess a high level of ILP.

Frameworks:

- We developed *Chainsaw* that contains three pieces: 1) *Chainsaw*-compiler that analyzes and extracts fused-instructions from program traces. *Chainsaw*-scheduler that statically schedules extracted fused instructions (details on Chapter 6.4) And *Chainsaw*-simulator which is a detailed cycle-accurate simulator that models the host core, the *Chainsaw* accelerator, and spatial fabrics of parameterizable size. We included Ruby from GEMS framework to model the memory system. Ruby models inclusive/exclusive cache hierarchies with various replacement policies, coherence protocol implementations, and interconnection networks. We assume that *Chainsaw* accelerator is coupled closely with the host core and communicates with the host core via the L1 cache.

  🐬 [https://github.com/sfu-arch/chainsaw](https://github.com/sfu-arch/chainsaw)

1.5.3 Publications

This dissertation includes work published at three peer-reviewed conferences. Each research chapter of this dissertation relates to one or more papers, which have been published. I have collaborated with my supervisors, Dr Arrvindh Shriraman, Dr Nick Sumner, and Dr Tony Nowatzki, for conducting the research projects in all chapters. The publications are listed below, along with the contribution of the authors:

- **MICRO2019** — μIR — An intermediate representation for transforming and optimizing the microarchitecture of application accelerators [182] was previously published
at the 52nd IEEE/ACM International Symposium on Microarchitecture with co-authors Reza Hojabr, Navid Rahimi, Sihao Liu, Apala Guha, Tony Nowatzki and Arrvindh Shriraman

- Amirali Sharifian: I was the main contributor of the software stack and hardware backend, including the idea, design, development, and evaluation of the work. The software stack contains a lowering pass from high-level programs to the proposed hardware intermediate representation. In terms of the backend, I designed and implemented the hardware accelerator modules in Chisel. In addition, I provided a software driver that can be used on a host CPU to administrate initialization and execution of the accelerator on the target FPGA and software simulation.

- Reza Hojabr, Navid Rahimi and Sihao Liu: I collaborated with my colleagues on the evaluation of $\mu$IR framework, Intel HLS and VLSI synthesis flow.

• MICRO2018 — TAPAS: Generating Parallel Accelerators from Parallel Program [131] was previously published at the 51st IEEE/ACM International Symposium on Microarchitecture with equal contribution with Steve Margerm and co-authors Apala Guha and Arrvindh Shriraman

- Amirali Sharifian: I was the main contributor for the TAPAS’s compiler framework and the compute and control components of the Chisel library including the idea, design and development.

- Steven Margerm: Steve, a former masters’ student at SFU’s computer architecture lab, was the main contributor to TAPAS’s hardware tasks, framework, and memory system, including the idea, design and development.

• MICRO2016 — CHAINSAW: Von-Neumann Accelerators to Leverage Fused Instruction Chains [183] was previously published at the 49th IEEE/ACM International Symposium on Microarchitecture with co-authors Snehashish Kumar, Apala Guha and Arrvindh Shriraman

- Amirali Sharifian: I was the main contributor for the software and the simulator backends, including the conception, design, development and evaluation of the work. The software stack contains an LLVM pass to trace programs and extract program traces at the runtime, a graph decomposer to form fused instructions, and a static scheduler to statically schedule fused instructions on the proposed core lanes. The simulator contains a framework for simulating architecture, baseline CGRA and baseline out-of-order cores.

- Snehashish Kumar and Apala Guha: I collaborated with my colleagues on the evaluation of Chainsaw framework.
1.6 Dissertation Organization

Chapter 2 provides background on existing languages for designing hardware accelerators. In this chapter we discuss the motivation for proposing different hardware languages with different levels of abstraction, their strengths and weaknesses, and how the hardware compilers are engineered, and we then we provide a qualitative comparison between these frameworks. In Chapter 3, we discuss the requirements for our improved hardware abstractions. We specify the implementation of this abstraction, $\mu IR$ intermediate representation, and describe the lowering process from high-level language to $\mu IR$. Chapter 4 discusses $\mu opt$, the $\mu IR$ compiler and optimizer framework, and the details of the transformations and hardware-specific optimizations that $\mu opt$ performs. In this chapter, we evaluate and compare $\mu IR$ with existing commercial HLS frameworks from different benchmarks that are implemented in different languages such as C/C++, Cilk, Halide and Tensorflow. Chapter 5 introduces the notion of a task graph. We discuss the task graph abstraction and the necessity of having task abstraction in $\mu IR$ to be able to support dynamic parallelism. We show that supporting dynamic parallelism allows us to support nested parallel loops and recursive functions in hardware. In Chapter 6, we discuss a new program abstraction; chain. We show that a dataflow model for accelerating programs does not always provide efficiency, particularly when irregular control flow exists. We discuss our new proposed accelerator architecture that is based on a von Neumann machine to accelerate this new abstraction. Chapter 7 outlines the software that was developed to help with conducting the research presented herein. Finally, chapter 8 presents concluding thoughts and directions for future work.
Chapter 2

Background And Motivation

In this chapter, we provide a brief overview of reconfigurable architectures, specifically FPGAs, and mention existing challenges with programming FPGAs. We then overview the designing hardware techniques for these devices and point out the pros and cons of each technique. Finally, we provide a comparison between these techniques and our motivation for proposing a new intermediate representation for designing hardware.

2.1 Hardware Accelerators

When we look at computing performance spectrum, we see General-Purpose Processors (GPPs) on one side that deliver moderate performance with inefficiency in terms of power but enabling programmability and supporting the various application from different application domains [83]. On the other extreme, however, we see application-specific integrated circuits (ASICs) that deliver high performance and low power by sacrificing programmability and being designated for a specific application. Fortunately, there exist other architectures fit in the performance spectrum in between. Coarse Grain Reconfigurable Architectures (CGRAs) and Field Programmable Gate Arrays (FPGAs) are two architectures promising higher performance per watt compare to GPPs in exchange for less reconfigurability in the spectrum. Usually, these devices contain programmable logic blocks or processing elements (PEs) that can be configured for a specific operation and then pass data between each other using internal interconnections.

Reconfigurable devices, initially, were considered as a low-volume, low-density replacements of ASICs. They were simple systems with a few tens of modest configurable logic blocks, and programmable interconnects. Thanks to Moore’s law, however, these devices have been becoming extremely complex chips. Recently, Xilinx has revealed a new accelerator platform, Versal, that integrates FPGA technology with ARM CPU cores, Digital Signal Processing (DSP) cores and Artificial Intelligence (AI) processing engines. The target applications for Versal are data-intensive workloads running on datacenters [214, 212, 213]. To have a better understanding of FPGA architectures, we have shown the underlying architecture of FPGAs in Figure 2.1.
There are two major building blocks in an FPGA, *Configurable Logic Block* and *Programmable interconnect*. Each logic block contains look-up tables (LUTs) and Flip-Flops (FFs) that are used to implement combinational and sequential logic. Apart from the homogeneous array of logic cells, most of the FPGAs also contains discrete components such as BRAMs (block RAMs), DSP (digital signal processing) slices, processor cores, and various communication cores.

The contrast in the performance of reconfigurable devices compare to processors comes from their execution model. In the processors, their execution model is base on Von Neuman machines. In this model, each program is written in a high-level language, and the processor has a fixed well-defined Instruction Set Architecture (ISA). The job of the compiler is to translate the input behavioral description of the program to the ISA. Finally, the processor fetches the compiled program from the memory to its pipeline, decodes and executes the instructions to run the program. In this model, for executing each single instruction the processor needs to go through all the pipeline stages and the overhead of these stages makes the processors inefficient [83]. Moreover, both instructions and data have to fetched from external memory into processor pipeline and the bandwidth between processor and memory is often the critical factor in determining the overall performance [211].

On the other hand, reconfigurable devices support explicit implementation of data and control path on chip as oppose to Von Neuman machines that implicitly supports arbitrarily control flow using fetch and decode pipeline. This allows the hardware to be power efficient by skipping unnecessary pipeline stages. The inherent fine-grained architecture of such devices is well suited for exploiting various forms of parallelism present in the application. Besides, these devices contain distributed memories that provide much-needed bandwidth to satisfy the demands of concurrent
logic. In summary, FPGAs can exploit: 1) multiple levels of nested parallelism, 2) data locality with custom data pipelines and 3) defining custom memory hierarchies [25, 69, 78].

Unfortunately, the use of such devices is restricted to a narrow segment of hardware programmers. The larger community of software programmers has stayed away from this technology, primarily because of the challenges experienced by beginners trying to learn and use FPGAs. For example, for designing a new accelerator designer must account for the timing between pipelined signals and the physically limited compute and memory resources available on the target device. The designer must also manage partitioning of data between local scratchpads and off-chip memory to achieve good data locality [132]. This shows that: "all the features that make reconfigurable architectures efficient also make them hard to program".

2.2 Hardware Accelerator Design Abstractions

Reconfigurable devices are predominantly programmed using Hardware Description Languages (HDLs) that offer very low level abstractions. In this paradigm, hardware designers should think about the design in terms of low-level building blocks such as gates, registers, and multiplexers. HDLs are well suited for describing a design at that level of abstraction. While these languages support hardware design at behavioral level, it is not recommended to design the hardware at this level and leaving its destiny in the hands of a synthesis tool. However, this is in complete contrast with the software programming languages, which have evolved since they introduced. Software languages are designed to describe the behavior of an algorithm for processors. In the backend, they benefit from well-established ISAs and advanced compilers that offer a much simpler programming experience. From the language side, they enjoy the existence of concepts such as object-oriented and polymorphism, as well as automatic memory management (garbage collection). One of the commonly-accepted solutions for closing this abstraction gap between hardware and software languages, as proposed by semiconductor roadmaps is to raise the level of abstraction in the design process [198]. To achieve the acceptable productivity gains and to bridge the semantic gap between higher abstraction levels and low-level implementations, the goal is to automate the design process as much as possible. To make automation possible we need to have: 1) A well-defined system abstraction level, like software programming languages, 2) well-known components of particular abstraction level, ISAs in processors, and 3) having a clear semantics to move from a higher level of abstraction to a lower level (synthesis). As a result, a higher level of abstraction can drastically increase designer’s productivity because it reduces the number of objects a designer needs to consider and allows the industry to design and manufacture complex application-oriented integrated circuits in shorter periods.

There are two primary reasons for emphasizing more abstract system design methodologies. The first is the fact that high-level abstractions are closer to a designer’s usual way of reasoning. It would be difficult to imagine, for example, how a designer could specify, model and communicate a system design through a schematic or hundred thousand lines of HDL code. The more complex
the design, the more difficult it is for the designer to comprehend its functionality when it is specified on register-transfer level of abstraction. On the other hand, when a system is described with an application-oriented model of computation as a set of processes that operate on abstract data types and communicate results through abstract channels, the designer will find it much easier to specify and verify proper functionality and to evaluate various implementations using different technologies. The second reason is that computation kernels are usually defined by the experts in the application domain who understand applications very well but only have basic knowledge of design technology and practice. Higher-level abstraction allows them to specify, explore, and verify hardware accelerators without expert knowledge of system engineering.

We first overview the techniques try to extend HDL languages and raise their level of abstraction by either introducing new components to the language like SystemVerilog [188] or adding new programming features such as meta-programming, like in Chisel [14]. We describe the compiler backend of these new languages and how they have been architect to enable the reusability of these abstractions across different hardware languages. We elaborate on HLS frameworks, their abstractions, their limitations and compare different existing HLS frameworks.

### 2.3 Hardware Construction Languages (HCL)

One of the common practices that have been done to increase HDL design productivity is to extend the HDL languages like Verilog and VHDL. SystemVerilog [188] was one of the successful attempts to extend Verilog language, for example. SystemVerilog adds two sets of components to Verilog: the synthesizable components that extend and add several features to Verilog’s type system and design parametrization; and use of object-oriented modelling to improve verification components of Verilog. Bluespec SystemVerilog [8], was another attempt to raise Verilog’s control abstraction and supports state machine inference from nested while loops. Chisel [14] perhaps is the most successful and generalized Hardware Construction Language (HCL) example, which is implemented as a domain-specific language (DSL) for describing hardware circuits embedded in Scala. Chisel provides modern programming language features such as meta-programming and object oriented concepts coupled with library availability for accurately specifying low-level hardware blocks, but which can be readily extended to capture many useful high-level hardware design patterns. The benefits that Chisel offers over classic HDLs can be folded in two points: 1) Chisel improved productivity through new language features, like object oriented programming, functional programming and availability of reusable libraries. 2) It improves specialization due to the hardware-compiler structure. Chisel allows designs to be more parameterizable and modular [95]. For instance, a designer can write a recursive Scala function to construct an adder-reduction tree, parameterized on bit-width. Unlike the explicitly unrolled version necessary in Verilog, the same generator could be reused anywhere an adder tree is desired. Besides, Chisel adds modularity to the design process. There are mature Chisel projects like Rocket-Chip [11] and Diplomacy [49] that can be used as libraries, as examples of Chisel’s modularity. Using these libraries, the user can
import a RISC-V microprocessor in the same way that he imports a graph library in a software program.

To compare Chisel with other HDL languages, we show hardware implementation of Finite Impulse Response (FIR) Digital Filter [209] in Chisel. Below is the mathematical operation that describes how FIR filter applies to an input sequence $x[n]$, if the function $h[k]$ is the response of this filter.

$$y[n] = \sum_{k=0}^{N-1} h[k] x[n - k]$$

![Figure 2.2: Finite Impulse Response (FIR) Digital Filter](image)

In Figure 2.3, we have shown an example of implementing FIR filter in Chisel using similar Verilog synthesizable primitives. The limitation with this design is that because the language is limited to make the coefficients parameterizable, it is challenging to reuse the same design across the design with a different coefficient. In Figure 2.4a, we have shown the same implementation of FIR filter using Chisel primitives and Figure 2.4b shows how it is possible to reuse the module across the different design.

```scala
// 3-point moving average implemented in the style of a FIR filter
class MovingAverage3(bitWidth: Int) extends Module {

  val io = IO(new Bundle {
    val in = Input(UInt(bitWidth.W))
    val out = Output(UInt(bitWidth.W))
  })

  val z1 = RegNext(io.in)
  val z2 = RegNext(z1)

}
```

![Figure 2.3: A chisel implementation of FIR with similar Verilog primitives [62]](image)
// Generalized FIR filter parameterized by the convolution coefficients

class FirFilter(bitWidth: Int, coeff: Seq(UInt)) extends Module {

    val io = IO(new Bundle {
        val in = Input(UInt(bitWidth.W))
        val out = Output(UInt(bitWidth.W))
    })

    // Create the serial-in, parallel-out shift register
    val zs = Reg(Vec(coeff.length, UInt(bitWidth.W)))
    zs(0) := io.in
    for (i <- 1 until coeff.length) {
        zs(i) := zs(i-1)
    }

    // Do the multiplies
    val products = VecInit.tabulate(coeff.length)(i => zs(i) * coeff(i))

    // Sum up the products
    io.out := products.reduce(_ + _)
}

(a) Chisel FIR Filter implementation [62]

val movingAverage3Filter = FirFilter(8.W, Seq(1.U, 1.U, 1.U)) // same 3-point moving average filter as before
val delayFilter = FirFilter(8.W, Seq(0.U, 1.U)) // 1-cycle delay as a FIR filter

(b) Chisel FIR reuse examples [62]

Figure 2.4: While Chisel provides similar base primitives as synthesizable Verilog, the power of Chisel comes from the ability to create generators, such as in FIR filter that is defined by the list of coefficients and use and re-use them across designs.

2.4 Hardware Compiler Framework (HCF)

The second advantage of Chisel compare to traditional HDLs is benefiting from supporting a Hardware Compiler Framework (HCF) that looks very much like LLVM [115] applied to hardware generation 2.5 as backend. The Chisel-to-Verilog process forms part of a multi-stage compiler. The Chisel front-end compiles Chisel to a circuit intermediate representation called FIRRTL (Flexible Intermediate Representation for RTL) [120]. FIRRTL mid-end then optimizes FIRRTL and applies user-custom transformations. Finally, the Verilog backend emits Verilog based on the optimized FIRRTL.
In this process, FIRRTL represents the standardized elaborated circuit that the Chisel HDL produces. FIRRTL represents the circuit immediately after Chisel’s elaboration but before any circuit simplification. FIRRTL is designed to resemble the Chisel after all meta-programming has executed. Thus, a user program that makes little use of meta-programming facilities should look almost identical to the generated FIRRTL. In fact, in this architecture, Chisel is mostly used for its meta-programming facilities; hence, Chisel front-end can be lightweight. Another advantage of FIRRTL is that other HLD languages, such as Verilog [219] can target it and reuse the majority of the compiler toolchain. There are other similar works like LLHD [179], CoreIR [134], LGraph [155], LCAST [207], RTLIL [210] and netlistDB [137], that try to propose a compiler framework for hardware compilation. However, maybe the main obstacle of HCF frameworks is the lack of existence of a unified IR that all the compiler frameworks are agreed on and build their transformations and optimizations on top that.

While these improvements allow for more powerful meta-programming compared to Verilog generate statements, users still write programs at a timed circuit level. This is still the most critical limitation of HCLs to improve the overall system design productivity.

In this work, we used Chisel as a backend language of our framework. The most two significant benefits of using Chisel in our frameworks are: 1) Chisel enables us to have a parameterized hardware design and target different hardware. For example, the off-chip memory data bit width varies from development SoC boards to AWS cloud FPGAs and supporting different backend for one design can always be challenging. However, with the use of the Chisel parametrization feature, we can easily swap the backend with no changes to the main design. The second benefit is that Chisel enables us to reuse our hardware modules in different designs and use our framework as a library to integrate with external projects such as RocketCore [11].

### 2.5 High-Level Synthesis (HLS)

High-Level Synthesis (HLS) refers to converting the algorithmic behavior of an application into a low-level RTL (register-transfer level) description. Over the past few decades, several high-level languages and HLS compiler frameworks have been proposed to ease the pain of hardware accelerator design at the RTL level. All these frameworks intend to raise the level of abstraction at which
the user can write a program in the high-level language and compile it down to HDL. We classify most of these frameworks into C-based frameworks and high-level language–based frameworks.

In pure C-to-gates HLS [26, 142, 222, 216, 36, 47] and Intel’s OpenCL-to-FPGA [199], front-end captures system behavior with a model of computation in a standard language such as C, C++, SystemC and OpenCL as an input. The language components in these frameworks are in the form of untimed mathematical expressions and nested serial loops, pipeline loops or parallel loops. The major challenge in these frameworks is that the input can always be suboptimal, and the compiler only supports a limited type of microarchitectural optimizations (e.g. constant propagation, dead-code elimination and bit-width analysis). In this case, the frameworks require the designers to sprinkle ad-hoc pragmas and types in the program to ensure that the tools produce good quality RTL (e.g., Xilinx’s \texttt{stream<T>} in a C++ program translate to FIFO queues in RTL). As a result, how to write a good C/C++ code targeting the HLS tool and annotate the program to guide the compiler to get the optimal performance [46] becomes the central barrier and makes it prohibitive to most software programmers.

In Figure 2.6, we have shown the same Chisel implementation of FIR filter (Figure 2.4a) in Vivado HLS. This implementation contains high abstraction such as for loops, \texttt{Shift\_accum\_Loop}, that iterates over coefficients and makes a pipelined implementation of FIR filter.

We recognized three major subproblems in existing HLS frameworks that need to be addressed by a new accelerator intermediate representation. The first problem is the absence of explicit parallelism in the C-based HLS frameworks and how to automate nested-loop pipelining. The second problem is that most of the HLS tools universally generate statically scheduled datapaths. Though, these datapaths have a hard time exploiting parallelism in code with potential memory dependencies, with control-dependent dependencies in inner loops, or where long latency control decisions limit performance. The third problem is that the existing HLS tools have limited support for concurrency. Their focus is mostly on exploiting static data-level or thread-level parallelism that can be determined and scheduled at compile-time and mapped to a fixed pipeline. Nevertheless, to achieve higher performance, HLS tools need to have support for dynamic parallelism generated at run-time rather than statically at compile time. In the next section, we elaborate more on each of these problems and their related works.

### 2.5.1 Absence of Explicit Parallelism

Usually, HLS tools adopt polyhedral tools to automate loop pipelining and memory banking decisions, and they try to reduce the need for adding structural description or explicit parallelism hints to the input code. However, such techniques are limited to only \textit{affine accesses within a single loop nest} [208]; it does not address non-affine cases or cases where accesses to the same memory occur in multiple loop nests. For instance, Pouchet et al. [157] combine HLS tools with polyhedral analysis to optimize input design for improving data reuse and drive design space exploration to find loop tile sizes, for tileable programs. This technique captures a more extensive design space
```c
#include "fir.h"

out_data_t fir_filter(inp_data_t x, coef_t c[N]) {
    static inp_data_t shift_reg[N];
    acc_t acc = 0;
    acc_t mult;
    out_data_t y;

    Shift_Accum_Loop:
    for (int i = N - 1; i >= 0; i--) {
        // acc+=x*c[0];
        shift_reg[0] = x;
        if (i == 0) {
            // acc+=x*c[0];
            shift_reg[0] = x;
        } else {
            shift_reg[i] = shift_reg[i - 1];
            // acc+=shift_reg[i]*c[i];
        }
        mult = shift_reg[i] * c[i];
        acc = acc + mult;
    }
    y = (out_data_t)acc;
    return y;
}
```

Figure 2.6: FIR filter in HLS [215]

than previous works by including tile sizes, but this is limited to the benchmarks with strictly affine data accesses.

To elaborate more on the restrictions on the absence of explicit parallelism in C-based HLS frameworks, we use the Gaussian discriminant analysis (GDA) kernel, Figure 2.7. In this example, all loops are parallel, and one valid design would be to implement $L_1$ as a coarse-grained pipeline with $L_{11}$ and $L_{121}$ as its stages. Vivado provides DATAFLOW optimization to optimize the flow of data between tasks (functions and loops), coarse-grain pipelining, and ideally pipelined functions and loops for maximum performance. Nevertheless, there are limitations to this directive. The directive does not support arbitrarily nested coarse-grained pipelines, multiple producers and consumers between stages, or coarse-grain pipelining within a finite loop scope [217], as required in the outer loop in Figure 2.7. This limitation is the inherent obstacle with the absence of explicit parallelism in the language, and as a result, it leads the compiler analyses to be conservative and produce sub-optimal designs. Hence, the HLS tool can not generate the optimal design for this example.

We propose a new generalized loop abstraction for representing nested loops in our intermediate representation. We show that using this new abstraction, we can implement arbitrarily nested
loops and pipeline them. To this end, our new loop abstraction encloses nested loops in a separate entity and disassociate them from their outer loop. Using generalized loop abstraction, each loop entity can be pipelined and be variable latency to the outer loop (Chapter 5).

### 2.5.2 Dynamic Scheduling

Commercial HLS tools almost universally rely on building datapaths that are controlled following static schedules; that is, the cycle when every operation is executed is fixed at synthesis-time [61]. There is an analogy between statically scheduled HLS frameworks with Very-Long Instruction Word (VLIW) compiler techniques [113]. Both of these frameworks seek to exploit fine-grain parallelism between operators at the compiler. These techniques serve well the applications that have relatively regular behavior. However, by looking at the broader classes of applications, the ability of dynamic scheduling to automatically extract parallelism becomes essential. With dynamic scheduling, not only complex loop transformations (and designer’s hints `pragma`) are often unnecessary, but more parallelism can be extracted in the presence of control and memory dependencies undecidable at compile time. Moreover, dynamically scheduled circuits allow speculative execution [102] to pipeline the loop iterations in the presence of irregular control flow. The examples of dynamically scheduled dataflow are Elastic Circuit [50, 195], CGPA [124], ASH [23, 22], AHRL [171], CHiMPS [167] and [99]. The idea in dynamic dataflow is to refine the dataflow to replace triggering the operators through a centralized pre-planned controller with embedding controller locally in each component and make scheduling decisions in the circuit as it runs.

We embed the notion of dynamic scheduling into our new intermediate representation nodes. These nodes expose dynamism to our backend compiler, and our compiler leverage this information to create a dynamically scheduled dataflow. We show that using this new information; our compiler can automatically pipeline the dataflow nodes with introducing latency insensitive ready-valid interfaces between each operation in the dataflow, (Chapter 3.2).
```c
while(!done) {
    spawn_pipe_stage() { //Stage-0
        chunk_t *chunk = next_chunk()
        if(chunk == NULL) { done = true; exit(0); }
    }
}

spawn_pipe_stage() { //Stage-1
    chunk->is_dup = dedup(chunk);
}

if(!chunk->is_dup) { //Stage-2
    spawn_pipe_stage() { compress(chunk);
    }
}

spawn_pipe_stage() { //Stage-3
    write_to_buffer(chunk);
}
```

Figure 2.8: Pseudocode for PARSEC’s [18] Dedup based on Cilk-P, modified to enhance clarity

### 2.5.3 Dynamic Parallelism

Overall, all the pure C-to-gates HLS techniques rely on capturing parallelism between fine-grain operations of sequential code by constructing the control and dataflow graph (CDFG) of the computation kernels. They use scheduling algorithms to extract parallelism between the operations that are provably independent in the CDFG. The highest performance hardware exploits both fine-grain and coarse-grain parallelism, and these techniques are less effective in capturing coarse-grain parallelism. To extract coarse-grain parallelism, main-stream HLS tools frameworks use parallel programming constructs and/or annotations. For example, CatapultC [165] and Vivado [142] accepts SystemC [148] processes and modules. Vivado, also accepts parallel functions/code-blocks communicating through dataflow channels. Altera openCL compiler [93] accepts single instruction multiple thread (SIMT) programming model. CMOST [221] is a C-to-FPGA framework that uses task-level modelling to exploit multi-level parallelism. The HLS tools create multiple hardware execution units onto which successive loop iterations are statically scheduled at the hardware construction time only if the number of input/output iterations and the execution time of each dataflow node for processing the input iteration is known statically. There are two limitations with this approach. The first limitation is that HLS tools must plan for the worst-case and allocate resources for all possible iterations regardless of whether they are executed, and must handle corner cases. Second, in many concurrent programs the parallelism evolves as the program...
runs, either due to control flow [125], or run-time non-determinism and statically it is not possible to predict the behavior of the application. The techniques that existing HLS techniques use to expose parallelism are not capable of capturing such type of parallelism. In Figure 2.8 we show Dedup example from PARSEC benchmarks [18]. The figure includes the commented pseudo code. HLS tools find this particular code sample challenging and cannot generate an optimal microarchitecture. There are three distinct characteristics in this example. First, the stages in the pipeline change based on the inputs. As shown in the task graph, for some iterations stage-2 could be entirely skipped based on the results from stage-1. Second, the stages have different ordering constraints and exhibit nested parallelism. Stage-2 is embarrassingly parallel while stage-1 enforces ordering across each sequence. Finally, the pipeline termination condition needs to be evaluated at runtime and cannot be statically determined (e.g., bounded loop).

We define a new hierarchical structural graph that includes first-class support for generic tasks. Using this hierarchical graph, we support arbitrary heterogeneous parallel patterns, including nested loops and recursion (Chapter 5).

2.6 High-Level Language–Based Frameworks

There are recently proposed frameworks that have been implemented as DSL languages embedded base on popular modern programming languages such as Java and C++. Among these frameworks Esterel [20], Kiwi [184], IBM’s Liquid Metal [12], Gorilla++ [116] and Spatial [109] for generating hardware circuits. These frameworks benefit from the high-level abstracts that their host languages offer, such as object-oriented programming, polymorphism and automatic memory management. For instance, Spatial embeds low-level abstraction into high-level primitives like parallel patterns [158]. Spatial shows that these particular primitives within the language are better suited for accelerators targeting applications with data locality and data parallelism. Figure 2.9 shows the FIR implementation in Spatial language. In this example, the filter is implemented with specific Reduce primitive and the memory hierarchy is defined using RegFile, StreamIn/StreamOut and DRAM constructs.

The challenge with Spatial and other hardware DSL languages is that they try to strike a balance between high-level constructs in a language like Chisel for improving programmer productivity versus low-level syntax for tuning performance. For example, in Spatial the types of control structures fall into four types: FSMs, Foreach, Reduce and MapReduce. As long as the designer can express the design in such patterns, the back-end compiler can reason about the available parallelism and automatically pipeline the design. However, these patterns are designed to capture regular and predictable parallel patterns and in many cases, such control structures are noted enough to express arbitrary designs, and the designer or synthesizer fails to express the design in these DSL languages.
object FIRFilter extends SpatialApp {
  override val target = targets.DE1
  val Nmax = 16

  @virtualize def main(): Unit = {
    val weightArray = loadCSV1D[Int]("weights.csv")
    val N = ArgIn[Int]
    setArg(N, weightArray.length)
    val weights = DRAM[Int](N)
    setMem(weights, weightArray)
    val in = StreamIn[Int](target.GPInput1)
    val out = StreamOut[Int](target.GPOutput1)
    Accel {
      val w = RegFile[Int](Nmax)
      val taps = RegFile[Int](Nmax)
      w load weights(0::N)
      Stream(*) { _ =>
        taps <<= in
        out := Reduce(0)(Nmax by 1 par Nmax){i => w(i) * taps(i)
          _+_
        }
      }
    }
  }
}

Figure 2.9: FIR filter in Spatial [187]

(a) Levels of abstraction on each domain.

(b) Y-tasks.

Figure 2.10: Gajski-Kuhn Y-chart [67].
2.6.1 Comparison between Spatial and HLS

In this subsection, we make a comparison between Spatial language and HLS frameworks. For this purpose we use Y-Chart [206], Figure 2.10b. Y-Chart divides design representation into three domains, three axes on the chart: 1) The behavioral domain describes the behavior, or functionality, of the design, ideally without any reference to how this behavior is achieved by an implementation. 2) The structural domain describes the abstract implementation or logical structure of the design as a hierarchy of components and their interconnections. 3) Physical, usually called layout or board design. The physical design describes the different dimensions of each component. The concentric circles of the Y-chart represent the different levels of abstraction (Figure 2.10a). The level of detail increases from the outside inward.

Spatial is a structural language that tries to raise the level of abstraction of Chisel language. The main difference between Spatial and HLS is that Spatial as a hardware language expects the designer to reimplement the design in this new structural language using its specialized primitives. The advantage of Spatial as a hardware language against other HDL languages like Chisel is that it formalizes the design space by defining specific primitives and allows the compiler to search in the design space to tune the design parameters. In this type of framework, there is a clear separation between algorithm and hardware description, which the designer still needs to have knowledge about the algorithm and implement the desired algorithm in this new language. Nonetheless, the HLS frameworks try to fill this gap by automating the design process by using a software language to describe the hardware accelerator and synthesize a new design in RTL. Figure 2.10a shows where each of these language abstractions stands on Y-Chart.

2.6.2 Image Processing DSLs

Another noteworthy class of hardware languages are image processing DSLs. Recently, there have been explicit proposed image processing DSLs due to regular data access and the static nature of signal processing and multi-media workloads. The narrow domain allows these DSLs to offer high-level specification and more concise abstractions for specifying stencil operations. These languages usually rely on source-to-source translation, and their implementation is a mixture of a DSL language (front-end) and standard HLS tools (back-end). Recent work on Halide [168] has demonstrated targeting heterogeneous systems by generating intermediate C++ and Vivado HLS [161]. Rigel [88], Darkroom [87] and AnyHLS [3] generate Verilog, and PolyMage [40] generates OpenMP and C++ for high-level synthesis. Rigel and Darkroom support generation of specialized memory structures, such as line buffers, in order to capture reuse. HIPACCC can infer memory hierarchy on GPUs from a fixed set of access patterns.

The limitation with these DSL frameworks is that their solution is not generalized, and it is limited for only a specific domain requiring dedicated support from the compiler to introduce new primitives. Therefore, these languages and compilers are restricted to a specialized application domain they have been designed for.
We show the generality of our approach by evaluating the framework against applications from different domains. We evaluate Halide, Tensorflow, irregular C-based, parallel Cilk and recursive applications (Chapter 4.5).

2.7 Comparison to Our Work

We conclude this chapter by making two comparisons between our framework with related works we overviewed in this chapter. Based on three high-level metrics, we make our first comparison between the three techniques, HDLs, HLS and DSLs. As we mentioned to feel the productivity gap in the process of designing hardware any technique needs to satisfy three metrics: 1) A well-defined system abstraction level (Abstraction Level), 2) well-known components of particular abstraction level that can be optimized (Performance) and 3) Having a precise semantics to move from a higher level of abstraction to a lower level (Portability). Figure 2.11 shows our comparison on rectangular diagrams. While using HDL the designer can hand-tune the design and benefit from the highest performance compared to other techniques, their low level abstraction, and the lack of clear semantics to synthesize a structural definition from the behavioral, makes the hardware design challenging. The HLS frameworks offer a higher level of abstraction than HDL languages. They have a well-defined semantic to synthesize a behavioral code, in most cases C-based languages, to the structural representation; however, they are limited to a suboptimal design due to their abstraction. Finally, DSLs offer the highest level of abstraction for each domain, except they have low portability because they are limited only to a specific domain. DSLs also provide decent performance in the domain that they fit in. In Chapter 4, we have compare our framework against commercial HLS frameworks and general-purpose processors in Section 4.5 and Section 4.6.6.

Figure 2.11: Comparison between different domain of languages to design hardware accelerators

Table 2.1 gives a summary of our second comparison. We break down the existing techniques into four categories: C-based HLS, Hardware DSLs, Image Processing DSLs and HCLs. We separated design features from the type of optimizations that these techniques can target. For design
features, we compare input, output and the abstractions of each framework. Our front-end supports C/C++ with Cilk extension, and using high-level compilers, we show that our framework is capable of supporting DSL languages such as Tensorflow and Halide. We explain our frameworks front-end in-depth in Chapter 3 and evaluate our results in Section 4.5. We have separated locality from pipelining and coarse grain parallelism for optimizations and compared them against other frameworks. In Section 4.5 and Section 4.2 we show how our compiler framework can iteratively applies different class of optimizations to our baseline design. We also compare these techniques base on their level of abstractions (Higher Ops). Specifically, we compare if the hardware language has support for complex data types instead of scalar operations with support for specialized operations, more details in Section 4.2.
### Table 2.1: Comparing the hardware oriented Intermediate-Representations

<table>
<thead>
<tr>
<th>Design</th>
<th>HLS Frameworks</th>
<th>Domain-specific IRs</th>
<th>Image Processing DSL</th>
<th>Modular RTL</th>
<th>µIR Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>C program</td>
<td>Domain lang.</td>
<td>Domain Lang.</td>
<td>Chisel, SystemVerilog, VHDL</td>
<td>C++ / Cilk / Tensorflow / Halide</td>
</tr>
<tr>
<td>Output</td>
<td>Static dataflow</td>
<td>Pattern-based dataflow</td>
<td>Pipeline of kernels</td>
<td>Static Dataflow</td>
<td>Hierarchical dataflow</td>
</tr>
<tr>
<td>IR Abstraction</td>
<td>Compiler ops (e.g., add, load, store)</td>
<td>Concurrency patterns, Memory</td>
<td>Kernels, Streams</td>
<td>Primitive Logic e.g., Mux, Adder</td>
<td>Polymorphic ops, Dataflow node, Memory.</td>
</tr>
<tr>
<td>Higher Parall.</td>
<td>–</td>
<td>✓Patterns e.g., forall</td>
<td>✓Kernels e.g., stencil</td>
<td>–</td>
<td>✓General</td>
</tr>
<tr>
<td>Op Pipelining</td>
<td>✓</td>
<td>–</td>
<td>–</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Locality</td>
<td>–</td>
<td>✓(streams)</td>
<td>✓(streams)</td>
<td>–</td>
<td>✓(partitioned, shared)</td>
</tr>
<tr>
<td>Higher Ops</td>
<td>–</td>
<td>✓</td>
<td>–</td>
<td>–</td>
<td>✓(Tensor ops)</td>
</tr>
</tbody>
</table>
Chapter 3

$\mu$IR Hardware Abstraction

In this chapter, we discuss our motivations to design a new microarchitecture abstraction for designing hardware accelerators with an example. We then provide an overview of $\mu$IR, a new proposed hardware abstraction for targeting reconfigurable hardware, and details of how this abstraction meets defined criteria.

3.1 Motivation for a New Hardware Abstraction

Here we discuss the possible design decisions and optimizations that $\mu$IR would help consider when creating the microarchitecture of an accelerator. We intend to illustrate representative optimizations that informed the design of $\mu$IR’s abstractions.

The example we consider is an image processing pipeline. The pipeline typically reads the input image row-wise into a chain of kernels that operate on a sliding data window. The algorithm/program in figure illustrates the operation of a single kernel Figure 3.2 represents stencil2D implementation from Machsuite Benchmark [169]. Figure 3.1 shows the structural block-level view of the pipeline. We now discuss some specific microarchitecture optimizations for this example. The optimizations themselves are not particularly novel and have been developed previously for both microprocessors [90] and spatial architectures [205, 180].

![Figure 3.1: Example top-level microarchitecture for image processing pipeline. The lists enumerate the class of design optimizations that influenced the design of $\mu$IR.](image)

Figure 3.1: Example top-level microarchitecture for image processing pipeline. The lists enumerate the class of design optimizations that influenced the design of $\mu$IR.
```c
void stencil (TYPE orig[row_size * col_size], TYPE sol[row_size * col_size], TYPE filter[f_size]){
    int r, c, k1, k2;
    TYPE temp, mul;

    stencil_label1:for (r=0; r<row_size-2; r++) {
        stencil_label2:for (c=0; c<col_size-2; c++) {
            temp = (TYPE)0;
            stencil_label3:for (k1=0;k1<3;k1++){
                stencil_label4:for (k2=0;k2<3;k2++){
                    mul = filter[k1*3 + k2] *
                           orig[(r+k1)*col_size + c+k2];
                    temp += mul;
                }
            }
            sol[(r*col_size) + c] = temp;
        }
    }
}
```

Figure 3.2: Stencil2D example from Machsuite [169].

**Data Buffer Organization**  A computer architect will desire to optimize both the organization of the line buffers between the kernels, and the internal organization of each line buffer. Hence, we designed $\mu IR$ with multiple nested-levels of representation, where components represented at the top-level (e.g., line buffer) could itself internally be represented as another graph of components (e.g., FIFOs, registers, SRAMs). Now, consider the internal design of line buffer itself — the write side of the line buffer receives data from the image in row-major order; however within each kernel it is accessed in column major. One possible option is we can design the line buffer array to match the read-side pattern i.e., column major. This column major organization would require incremental writes to the entries.

However, the row major organization has a single port, fetches a wide amount of data every few cycles, and benefits from lower activity rate (power consumption). In Section 3.2.5 we elaborate on memory operations and memory abstractions with $\mu IR$.

**Dataflow scheduling**  Another critical design question is how to improve performance by increasing the compute density. One option is to put additional functional units in each kernel block and try to schedule multiple operations per cycle. These functional units can be controlled either by i) aggressively unrolling the kernel and scheduling the multiple operations in the dataflow [48], or ii) by fusing statistically prevalent dependent operations into a single new operation [74]. If the operations fused are temporally far apart, it effectively increases the number of pipeline registers, which increases the power consumption. In section 3.2.4, we will highlight the abstractions within $\mu IR$ that we included to permit dataflow operations to be pipelined and fused.
**SIMD Execution Units**  
SIMD increases the compute density and enables overlap of data movement and computation. $\mu$IR also permits computer architects to SIMDize and widen the width of any operation within the dataflow kernel (§ 3.2.4). However, SIMDization makes makes the earlier linebuffer organization we discussed. When the compute units are SIMDized, the buffer does not need to supply pixels every cycle since the wider compute operations only need a sub-row of pixels every few cycles. This reduced memory bandwidth, combined with intrinsic wide fetches, makes a simple single-ported scratchpad RAM just as efficient as a line buffer.

**Design Space Exploration**  
As with any hardware design, accelerator design spaces can be extremely large and cumbersome to explore. While making optimizations like loop pipelining and memory banking automatic help to improve productivity, these transformations leave the compiler with numerous choices about how to allocate resources. These decisions can accumulate large performance/area tradeoff spaces which combine exponentially with application complexity. In a fixed implementation of general matrix multiplication, there is a large design space that includes the dimensions of on-chip tiles that hold portions of the full matrices and decisions about the parallelizations of loops that iterate over tiles as well as loops that iterate within these tiles. In section 3.2.2, for each $\mu$IR’s abstract, we discuss the design tradeoffs that $\mu$IR can explore.

Our goal is to effect these microarchitecture and algorithm/program optimizations in a more generalized manner without being restricted by a specific application domain or architecture template.

### 3.2 $\mu$IR Design

$\mu$IR is a structural language for the design of accelerators implemented on reconfigurable spatial architectures, including FPGAs. The $\mu$IR graph represents the accelerator architecture as a latency-agnostic structural graph. Components in a $\mu$IR graph execute in parallel and communicate via sequence of atomic tokens passed over unbounded edges. The main benefits of $\mu$IR are i) designer can transform the $\mu$IR graph, prior to RTL generation, and this permits many microarchitectural designs to be explored. ii) components can be locally refined during performance tuning without requiring global schedule changes e.g., change the number of execution units to improve throughput, iii) it promotes modular, re-usable hardware components for accelerator design.

#### 3.2.1 $\mu$IR Abstractions

The aim of $\mu$IR is to simplify the accelerator design process, allowing domain experts to quickly develop, test, optimize, and deploy hardware accelerators, by targeting $\mu$IR from higher level language such C/C++ and Cilk, and expressing his transformations as a set of iterative optimizations passes over $\mu$IR graph. $\mu$IR represents the accelerator architecture as a hierarchical dataflow graph with shared memory, Figure 3.3 shows the primitive components of $\mu$IR. At the top level, the accelerator circuit is represented as a dataflow graph of *task modules, microarchitecture structures,*
junctions and connection required across different task modules. In the rest, we describe each of these components:

Task modules: intends to capture the concurrency within the accelerator functionality. For this propose, $\mu IR$ uses notion of task in the design. A task module is analogous to a closure in software which takes arguments and produces a value after running the internal dataflow to completion and may potentially modify memory.

Structures: encompass hardware elements that have no representation in the software. Currently $\mu IR$ includes structures representing compute operations, network, and scratchpad/cache on the accelerator. An architect is permitted to include accelerator specific structures (implemented in either Verilog or Chisel). The task’s dataflow includes connection between the operation nodes and connections to structures.

Node: can either be i) a leaf node represents operations whose logic is inlined within the task module (e.g., fixed point arithmetic). Currently, our hardware accelerators support all operations specified by the LLVM IR, including IEEE754 compatible floating point. Or ii) external (e.g., memory ops, calls to other tasks) where the nodes need to interface with a different task module or hardware structure.

Connections: represent logical 1-1 dataflow between a producer and consumer nodes. Connections in the $\mu IR$ are lowered to different hardware implementations in Chisel based on what components they interface — i) connections between leaf nodes within a task are mapped to combinatorial wires or decoupled ready-valid interfaces. ii) connections among task modules map to call/return (serial semantics) or spawn/sync (parallel semantics) interfaces, iii) external node operations use request-response connections to interface with hardware structures outside the task module (e.g., a load interfacing with the cache).

Junctions: are structures that represent 1:M, M:1, and M:N connections between nodes in a task module and shared hardware structures such as caches. Junctions essentially represent static
networks which route requests and response across multiple source and destinations. We elaborate on the hardware implementation in the next section.

There are benefits with the associated hierarchical representation: i) It captures the important kinds of coarse- and fine-grain parallel patterns from the software. ii) It enables architecture optimizations to be expressed as a set of spatial and temporal transformations of the graph at varied granularity (e.g., task pipeline vs. intra-task dataflow), iii) The hierarchy provide an implicit scope for many hardware optimizations and ensure correctness. For instance, compiler can statically schedule nodes as long as they do not cross task boundaries.

Finally, \( \mu IR \) contains both in-memory and textual representation. The in-memory structure of a \( \mu IR \) design significantly influences how easily transformations are written. As is commonly done in software compilers, a \( \mu IR \) design is internally represented with a hierarchical graph, tree, where transformations recursively walk nested elements to manipulate the graph. If non-local information is necessary, transformations first walk the tree to build a custom data structure, then walk the tree a second time to manipulate the graph.

### 3.2.2 Hardware Implementation of \( \mu IR \)

In this section, we use a Cilk [119] parallel program to illustrate how the different components in \( \mu IR \) are implemented in hardware. For each component, we describe the tradeoffs a designer has to deal with as well. In the next section, we implement various passes to transform the \( \mu IR \) and optimize the accelerator.

```c
// 32bit scalar operations
int32_t a[N], b[N], c[N];
parallel_for (int i = 0; i < N ; i++ ) {
    spawn_task{ c[i] = a[i] * b[i];}
}
 sync;
```

Figure 3.4: A simple input parallel Cilk Example to \( \mu IR \)

Figure 3.4 provides our code example. A parallel loop; the loop spawns a task for performing \(<\text{INT}_32>\) multiplications. The spawns create concurrent tasks for the statement while the loop continues onto subsequent iterations. Figure 3.5 shows in-memory representation of \( \mu IR \) and Figure 3.6 represents the top-level circuit description of input Cilk example 3.4 in \( \mu IR \). The in-memory structure of a \( \mu IR \) design significantly influences how easily transformations are written. As is commonly done in software compilers, a \( \mu IR \) design is internally represented with a hierarchical graph, tree, where transformations recursively walk nested elements to manipulate the graph. If non-local information is necessary, transformations first walk the tree to build a custom data structure, then walk the tree a second time to manipulate the graph. It includes the following components: i) Task modules (Section 3.2.3): The accelerator functionality described in the Cilk program has been decomposed into a set of tasks. The task modules are demarcated based on
where the software has indicated asynchronous concurrency is required (the spawns in the Cilk program) ii) Nodes(Section 3.2.4): Each task module is internally constructed as a dataflow graph of nodes. The dataflow within a task module enables fine-grain parallelism. iii) Memory nodes (Section 3.2.5): The task modules may internally include multiple memory operations. The task module interfaces with scratchpad or cache through an request-response (<=>) interface.

Figure 3.5: The (...) statement is a block statement that only contains multiple children statements - this node makes it easy to replace a single statement with multiple statements in a single walk of the graph.

```java
class Example(val p: Parameters) extends circuit {
   /*-------------- IO Definitions ------------*/
   val IO = {
      Mem = new Port[UInt32]()
   }
   /*-------------- Task Blocks ------------*/
   val task_for = new Main()
   val task_scalar = new Sum()
   /*-------------- Structures ------------*/
   val scal_mem = new Scratchpad[UInt32]()
   /*-------------- Connections ------------*/
   task_scalar.io.task <||> task_for.io.task
   scal_mem.io.Mem <=> task_scalar.io.Mem
   io.Mem.port <=> scal_mem.io.AXI
}
```

Figure 3.6: An example µIR design in its hierarchical representation versus textual representation from Figure 3.4 - Schematic: Blue: Task blocks, Yellow: Hardware structures, Red: Ports. <||> : task Spawn/Sync interface <=> : Req.-Resp. interface with hardware structures.
3.2.3 Task Modules

The Cilk program is partitioned into task modules by TaskGraph during the translation to the \( \mu IR \) specification. In this particular program, there are two task modules, the root task in the circuit is the actual for-loop itself, and a task for the scalar multiplication. Tasks communicate either through memory or through arguments passed through registers in the the connection interface (\(<||>\)). Inter-task Connections (\(<||>\)) establish a logical parent-child relationships between tasks. In the overall execution parents spawn children (trigger dataflow to execute concurrently) and children terminate and return values to parents at \( \text{syncs} \).

Tasks partition the accelerator hardware execution into asynchronous latency tolerant regions. Task modules form closures and encompass both the functionality and state. \( \mu IR \)'s goal to represent microarchitecture at a more functional level means it must provide a way to represent an execution block. Tasks modules are a compact representation to represent the computation as a dynamic asynchronous (latency-agnostic) pipeline along with the state (registers and memory locations) manipulated the dataflow nodes. This helps decouple task executions from each other and avoids centralized pipeline controllers in the hardware. Task modules are self-scheduled hardware dataflow blocks and start up when it has valid input arguments and runs to completion. Tasks help partition the accelerator’s execution into independent dataflows that are coordinated through asynchronous concurrency control. \( \mu IR \) supports arbitrarily nested loop and heterogeneous asynchronous parallelism in the accelerator hardware. Tasks are inspired by seminal work that merged threads and dataflow execution [51]. Recent work has leveraged a similar model for for heterogeneous CPU-GPU processing [190, 55, 66]. Most prior work in spatial accelerators [141, 218] completely unroll loops and flatten code into a single synchronous dataflow.

![Diagram of Task Modules](image)

**Figure 3.7:** \( \mu IR \) enabled design tradeoffs when implementing connections (\(<||>\)) between task-modules.

**Design Tradeoffs:** \( \mu IR \) permits the computer architect to make different tradeoffs when lowering the task-module connections to hardware (see Figure 3.7.) The first option would be to implement \(<||>\) as a decoupled interface between Main and Scala task. In this case, the initiation interval to fire a new task would be limited to the rate of slowest task, in this example would be scalar task. The second option, is placing a FIFO queue and implement the \(<||>\) between the for-task and the
scalar multiplications as a synchronouse pipeline. As a third option the designer cal also transparently add more execution units for each task to optimize the overall performance. The final option would be convert the connection to a no-op and the scalar-multiplier task could be completely inlined into the parent. This will simplify the dataflow.

### 3.2.4 Nodes and Dataflow in a Task-Module

Here we describe the data and control path within each task module. Figure 3.8 illustrates the $\mu$IR for the module and the associated nodes and hardware structures; this has been elaborated from the description in line 8—Figure 3.6 of the top-level circuit. The task module is implemented as a collection of nodes. Nodes represent the compute, control and memory operations in the module. $\mu$IR supports all the instructions within LLVM IR, including fixed-point, floating point, vector, memory and branches.

Connections between the nodes represent the dataflow. Node connections ($<>$) by default map to a decoupled ready-valid pipeline handshaking interface. This leads to two distinctive benefits. First, they permit nodes with multiple cycle latency (e.g. floating point operations), and unpredictable latency (e.g., memory operations) to be connected without the designer having to concern themselves with correctness issues related to clock cycles. Second, they localize the control logic within a module and pipeline the overall task-module. Multiple invocations can be time-multiplexed on a single task module.

**Figure 3.9: An example $\mu$IR hierarchical representation of a task module**

**Design tradeoff in connections:** We currently make it feasible for the computer architect to choose from four different connection implementations based on the module’s dataflow. i) The baseline connection implementation supports the majority of the $\mu$IR nodes need 2 input — 1 output operation, a consequence of deriving the basic operations from the LLVM IR. ii) Hardware accelerators [81] may potentially include compound domain-specific nodes that hence require multiple input - connections. We elaborate more on this optimization in Section 4.6.1. iii) The nodes may fans-out the result to many sink operations i.e., single producer - multi-consumer pattern
Figure 3.8: Auto-generated \(\mu IR\) for Tensor2D task module. Expansion of Line 8 — Figure 3.6.
and require connections with multiple outputs and iv) Connections may be required to implement control flow and define ordering between operations. In such cases a 1-bit token serializes the two operation nodes: \( \text{Pred} \rightarrow \text{Op} \rightarrow \text{Succ} \).

Design tradeoff in intrinsic operations: Types and intrinsic introduce another important design question. How do we implement the Tensor2D multiplication in the dataflow, for instances. The \( \mu IR \) decouples hardware intrinsics from the microarchitecture transformations and introduces them as first-class object. \( \mu IR \) only specifies the “what” and not the “how”? This separation of the functionality of the intrinsic nodes from the pipeline is an important innovation in \( \mu IR \). The pipeline interacts with the intrinsic through the latency agnostic ready-valid handshaking (Figure 3.10). The hardware designer has to concern themselves only with the functionality concentrated in the white boxes. \( \mu IR \) implicitly deals with the connections and handshaking which are also typed and specify the logical data movement. During the lowering to hardware we optimize the shape of the data movement over physical wires (e.g., transferring the 4 words for the tensor over 4 cycles). A hardware designer has full freedom to specify the implementation of the multiplication for the Tensor2D4x4 and we will include these when during the final lowering to Chisel. In Section 4.6.3 We evaluate the benefits of intrinsic operations.

To show the intrinsic operations, we modified the first Cilk example and make it conditional with mixed type of UInt32 and Tensor2D, Figure 3.11. In the modified example there is a
32bit scalar operations

```c
uint32_t a[N/2], b[N/2], c[N/2];
```

// Tensor2D operations

```c
Tensor2D a2D[N/2], b2D[N/2], c2D[N/2];
```

```c
parallel_for (int i = 0; i < N; i++) {
    if (i % 2 == 0)
        spawn_task { c[i/2] = a[i/2] * b[i/2];}
    else
        spawn_task { c2D[i/2] = a2D[i/2] * b2D[i/2];}
}
```

```c
sync;
```

Figure 3.11: A mixed type input parallel Cilk Example to µIR

parallel loop; in the odd iterations, the loop spawns a task for performing 2D tensor (2×2 tiles) multiplications and in the even iterations it performs an integer multiplication. The spawns create concurrent tasks for the statement while the loop continues onto subsequent iterations. Figure 3.13 shows the µIR textual representation of the this example.

Figure 3.12: µIR graph for a mixed type input specification

### 3.2.5 Memory: Load/Store nodes, Junctions and Scratchpads

A key design issue in µIR is how to connect the load and store nodes with memory structures such as cache and scratchpads. Programs typically view memory as a two-level hierarchy (register or memory) and the compiler IR in SSA form treats all the registers as a logically centralized resource. It is necessary to map the shared memory in the program to physical hardware structures, both temporally and spatially [159, 208]. µIR also include the notion of a coherent L1 cache through the AXI protocol to handle cases when the load or store cannot be mapped to a scratchpad.

We first describe the flow of a load or store operation. First, load and store operations in the task-module’s dataflow need to interface with hardware structures outside the task-module definition and hence include request-response interfaces. Memory operation nodes require extensive
Figure 3.13: \(\mu IR\) design for a mixed type input specification

logic for handling unaligned accesses and masking logic due to handle half words and bytes. This arises due to the fact that the memory interface to cache or scratchpad may be a different width than the compute operations. Intrinsic Tensor2D load and store operations require more complex logic since they read multiple sequential words from the memory interface. We chose to group this common logic for multiple loads and store operations into the data box to minimize resource requirements.

Since multiple operations within the task module may need to interface with a databox representing an N:1 connection. For this the \(\mu IR\) includes junctions for routing requests and responses between the loads/stores and the databox. Junctions in the \(\mu IR\) are generic (1:N, N:1, and M:N) connections. One possible implementation of junctions is a static network. We elaborate more on the notion of the junction and the optimizations it permits in section 4.6.4.

The databox is a table of staging buffers. Each buffer entry: 1) converts the type (e.g., Tensor2D) to word granularity accesses and issues them to the cache/scratchpad 2) coalesces the responses required from the cache to complete a request (e.g., 4 word responses for a Tensor2D), and 3) shifts
and masks the data to handle alignments and sub-word operations. Each staging buffer fetches words in parallel, and multiple staging buffers operate concurrently, which improves MLP.

**Design tradeoff:** The connections between the load/store nodes and storage permit multiple optimizations. First, we expose the junctions (i.e. routing), the load/store nodes and the physical scratchpad in the $\mu IR$. This permits us to spatially partition the storage. For instance, in the $\mu IR$ code for the accelerator, Figure 3.6, we have allocated separate scratchpads for each task-module; we elaborate more on this optimization in Section 4.3. Second, we have parameterized the junction implementations in $\mu IR$ so that the designer can control the physical network should the junction be lowered into; we explore this optimization in Section 4.6.5. Finally, $\mu IR$ also exposes the notion of a L1 cache shared between different task modules. The $\mu IR$ permits transformations that can manipulate the organization of these banks; see Section 4.3. Finally, the data boxes are extensively parameterized to control the number of parallel memory operations in the accelerator.

![Figure 3.14: Connections between memory nodes in the task module and transfers to/from a cache or scratchpad.](image)

**3.2.6 Execution and Memory Model:**

The execution flows need to be considered at two levels, whole-accelerator level, and the local dataflow within each task. Figure 3.15 provides an exploded view. $\mu IR$ represents the whole accelerator as a graph of concurrently running dynamic task blocks. In this example, only the child tasks (`task_scalar` and `task_tensor`) perform actual work, and the parent `task_for` is used only for creation and coordination of the workers. `task_for` creates N/2 instances of `task_scalar` and N/2 instances of the `task_tensor`. $\mu IR$ models each task block as having a local task queue that stores ready and pending tasks. The task block is free to process the
ready tasks in any order. In the overall execution, parents spawn children to run concurrently and children terminate and return values to parents at sync. Tasks communicate either through memory or through registers in the connection.

![Figure 3.15: Execution Model of \( \mu IR \) at all levels. We have exploded each component to show the internal execution flow.](image)

The memory model is a partitioned global address space. A hardware designer could introduce any number of memory spaces in the \( \mu IR \) graph to interface with the task blocks. These address-spaces are incoherent with each other, but coherent with DRAM and the CPU through AXI. \( \mu IR \) includes two abstractions for representing local address spaces, scratchpads and caches. Caches are implicitly managed by a hardware controller, while scratchpads are managed with DMA.

The execution within each task block is modeled as a pipelined latency-agnostic dataflow. Individual nodes in the dataflow handshake with each other through a ready/valid control protocol. The control can apply back pressure to handle stalls (like control signals in a microprocessor pipeline) and permits arbitrary insertion and removal of buffering between nodes. Every node in the dataflow operation is considered to occur completely asynchronously. The pipelined dataflow enables multiple concurrent invocations to be outstanding at the same time on dataflow and improves throughput. Unlike a tagged dataflow architecture [9], concurrent invocations complete in-order of invocation. This leads to a simpler RTL implementation.

### 3.2.7 Control Flow and Loops

\( \mu IR \) supports arbitrary control-flow from input algorithm. For forward branches, \( \mu IR \) implements dataflow predication i.e., trigger the node in dataflow for control, but bypass the actual logic and poison the output. With backward branches and loops there are three challenges i) Live ins: We extract each loop into the a task block, buffer the live ins from other parts of the circuit, and feed it into the dataflow. ii) Loop carried dependencies: We introduce buffering, latency-insensitive edges, and registers to break the combinatorial loop when implementing backward edges. This is similar to Arvind and Nikhil’s seminal work on dataflow machines [9]. iii) Loop nests: In \( \mu IR \) each nested loop is disassociated from its outer loop, and is encapsulated within a task block. Intuitively,
each nested loop is enclosed in a separate function that can run in pipeline parallel fashion with
the parent. To the outer loop, the nested loop appears as a variable latency non-deterministic
operation with request-response interface. Finally, recursion is handled similar to loops. We use
LLVM to convert recursion to a iterative pattern prior to translating the program into an \( \mu IR \) graph
(see recursive mergesort and fib in § 4.5).

3.2.8 \( \mu IR \) Compiler: Transforming programs to \( \mu IR \) graph

**Algorithm 1:** Generating \( \mu IR \) graph from Compiler IR.

```plaintext
1 function Stage1_\mu IR_Taskgraph:
2 \mu IR_TaskGEdges = Map{}
3 \mu IR_TaskGNodes = Map{}
4 TaskQueue = {main()}
5 while TaskQueue != φ :
6   Current = TaskQueue.pop()
7   \mu IR_TaskGNodes[Current] = List{}
8   for bb in Current.BasicBlocks :
9      if StaticSchedule(bb) :
10         \mu IR_TaskGNodes[Current].add(bb)
11      else:
12         Child = new Task(bb)
13         \mu IR_TaskGEdges[Current].add({Current, Child})
14         TaskQueue.push(Child)
15   end
16 end

19 function Stage2_Schedule(Task node):
20 ComputeNodes = {}; DataflowEdges = {}; ControlNodes = {}; ControlEdges = {};
21 MemoryNodes = {}
22 for bb in Task.BasicBlocks :
23   for node in bb :
24      if node is Compute :
25         ComputeNode.add(node)
26         DataflowEdges.add({node, node.dependents})
27     elif node is Control :
28         ControlNodes.add(node)
29         ControlEdges.add({node, node.target})
30     elif node is Memory :
31         MemoryNodes.add(node)
32         GlobalMemory.connect(node)
33   end
34 end
```

47
The generation of $\mu IR$ graph from compiler IR proceeds in three stages. In **Stage 1** we transform the compiler IR to a $\mu IR$ task graph. A task block in $\mu IR$ represents a set of basic blocks that needs to be asynchronously scheduled in hardware, either because the amount of work is statically unknown or it may be profitable to dynamically schedule.

Algorithm 1 shows the pseudocode of step 1. $\mu IR$ _TaskGEdges_ and $\mu IR$ _TaskGNodes_ collectively represent the task-level microarchitecture graph. We iterate over LLVM program-dependence-graph in breadth-first fashion and aggregate basic blocks(line 9: _if_ block). Basic blocks that terminate dynamically schedulable regions e.g., loops, function calls, concurrent tasks in Cilk, Tensor-flow intrinsics, start a new task and restart the aggregation process (line 11: _else_ block). Our compiler pass then extracts the task’s basic blocks from the surrounding program-graph and creates a closure that captures the scope i.e., live-ins, live-outs and control dependencies. This enables the task region to be invoked through a timing-agnostic asynchronous interface. The asynchronous interface lowers to a hardware issue-queue (during the Chisel elaboration stage). Based on program flow the hardware queue, at run time, determines if an execution tile has to assigned for the task region, and if so which execution tile (see Section 3.2.2:Execution Model). The design is fully parameterized and a user can vary the number of execution tiles for each task region.

In **Stage 2**: we create the datapath for each task-block. $\mu IR$ _TaskGNodes_ is a dictionary that specifies for each node in the task graph, the corresponding region of basic blocks in LLVM. Algorithm 1 lists the pseudocode, Stage2 _Schedule_. In this stage, the body of each task block contains only forward branches, loops are treated as self-scheduling asynchronous tasks. We lower the set of basic blocks to a hyperblock and embed it as sub-graph within the node in a task graph. The conversion to $\mu IR$ graph is a literal translation of the data flow graph. In the baseline, every compiler op lowers to a decoupled node, every node internally implements the functional unit and edges are pipelined connections between the function units. Subsequently, we connect memory operations to a global memory unit at the top-level of the graph. Section 3.2.7 and 3.2.5 provides more details on control-flow and memory implementation of stage 2.
Chapter 4

The \( \mu \)IR Compiler

In the previous chapter, we described \( \mu \)IR hardware abstraction. In this chapter, we explain the lowering process and the details of the \( \mu \)IR compiler. We call \( \mu \)IR compiler as \( \mu \)opt. The goal of \( \mu \)opt is to provide a library of passes that are used to simplify, transform, or specialize accelerator architecture specified in \( \mu \)IR.

4.1 \( \mu \)opt Pass Manager

The \( \mu \)opt Pass Framework is an important part of the \( \mu \)IR system, because \( \mu \)opt passes are where microarchitectural specific transformations happen. Passes perform the transformations and optimizations that make up the \( \mu \)opt they build the analysis results that are used by these transformations, and they are a structuring technique for \( \mu \)opt. All \( \mu \)opt passes are subclasses of the Pass class, which implement functionality by overriding virtual methods inherited from Pass. Depending on how designer’s pass works, he should inherit from the AccelPass, LoopPass or OpPass, which gives the system more information about what the pass does, and how it can be combined with other passes.

An optimization pass is an iterative transformation of the microarchitecture graph before lowering \( \mu \)IR graph to Chisel implementation. The input to \( \mu \)opt is an accelerator design specified in \( \mu \)IR graph with a set of associated optimization passes. The designer express these optimization passes in the form of graph transformations using \( \mu \)opt pass framework, and the output is a functionally equivalent, but optimized microarchitecture in \( \mu \)IR. The final pass in \( \mu \)opt lowers the \( \mu \)IR graph to the Chisel implementation.

One of the benefits of \( \mu \)opt pass framework is that for analysis purposes \( \mu \)opt can leverage the analysis passes from higher level compiler. For instance, in the case of memory transformations, \( \mu \)opt needs information about relations between memory ops and how they depend on each other. In this example to determine whether or not two pointers ever can point to the same object in memory, \( \mu \)opt inherent and reuse pointer analyses [123, 192, 147] from software compiler and then use them to apply microarchitectural specific optimizations on \( \mu \)IR abstraction. In below we have summarized the other benefits of \( \mu \)opt pass based approach:
1. Accelerator designers can manipulate the architecture without requiring knowledge of HDLs such as Chisel.

2. It helps computer-architects decompose their ideas into a set of structured graph transformations.

3. It algorithmically represents architecture optimizations and automates the repetitive task of porting an optimization to different application accelerators.

### 4.2 \( \mu IR \) Compiler — (\( \mu OPT \))

The crucial benefit of \( \mu IR \) is the ability to generate multiple microarchitectures with different design tradeoffs for the same software functionality. In this section, we demonstrate five microarchitecture optimizations from the \( \mu opt \) framework that successively expose opportunity for each other — Figure 4.1 shows the order in which these passes optimize the design.

**Pass 1 — Task Block Queuing:** The hardware designer has the ability to modify the queuing and asynchrony between tasks in the whole-accelerator circuit. This permits the individual task blocks to proceed at different rates and enables subsequent optimization passes. This pass is achieved by controlling how inter-task connections (\(< || >\)) in \( \mu IR \) map to RTL. One choice would be to introduce FIFO queues on the interface between the for-loop task block and tensor task block.

```cpp
parallel_for (i = 0; i<N; i++) {
  if(i % 2 == 0)
    spawn_task{
      c[i/2]=a[i/2]*b[i/2];
    }
  else
    spawn_task{
      c2D[i/2]=a2D[i/2]*b2D[i/2];
    }
}
```

Figure 4.1: Overview of \( \mu opt \) transformation passes. Each pass applies a specific transformation. Passes run in the order 1→5.
only, while leaving the low latency scalar block coupled. The tensor block has higher latency and we require more decoupling to ensure the for-loop block can run at a higher rate.

**Pass 2 — Execution Tiling:** The higher latency of the tensor block could potentially lead to longer queuing delays. Hence, we need to increase the throughput of the tensor block by replicating it by N (a tunable parameter). The key challenge that \( \mu IR \) deals with during the RTL generation is creating buses and crossbar to route tasks to different execution units. This change can be achieved locally without affecting the other parts of the accelerator circuit.

**Pass 3 — Localized Type-specific scratchpads:** The shared scratchpad compromise the execution of both the tensor and scalar blocks, due to contention. The tensor block (multiplying 2\times2 tiles) reads 8 words and writes back 4 words per cycle, while the scalar reads 2 words and writes back 1. To solve this, \( \mu IR \) creates local per-task scratchpads. The scratchpads also expose their type to \( \mu IR \) and during RTL generation, we optimize the shape of the data movement over physical wires and change the data organization. Another option would be introducing a separate writeback buffer for writing out the data.

**Pass 4 — Scratchpad Banking:** To deal with the higher throughput introduced by multiple execution blocks in Pass 2, we also have to increase the throughput of the tensor memory system. For the tensor scratchpad we need to supply 2 tiles (each four words) in a cycle. The options are to either use four dual-port SRAM blocks and stripe the words across them or use a dual-ported SRAM with wide eight-word reads. The scalar scratchpad will use two port SRAM.

**Pass 5: Op Fusion and Pipelining** Finally, the for-loop block is on the critical path of the entire accelerator circuit. The dataflow itself is entirely serial and the pipeline has five stages: Buffer \( \rightarrow \phi \rightarrow i++ \rightarrow i==0 \rightarrow \text{Cond-Branch} \). This implies that each iteration takes at least five cycles, and limits the throughput of the scalar task (only 2 cycles for execution). To re-time the pipeline to two stages, the pass fuses all of the operations into a single node. \( \mu IR \) enables re-time the pipeline with having to modify the RTL.

### 4.2.1 \( \mu opt \) Transformations

In algorithm 2, we illustrate a transformation pass that uses iterators for both the whole-accelerator and local dataflow of each task block. Algorithm 2 shows the pseudocode for this optimization. The optimization codifies pass 3 and 4 in Figure 4.1 — the goal of this transformation is to partition the address space and direct un-related loads to different scratchpad banks. The optimization itself requires two sub-passes i) *Analysis*, which identifies the memory space to which each memory operation belongs. For this purpose we used LLVM pointer analysis passes [123] to extract memory information from input program ii) *Transformation*, which creates separate scratchpads in the microarchitecture graph for each memory space. In the analysis pass, first we invoke \texttt{LLVMPointsto} to identify independent memory spaces. In the transformation pass we show using the information from analysis pass and using \( \mu IR \) abstraction how we create independent scratchpads. Later, using \( \mu IR \) abstraction, we tune scratchpad parameters such as number of ports.
Algorithm 2: Scratchpad Banking

// Temporary map from address space to list of memory ops. ID 0: Global. 1–N: Different address spaces

global: Mem_groups = Map[ID, List(MemOps)]

Analysis:
1 def getMemoryAccess(Circuit):
2     foreach task in Circuit do
3         foreach mem in task.getMemops() do
4             space_id = LLVMPointsto(mem)
5             Mem_groups[space_id].insert(mem)

Transformation:
7 def scratchpadBanking(Circuit):
8     foreach (ID, items) in Mem_groups do
9         Param = getMemParams(items)
10        Mem = new RAM(Param)
11     foreach op in items do
12         op.connect(Mem)

and number of banks. Another advantage of using $\mu$opt is the ability of applying repetitive transformations on a same graph.

4.3 $\mu$opt In Practice

To show $\mu$opt optimization passes in practice, we use dot product as an example and show how $\mu$opt successively applies transformations to a baseline design (Optimization 1 to 4). We start the design by describing the baseline architecture that $\mu$IR generates. Next, we describe each transformations in detail and how it opens up new opportunity for another optimization.

Dot product is a basic linear algebra kernel, defined as the sum of the element-wise products between two vectors of data. Figure 4.3 shows the most basic implementation of vector dot product. In this implementation at each cycle we load two values from DRAM, multiply them together and accumulate them with the current value. Later, we store the result in DRAM and fetch two new data. In the rest, we use this base implementation to demonstrate some of the microarchitectural transformation that helps to build more efficient design.

Optimization 1 - Tiling: In most applications, there are some properties of the algorithm that are not known at compile time. Data structures may be dynamically sized and larger than available memories on the FPGAs. Hence, we want to include as much flexibility for a given algorithm as possible. The first way to incorporate flexibility is through tiling. This means we can allow data structures in the host to be dynamic while keeping properties in the accelerator fixed.
```c
int innerProduct(int vect_A[], int vect_B[])
{
    int product = 0;
    // Loop for calculate dot product
    for (int i = 0; i < n; i++)
        product = product + vect_A[i] * vect_B[i];
    return product;
}
```

Figure 4.2: Dot product implementation in C

![Figure 4.3: Baseline implementation of Dot product in \( \mu IR \)](image)

First, we fix the tile size in the accelerator. For example in this case, the tile size is equal to \( N \). In the accelerator there needs to be a control that will loop over all of the tiles. Then, within a tile, we will run the multiply, accumulation. As a result, in local memory we can only fetch one tile at a time.

![Figure 4.4: Dot product with dual buffer local memory](image)

**Optimization 2 - Pipelining:** In the case of dot product, there are two types of parallelism. The coarse grain parallelism, outer controller, that loads data from DRAM to local buffers and fine-grain parallelism, inner controller, that multiplies and accumulates the operands. In the outer controller, pipelining means that the stages will execute in this manner: Both loads will execute the for in parallel tile = 0 (since they have no data dependencies). When they are both done, the
inner controller will execute with tile = 0 while the loads will then execute with tile = 64, and so on. In software terminology, this particular case of coarse-grain pipelining is known as **prefetching**, but the pipeline concept indicates overlapping any operations with any other operations with any pipeline depth. The challenge with making the outer controller pipeline is that we need to insert N-buffered memories. In this example, the both SRAMs, the blue and the red, are double-buffered. Therefore, while in the second stage, the multiply-accumulate operation is happening, the data for the next tile is loading to the buffer in the first stage.

For the inner controller, this means that the hardware will issue a new value for i at each cycle. Even though the multiply operation may have a latency of 6 cycles, we could expect to get 6 new results in 6 cycles once the pipeline is running at steady-state. The Figure 4.5 demonstrates the execution of these transformation on the baseline architecture.

![Figure 4.5: In this moment, the Tile 3 is been loaded from DRAM to the local buffer in Stage 2, while in Stage 1 the value from Tile 2 is accumulating.](image)

**Optimization 3 - Parallelization:** In the next transformations, we show the effect of **unrolling** to increase the parallelism of the basic implementation. When we use parallelization in the microarchitectural context, we should think of the associated iterator (in this case tile and i) as being vectorized by the parallelization factor. Rather than exhibiting one value at a time, the iterator will exhibit multiple consecutive values simultaneously. This means that on the first iteration of the outer loop, we will process tiles = 0 and tiles = 64 in parallel.

To increase the parallelism for dot product, we unroll the outer loop by 2, which means \( \mu_{opt} \) will duplicate whatever control structures are inside of it to provide the hardware to run in parallel. Note that for the SRAMs, We need unique SRAMs for both lanes of this controller, so we want them to get unrolled with everything else. Figure 4.6 shows the unrolled version of previous design.

The inner loop has a parallelization of 4, which means we will issue 4 elements into the map function at once (i.e.- \( i = 0, i = 1, i = 2, \) and \( i = 3 \)), and a summing tree of depth \( \log_2(4) = 2 \) will be generated to accumulate all of the lanes with the shortest latency possible. Figure 4.7 shows the hardware with the inner parallelization.
4.4 $\mu$IR Compilation Flow

To summarize, we overview $\mu$IR framework and explain how a designer can start from an input a program, get $\mu$IR baseline design, optimize the design and then from an optimized design he can get Chisel design. We broke down $\mu$IR framework into 5 steps, Figure 4.8 that we describe below.

**Input program**

In $\mu$IR framework, the accelerator design process starts by expressing the design in the form of a C program. The $\mu$IR framework accepts the behavioral description of the algorithm in C language...
Software compiler

In the next step, when the designer expresses his design in the form of high-level language like C, we pass the input program to the software compiler and represent the input program as target independent IR. In \( \mu \text{IR} \) framework, we used LLVM as the software compiler. In this stage, we aggressively optimize and transform the input program to exploit more parallelism.

Task Extraction

In task extraction step, we used Tapir [177] as front end software compiler to automatically decompose the input programs to multiple tasks. Tapir provides the front-end language bindings that translates Cilk/OpenMP programs to the LLVM IR. Tapir adds three instructions to the LLVM
IR, detach (or spawn), reattach and sync. Spawn and reattach together delineate a task. A detach instruction terminates the block that contains it, spawns a new task starting from the target block and continues execution in parallel from the continuation. The reattach terminates the task spawned by a preceding detach instruction. Since Tapir assumes a generic threadpool execution model, it leaves the markers in place in the original PDG (Program Dependency Graph). We leverage the markers to perform reachability analysis and extract an explicit task graph, which is the architecture blueprint for our parallel accelerator. Nesting loops and irregular flows are analyzed in this stage and in the resulting task graph all task relations and the basic blocks that constitute a task are explicitly specified. We perform live variable analysis to extract and create the requisite arguments that need to be passed between tasks; these are used to parameterize the spawn port and args RAM for each task unit.

**μIR Baseline**

After applying standard software optimization to the input program, running O3 optimizations at the software compiler, we form μIR graph. Our lowering process uses μIR abstraction we described in Chapter 3.2, and lower the optimized LLVM IR to μIR graph. Along with the lowering process, we pass LLVM pointer analysis information to μIR graph since in the next step μIR compiler, μopt uses this information to specifically optimized the μIR baseline design to exploit available memory bandwidth on the chip.

**μoptPass**

As we described in this chapter, μopt Pass Framework is an essential piece in μIR framework. μopt passes are where the designer applies microarchitectural specific transformations to the μIR baseline graph after lowering software IR to μIR graph. Passes perform the transformations and optimizations that make up the μopt they build the analysis results that are used by these transformations, and they are a structuring technique for μopt. All μopt passes are subclasses of the Pass class, which implement functionality by overriding virtual methods inherited from Pass. Depending on how the pass works, the designer should inherit from the AccelPass, LoopPass or OpPass, which gives the system more information about what the pass does, and how it can be combined with other passes.

**μopt Backend Generation**

At this stage, after optimizing the baseline design using μopt we lower the optimized μIR graph to a form of dynamic dataflow implemented in Chisel language. This graph is the structural implementation of μIR that can be synthesized and map to FPGA devices. At this level, all the operations and edges in μIR graph become an explicit implementation in Chisel. Moreover, all the graph nodes properties become configuration parameters in the hardware design.
4.5 Quantitatively Evaluating $\mu$opt and $\mu$IR

Methodology: The $\mu$IR compiler provides translations from applications or IR in the $\mu$IR abstraction to synthesizable hardware descriptions in Chisel RTL. The generated Chisel RTL is then compiled by the third party Chisel compiler and used to generate Verilog, which can in turn be synthesized on an FPGA target using vendor tools. $\mu$IR also generates host code in C++ for FPGA runtime administration and data management.

The primary purpose of $\mu$IR and $\mu$opt is to provide a fertile playground for computer architects to see their ideas reflected in RTL of accelerators. Here we try to answer the following: i) What is the quality of the baseline accelerators (no optimizations). Our goal is to establish a performance bar to isolate the benefits of the individual optimizations, ii) How do the baseline accelerator architectures compare to those generated by commercial HLS tools. iii) How do different optimization passes improve the performance. We consider each optimization individually, § 4.6.1—§ 4.6.4, and together§ 4.6.5

4.5.1 What is the quality of baseline accelerators?

Observation 1: $\mu$IR-generated accelerators can attain high frequency. 200-500 Mhz on FPGA. 1.6—2.5Ghz on ASIC.

Observation 2: $\mu$IR-generated accelerators can achieve low power. 500-1200 mW on FPGA, 20-150 mW on ASIC.

Here, we try to establish the performance and power characteristics of the baseline accelerator, which we further optimize in § 4.6. We evaluate on two backends, an FPGA Intel Arria 10 and ASICs synthesized using the Synopsys Design Compiler (UMC 28nm technology). Table 4.1 summarizes the results. All our workloads were unmodified.

Overall, even pre-optimization, $\mu$IR produces competitive accelerators. The floating point workloads (benchmarks with $F$) attain frequency between 350-400Mhz. For the FP macros, during RTL generation $\mu$IR plugs in the IP cores and for ASIC we use an in-house version of Berkley hardfloat. Here we use single precision throughout. For FP workloads, ASICs improve the power 20—50x vs. the FPGAs.

Compute intensive workloads use plenty of registers and FPGA ALMs and typically consume 1—1.2W in power on the Arria 10 e.g., COVAR, CONV, 2MM 3MM, SOFTM, FFT. In workloads with higher compute density and simpler operations, SAXPY, CONV, SOFTM, RELU, 2MM[T] the ASICs improved clock frequency 4—6×, compared to the FPGA.

Finally, the Cilk accelerators, achieved a lower target frequency, between 200-300Mhz on the FPGA (compared to non-Cilk workloads). This is primarily a result of the queueing and buffering logic required to manage asynchronous task blocks being on the critical path.
4.5.2 $\mu$IR vs High-level synthesis

**Observation 1** Starting from the same program specification, $\mu$IR-generated accelerators attain 20% higher Mhz compared to HLS toolchains, due to dataflow execution.

**Observation 2** Many workloads exploit higher clock to achieve overall better performance (10—30%). On some workloads, HLS can generate better streaming buffers and achieves 10% better performance.

An apples-to-apples comparison with prior HLS toolflows is not feasible since i) HLS primarily targets loop parallelism, and ii) they rely on streaming memory behavior (otherwise memory accesses are serialized). To ensure a fair comparison, we manually modified programs to be HLS compatible. We ported all the Machsuite and Tensorflow to HLS.\footnote{We had to rely on two toolchains, Legup\cite{26} and Intel HLS, as neither one supported all the workloads.} Our set-up, i) we switched on all the compiler optimizations, since HLS relies on them, ii) we disable all $\mu$opt optimizations and tool-specific optimizations\cite{104, 37, 15, 222}, iii) we ensure that for RAMs and FP we use vendor-specific IP.

Figure 4.9 plots the baseline $\mu$IR’s (no optimizations) performance normalized to HLS. The main reason for the performance improvement under $\mu$IR is the fundamentally different execution models and architecture generated by $\mu$IR. HLS relies on a state machine to coordinate execution. $\mu$IR however adopts a decentralized dataflow-based execution model. This leads to deeper operation pipelines and hence 20% higher frequency than HLS. In workloads like fft, gemm, 2mm and 3mm, which they have nested loops the $\mu$IR’s pipeline depth is 30 (2MM) — 40(GEMM) stages; even workloads with few loops such as Dense8 have 15 stages. GEMM, Covar, 2MM, and 3MM: $\mu$IR exploits better operation parallelism as HLS serialize the nested loop executions — overall performance improvement is 20–30% (execution cycle improvement and clock frequency improvement contribute equally). In Conv, $\mu$IR’s dataflow achieves nearly 80% improvement in target clock (overall 60% improvement in execution time). In FFT and Dense, HLS generates streaming buffers and improves the memory system (we were unable to turn it off), $\mu$IR relies on a less efficient cache.
<table>
<thead>
<tr>
<th>Bench</th>
<th>MHZ</th>
<th>mW</th>
<th>ALMs</th>
<th>Reg.</th>
<th>DSP</th>
<th>$nm^2$</th>
<th>mW</th>
<th>Ghz</th>
</tr>
</thead>
<tbody>
<tr>
<td>GEMM$^F$</td>
<td>422</td>
<td>946</td>
<td>4480</td>
<td>7936</td>
<td>1</td>
<td>42.9</td>
<td>28</td>
<td>1.66</td>
</tr>
<tr>
<td>COVAR.$^F$</td>
<td>354</td>
<td>1496</td>
<td>11448</td>
<td>20415</td>
<td>4</td>
<td>73.6</td>
<td>150</td>
<td>1.66</td>
</tr>
<tr>
<td>FFT$^F$</td>
<td>425</td>
<td>1109</td>
<td>6548</td>
<td>12347</td>
<td>4</td>
<td>91.3</td>
<td>47</td>
<td>1.66</td>
</tr>
<tr>
<td>SPMV$^F$</td>
<td>388</td>
<td>868</td>
<td>3386</td>
<td>6048</td>
<td>2</td>
<td>36.9</td>
<td>24</td>
<td>1.66</td>
</tr>
<tr>
<td>2MM$^F$</td>
<td>385</td>
<td>1080</td>
<td>6126</td>
<td>12198</td>
<td>4</td>
<td>84.5</td>
<td>24</td>
<td>1.66</td>
</tr>
<tr>
<td>3MM$^F$</td>
<td>377</td>
<td>1202</td>
<td>8150</td>
<td>15216</td>
<td>1</td>
<td>97.38</td>
<td>52</td>
<td>1.66</td>
</tr>
</tbody>
</table>

Cilk Benchmarks

<table>
<thead>
<tr>
<th>Bench</th>
<th>MHZ</th>
<th>mW</th>
<th>ALMs</th>
<th>Reg.</th>
<th>DSP</th>
<th>$nm^2$</th>
<th>mW</th>
<th>Ghz</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIB</td>
<td>307</td>
<td>751</td>
<td>1818</td>
<td>2614</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>M-SORT</td>
<td>314</td>
<td>959</td>
<td>4943</td>
<td>6767</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>SAXPY</td>
<td>214</td>
<td>609</td>
<td>1667</td>
<td>2252</td>
<td>2</td>
<td>17.4</td>
<td>20</td>
<td>2.5</td>
</tr>
<tr>
<td>STENCIL</td>
<td>207</td>
<td>812</td>
<td>4123</td>
<td>6134</td>
<td>2</td>
<td>82.6</td>
<td>94</td>
<td>2.5</td>
</tr>
<tr>
<td>IMG. Scale</td>
<td>206</td>
<td>705</td>
<td>2484</td>
<td>3582</td>
<td>2</td>
<td>83.4</td>
<td>96</td>
<td>2.5</td>
</tr>
</tbody>
</table>

Tensorflow Benchmarks [138]

<table>
<thead>
<tr>
<th>Bench</th>
<th>MHZ</th>
<th>mW</th>
<th>ALMs</th>
<th>Reg.</th>
<th>DSP</th>
<th>$nm^2$</th>
<th>mW</th>
<th>Ghz</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONV.</td>
<td>363</td>
<td>1071</td>
<td>6354</td>
<td>10698</td>
<td>1</td>
<td>75.3</td>
<td>60</td>
<td>2.5</td>
</tr>
<tr>
<td>DENSE8$^F$</td>
<td>362</td>
<td>914</td>
<td>4025</td>
<td>7218</td>
<td>1</td>
<td>45.8</td>
<td>37</td>
<td>2</td>
</tr>
<tr>
<td>DENSE16$^F$</td>
<td>381</td>
<td>923</td>
<td>4070</td>
<td>7222</td>
<td>1</td>
<td>46.5</td>
<td>37</td>
<td>2</td>
</tr>
<tr>
<td>SOFTM8$^F$</td>
<td>375</td>
<td>1171</td>
<td>7787</td>
<td>13232</td>
<td>0</td>
<td>60.6</td>
<td>54</td>
<td>2</td>
</tr>
<tr>
<td>SOFTM16$^F$</td>
<td>347</td>
<td>1171</td>
<td>7842</td>
<td>10698</td>
<td>0</td>
<td>61.2</td>
<td>53</td>
<td>2</td>
</tr>
</tbody>
</table>

Halide [168]

<table>
<thead>
<tr>
<th>Bench</th>
<th>MHZ</th>
<th>mW</th>
<th>ALMs</th>
<th>Reg.</th>
<th>DSP</th>
<th>$nm^2$</th>
<th>mW</th>
<th>Ghz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conv3X3</td>
<td>207.64</td>
<td>864</td>
<td>3629</td>
<td>4535</td>
<td>8</td>
<td>97.38</td>
<td>52</td>
<td>2</td>
</tr>
<tr>
<td>StridedConv</td>
<td>201.78</td>
<td>1223</td>
<td>8522</td>
<td>10837</td>
<td>28</td>
<td>86.3</td>
<td>52</td>
<td>2</td>
</tr>
<tr>
<td>UpSample</td>
<td>207.17</td>
<td>851</td>
<td>3458</td>
<td>4398</td>
<td>4</td>
<td>77.8</td>
<td>59</td>
<td>2</td>
</tr>
<tr>
<td>DownSample</td>
<td>198.93</td>
<td>832.66</td>
<td>3343</td>
<td>4078</td>
<td>4</td>
<td>84.2</td>
<td>67</td>
<td>1.66</td>
</tr>
<tr>
<td>ConvlLayer</td>
<td>205.97</td>
<td>1020</td>
<td>6238</td>
<td>7828</td>
<td>10</td>
<td>95.5</td>
<td>66</td>
<td>2</td>
</tr>
<tr>
<td>Harris</td>
<td>203.51</td>
<td>1020</td>
<td>6238</td>
<td>7828</td>
<td>10</td>
<td>95.5</td>
<td>66</td>
<td>2</td>
</tr>
</tbody>
</table>

In-house

<table>
<thead>
<tr>
<th>Bench</th>
<th>MHZ</th>
<th>mW</th>
<th>ALMs</th>
<th>Reg.</th>
<th>DSP</th>
<th>$nm^2$</th>
<th>mW</th>
<th>Ghz</th>
</tr>
</thead>
<tbody>
<tr>
<td>RELU[T]</td>
<td>460</td>
<td>547</td>
<td>840</td>
<td>1169</td>
<td>14</td>
<td>15.9</td>
<td>19</td>
<td>2.5</td>
</tr>
<tr>
<td>2MM[T]</td>
<td>496</td>
<td>568</td>
<td>1073</td>
<td>1566</td>
<td>14</td>
<td>91.1</td>
<td>27</td>
<td>2.5</td>
</tr>
<tr>
<td>CONV[T]</td>
<td>397</td>
<td>618</td>
<td>1717</td>
<td>2714</td>
<td>14</td>
<td>14.6</td>
<td>17</td>
<td>2.5</td>
</tr>
</tbody>
</table>

Table 4.1: Synthesizing Baseline $\mu IR$ on Arria10 FPGA — $F$ - Floating point benchmarks $[T]$: Tensor operations. Synopsys DC Compiler. ASIC umc 28nm.
4.6 Evaluating the benefit of passes ($\mu_{opt}$)

Table 4.2 summarizes the different optimization passes we study. It is not our intention to claim that the optimizations themselves are novel. Our goal is to implement and evaluate these optimizations as an $\mu IR$ graph transformation.

Our microarchitecture transformations can be broadly classified into three categories. i) Timing: The microarchitecture graph exposes latency and contention through connection edges. We study three specific optimizations that improve the communication latency between operations (op-fusion, scratchpad/cache banking). ii) Spatial: All nodes and structures in the $\mu IR$ graph can be replicated to improve throughput. We study spatial optimization in four components, function units, scratchpad, cache and task blocks. iii) Higher-Order Ops: $\mu IR$ provides an opportunity for specializing the compute node implementations on a per-type basis (for example, single-cycle tensor operations). For each pass we list the benchmarks that benefited from our optimization.

4.6.1 Auto Pipelining and Op-Fusion

**Result:** Reduces execution time between $1.17 - 1.7\times$

**Related research:** [81, 28, 91]

**$\mu IR$ scope:** Task dataflow, nodes and connections

This pass iterates over and transforms the dataflow graph of a task block. We auto balance the dataflow pipeline and fuse nodes in a greedy fashion. The baseline $\mu IR$ makes no scheduling decisions and hence requires pipeline handshaking on all dataflow edges. Fusing nodes, to balance...
Table 4.2: Summary of \( \mu_{opt} \) passes

<table>
<thead>
<tr>
<th>Opt</th>
<th>Type</th>
<th>Bench.</th>
<th>Sec.</th>
<th>Perf Impro.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op fusion</td>
<td>Timing</td>
<td>FFT, SPMV, COVAR., SAXPY</td>
<td>§ 4.6.1</td>
<td>1.4×</td>
</tr>
<tr>
<td>Task tiling</td>
<td>Spatial</td>
<td>STENCIL, SAXPY, IMG. SCALE, FIB, M-SORT</td>
<td>§ 4.6.2</td>
<td>6×</td>
</tr>
<tr>
<td>Tensor Ops</td>
<td>Higher Ops</td>
<td>RELU[T], 2MM[T], CONV.[T]</td>
<td>§ 4.6.3</td>
<td>8×</td>
</tr>
<tr>
<td>Memory localization</td>
<td>Timing &amp; Spatial</td>
<td>SPMV, CONV., SAXPY, COVAR.</td>
<td>§ 4.6.4</td>
<td>1.3×</td>
</tr>
<tr>
<td>Cache banking</td>
<td>Timing &amp; Spatial</td>
<td>SAXPY, RELU, RGB2YUV</td>
<td>§ 4.6.4</td>
<td>1.5×</td>
</tr>
<tr>
<td>All Opt</td>
<td></td>
<td></td>
<td>$ 4.6.5</td>
<td></td>
</tr>
</tbody>
</table>

the pipeline eliminates the handshaking and pipeline register (Figure 4.10). The op fusion pass iterates over the dataflow in depth first fashion to search for opportunities to fuse nodes with their successors. During fusion, we seek to try to ensure that the resulting fused pipeline’s frequency is not penalized (compared to the baseline). Combining multiple low-latency nodes together reduces the number of pipeline stages (and consequently latency) without introducing frequency-robbing critical stages.

![Figure 4.10: Illustration of Auto-Pipelining and Op-Fusion pass.](image)

Figure 4.10 shows the normalized execution comparison with the baseline implementation on four benchmarks, FFT, SPMV, COVAR, and SAXPY. The overall execution time reduces by 1.2× to 1.6×. These were chosen due to their compute intensity. This pass primarily target compute intensive dataflows where there are long chains of fusible nodes and inexpensive operations like shift and bit-wise operations.

4.6.2 Concurrency Tiling

**Result:** Reduces execution time between 1.5—6×

**Related research:** [100, 51, 66, 190, 175]

**\( \mu_{IR} \) scope:** Task blocks
Figure 4.11: Execution time improvement due to Op-Fusion — Baseline $\mu IR = 1$

Figure 4.1: Pass 2 in § 4.2 provides an overview of this optimization. $\mu IR$ permits each task block to independently increase the number of execution units. This effectively achieves a “multi-core” effect with multiple execution units running and completing in parallel.

Figure 4.12 plots the performance when varying the number of execution units per task. The baseline accelerator $\mu IR$ specifies 1 execution unit for each task. We only study Cilk benchmarks as they exploit higher-level parallelism. The accelerator exploits all the available parallelism exposed by the applications and scale with increasing FPGA resources ($1.5–6 \times$). Saxpy improves with the addition of a second tile, but the benchmarks become quickly memory bound. Stencil and Image-scaling accelerators are more computationally intense (scale up to 8 cores). Both fib and merge-sort have extensive parallelism and scale well up to 4–8-way parallelism before being limited by the memory system.

4.6.3 Tensor Higher-Order Ops

**Result:** Reduces execution time between $4 – 8 \times$

**Related research:** [32]

$\mu IR$ scope: Nodes and connections

In this section, we introduce tensor operators in the microarchitecture. They are highly optimized hand designed library (in Chisel) of operations that $\mu IR$ can incorporate during the construction of the dataflow. Our $\mu IR$ library includes support for 2D tensors, whose shape the designer can control i.e., for instance $2 \times 2$ in this example (Figure 4.13). In $\mu IR$ all the microarchitecture components are typed and reflect the type specification of the macro operation. The microarchitecture transformations do not have to be involved in the actual implementation of the operations.
itself, but can flexibly use them within the dataflow. The actual hardware function unit for the tensor operations are incorporated in from the library of components. Figure 4.13 lists tiled matrix multiplication.

```
for (int i = 0; i < NTiles; i++)
    for (int j = 0; j < NTiles; j++)
        for (int k = 0; k < NTiles; k++) {
            /** Tensor Intrinsics **/  
            Tensor2D* a = loadTile(&A[i][j]);
            Tensor2D* b = loadTile(&B[i][j]);
            Tensor2D* mul = multile(a,b);
            storeTile(addTile(&C[i][j],mul),&C[i][j]);
        }
```

Figure 4.13: Implementation of 2MM with Tensor ops

Figure 4.14 shows an optimized reduction-tree implementation of tensor multiplication for 2×2 shapes. Compared to the baseline which implements the operation through the pipeline, this is more efficient and also embarrassingly parallel. µIR also parameterizes the type of the intrinsic itself i.e., one of the parameters for a scratchpad is the shape of the data (2×2 in this case). µIR autogenerates RTL for the appropriate RAMs.
We implemented operator for all common tensor math (e.g., +, *, conv) and evaluated their benefit in improving 3 benchmarks RELU[T], 2MM[T], and CONV[T]. We find that the tensor operator increases leads to $4 - 8 \times$ improvement in performance (Figure 4.15). The cause for this improvement i) $\simeq 4 \times$ increase in computational density and DSP blocks compared to baseline ii) the operand networks are all widened to implicitly transfer all the elements of the Tensor2D at one time iii) the fusion of scalar ops into a single higher order operator eliminates the pipeline handshaking.

![Multiplier unit for Tensor2D](image)

**Figure 4.14**: Multiplier unit for Tensor2D. $C_{2\times2} = A_{2\times2} \times B_{2\times2}$

![Performance improvement](image)

**Figure 4.15**: Performance improvement due to tensor ops — Baseline IR = 1
<table>
<thead>
<tr>
<th></th>
<th>Execution Tile 1 to 2</th>
<th>Add one more SRAM</th>
<th>Fused Operation</th>
<th># Graph</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>µIR</td>
<td>FIRRTL</td>
<td>µIR</td>
<td>FIRRTL</td>
</tr>
<tr>
<td>Saxpy</td>
<td>1</td>
<td>4</td>
<td>39</td>
<td>92</td>
</tr>
<tr>
<td>Stencil</td>
<td>1</td>
<td>4</td>
<td>68</td>
<td>144</td>
</tr>
<tr>
<td>Image SCA.</td>
<td>1</td>
<td>4</td>
<td>46</td>
<td>128</td>
</tr>
</tbody>
</table>

Table 4.3: Conciseness of µIR vs FIRRTL (All µopt)
4.6.4 Localizing and Banking Memory

**Result:** Reduces execution time between 1.05—1.8×

**Related research:** Universal $\mu IR$ scope: Memory

The baseline microarchitecture used a shared scratchpad for local accesses and an L1 cache for all global accesses. Here, we focus on further increasing the number of scratchpads and L1 cache banks. First, we leverage algorithm 2 listed in section 4.2 to create multiple local memory address spaces. Second, we bank the L1 cache to parallelize the global accesses. $\mu IR$ auto-generates the RTL logic for i) for routing loads/stores to the different memory banks, and ii) managing shared ports.

Figure 4.16 shows the performance improvement of these optimizations. SPMV, SAXPY, and CONV2D benefit from localized scratchpads as they stream data. SAXPY and CONV2D read in two matrices and hence do not benefit from four-way memory partitioning. The amount of improvement for benchmarks depends on memory level parallelism of each workload and whether working set size fit in cache (64KB here). For instance workloads such as GEMM and FFT benefit from parallel access to local caches. 2MM, and 3MM see no benefit because the data maps to the same cache bank and does not benefit from the increased port and banks. COVAR. is compute intensive.

![Figure 4.16: Effect of cache banking (1–4 Banks) — Baseline IR = 1](image-url)
4.6.5 Stacking Multiple optimizations

**Observation**: Applying multiple optimizations, leads to cumulative benefits — overall between 20%—4.2× improvement in performance. Our goal is to study the best performance achievable with the set of optimization we study in this paper. Figure 4.17 shows the overall performance improvement. We group together all the Cilk accelerators, as the concurrency tiling optimization applies only to them. SAXPY and STENCIL: the tiling pass increases parallelism. To accommodate the higher compute parallelism, memory localization increases the memory parallelism as well. GEMM, FFT and SPMV: Most of the performance improvement is attributable to Op-Fusion since it improves loop initiation interval and improves pipeline parallelism between loops. In 2MM and 3MM, both Op-fusion and localization are helpful to improve the performance. Finally, in COVAR, CONV, DENSE and SOFTM, the primary optimization that results in performance improvement is memory localization and banking.

![Figure 4.17: Effect of stacking multiple μopt optimizations](image)

4.6.6 μIR vs. an ARM A9

μIR accelerators perform 2–17× better than an ARM A9.

Figure 4.18 compares the perform of μIR-optimized accelerators against an ARM A9 1Ghz dual issue out-of-order processor. In this case, we compare the best version of each accelerator with all the μopt optimizations applied. There are three main reasons for the better performance of μIR accelerators i) More ILP: GEMM, FFT, RELU and 2MM accelerators can issue more operations per cycle than the dual-issue ARM. ii) More compute density: Relu[T], 2MM[T], and Conv[T] lever-
age tensor function unit to pack more ops/cycle into the execution; CPU pipeline limits compute density. iii) Reduced overhead: the dataflow execution model eliminates the latency penalty of the front-end in CPUs.

Figure 4.18: Optimized $\mu IR$ vs ARM A9 1Ghz. ARM = 1. > 1: $\mu IR$ is better. < 1 ARM is better. Note: ARM does not support Cilk.
Chapter 5

TaskGraph

In this chapter, we discuss our abstraction to represent parallel accelerators from programs with dynamic parallelism. Our abstraction is built on top of software parallel compiler [131], which embeds fork-join parallelism into the compiler’s intermediate-representation. It represents hardware accelerators from parallel programs that contain on-the-fly or dynamic parallelism. We illustrate that the dynamic task-based accelerator has flexibility for realizing nested, heterogeneous, recursive, irregular or regular concurrency patterns. We discuss how TaskGraph compiler handles the challenges of generating hardware for a dynamically pipelined program. Finally, we evaluate the performance and flexibility of TaskGraph on FPGA SoC boards and compare our results with commercial HLS tools and multicore processors.

5.1 Introduction

A key limitation of HLS tools is their approach to concurrency. Accelerators attains high performance by instantiating multiple execution units that effectively support both coarse-grain and fine-grain concurrency [160, 15, 218] (relative to software). Unfortunately, current HLS tools do not effectively support concurrent languages. HLS tools also require an extensive set of annotations to generate parallel architectures. concurrency [30]. High-level-synthesis (HLS) tools with C interface typically analyze loops and employ techniques such as unrolling and pipelining [218]. Both Intel and Xilinx have targeted HLS at data parallelism [104].

HLS tools were aware of the challenges introduced by concurrency and have sought to exploit higher-level parallelism. LegUp [2, 35] includes support for a subset of the OpenMP and pthread APIs, and seeks to benefit from thread-level parallelism. IBM’s liquid metal [15] supported streaming kernel parallelism. Both toolchains are limited to static concurrency patterns, i.e. the parallelism structures are known during hardware generation and the structures cannot change during execution.

Recent works and industry-standard HLS tools have adopted fixed hardware templates that target specific concurrency patterns. Common templates include data parallelism, loop parallelism and loop pipelining [122, 158, 204]. The application programmer is expected to annotate and mod-
while(!done) {
  spawn_pipe_stage() {
    chunk_t *chunk = next_chunk()
    if(chunk == NULL) { done = true;
        exit(0);
    }
  }
}

spawn_pipe_stage() {
  chunk->is_dup = dedup(chunk);
}

if(!chunk->is_dup) {
  spawn_pipe_stage() {
    compress(chunk);
  }
}

spawn_pipe_stage() {
  write_to_buffer(chunk);
}

Stage-0: Pipeline Start/End
Stage-1: Ordered
Stage-2: Conditional & Pipeline
Stage-3: Receive data from S1 and S2

Figure 5.1: Pseudocode for PARSEC's [18] Dedup based on Cilk-P

ify the application to fit the template. Template-based HLS adopts a construct-and-run approach in which the concurrency and operations are scheduled statically at hardware generation time. Unfortunately, in many concurrent programs the parallelism evolves as the program runs, either due to control flow [125], or run time non-determinism [39, 42] (see example in Figure 5.2).

Current HLS tools are built on a sequential compiler i.e., compiler intermediate representation and passes restricted to a sequential program-dependence-graph. Hence, prior tools largely focused on programs with static parallelism that can be expressed through templates (e.g., pragma pipeline) or library calls (e.g., OpenCL). Our work focuses on programs with irregular fine-grain parallelism expressed implicitly within the program and it has been built on a parallel compiler released in 2017 [176]. We demonstrate that for programs with dynamic concurrency, FPGAs can achieve higher performance/watt than a multicore.

Our Approach

Our work focuses on synthesis of hardware accelerators from parallel programs that contain on-the-fly or dynamic parallelism. TaskGraph is a complete HLS framework that leverages parallel IR [196] to generate the RTL for a parallel task-based accelerator. The accelerator architecture includes support for spawning and synchronizing both homogeneous and heterogeneous tasks at run time. TaskGraph leverages the parallelism markers embedded by Tapir to generate RTL in two
Figure 5.2: Parallel accelerator generated by TaskGraph for dynamic pipeline parallelization of PARSEC’s Dedup

stages. The first-stage analyzes the parallel IR to infer the task dependencies, required synchronization, and generates a top-level architecture at the granularity of tasks. In the second stage, it generates the dataflow execution logic for each task; we permit arbitrary control flow (including loops) and memory operations. The microarchitecture generated by TaskGraph is specified in parameterized Chisel [14] and permits the designer to vary the number of tiles dedicated-per task, resource per task (e.g., queue depth, registers, scratchpad) and memory system capacity.

We illustrate that the dynamic task-based accelerator has flexibility for realizing nested, heterogeneous, recursive, irregular or regular concurrency patterns. We briefly discuss how TaskGraph handles the challenges of generating hardware for a dynamically pipelined program, Dedup from PARSEC (see Figure 5.2); the figure includes the commented pseudo code. HLS tools find this particular code sample challenging and cannot generate an optimal microarchitecture. First, the stages in the pipeline change based on the inputs. As shown in the task graph, for some iterations stage-2 could be entirely skipped based on the results from stage-1. Second, the stages have different ordering constraints and exhibit nested parallelism. Stage-2 is embarrassingly parallel while stage-1 enforces ordering across each sequence. Finally, the pipeline termination condition (see line 4) needs to be evaluated at runtime and cannot be statically determined (e.g., bounded loop).

To handle dynamic parallel patterns TaskGraph generates a hierarchical microarchitecture that includes first-class support for generic tasks. At the top-level the accelerator’s microarchitecture consists of a collection of unique atomic task units (one for each heterogeneous task in the system). Each task unit internally manages the dataflow logic for executing the task. The generated architecture has the following benefits: i) dynamic task spawning enables the program control to
skip stages entirely and change the pipeline communication pattern, ii) the hierarchical task logic organization permits concurrent tasks to be nested. TaskGraph permits Dedup’s stage-2 to be internally parallelized while ordering the tasks for stage-1. iii) The architecture eliminates dedicated communication ports, and allocates local RAM for communicating data between the tasks. This permits Dedup’s stage-1 to directly pass data to stage-3 when stage-2 is bypassed conditionally. iv) Finally, the architecture does not require any separate control for managing the task dependencies. TaskGraph derives the concurrency control from the compiler IR and embeds it within the tasks e.g., pipeline exit function is the next_chunk() dataflow embedded within stage-0.

5.2 Background and Scope

There is a gap in the quality of the hardware generated by HLS and human-designs a) partly due to the inability of the HLS tool to comprehend and exploit the parallelism available in software, and b) partly due to underlying abstractions being not available in the hardware architecture. As resource abundant FPGAs appear in the market it becomes imperative for HLS to support dynamic parallelism where the user only specifies what tasks can run in parallel, instead of how the parallel tasks are mapped to execution units.

5.2.1 Why dynamic task parallelism in TaskGraph?

A question that naturally arises when generating a parallel architecture is how does one specify parallelism to an HLS tool? There appears to be no consensus among current toolchains. This is primarily a result of there not being a standard framework to support concurrent execution, as is true for CPUs. Common frameworks such as OpenMP [2, 19] and Intel TBB are implemented using threads. However, it is unclear if the requirements of threads (e.g., precise register context, shared memory, per-thread stack) can be supported on non-CPU architectures at low overhead. Recent works have included support for threads in OpenMP loops [196, 10, 220].

We present an alternate vision based on the task abstraction. Please note that the notion of dynamic tasks [64, 10] we discuss here is different from the notion of static tasks explored in prior work [15]. Intuitively, a task is analogous to some encapsulated computation in software (not unlike a function call) which takes arguments and produces a value after running to completion. Every task is described as a three tuple (f(), args, sync). The function f() represents a scoped subset of the program dependence graph which implements the functionality. Args[] is a set of arguments passed to the function and sync is a run time field that represents other tasks that need to be synchronized. The key feature of tasks is that tasks can dynamically spawn new child tasks at run time. There has been extensive work in supporting static task parallelism in FPGAs [17, 29, 30, 60]. Prior works statically scheduled these tasks on the underlying execution units and relied on the task abstraction for understanding the static concurrency pattern. TaskGraph provides direct support in hardware for creating and synchronizing tasks dynamically based on accelerator execution. Task-
Graph adopts a dynamic software task model that has been previously explored in the context of [112, 98, 173], vector architectures [189] and GPUs [143].

5.2.2 Dynamic Parallelism vs Static Parallelism

The term dynamic parallelism refers to enabling tasks to vary at run-time, both in terms of the type of child task spawned and the number of child tasks spawned. Prior HLS tools adopt static parallelism in which any concurrent thread/task is created and scheduled up-front. We use a commonly used parallel for-loop (Figure 5.3) to illustrate the differences between dynamic and static parallelism. The loop exhibits dynamic parallelism. First, the loop bounds are determined by a parameter len which is known only during execution and which varies the number of parallel loop iterations. Second, the bar() function is invoked only if node[i] is valid. The figure shows how TaskGraph handles this pattern. TaskGraph creates a root task for the loop control, which spawns a child task, f(), only when required (i.e., node is valid) and up to the dynamic maximum of len. Current HLS tools will unroll the loop to exploit the static parallelism. When a loop is unrolled, the HLS tools create multiple hardware execution units onto which successive loop iterations are statically scheduled at the hardware construction time (in Figure 5.3 unroll factor is 2). Therefore, they must plan for the worst case and allocate resources for all possible iterations regardless of whether they are actually executed, and must handle corner cases (e.g. len < unroll).
5.2.3 Static vs Dynamic Scheduling

Another limitation of current industry-standard HLS tools [45] is the lack of dynamic scheduling i.e., the ability for the dataflow to handle variances in instruction latency (e.g., cache misses). Even the HLS tools that support threads [37], only support static scheduling of instructions. Since memory instructions also need to have deterministic scheduling, prior HLS tools primarily support a streaming memory model in which data is loaded into a scratchpad ahead of invocation. While the combination of static scheduling, static concurrency, and streaming memory model leads to high efficiency, but it limits the type of workloads that HLS can target. A pre-requisite for supporting dynamic task parallelism is shared memory and caches. Consequently, TaskGraph needs to handle non-deterministic latency in memory operations (see Section 5.3.3). Concurrent work from Josipović et al. [101] at FPGA 2018 has started investigating the benefits of dynamic scheduling of instructions. However, their work only exploits static parallelism from loops in sequential programs. TaskGraph’s focuses on dynamic parallelism and parallel programs. TaskGraph includes support for the task abstraction in the compiler that makes it feasible to target parallel languages such as Cilk. TaskGraph also supports dynamic scheduling, however this is not our focus.

```c
float d, s = 0.0;
int i;
for (i=0; i<100; i++){
    d = A[i] - B[i];
    if (d >= 0)
        s += d;
}
```

Figure 5.4: Limitations of static scheduling

```c
float d, s = 0.0;
int i;
for (i=0; i<100; i++){
    #pragma pipeline
    d = A[i] - B[i];
    if (d >= 0)
        s += d;
}
```

Figure 5.5: Static Schedule: a) no pipeline, b) pipeline

More recent works like Elastic Circuit [50, 195], CGPA [124] and [99] use dynamically schedule circuit to avoid the limitations in inferring loop pipelining. The idea is to refine from triggering the operators through centralized pre-planned controller but to take scheduling decisions locally in the circuit as it runs. Therefore, as soon as all the conditions for execution of an operation is
satisfied the node has to execute. Figure 5.6 shows the execution of a dynamically scheduled HLS circuit. The key reason to a good execution of this loop is that, ideally, a new value of `i` should be used to start computing `A[i] - B[i]` on every cycle. While this technique can dynamically provides better schedule compare to other statically scheduled HLS techniques. But it is clear that, as in the case of processors, taking scheduling decisions dynamically costs resources and time such as the area and delay of the control elements.

![Dynamic Schedule](image)

Figure 5.6: Dynamically schedule

Overall, all the pure C-to-gates HLS techniques described up to here, rely on capturing parallelism between fine-grain operations of sequential code by constructing the control and dataflow graph (CDFG) of the computation kernels. Then, they use either scheduling algorithms or dynamic techniques to extract parallelism between the operations that are provably independent in the CDFG. However, these techniques are less effective in capturing coarse-grain parallelism. Hence, the Quality of Results (QoR) of these HLS tools are often lower than that of manual design process using low-level hardware-description languages. Because the highest performance hardware exploits both fine-grain and coarse-grain parallelism.

To extract coarse-grain parallelism, mainstream HLS tools accept parallel programming constructs and/or annotations. For example, CatapultC [165] and Vivado [142] accepts SystemC [148] processes and modules. Vivado, also, accepts parallel functions/code-blocks communicating through dataflow channels. Altera openCL compiler [93] accepts single instruction multiple thread (SIMT) programming model. CMOST [221] is a C-to-FPGA framework that uses task-level modeling to exploit multi-level parallelism. In all these works, the number of input/output iterations and the execution time of each dataflow node for processing the input iteration need to be static and known at compile time. The HLS tools create multiple hardware execution units onto which successive loop iterations are statically scheduled at the hardware construction time. The down side of this approach is HLS tools must plan for the worst case and allocate resources for all possible iterations regardless of whether they are actually executed, and must handle corner cases.

### 5.2.4 TaskGraph vs Prior Work

Table 5.1 summarizes the feature set of current HLS tools. Many HLS tools [45] primarily target sequential programs and unroll loops to exploit instruction parallelism. A parallel architecture is often realized by using the HLS compiler to synthesize a single hardware core, and then typically requires an expert to manually instantiate multiple instances of the core in a hardware description
Table 5.1: Comparing the features of HLS tools

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Scope</td>
<td>Seq.</td>
<td>Kernel</td>
<td>Pattern</td>
<td>Parallel Prog.</td>
</tr>
<tr>
<td>Target</td>
<td>Loops</td>
<td>Thread</td>
<td>Pattern</td>
<td>Tasks</td>
</tr>
<tr>
<td>Hints</td>
<td>#pragma</td>
<td>Kernels</td>
<td>Pattern</td>
<td>Spawn/Sync</td>
</tr>
<tr>
<td>Dynamic Loops</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Heterogeneity</td>
<td>—</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nested parallel</td>
<td>—</td>
<td></td>
<td>Limited</td>
<td>✓</td>
</tr>
<tr>
<td>Lightweight</td>
<td>✓</td>
<td></td>
<td>Limited</td>
<td>✓</td>
</tr>
<tr>
<td>Multithreading</td>
<td>—</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Multicore</td>
<td>—</td>
<td></td>
<td>✓ Multiple execution units</td>
<td></td>
</tr>
</tbody>
</table>

language. To avoid this, both Xilinx Vivado and Intel HLS unroll and pipeline loops to convert loop parallelism to instruction level parallelism. Achieving efficient hardware requires the software developer to identify where it might be feasible to exploit loop parallelism and add additional hardware-oriented pragmas. HLS tools have anticipated the need to target higher levels of parallelism. Recent works have supported a subset of OpenCL, OpenMP or Pthreads. The primarily target is data parallel kernels. Current HLS tools schedule the concurrent operations statically and do not support dynamic spawning, asynchronous behavior, or nested parallelism. Furthermore, since the HLS tools statically schedule the memory operations, they require code annotations to help identify the streaming and FIFO access patterns between functions [38]. Finally, a promising avenue of research is HLS for domain-specific patterns. The hardware expert designs a parameterized template that targets a parallelism pattern (e.g., pipeline) and the software developer modifies the applications to ensure the program structure matches the pattern. Unfortunately, patterns always risk becoming obsolete.

TaskGraph targets parallel programs (not a particular pattern) and only requires the programmer to identify concurrent tasks. It is built on a parallel compiler and leverages the information to automatically synthesize parallel hardware for arbitrary task graphs. The key novelty of TaskGraph is that tasks can dynamically spawn and sync with other tasks. This enables TaskGraph to handle a variety of common programming patterns including nested, recursive and heterogeneous parallelism. Finally, TaskGraph supports dynamic scheduling of operations and handles non-determinism to enable a cache-based memory model.
Figure 5.7: Overview the TaskGraph framework. The input to the framework is a concurrent program written in Cilk. TaskGraph is organized in three stages. The input to Stage 1, is program dependence graph from LLVM IR. In Stage 2, the input is the generated task level architecture. At this stage the framework, optimizes the task level architecture, generates TXUs and connect them to memory hierarchy. Finally, at Stage 3, the framework sets the proper parameters and generate RTL design.
5.3 TaskGraph: High-Level-Synthesizing Dynamic Parallel Accelerators

TaskGraph is a hierarchical HLS toolchain for generating the RTL for a parallel application-specific accelerator (see Figure 5.7). TaskGraph is language agnostic since it relies on Tapir-LLVM to parse the parallel program and generate compiler IR with additional markers indicating the parallelism. The input to TaskGraph is a parallel program with markers for tasks and parallel loops; currently our infrastructure has been tested using Cilk, OpenMP and Cilk-P. Figure 5.7 shows the stages in TaskGraph and RTL generation for a program with nested parallel loops. TaskGraph consists of three stages. In Stage-1, (Section 5.3.1) TaskGraph analyzes the compiler IR, extracts the task dependencies, and generates the top-level RTL. The task units are declared and wired to the memory system. In Stage-2 (Section 5.3.3) the program graph of each task is analyzed, and the RTL is generated for the dataflow of each task unit. Finally, in Stage-3 we configure and set the hardware parameters (e.g., number of execution cores) based on a specific deployment (e.g., LUTs available on the FPGA) and generate FPGA bitstream.

TaskGraph-generated accelerators support dynamic parallelism, dynamic scheduling, and caches. We restrict all communication between the ARM and the accelerator to occur through shared memory. Currently, TaskGraph maps the accelerators to the FPGA on an SoC board. The ARM and the FPGA share a 512KB L2 cache. We synthesize a 16K L1 cache for the accelerator which is kept coherent with the L2 through AXI. TaskGraph generates a binary for the program regions/functions that cannot be offloaded (e.g., due to system calls) and they run on the ARM. TaskGraph does not rely on any hard logic in the FPGA and synthesizes the logic required to support the parallelism. This enables a flexible execution model that is independent of the processor and enables TaskGraph to target different FPGA boards.

5.3.1 Stage 1: Task Parallel Architecture

TaskGraph relies on Tapir [176] to comprehend the semantics required by the task-based accelerator architecture. Tapir adds three instructions to LLVM IR, detach, reattach, and sync, to express fork-join parallel programs. Using these three instructions TAPIR can support dynamic task spawning (create a concurrent task) and sync (synchronize parent and child). We describe the front-end task compiler pass in more detail in §5.3.6 and focus on the hardware generation itself once the task dependencies are known. The generated accelerator consists of multiple task units at the top-level, and each task unit represents a unique task. Figure 5.8 illustrates the top-level RTL, the interface and the parameters associated with the interface. TaskGraph supports time multiplexing (equivalent to simultaneous multithreading) of multiple tasks on an execution unit, dynamic tiling and assignment of tasks at runtime to different execution units (equivalent to multicore). A task unit is an execution engine for a single task and serves as the basic building block in the architecture. The accelerator can consist of any number of task units interacting to create different task graphs. There are four main components within each atomic task unit: i) The task queue which...
manages spawned tasks, ii) Parent task interface (Spawn/Synchronization ports), iii) Child task interface (Spawn/Synchronization ports), and iv) Task Execution Unit (TXU) which represents a pipelined dataflow execution unit.

```scala
class NestedAccelerator(implicit p: Parameters)
  extends Module {
    // Simple L1 Cache
    val SharedL1cache = new Memory[Cache]()
    // DRAM. AXI4 interface
    val DRAM = new Port[AXI]()
    // Wire DRAM and L1cache to AXI

    // Initialize task units
    val Task_0 = new Task0(Nt=32, Nr=32, ParentPort = 1, ChildPort=1, new T0-DF())
    val Task_1 = new Task1(Nt=32, Nr=32, ParentPort = 1, ChildPort=1, new T1-DF())
    val Task_2 = new Task2(Nt=32, Nr=32, ParentPort = 1, ChildPort=1, new T2-DF())

    //********** Connect Task Units **********/
    Task_0.io.in <> io.in
    Task_0.io.out <> Task_0.io.out
    Task_1.io.detach.in <> Task_0.io.spawn.out
    Task_0.io.sync.in <> Task_0.io.out
    Task_2.io.detach.in <> Task_1.io.spawn.out
    Task_2.io.sync.in <> Task_1.io.out

    //******** Connect Cache to Task units********/
  }
```

Figure 5.8: TaskGraph generated microarchitecture in μIR.

5.3.2 Execution Example

We describe the functionality of each of the components in the task unit by considering the implementation of the nested loop example (see Figure 5.9). The task graph in the figure illustrates three tasks T0, the outer loop control and spawner of N instances of inner loop. T1 is the inner loop control and spawner of N instances of T2, the body. Finally T2 performs the actual work, reading elements from the A[i][j], B[i][j] and adding them. In this application, N dynamic instances of task T1 will be created (for each iteration of the outer loop) and each dynamic instance of T1 will create N instances of T2 (total: N² instances).

A task in the queue can be in of the following states • READY: spawned, but not allocated a TXU • EXE: TXU allocated, but task has not complete • COMPLETE: execution complete and need to synchronize with the parent • SYNC: Waiting on synchronization with child tasks. The task
queue metadata consists of a child join counter (Child#), the Parent ID and Args[] RAM (argument RAM). \textbf{1} illustrates a spawn operation, with T0 initiating the task corresponding to the inner loop iteration, T1. A spawn is a tuple, Args[] and ParentID. The Parent ID consists of [SID, DyID]. The SID refers to the name of the parent task (in this instance T0) and the DyID corresponds to the task queue entry allocated to the instance of the parent task (index 0 here). This corresponds to dynamic task T0:0 spawning an instance of T1 (a j-loop, when \(i = 0\)). The Parent ID metadata available in the spawn is noted down in the allocated to the spawned T1 task and will be used during synchronization. In \textbf{2} the dynamic instance T1:0 (corresponding to the inner j loop with \(i = 0\)) creates N instances of the inner body T2. The field C# (Child#), in task T1:0’s queue entry corresponds to the count of the children tasks that are created by dynamic task T1:0. In this example, N instances of T2 are created corresponding to loop iterations \(i=0,j=0...N-1\). Note that, task T0 may concurrently create other instances T1:1,T1:2,... (inner j loop for \(i=1,i=2,...\)iteration) if there is enough queuing available. The task unit asynchronously assigns task execution units for the ready tasks.

In \textbf{3} as the instances of T2 complete they synchronize with their parent task that spawned them. Each task will only synchronize and join with the parent task that created it. Here, the T2 instances T2:0...T2:N-1 (corresponding to tasks \(i=0,j=0...N-1\)) will join on completion with the dynamic instance of their parent T1:0 (\(i=0, j\)-loop control). Joining entails decrementing the counter in the queue entry (index 0) corresponding to T1:0. The purpose of noting down the SID and DyId when the T2 tasks were spawned is clear now. The SID permits composability and allows heterogeneous task units to communicate with and dynamically spawn a shared task. The SID serves as the network id of the parent task unit to route back on a join. The DyID serves as the index into the queue within the task unit corresponding to the SID. Finally in \textbf{4}, once T1:0 has joined with all its spawned children, it proceeds to move from SYNC to COMPLETE status and reattaches.

Figure 5.9: Execution flow of Nested-loop accelerator generated by TaskGraph
back with its parent, T0. The task queue interfaces decouple task creation from task execution. The spawn and sync are asynchronous, and employ ready-valid signals. The asynchronous design permits us to vary the resource parameters per task without having to reschedule the tasks to deal with changes in latency.

### 5.3.3 Stage 2: Generating Task Exe Unit (TXU)

![Figure 5.10: Task-2’s Task Execution Unit (TXU).](image)

TXU is the representation of execution engine within each task unit. Each TXU is a fully pipeline execution unit which permits multiple dynamic instances of a task execute simultaneously. The TXUs only communicate at the task boundaries with each other. All the inter-TXU communication is marshaled through a shared scratchpad or the cache.

TaskGraph generates the logic for the TXUs based on the per-task sub-program-dependence-graph earmarked by our compiler. Each TXU is a dataflow that enables fine-grain instruction level parallelism to be mined. TaskGraph HLS dynamically schedules the operations in the TXU. An automatic pipelining process introduces latency insensitive ready-valid interfaces between each operation in the dataflow. A dataflow graph mapped to the TXU may contain nodes with multicycle latency (e.g. floating point operations), and non-deterministic latency (e.g., memory operations). This approach is in contrast to current industry strength HLS tools which try to schedule the timing all operations statically; concurrent work in FPGAs has begun to analyze the potential of dynamic scheduling [101].

Figure 5.10 shows the add function unit from the file created by TaskGraph, from \( C[i][j] = A[i][j] + B[i][j] \). The add function unit communicates with Load \( A[][] \), Load \( B[][] \), + and Store \( C[][] \) via decoupled handshaking signals which contain ready and valid signals in addition to data. The handshaking interface is governed by a simple state machine. This dataflow permits multiple concurrent T2 tasks to be outstanding at the same time on the
execution unit. Task pipelining is illustrated in Figure 5.11. The dynamic task ids correspond to the queue index allocated at run time. Note that the pipeline of a TXU is in dataflow order and tasks complete in order of issue. Any load stalls cause the pipelined dataflow to throttle and eventually stall; however this leads to a simpler implementation compared to dynamic dataflow [59].

### 5.3.4 Stage 3: Parameterized Accelerator

TaskGraph is a parameterized hardware generator and seeks to permit late stage parameter binding. As hardware designs grow in complexity, modularity becomes necessary. The asynchrony and latency insensitivity permits each of the task units to be parameterized independently. As shown in Figure 5.8 every task unit provides the mechanism for passing parameters prior to hardware elaboration and bitstream generation. While each tile has multiple parameters that can be set including the width and types of the args RAM, there are primarily two parameters that are set at this stage in toolchain, the task queue size ($N_{tasks}$) and the number of task execution units ($N_{tiles}$).

We permit the user to vary the parameters on a per-task basis. The latency of the individual tasks and task dependencies will determine $N_{tasks}$. Determining $N_{tiles}$ is more involved as it depends on the the processing rate required of that particular task unit and how many active tasks are required to potentially hide memory latency.

### 5.3.5 Task Memory interface and Memory Model

In this study, we consider a heterogeneous SoC where both processing cores and accelerator are integrated into a single chip. Each of the accelerator’s caches is connected to the last-level cache, which is shared with the ARM processor over the AXI bus. A key question is how does the memory model look like on the accelerator side. TaskGraph permits arbitrary task graph patterns to be converted into accelerators and thus needs to support a more flexible cache-like interface.

In TaskGraph all the task units share an L1 cache; it is conceivable that this model is most suitable for handling a programming model most familiar to software programmers [164, 166]. Generating an optimal cache hierarchy is beyond the scope of this paper and we primarily focus on how to route values from the cache to the TXUs.

The data box (see Figure 5.12) connects a memory operation in the TXU to a memory interface. We currently support both cache and scratchpad (we only evaluate the cache memory model
Figure 5.12: Data Box. Interfaces with memory operations in logic box and transfers operations to/from a cache or scratchpad.

in this paper). We choose to group the common logic for multiple memory operations (e.g., misalignment) into the data box to minimize resource requirements. Figure 3.14 shows the architecture of the data box. Each data box consists of the following parameterized microarchitecture components: i) an in-arbiter tree network that arbitrates amongst requests to the memory interface, ii) an out demux network that routes responses back to the memory operations in the TXU’s dataflow, and iii) a table of staging buffers that contain the actual logic for reading the required bytes from the cache/AXI memory interface (which only supports word granularity accesses). Both the request and response networks are statically routed.

5.3.6 Compiler Front-end: Tasks from IR

TaskGraph is language agnostic and relies on the parallel IR introduced by Tapir [176]. Tapir provides the front-end language bindings that translates Cilk/OpenMP programs to the LLVM IR. Tapir adds three instructions to the LLVM IR, detach (or spawn), reattach and sync. Spawn and reattach together delineate a task. A detach instruction terminates the block that contains it, spawns a new task starting from the target block and continues execution in parallel from the continuation. The reattach terminates the task spawned by a preceding detach instruction. Since Tapir assumes a generic threadpool execution model, it leaves the markers in place in the original PDG (Program Dependency Graph). We leverage the markers to perform reachability analysis and extract an explicit task graph, which is the architecture blueprint for our parallel accelerator. Nesting loops and irregular flows are analyzed in this stage and in the resulting task graph all task relations and the basic blocks that constitute a task are explicitly specified. We perform live variable analysis to
extract and create the requisite arguments that need to be passed between tasks; these are used to parameterize the spawn port and args RAM for each task unit.

```c
/* current: Basic block processed TaskDep: Edges between tasks TFG: Task list and sub-pdg */
current = {BB-start}; TaskDep = {Root}; TFG = {};
while(current != EMPTY) {
    startBB = current.top
    visited[startBB] = true
    TaskDep.top.insert(bb)
    for-each-successor succ of bb:
        if (visited[succ] == true) continue;
        current.push(succ)
        if (Edge(bb, succ) == SPAWN)
            child = new Task(succ)
            TaskDep.push(child)
            TFG.insert(child)
            TFG.insertEdge(TaskDep.top, child, SPAWN)
    } elseif (Edge(bb, succ) == REATTACH) {
        Task currentTask = TaskDep.pop()
        TFG.insertEdge(tst.top, currentTask, REATTACH)
        TST.insert(succ)
    } else {/*regular edge*/
        TST.top.insert(succ)
        dfs(TFG, current, TST)
    }
} Input.pop()
```

Figure 5.13: TaskGraph pass for extracting tasks from Tapir

5.4 TaskGraph-generated Accelerators

There are no standard benchmark suites that target only dynamic parallelism. Table 5.2 gives a brief summary of the accelerator benchmarks and shows the characteristics of each application. Our emphasis is on being able to implement accelerators for these workloads without requiring additional effort from the programmer. Here we study applications that use common software patterns that current HLS tools either find challenging and may throw errors.

5.4.1 Nested Parallel and Conditional Loops

Related benchmarks: Matrix addition, Stencil, Image scaling, Saxpy Stencil

Multiple workloads employ the similar pattern of nested loops. However, there are variations based on the parallelism of the loop nests, loop depth and conditional loop entry/exit. Here, we briefly discuss Stencil. Stencil is an iterative kernel (Figure 5.14a) that updates array elements in a loop.
Table 5.2: Benchmark Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>HLS Challenge</th>
<th>Memory Pattern</th>
<th>Per-Task</th>
<th># Inst</th>
<th># Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix Add</td>
<td>Nested loops</td>
<td>Regular</td>
<td></td>
<td>49</td>
<td>21</td>
</tr>
<tr>
<td>Image Sca.</td>
<td>Nested, If-else loops</td>
<td>Regular</td>
<td></td>
<td>52</td>
<td>25</td>
</tr>
<tr>
<td>Saxpy</td>
<td>Dynamic exit loops</td>
<td>Regular</td>
<td></td>
<td>29</td>
<td>16</td>
</tr>
<tr>
<td>Stencil</td>
<td>Nested parallel/serial</td>
<td>Regular</td>
<td></td>
<td>23</td>
<td>16</td>
</tr>
<tr>
<td>Dedup</td>
<td>Task Pipeline</td>
<td>Irregular</td>
<td></td>
<td>180</td>
<td>72</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>Recursive parallel</td>
<td>Regular</td>
<td></td>
<td>36</td>
<td>52</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>Recursive parallel</td>
<td>Regular</td>
<td></td>
<td>26</td>
<td>19</td>
</tr>
</tbody>
</table>

While the loop is embarrassingly parallel, the loop bounds are variable which introduces dynamic parallelism HLS tools parallelize this pattern in two ways: 1) flattening the inner-loops to a single level or 2) having the HLS tool only parallelize the innermost loop (which is not parallel in this case), while executing the other levels serially. In contrast, TaskGraph decomposes nested loops into multiple task units. Each task unit is asynchronous and can independently configure the number of tiles to exploit all the available parallelism. TAPAS supports arbitrary nesting of loops; serial and parallel loops can be nested in any order to any depth (resources permitting). Since the task unit exposes an asynchronous spawn/sync interface, TaskGraph can set up each loop independently. Every loop can execute in parallel or serial fashion based on program semantics, without requiring any changes to the outer loops.

5.4.2 Pipeline Parallelism

Dedup code and hardware accelerator are outlined in Figure 5.2. Dedup has an irregular pipeline pattern. The main challenges posed by Dedup are:

- **Task-Level Pipeline**: HLS tools have limited support for task-level pipelines and primarily target loop pipelining. Dedup is parallelized using tasks that have have non-trivial entry and exit logic, making it challenging to convert them to loops.

- **Conditional stages**: Emerging research [35] has sought to support functional pipelines using FIFO queues that require the program to be rewritten. Unfortunately, dedup also has conditional pipeline stages (see S2 in Figure 5.2) which FIFO queues cannot support. FIFO queues fix producer and consumer stages and cannot handle conditional pipelines.

- **Intra-stage parallelism**: The FIFO ports are ordered and this would lead to dedup losing parallelism in the S2 stage, which permits out-of-order chunk processing (see task-dependencies in Figure 5.2).
```c
void stencil () {
/* Parallel for loop */
PL: cilk_for (pos = 0; pos < NROWS * NCOLS; pos ++) {
/* Serial for loop */
L1: for (nr = 0; nr <= 2*NBRROWS; nr ++) {
/* Serial for loop */
L2: for (nc = 0; nc <= 2*NBRCOLS; nc ++) {
    int row = (pos/NCOLS) + nr - NBRROWS;
    int col = (pos & (NCOLS-1)) + nc - NBRCOLS;
    if ((row < NROWS)) {
        if ((col < NCOLS)) {
```

(a) Parallel Stencil using Cilk

(b) Stencil Accelerator

Figure 5.14: Stencil example using μIR and TaskGraph

- **Pipeline control**: Finally, the pipeline termination condition is dynamically determined by an exit function (get_next_chunk()). HLS tools do not support dynamic exits, since they statically schedule operations.

  *TaskGraph* does not suffer from these limitations since it supports dynamic spawning/syncing of tasks and execution units are assigned at runtime. Furthermore, all tasks communicate with each other through shared memory and the parallelism is not limited by extraneous hardware structures such as FIFO.

### 5.4.3 Recursive Parallelism

TAPAS can effectively generate accelerators for recursively parallel programs. HLS tools have traditionally not supported recursion [58, 35] due to the lack of a program stack. Nothing precludes
the addition of a stack, but it would require changes to the HLS compilation framework. Figure 5.15 illustrates how TAPAS can support recursively parallel mergesort.

In mergesort, the primary function employs a divide and conquer strategy. It partitions an array into two halves, and recurses on each half in parallel. The parent function then waits on the children and merges the sorted halves. To implement recursion it must be possible for more than single invocation of the same function to exist at the run time. Further, the data also has to be implicitly passed via a stack. TaskGraph achieves this through the following: i) TaskGraph precisely captures the state needed by a recursive task from the LLVM IR and implicitly manages the stack frames in a scratchpad. ii) The task controller supports dynamic scheduling and asynchronous queuing, which permits a task to spawn itself without logic loops. iii) The task controller tracks the dynamic instances to support implicit parent-child synchronization iv) Finally, all return values from the recursion are passed through shared cache.

```c
void mergeSort(...) {
    if (start < end) {
        int mid = start + ((end - start) / 2);
        /* Spawn self to sort 1st half */
        cilk_spawn mergeSort(list, start, mid);
        /* Spawn self to sort 2nd half */
        cilk_spawn mergeSort(list, mid + 1, end);
        /* Parent waits for children */
        cilk_sync;
        merge(list, start, mid, end)
    }
}
```

![Figure 5.15: Accelerator for recursive mergesort.](image)
5.5 Evaluation

It is challenging to find a fair baseline since dynamic parallelism is not supported by existing HLS toolchains. Running our benchmarks on the FPGA would entail changing the algorithm and memory model, and a program re-write. Even finding a baseline CPU is challenging since the Cilk and Tapir are currently x86-only. The x86 multicore’s cache hierarchy is deeper and larger than the FPGAs, which makes it challenging to understand the impact of dynamic parallelism independent of the memory system. We answer the following questions: i) Can TaskGraph support fine-grain tasks? How fine-grain can the tasks be (§ 5.5.1)? ii) Is the performance improvement attributable to low task spawn latency or speeding up individual tasks with dataflow execution? (§ 5.5.3) iii) What is the baseline performance compared to an Intel i7 quad core. (§ 5.5.3) iv) What is the energy consumption and performance/watt benefit compared to a Intel i7 (§ 5.5.4). In all the cases, we use the same unmodified Cilk programs. v) How does static parallelism with prior HLS tools compare against dynamic parallelism in TaskGraph (§ 5.5.5)

5.5.1 Parallel Task Overhead

Result: The overhead of spawning a task on an FPGA is significantly less than a software. This enables small, fine grain tasks to scale better.

```c
void scale(int *a, int n) {
    cilk_for(int i=0;i<n;++i) {
        a[i]++;
    }
    return;
}
```

Figure 5.16: (a) Test Code, (b) Parallel Task Tiling

The microbenchmark in Figure 5.16(a) was synthesized to see how fast tasks can be spawned. Figure 5.16(b) provides a top-view of the generated architecture. We incrementally varied the amount of work (“+” operations) in the loop body. The performance is plotted against an increasing number of worker tiles (Figure 5.17). On a Arria 10 (≈300Mhz) target device, we achieve a maximum spawn rate of 40 million spawns/second. Even for fine-grain tasks (50 instructions),
the performance scales monotonically with the addition of parallel worker tiles. 'Software' in the plot (Figure 5.17) refers to spawning a task of 50 increments; the program was run on a Intel i7-3.4Ghz, 8MB L2 (four cores). At such fine-granularity, the software runtime for Cilk provides zero benefit due to task spawning overheads. TaskGraph exploits the low overhead of task spawning on an FPGA and enables fine-grain parallelism not exposed in software.

5.5.2 Resource Utilization

**Result:** Arria 10 FPGA board can support \( \approx 100 \) parallel tasks each containing 100 integer operations.

Table 5.3 shows the per-instruction and per-tile resource utilization of the accelerator. The test code was synthesized for two Intel SoC FPGAs, the Cyclone V and Arria 10 (see Table 5.3). The primary sources of overhead in TaskGraph are the task controller logic and memory arbitration. On the smaller Cyclone V SoC, 10 tiles of 50 integer operations each filled 85% of the chip (153Mhz). On the larger Arria 10 board the accelerator could achieve 308 MHz and occupied 12% of the chip. A single M20K block RAM consumed is for queuing the spawned tasks in the task controller logic.

Figure 5.18 shows the relative amount of ALM resources (aka. LUTs and registers) used by each sub-block of the design. In the extreme case (1 operation/task), 60% of the logic is non-compute overhead; at 50 operations/task, the overheads is \( \approx 20\% \). As the number of execution tiles increases, the overhead of the control logic is amortized and at 10 tiles the control overhead is reduced to 3%. The memory network required to support shared memory access is less than 10% of overall chip.
resources. The network is primarily needed to support dynamic scheduling and routing values back and forth from the shared cache to the internal nodes in the task execution unit.

### 5.5.3 Scalability and Performance

**Result 1:** TaskGraph generated accelerators exploit all the available parallelism exposed by the applications and scale with increasing hardware resources (1.5–6×).

**Result 2:** To improve accelerator performance compared to an Intel i7 multicore a better cache hierarchy is required.

Figure 5.19 plots the performance when varying the number of execution tiles per task. A performance increase from the baseline is seen in all examples with the exception of Dedup. In Dedup even the baseline case (1 tile) has four heterogeneous task units (Figure 5.7) organized as a pipeline, with one execution tile per task unit. Any further improvement with increasing tiles/task is feasible only if the pipeline stages are unbalanced (not the case here). The saxpy and matrix

---

<table>
<thead>
<tr>
<th>MHz</th>
<th>Tile</th>
<th>Ins.</th>
<th>ALM</th>
<th>Reg</th>
<th>BRAM</th>
<th>%Chip</th>
</tr>
</thead>
<tbody>
<tr>
<td>185.46</td>
<td>1</td>
<td>1</td>
<td>1314</td>
<td>1424</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>178.09</td>
<td>1</td>
<td>50</td>
<td>2955</td>
<td>3523</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>153.61</td>
<td>10</td>
<td>1</td>
<td>7107</td>
<td>8547</td>
<td>1</td>
<td>24</td>
</tr>
<tr>
<td>159.24</td>
<td>10</td>
<td>50</td>
<td>24738</td>
<td>27604</td>
<td>1</td>
<td>85</td>
</tr>
</tbody>
</table>

**Arria 10 (10AS066)**

<table>
<thead>
<tr>
<th>MHz</th>
<th>Tile</th>
<th>Ins.</th>
<th>ALM</th>
<th>Reg</th>
<th>BRAM</th>
<th>%Chip</th>
</tr>
</thead>
<tbody>
<tr>
<td>308</td>
<td>10</td>
<td>50</td>
<td>28844</td>
<td>27659</td>
<td>1</td>
<td>12</td>
</tr>
</tbody>
</table>

**Table 5.3: FPGA Utilization**

![Figure 5.18: ALM Utilization by Sub-block](image-url)
addition improve with the addition of a second tile, but the benchmarks quickly saturate the cache bandwidth as their inner loops are small and dominated by memory reads and writes. In contrast, the Stencil benchmark is more computationally intense and consequently scales well even up to 8 tiles and beyond.

Q2. In Figure 5.20, we compare the execution time of accelerators against an Intel i7 quad-core (3.4Ghz,8MB L2,21GB/s DRAM). Identical Cilk benchmarks were used for both the i7 runs and TaskGraph. We set the concurrency to be identical (four cores for i7 and four tiles for TaskGraph). Accelerator designs were generated for both Cyclone V SoC FPGA and Arria 10 SoC FPGA. On the Cyclone V the accelerators performed at approximately 50% of the multicore (even achieving speedup in a few cases). On the Arria 10, the generated accelerators performed on par with the i7 as a result of higher frequency (300Mhz vs 150Mhz for the Cyclone V). The Dedup accelerator achieved best speedup since the accelerator implemented the pipeline more efficiently than software. The mergesort accelerator performed poorly in comparison to the Intel i7 since it is completely memory bound and limited by the memory system on the FPGA.

![Figure 5.19: Performance Scalability.](image)

To understand the impact of the memory system, we ran the non-Cilk, sequential programs on the ARM CPU of the SoC board (same memory hierarchy as FPGA) and find it to be $13 \times$ slower than the i7. Any performance difference of our accelerators with the i7 is a result of: slower clock (FPGA 150Mhz vs 3.4Ghz i7) and smaller, less sophisticated cache hierarchy.
5.5.4 Energy Consumption

Result: TaskGraph-generated accelerators exceed the energy efficiency of the multicore often by 20\times.}
In Table 5.4 we report the absolute power consumption and resource utilization of the generated accelerators. The power is obtained using using Intel Quartus PowerPlay. It is an estimate of total power (static and dynamic) based on signal activity levels derived from gate-level simulation. The tabulated data shows how even with varied parallel patterns such as Stencil (nested loops) and Mergesort (recursive) we can effectively exploit the available resources (50% of the Cyclone V FPGA chip). Mergesort is the largest design using roughly half of the available chip resources and consuming approximately 1.5W of power. We compare the performance/watt (Figure 5.21) of the accelerator against a multicore; in both cases we set the concurrency level to four. The power for the multicore is directly measured through the RAPL interfaces. TaskGraph accelerators often achieve over 20× better performance/watt than the multicore.

<table>
<thead>
<tr>
<th>Bench</th>
<th>Tile</th>
<th>MHz</th>
<th>ALMs</th>
<th>Regs</th>
<th>BRAM</th>
<th>Power(W)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAXPY</td>
<td>5</td>
<td>149</td>
<td>7195</td>
<td>9414</td>
<td>3</td>
<td>0.957</td>
</tr>
<tr>
<td>Stencil</td>
<td>3</td>
<td>142</td>
<td>11927</td>
<td>11543</td>
<td>3</td>
<td>1.272</td>
</tr>
<tr>
<td>Matrix</td>
<td>3</td>
<td>223</td>
<td>4702</td>
<td>7025</td>
<td>3</td>
<td>0.677</td>
</tr>
<tr>
<td>Image</td>
<td>4</td>
<td>141</td>
<td>4442</td>
<td>5814</td>
<td>3</td>
<td>0.798</td>
</tr>
<tr>
<td>Dedup</td>
<td>3</td>
<td>153</td>
<td>10487</td>
<td>6509</td>
<td>3</td>
<td>1.014</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>4</td>
<td>120</td>
<td>5699</td>
<td>9887</td>
<td>62</td>
<td>1.155</td>
</tr>
<tr>
<td>Mergesort</td>
<td>4</td>
<td>134</td>
<td>14098</td>
<td>24775</td>
<td>74</td>
<td>1.491</td>
</tr>
</tbody>
</table>

Table 5.4: FPGA Resources (Board: Cyclone V)

5.5.5 Intel HLS vs TaskGraph

An apples-to-apples comparison with prior HLS tools is challenging since: i) HLS tools only support static parallelism. It is not feasible to convert some applications to use static parallelism (e.g., recursive mergesort), and with others (e.g., Dedup) the conversion changes the algorithm entirely. ii) HLS tools deploy a streaming memory model since they statically schedule all instructions with known latencies. TaskGraph employs caches and shared-memory, which are a pre-requisite for dynamic parallelism.

To attempt a quantitative comparison we use two benchmarks, SAXPY and Image scaling. Among our benchmarks these were amenable to static parallelism. We used the Intel HLS Compiler (v17.1) and employ the suggested streaming DRAM interface. We set up the DRAM latency for both the Intel HLS and TaskGraph to 270ns (150Mhz FPGA clock). We also set the same concurrency level. In HLS the loops was unrolled 3 times and TaskGraph was configured to use 3 tiles. The results are listed in Table 5.5. The results indicate that TaskGraph is pretty competitive. It may be feasible to hand optimize the HLS implementation further, but we could also optimize TaskGraph. The most notable difference is where the block RAMs are utilized. Intel HLS appears

---

The part_3_ddr_masters.cpp example included with Intel HLS
to generate large stream buffers in its load and store interfaces. In contrast, TaskGraph uses a 16K L1 cache shared by all task units, but also expends block RAMs in the task queue.

<table>
<thead>
<tr>
<th>Bench</th>
<th>Tool</th>
<th>MHz</th>
<th>ALMs</th>
<th>Reg</th>
<th>BRAM</th>
<th>ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image</td>
<td>Intel HLS</td>
<td>155</td>
<td>5467</td>
<td>10466</td>
<td>67</td>
<td>20ms</td>
</tr>
<tr>
<td></td>
<td>TaskGraph</td>
<td>152</td>
<td>4543</td>
<td>7126</td>
<td>10</td>
<td>21ms</td>
</tr>
<tr>
<td>SAXPY</td>
<td>Intel HLS</td>
<td>181</td>
<td>3799</td>
<td>7961</td>
<td>38</td>
<td>103ms</td>
</tr>
<tr>
<td></td>
<td>TaskGraph</td>
<td>146</td>
<td>4254</td>
<td>5718</td>
<td>11</td>
<td>99ms</td>
</tr>
</tbody>
</table>

Table 5.5: Intel HLS vs TaskGraph (Board: Cyclone V)
Chapter 6

Chain Hardware Abstraction

In this chapter, we describe our new instruction abstraction Chain and propose a new Von-Neumann based accelerator and demonstrate that many of the fundamental overheads (e.g., fetch-decode) can be amortized by adopting Chain instruction abstraction. We show that chains adapt to different acceleration behaviors by varying the length of the chains and the types of instructions that are fused into a chain. Finally, we explain how Chains convey the producer consumer locality between dependent instructions, which the new architecture then captures by temporally scheduling such operations on the same execution unit and uses pipeline registers to forward the values between dependent operations.

6.1 Introduction

For many years, general-purpose processor architectures and micro-architectures took advantage of Dennard scaling and Moore’s Law, exploiting increased circuit integration to deliver higher performance and lower energy computation. Even though the improvement was remarkable but it is getting increasingly difficult to maintain current exponential improvement.

Conventionally, designers used available transistors to increase the speed of superscalar processors by using different approaches such as speculative execution or bigger register files. After a while power hunger structure became a major problem. Dennard observed that voltage and current should be proportional to the linear dimensions of a transistor. As a result, he modeled power in CMOS chips as:

\[ P = QfCV^2 + V I_{\text{leakage}} \]

Where, \( Q \) is the number of transistor devices, \( f \) the operating frequency of the chip, \( C \) the capacitance and \( V \) the operating voltage. The leakage current \( I_{\text{leakage}} \) could be neglected until 2005 with device structures larger than 65nm. Before 2005, the size of the transistors shrunk, and the voltage was reduced. Therefore, circuits could operate at higher frequencies at the same power. However, these rules could no longer be sustained, because of the exponential growth of the leakage current. With Post-Dennard scaling, like with Dennard scaling the number of transistors grows with \( S^2 \) ratio.
and the frequency with $S$ from generation to generation, i.e. the potential computing performance increases by $S^3$ between two generations. Transistor capacitance also scales down to $\frac{1}{S}$ under both scaling regimes. However, as the threshold and operating voltage cannot scale any longer, it is no longer possible to keep the power budget constant from generation to generation and simultaneously reach the potential performance improvements. Whereas with Dennard scaling power remains constant between generations, Post-Dennard Scaling leads to a power increase of $S^2 = 2$ per generation for the same die area [197]. At the same time utilization of a chip’s computing resources decreases with a rate of $\frac{1}{S^2}$ per generation. Therefore, these microprocessor design techniques eventually hit the Power Wall.

6.1.1 Optimizing processor

Traditional general purpose processors consist two separate parts, back-end, and front-end. In order to execute an application, processor starts to fetch instructions from instruction cache sequentially. The next step is to decode the fetched instruction and decomposed information which has been embedded into the instruction. This information consists type of operation, source, and destination for the operands. If the size of instruction window is equal to one, then the processor has no control on scheduling the instruction dynamically. However, in order to achieve better performance the processor doesn’t look at single instruction and instead it looks at an instruction window to schedule what the execution unit needs to do for the next steps. As much as the instruction window is larger, the processor can have better scheduling. The minimum size of the instruction window can be equal to issuing width of the processor. However, this size can go further by using more complicated structures such as re-order-buffer (ROB) which is the case in superscalar processors [185]. Having better scheduling means that the processor can run instructions in an out-of-order fashion and therefore it can exploit more instruction level parallelism. The processor also can utilize the pipeline more aggressively by using speculation. The superscalar processors often use speculation with out-of-order execution so that they can schedule the speculated instructions. However, in this approach the processor should also pay the potential recovery cost in mis-speculations. Processor’s pipeline front-end consists all the above stages, fetch, decode and issue. Finally, the execution units read from register file and data cache for getting the operands and write back the result values into them. For generating control signals of the execution and storage units back-end relays on the front-end.

Improving each of these stages will improve overall performance of the processor. For instance, in front-end using Single-Instruction-Multiple-data (SIMD) execution model is an effective approach to reduce dynamic scheduling cost of front-end. Another way of optimizing the processor efficiency is to optimize the data acquisition mechanisms. There are proposed specialized memory [130, 126], registers [170] and scratch pad memory [85, 54] which only focuses on data acquisition. For instance, [105] uses optimized memory and memory interface with a simple in-order Von-Neumann core.
However, at [13] Azizi et al. shows that using different microarchitectures techniques such as increasing issue width or improving predication accuracy, won’t provide significant changes in energy consumptions.

6.1.2 Specialized Hardware

Altering general purpose processor with a specific optimization is an alternative approach to improve processor efficiency. A central tenet of these optimizations is to split up the program into multiple phases [140], and specialize the architecture for each behavior. Heterogeneous multicore systems (Big and small cores) [128, 146]) adopt this approach, but they tend to use a conventional front-end which dominates overall energy consumption [84]. Other approaches have sought to improve the general-purpose processor (here is referred to as OOO) efficiency by detecting and caching the loops in a smaller µop cache [86]. However, such approaches continue to rely on the energy-hungry backend of the processor such as a centralized register file and reorder buffer. Addressing these concerns, many hardware accelerator based approaches have eschewed the fetch-decode instruction model in favor of dataflow architectures [77, 139, 202]. SIMD-based design amortizes the cost of instructions across multiple data parallel operations but restrict the types of instructions that can be bundled and are closely coupled to the hardware function unit. A key limitation of such dataflow approaches is designing for programs with varying levels of instruction-level parallelism (ILP). A reconfigurable functional-unit fabric [77] may have low utilization (and consequently higher static power) when the ILP in the programs do not match the peak ILP available from the hardware. It is hard to see how dataflow-based accelerators[203, 202, 81] can adapt to varied instruction parallelism both across applications and within an application. Section 6.3 discussed the tradeoffs in dataflow accelerators.

In recent years, hardware specialization has become a promising paradigm for continuing efficiency improvements. The insight of this paradigm is that, depending on the type of program or code, relaxing certain capabilities of the general purpose core, while augmenting it with others, can eliminate energy overheads and greatly improve performance. The general attitude of specialization is altering general purpose processor with specific optimization.

This dissertation seeks to provide an answer for the following question. While it is clear that using customized hardware accelerators which exploits specific program behaviors is a promising way forward [141], it is not clear what is the particular accelerator microarchitecture and how can we achieve this efficiency while attaining the generality needed to support different applications.

We have made great strides in cases where the hardware targets an already mature application domain (e.g., SIMD or GPUs). However, it is not clear how accelerators can be developed to address other programs that exhibit diverse control and memory behavior and instruction parallelism.

A promising approach to specialization is the notion of “custom or magic instructions” [84, 44, 75]. The key idea behind “magic” instructions is to use a single instruction to concisely express the parallelism and communication amongst frequently used groups of operations. Magic instructions require compound function units with associated lightweight storage elements that can execute
Figure 6.1: Chainsaw overview. Our compiler constructs control-free superblocks [16, 81] to eliminate branches from the offload region. The compiler then fuses sequences of instructions in the dataflow graph to construct chains (C1, C2, C3 etc.) and statically schedules them on the Chainsaw multi-lane architecture at chain granularity. The unaccelerated program regions continue to run on the OOO. Compared to prior work that fused subgraphs [75, 73, 44] chains only fuse only sequences of operations and do not require specialized compound function units.

these instructions efficiently. Magic instructions effectively amortize the cost of instruction fetch and decode across many operations. However, they tend to be application specific. Finding these magic instructions and then designing the custom function units that are widely used is challenging and raises questions about the generalizability of this approach. Nevertheless, the notion of using magic instructions to express more information about the program’s operation flow to the hardware is a promising approach; except we focus on the question of what property of the dataflow graph should magic instructions convey to the hardware and how do we decouple the hardware from the magic instruction itself.

6.1.3 Contribution

In this chapter, we explore Chainsaw [183], a von neumann-style accelerator, for executing Chains, which is a special type of magic instruction. The key contribution is that chains are decoupled from functional unit design, and are discovered at compile-time, thereby eliminating the tension between magic instruction efficiency and generality, application coverage and hardware design cost. A chain is a set of instructions that exhibits a strictly sequential dependence pattern, i.e., each instruction in the chain strictly communicates only with the next instruction in the sequence. Chains are a generalization of the widely used fused multiply-and-accumulate instruction (a chain of an add and a multiply operation) or paired µops [91]. In contrast to SIMD or VLIW instructions which express parallelism, chains express the lack thereof. Figure 6.1 shows our overall LLVM-based compiler and architecture framework and illustrates chains in the dataflow graph.

A chain concisely expresses producer-consumer locality between operations similar to dataflow. The limited single-producer to single-consumer locality expressed by chains can be i) more easily
expressed with narrower instructions (i.e., no destination ops need to be specified) and ii) readily exploited using pipeline registers. We reduce back-end costs by temporally scheduling the entire chain on a single functional unit and then leverage pipeline registers to directly forward values between the instructions in the chain. This stands in direct contrast to energy-hungry writes to a register file in an OOO and the operands transfers typically needed over a dataflow fabric [174]. While chains can have a varied number of operations of various types, we restrict the number of live-ins and live-outs per chain to simplify the chain wakeup. Our accelerator, Chainsaw, exploits these chains to deliver highly efficient execution. Note that there are sections of the program where Chainsaw is not the most efficient execution engine (e.g., SIMD, unpredictable control).

Chainsaw itself is a simple multi-lane architecture not too different from clustered microarchitectures [72, 53, 108]; each lane is a simple 4-stage in-order pipeline. While chains are mapped to individual lanes; the lanes execute at the granularity of the individual operations within a chain. Each lane executes only one chain at a time and does not interleave operations between chains which minimize chain wakeup costs. The instruction parallelism is exploited across the lanes. Registers are only needed for inter-chain communication; bypass registers capture the intra-chain data movement. Similar to embedded processors [53], Chainsaw fixes the maximum number of instructions that can be mapped to a lane to minimize the fetch-decode costs. Chainsaw performs within 81% of an ideal CGRA accelerator. Chainsaw overhead to the processor core while saving 45% of the energy. Compared to a CGRA with 8× the resources Chains save between 24–54% of the communication power by localizing the communication and 21% of the static power by improving utilization of function units.

6.2 Background and Motivation

Chainsaw as a hardware accelerator has been inspired by different types of proposed accelerators. We have categorized accelerators in three different groups. The first group focuses on ISA extension and the instruction abstraction by using compiler ability to extract program control and dataflow structure. Second group focuses on improving general purpose processors performance and energy efficiency by optimizing resource allocation and reducing the cost of instruction to instruction communication. And the last group which uses dataflow accelerators as a co-processor beside the main processor only runs region of interest efficiently and leaves the rest of the code to run on the main processor. For the rest of the section, we explain each of these accelerator types and try express their inspiring features for Chainsaw.

6.2.1 Instruction set customization

Von-Neumann architecture introduced the concept of instruction to hardware design. As a result, the compiler could find a chance to embed information about program control flow and program data flow inside the instruction abstract. For instance, RISC and CISC they are two different strategies for designing processors using Von-Neumann architecture and instruction abstract. The big
The difference between these two approaches is the amount of information which has been embedded in the instruction. RISC only supports simple instruction format which can describe only single operation per each instruction. However, CISC picks more complex instruction format. At high-level view can say CISC’s instructions they have formed from multiple RISC or µops. CISC philosophy with a modern twist—specifically uses encodings that convey more program-level intent to hardware, including dataflow and inherent concurrency information that hardware would otherwise require extra work to extract. Such “big instructions” can encode the relationships among large numbers of low-level operations. They can also capture higher-level operations (e.g., FFT, MPEG encoding) amenable to execute on specialized hardware. Both styles of “big instruction” reduce the energy per arithmetic operation and enable more concurrent operations within the same power budget.

SIMD instruction set is a more advanced example of instruction abstraction which not only has information about the type of operations but also it can express data parallelism [72, 191, 114] between operands. Using SIMD instruction set helps to reduce the amount of fetch and decode needs to be performed by fetching single instruction and applying on multiple data. Another example for customizing instruction set is VLIW instructions which instead of seeking for data level parallelism inside the program the instruction set expresses available instruction level parallelism.

Even though either of these examples can provide improvement in performance and energy but base on 90/10 locality rule a program executes about 90% of its instruction in 10% of its code [89]. For getting better improvement from the code we need to have an architecture which can run the region of interest efficiently.

Application-specific instruction set extensions are an effective way of improving performance and saving energy. In this approach, critical computation subgraphs are accelerated by collapsing the subgraphs into new instructions which can execute on specialized function units specially dataflow architectures [174, 77, 152]. However, a pertinent question is what information about the dataflow structure do we expect from instructions to express, can the information be expressed without increasing the size of the instruction, and does it generalize?

Tensilica [73] tries to generate application specific functionality at the time the hardware is designed. Before building the hardware Tensilica extracts instruction set additions automatically by analyzing the benchmark programs and lets the designer define new system-specific instructions if preexisting features don’t provide the required functionality. By this approach, Tensilica first collapses the subgraphs into single instructions. This approach has two advantages. First, it can reduce the length of computation. Second, it reduces the number of intermediate results stored in the register file. However, the main problem with this approach is that for each application the new processor should be generated which is not a practical approach.

An alternative approach is using reconfigurable architectures besides of the main processor so that designer does not need to generate a new processor for each application. FPGAs can be an example for reconfigurable devices. However, because of coarse granularity of operations (bit-level) which FPGAs can support they suffer from configuration time overhead, and it’s also hard
to completely utilize area and power consumption of them. CCA [44] and DSFU [75] use a configurable array of function units (FU) which reduces their flexibility in compare to FPGAs but that enables them to accelerate a broad range of dataflow subgraphs. CCA extracts computational subgraphs from the code statically or dynamically. However, dynamic approach uses trace cache which is a power hungry and inefficient component, and after that, it configures function units to support extracted subgraphs. To increase utilization, CCA instead of using square shape of FUs, uses triangular shape which helps them to avoid idleness of the units. Limitation in the area is another problem for CCA. According to the proposed architecture, each FU in CCA can only support limited set of operations. The reason is that if each FU supports all type of operations, especially operations involve complex circuits such as multiplier or divider, then they have to reduce the number of FUs because the area and power budget is constant and as a result they limit themselves to only tiny subgraphs. If they choose another approach and for each FU they support only limited set of Ops then can support bigger subgraphs inside the program, and they are not limited by the size of the subgraphs. But in the other side, they limit the subgraphs to their type, and they can only support few specifics type of subgraphs. As we can see these type of architectures, they are limited by either size of the subgraphs or type of them. While this approach is promising for specific application domains but since they are limited by the type of instructions inside the dataflow graph and also the shape of the extracted subgraphs it is not quite clear how it can be generalized.

Another approach is the separation of compile-time ISA from hardware ISA, an approach pioneered by Transmeta. Some have applied this idea [91] in a limited context for dynamically fusing pairs of x86 micro-ops in the decode stage and handled as one internal operation. For instance, the processor front-end detects compare-and-branch instructions in the instruction stream and then "fuse" them together into a single micro-op. Therefore, the processor can handle these two instructions as a single instruction. As a result, the front-end overheads of an OOO reduces.

Observations: Instruction-based specialization is an effective approach to reduce the overheads of the Von-Neumann architecture. Chainsaw generalizes this approach and discovers new instructions from applications at compile time by aggressively fusing a variable number of operations and types of operations. An important challenge to be addressed is state management; as an increase in the number of operations expressed by an instruction makes it harder to manage the associated state. In this work, we use the abstraction of chains which fuses only sequences of operations minimizing the amount of state that needs to be maintained.

6.2.2 Efficient General-purpose Processors

Clustering of execution resources [108, 72, 94, 200] seeks to scale up execution resources, localize communication between instructions, and minimize the cost of issuing instructions. These works reduced instruction-instruction communication by steering instructions to individual clusters of execution resources. In the past, these approaches have been pursued mainly because of wire delays challenging pipeline design. Today, similarly with wire energy dominating overall
instruction execution, clustering methods can help improve energy efficiency. Another line of work [146, 128, 7] has used heterogeneous backends and schedule low-ILP or moderate ILP code regions on an in order backend to save energy. A fundamental limitation of past work is that they primarily focused on clustering backend resources while minimizing changes to frontend which expends a significant fraction of the processor’s energy. Loop-accelerators [86] recognized that the key to improving energy efficiency is to disable the front-end for repeating instructions; they also continue to use the backend of a general-purpose processor. Loop-accelerators do not localize communication between instructions and continue to use centralized register files, issue queues, and execution units. Another work on increasing the front-end efficiency [145] has used out-of-order cores to generate schedules for in-order cores to execute. However, our focus is not just on in-order execution but also on minimizing back-end energy by localizing communication. There has been work in multi-level register files that have drawn ideas from embedded computing to exploit data locality at the fine-grain level between dependent instructions [68, 53]. A key benefit of this approach is that it generalizes how compound function units [44, 139, 81] achieve energy efficiency by using low energy operand registers to help dependent instructions directly communicate with each other.

Observations: When designing a Von-Neumann based execution engine, we need to distribute the front and backend of execution to minimize the fixed overheads per instruction. Chainsaw uses a distributed lane-based execution model to localize communication between instructions and minimize the energy required to move data between dependent instructions. Chainsaw uses the compiler to identify and exploit the locality when moving values between dependent instructions. Compared to prior work that also sought to leverage fine-grain operand registers [53], Chainsaw develops a compiler framework to carefully fuse and organize the instructions to guarantee dependent instructions are scheduled on a same lane.

6.2.3 Dataflow accelerators

Many recent proposals in hardware accelerators [77, 31, 152, 151] have been inspired by past work in dataflow architectures [194, 174]. These approaches have sought to switch between the Von-Neumann execution on the general-purpose processor and the dataflow-based accelerator, Figure 6.2, in a fine-grained manner [139]. However, dataflow accelerators they face with multiple problems. In principle dataflow accelerators cannot reach higher parallelism than a VLIW core. To overcome this limitation, designers need to take into account loop-level parallelism [135], but still due to the limitation regarding architecture or applicability this problem cannot be solved completely. Moreover, by adding loop-level parallelism and pipelining loop iterations we need to add extra latches to the design which is not free. Dataflow accelerators statically map the program dependence graph to a fabric of homogeneous or heterogeneous function units at compile-time to eliminate the overhead of fetch-decode. But mapping the instructions into the fabric itself is a challenging problem [135]. They also distribute the execution and register resources to improve scalability. However, dataflow accelerators tend to spatially distribute dependent operations and
expend energy in moving data between the individual function units over the communication network. By design dataflow accelerators are more optimal when available ILP in the program region is equal or close to pick ILP supported by the accelerator; when there are only moderate levels of ILP or the code region inherently has sequential behavior they tend to idle the function units and have low utilization (and consequently higher static power).

![Array of FUs](image)

**Figure 6.2: Array of FUs [75].**

**Observation:** A key challenge with dataflow accelerators is the energy required for moving values between dependent operations mapped across function units. *Chainsaw* temporally maps multiple dependent operations to the same function unit to minimize data movement. Dataflow architectures may seek to improve utilization by mapping multiple operations temporally to the same function unit; however, doing so would require a complex packet-based network [174]. Current accelerators use fine-grain instruction granularity PEs and implement circuit-switching; however, such designs require a data transfer over the network for each producer-consumer dependency.

In Table 6.1 we summarized the related works to *Chainsaw* and base on our observation we made a comparison between the classes of the works that have inspired *Chainsaw*.

### 6.3 Dataflow Graph Execution

We motivate the chain abstraction for instructions using the DFG in Figure 6.3. We are focusing on frequently executed regions that are free of control flow i.e. on hot paths or *traces* since these regions are the best candidates for acceleration. Therefore, the example dataflow graph (DFG) only depicts data dependencies. The DFG has a typical structure that is representative of hot paths in a wide variety of applications. It is an inverted tree that consumes several input values to output a few values at the bottom. The example DFG uses values computed by nodes 4, 13, 10, 7, and, 23, and produces values that are visible outside the DFG in nodes 26, and, 27. It has an ILP of five in the early three levels; in subsequent levels, the ILP tapers off to two and then one.
Figure 6.3: DFG example.

Table 6.1: Comparing the hardware oriented Intermediate-Representations

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Low</td>
<td>Low</td>
<td>High</td>
<td>–</td>
<td>✓Minimal ISA</td>
</tr>
<tr>
<td>Instruction Level Parallelism</td>
<td>Limited</td>
<td>✓Multi-Proc</td>
<td>Dataflow</td>
<td>✓Lane Based</td>
</tr>
<tr>
<td>✓Specialized Ins.</td>
<td>✓Specialized Ins.</td>
<td>SIMD Ins.</td>
<td>Limited(PEs)</td>
<td>✓Chain Ins.</td>
</tr>
<tr>
<td>Data Transfer Cost</td>
<td>✓Localized</td>
<td>High (Reg. File)</td>
<td>High (Switch)</td>
<td>✓ByPass Reg.</td>
</tr>
</tbody>
</table>

Figure 6.4 and Figure 6.5 illustrate the differences between dataflow execution and Von-Neumann execution applied to a subset of the DFG nodes. In the dataflow execution, functional units (FUs) are configured for specific operations, each node is statically mapped to a specific FU, and the dependencies are converted to data value transfers between the FUs. The Von-Neumann execution shown here assumes just two FUs. These FUs are temporally reused by different nodes. To enable such temporal scheduling, instructions are stored in local instruction buffers and fetched, decoded and issued in order of dependence. The buffers may be local to FUs (as shown in the diagram) or
global (fused). The results produced may be stored on the FU until it gets overwritten or in local or global register files.

![Figure 6.4: Execution on a spatial fabric.](image)

![Figure 6.5: Execution on a von Neumann architecture.](image)

### 6.3.1 Fabric Utilization and Static Power

Executing the example DFG on a dataflow architecture with a single configuration pass will require at least as many FUs as there are nodes in the DFG. However, since the maximum ILP of this DFG at any level is five, at most five of the FUs can be active at any given cycle. All other FUs will be idle i.e. at least 17 FUs will be idle in any given cycle. Indeed, multiple instances of the DFG can be pipelined onto the dataflow architecture to reduce idleness. Even tough, the amount of reduction depends on the initiation interval. Unlike processor pipelines which are seeking to overlap pipeline stages, pipelining a dataflow graph requires the loop around the dataflow graph to be able to pipeline i.e., overlapping multiple operations depends on the loop carried dependencies and unrolling factor. Even after pipelining iterations, the utilization of individual function units may be affected by the longest critical path in the DFG. For example, the FU allocated to node 12 must hold the result
for an additional cycle (in cycle 3) while it waits for the result of \( 16 \) to materialize. Thus, this FU must incur one cycle of idleness as it is not on the critical path. Similarly, the function unit mapping \( 25 \) in Figure 6.4 has to wait for \( 20 \) to complete. Otherwise, a customized number of pipeline latches are needed between \( 25 \) and \( 27 \), adding new challenges to the design [79].
<table>
<thead>
<tr>
<th></th>
<th>#Ops in DFG</th>
<th>Avg ILP</th>
<th>CGRA 4X4</th>
<th>CGRA 8X8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Perf. Cycles</td>
<td>Idle Cycles</td>
</tr>
<tr>
<td>gzip</td>
<td>59</td>
<td>5.1</td>
<td>117</td>
<td>1041</td>
</tr>
<tr>
<td>art</td>
<td>54</td>
<td>3.9</td>
<td>166</td>
<td>2076</td>
</tr>
<tr>
<td>mcf</td>
<td>29</td>
<td>3.6</td>
<td>93</td>
<td>441</td>
</tr>
<tr>
<td>equake</td>
<td>24</td>
<td>2.6</td>
<td>119</td>
<td>1106</td>
</tr>
<tr>
<td>parser</td>
<td>38</td>
<td>4.7</td>
<td>102</td>
<td>1236</td>
</tr>
<tr>
<td>bzip2</td>
<td>321</td>
<td>12.4</td>
<td>680</td>
<td>17856</td>
</tr>
<tr>
<td>gcc</td>
<td>126</td>
<td>10.5</td>
<td>271</td>
<td>1782</td>
</tr>
<tr>
<td>mcf</td>
<td>30</td>
<td>3.3</td>
<td>79</td>
<td>354</td>
</tr>
<tr>
<td>namd</td>
<td>78</td>
<td>6</td>
<td>161</td>
<td>1158</td>
</tr>
<tr>
<td>soplex</td>
<td>39</td>
<td>3.25</td>
<td>128</td>
<td>1522</td>
</tr>
<tr>
<td>povray</td>
<td>95</td>
<td>6.78</td>
<td>191</td>
<td>1307</td>
</tr>
<tr>
<td>hammr</td>
<td>147</td>
<td>13.3</td>
<td>499</td>
<td>7792</td>
</tr>
<tr>
<td>sjeng</td>
<td>99</td>
<td>5.8</td>
<td>219</td>
<td>4302</td>
</tr>
<tr>
<td>h264ref</td>
<td>64</td>
<td>5.3</td>
<td>175</td>
<td>3402</td>
</tr>
<tr>
<td>lbm</td>
<td>207</td>
<td>23</td>
<td>412</td>
<td>1847</td>
</tr>
<tr>
<td>sphinx3</td>
<td>44</td>
<td>3.6</td>
<td>91</td>
<td>1350</td>
</tr>
<tr>
<td>blacksc.</td>
<td>273</td>
<td>5.5</td>
<td>368</td>
<td>18257</td>
</tr>
<tr>
<td>bodytra.</td>
<td>81</td>
<td>4.7</td>
<td>173</td>
<td>3780</td>
</tr>
<tr>
<td>dwt53</td>
<td>33</td>
<td>3.6</td>
<td>102</td>
<td>1747</td>
</tr>
<tr>
<td>fluidan.</td>
<td>70</td>
<td>4.1</td>
<td>184</td>
<td>2860</td>
</tr>
</tbody>
</table>

Table 6.2: Characteristics of CGRA execution (4x4 and 8x8). Performance (cycles), Idle Cycles (cumulative function unit idle cycles). ILP: dataflow instruction parallelism
Table 6.2 shows the idle cycle count for spatial fabrics of size 4X4 and 8X8. In this table, idleness is defined as the total number of cycles for which the function units are idle. In this case, we are modeling an instruction granularity CGRA with one function unit per PE, and we assume an ideal memory system to eliminate the stalls due to the memory system. The idle cycles in the table are indicative of the idleness caused entirely by the mismatch between the ILP available in the program and the ILP of the CGRA; making the CGRA smaller would increase the reconfiguration frequency and overhead. The idleness in the 8X8 (64 units) fabric is average 2× compared to the 4X4 (16 units) fabric (max: povray. 4× the idleness). Idleness is a critical factor because it determines the static power consumption of the fabric and as can be seen, the cumulative idle cycles are many times the execution latency. In some cases, the 4X4 can have idleness higher than the 8X8; this may seem counter-intuitive; we discuss the reason in the following section.

6.3.2 Reconfiguration costs

Idleness could be reduced by reducing the number of FUs available, however, in that case, the dataflow architecture must be reconfigured multiple times to execute a single instance of the DFG, and the pipelining opportunities are also limited. Since, reconfiguration costs are significant, this is not always a viable solution.

Table 6.2 shows the average number of operations in the dataflow fabric. The fabric may need multiple passes to map the dataflow graph since the number of operations exceeds the number of function units in the fabric. However, the ILP column indicates that the dataflow graph will typically not be able to keep all the function units busy leading to uneasy tradeoff between static power and reconfiguration overheads. We illustrate the counter intuitive case introduced by the reconfiguration in gzip; in gzip the 4X4 demonstrates more idleness than the 8X8. Multiple reconfigurations introduce an unbalanced execution in the 4X4; gzip has 59 ops requiring 3 reconfigurations on a 4X4 (16 units) fabric with the final configuration leaving 5 PEs free for the entire duration of running the final 11 operations on a small fabric. All 59 ops can be mapped to 8X8 at once leading to lower idle cycles.

6.3.3 Data Movement Energy:

Dataflow accelerators at the instruction granularity maintain a one-to-one mapping between operations and function unit [79] to enable the use of a circuit switched network for the dataflow transfers. While this minimizes the per-transfer overhead, every producer-consumer operation communication requires a network transfer. Under current technology constraints, we show that this can consume a significant fraction of the overall power consumption [52].

Interestingly, while Von-Neumann architectures temporally map multiple operations to the same function unit. It can capture the locality between back-to-back dependent operations by forwarding values through pipeline registers. For example, just five FUs can capture the maximum ILP presented by the example DFG, minimizing idle cycles. The efficiency of Von-Neumann architectures depends heavily on instruction granularity [91]. Coarse-grain instructions can potentially
reduce front-end overhead and exploit locality between operations [84]. However, they may adversely affect scheduling complexity due to dependency checks i.e. they consume and produce more values. To make coarse-grain instructions feasible, we need to group operations without increasing the number of external dependencies. Hot regions in programs typically exhibit instruction sequences that meet these requirements. The example DFG is composed of several chains of computation that have the properties we are looking for: \((4, 5, 6), (13, 14, 15, 16), (10, 11, 12), (7, 8, 9), (23, 24, 26), \) and, \((19, 20)\). The Chainsaw accelerator leverages such chains for greater efficiency.

6.4 Chains

Figure 6.6 shows a possible decomposition of the dataflow graph in Figure 6.3 into chains; there are five chains, \(C_0–C_4\). This decomposition has been produced using Dilworth chain decomposition algorithm [65]. Note that the original Dilworth decomposition may produce dependency cycles among chains, which are not allowable in our case since such chains cannot be temporally scheduled. Therefore, we applied Dilworth decomposition and then broke the cycles by breaking the chains at cycle-forming dependencies.

![Dilworth Decomposition](image)

Figure 6.6: Chains built using Dilworth decomposition followed by cycle removal. Chain schedule on two FUs; the height of a chain is proportional to the number of ops in it. Fusing operations to chains reduce inter-chain register writes (21 to 4). Chains can exploit ILP with only two FUs (latency: 16 cycles for DFG).

Figure 6.6 also depicts a possible chain schedule if two FUs are available. Each chain node has a height that is directly proportional to its latency. Chains \(C_4, C_3\) and \(C_1\) are scheduled on one functional unit in that order, while \(C_0\) and \(C_2\) are scheduled on the other unit in that order. Note
that in chain $C4$ supplies a value to chain $C1$. This value is produced before $C4$ completes execution; however since $C4$ can only communicate the value at the chain boundary, $C1$ is stalled until all the instructions in $C4$ complete.

Chains are essentially an ISA abstraction between the compiler and the hardware, i.e. they do not need any specialized function units. The computation within the chain is expressed as stripped out instructions of the original processor ISA. The communication to the operations within a chain is restricted. One of the operands for each internal instruction will be the produced by the predecessor instruction in the chain. Leveraging this observation, we eliminate the bits reserved for an instruction for register ids, compresses the instruction and reduce the fetch-decode penalty.

Chains provide the following benefits: 1) they localize a large fraction of communication between dependent instructions and eliminate many registers accesses 2) they reduce the number of front-end events and exploit the ILP effectively. For instance, while the dataflow graph had 21 dependencies communicated through registers before chaining, chaining reduces the number of register writes to 4. Finally, due to the adoption of the Von-Neumann model, we need fewer FUs than a spatial fabric by temporally mapping multiple operations to the same function unit, resulting in better hardware utilization and larger dataflow graph mapping.

There are two question questions which their answers are critical to the success of chains:

1. Are chains potentially beneficial?

2. Is chain management hardware-friendly?

### 6.4.1 Chain benefits

Before we explore the potential benefits of chaining, we discuss the potential drawbacks and our strategies for minimizing these drawbacks. Chaining potentially decreases available ILP because instructions may need to wait for their chain to be activated even though their input operands are already available. The chain is activated only when the input operands are available for all the constituent instructions. This impact is visible in Figure 6.6. Although instruction 13 is ready to run at the very outset, it cannot execute until the chains $C0$, $C2$, $C3$, and, $C4$ have finished.

Therefore, to recover lost ILP, we break chains at inter-chain dependencies as shown in Figure 6.7. For instance, instructions 16 and 17 have been assigned to different chains. Now, chain $C1$ can be scheduled at the beginning since it does not depend on any other chain. Indeed, the latency of the region decreases from 16 cycles to 13 cycles, as shown in Figure 6.7. We also ensure that both the live-in count and the live-out count of each chain are limited to two, for reasons that are discussed in Section 6.4.2. Breaking at every live-in limit the number of chain inputs to two because now a chain has at most one node that consumes live-ins, and operations need at most two operands. However, the value produced by a node may be consumed by more than two nodes. In such a case, we introduce dummy fan-out nodes to limit the fan-out of each node to two. Essentially, each fan-out dummy node acts as a switch with a fan-in of one, and a fan-out of two. We denote this strategy as MaxILP.
Figure 6.7: Strategy MaxILP breaks chains at live-ins and live-outs leading to shorter chains and a higher chain count. More inter-chain data movement but critical path reduces to 13 ops, identical to the unchained DFG. Inter-chain register writes increased to 9.

Splitting chains reduces some of the benefits because intra-chain dependencies are converted into inter-chain dependencies which need register updates. For example, the MaxILP strategy produces 9 inter-chain dependencies while the baseline decomposition required 4. To offset the loss,

Figure 6.8: Strategy MaxSize merges chains greedily to reduce data movement. Register writes decreased to 4. Critical path length increased to 14 ops.
we merge back chains greedily as long as they do not form cycles. We continue to limit both the live-in and live-out counts of each chain to two. Algorithm 3 lists the pseudocode for this strategy. It iterates over each edge, and if the edge happens to be an inter-chain dependency, the algorithm explores concatenating the source and sink chains. We denote this chain formation strategy as \textit{MaxSize}. For example, the chain decomposition in Figure 6.7 is converted to the chain decomposition in Figure 6.8. The overall latency increases to 14 cycles, but the number of external dependencies reduces to four.

Algorithm 3: Algorithm for the \textit{MaxSize} strategy.

\begin{verbatim}
Input: Dilworth decomposed graph
Output: MaxSize decomposed graph
1 foreach edge do
2   srcNode = source(edge);
3   tgtNode = target(edge);
4   srcChain = chain(srcNode);
5   tgtChain = chain(tgtNode);
6   if srcChain == tgtChain :
7      continue;
8   tmpChain = concat(srcChain, tgtChain);
9   if liveIns(tmpChain) > 2 :
10      continue;
11   if liveOuts(tmpChain) > 2 :
12      continue;
13   remove srcChain from dfg;
14   remove tgtChain from dfg;
15   add tmpChain to dfg;
16   if dfg has cycle :
17      remove tmpChain from dfg;
18      destroy tmpChain;
19      add srcChain to dfg;
20      add tgtChain to dfg;
\end{verbatim}

Figure 6.9 shows the chained graph for a frequently executed region in \textit{gzip}. The colors red, blue, green, and, yellow depict the nodes and edges belonging to a particular chain. Some binary operations such as multiply are shown to have zero or one inputs because either the inputs are data values produced outside the region, or are constants. The dataflow graph shows varying amounts of ILP at different levels which is unsuitable for spatial fabrics. At the same time, the dataflow graph exhibits long chains that can be exploited by \textit{Chainsaw}.

There is a tension between chain size and average ILP. Chains are beneficial if the ratio of intra-chain dependencies to inter-chain dependencies is high and the impact on ILP is minimal. Figure 6.10 shows the relative proportion of internal and external dependencies using the MaxILP and MaxSize strategies respectively. For the MaxSize strategy, 70-80\% of the dependencies have
Figure 6.9: The chained graph for a frequently executed region in gzip. The DFG has varying levels of ILP which is unsuitable for spatial fabrics, and has several long chains.

been converted to intra-chain operations, which will lead to high-efficiency benefits in chained execution. The conversion rate is significantly lower at 40-60% for the MaxILP strategy.

Figure 6.10: Relative proportion of inter-chain and intra-chain dependencies. Fusing ops into chains localizes communication.

Figure 6.11 elucidates the reason for the high rate of conversion by the MaxSize strategy. Each stacked bar shows the relative proportion of dataflow graph nodes in chains of different lengths i.e. the percentage of computation covered by chains of different lengths. For MaxSize strategy, 50-80% of the nodes belong to chains of length three or more that implies most nodes belong to reasonably long chains. *bzip2, soplex, lbm* and *dwt53* have 50% of the chains with more than 5+
ops. *equake*, *blackscholes*, and *swaptions* have dataflow graphs that are closely interleaved leading to small chains; $\approx 35\%$ of the operations have only one op.

Figure 6.11: Computation coverage by chains. Histogram showing the percentage of ops subsumed by chains of different lengths. 50–80\% of operations are subsumed by chains of length 3 or more.

Figure 6.12 shows the average ILP in dataflow graphs for both the strategies as compared to the ILP in the Unchained state. Since the MaxILP algorithm forcefully breaks the chain when any internal instruction has an external dependency it attains as much ILP as the original Unchained dataflow graph. The MaxSize algorithm may potentially lose ILP, when an internal operation in the chain is delayed from waking up a remote chain. The loss in ILP may also manifest as an increase in the critical path when compared to the ideal unchained dataflow graph. Our model here assumes every instruction in the critical path has the same latency to eliminate effects of memory operations. Overall, we find that in 6 applications chains increase the critical path by $< 20\%$ and lose Avg. 20\% of the ILP. In 5 applications (sphinx3, sar-pfa, dwt53, soplex, bzip2) we reduced the ILP by $2 \times$ compared to the ideal dataflow graph. Note that these are averages and in many cases as long as we don’t restrict the ILP at particular points of the dataflow graph (e.g., memory ops) the overall performance won’t necessarily suffer. To conclude, chains have a high potential for improving energy-efficiency.
6.4.2 Chain Distribution

Figure 6.14 shows the distribution of chains by size per application. The average chain size per application is presented at the top of the chart. Overall, the average chain length is 2.6 operations across benchmarks. Apart from 4 applications (183.equate, blackscholes, sar-pfa and swaptions), all other applications have 50% or more chains with size greater than 2. Figure 6.13 shows the number of chains per application. The MaxILP approach produces significantly more chains than the MaxSize approach as described previously in section 6.4. There are 30 chains per workload on average, 10 workloads have fewer than 20 chains. Only 482.sphinx3 has more than 60 chains (avg. size 3). In conclusion, the number of chains per workloads is tractable for frequently executed regions.

![Graph showing chain distribution](image)

<table>
<thead>
<tr>
<th>Critical Path Increase (MaxSize vs Ideal Dataflow)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;20%</td>
</tr>
<tr>
<td>20–40%</td>
</tr>
<tr>
<td>50%</td>
</tr>
</tbody>
</table>

Figure 6.12: The amount of ILP mined from the dataflow graph by the MaxILP and the MaxSize algorithms. MaxILP has average ILP that is equal to the unchained dataflow graph’s ILP. MaxSize shows notable ILP loss in some applications.
Figure 6.13: # of chains generated by the MaxILP algorithm and the MaxSize algorithms. MaxSize typically creates fewer chains enabling.

Figure 6.14: Average chain lengths generated by MaxSize algorithm.
6.5 Chainsaw architecture

Figure 6.15 describes the overall design architecture of Chainsaw. Chainsaw is a multi-lane design consisting of lanes as an execution unit. Each of the cores is designed based on Von-Neumann architecture. Each lane contains 4-stage single-issue pipeline, and it runs instructions in order which they have scheduled. Each lane fetches and decodes instructions from a pre-loaded instruction buffer and then executes the instructions. However, Chainsaw emphasizes on the notation of chain and it works based on the chain abstraction. Therefore, in the scheduling phase, each chain scheduled entirely on a single lane to exploit the intra-chain locality within the lane’s pipeline registers. Therefore, the instruction buffer stores the execution sequence for multiple chains. Having multiple lanes not only exploits available parallelism across the chains but also make opportunely to execute the chains out-of-order across the lanes. Chains communicate with each other through registers; the live-in register bank holds the values for the duration of a chain’s execution. Chains only write out the live–out values into the register once it completes all the instructions in the sequence. Each lane has a pair of input registers, IN0, and, IN1 to hold the live-in values for the duration of an executing chain; these are refilled from the live-in register file when a chain is scheduled on the lane. If a chain produces values that will be consumed outside itself, these values are placed in a pair of output registers, OUT0, and, OUT1. The INs and the OUTs essentially act as a local register file (refilled and written back at chain boundaries).

Limiting number of local registers has two benefits. First, it reduces the cost of accessing network. If all the lanes needed to share a global register file, for each access they had to send their read and write request over the network. As a result or read and write cost would be accessing register file plus accessing network. However, by using local register file for each lane, it gives a chance to avoid accessing the network for each single request. Another benefit of limiting number of local registers is reducing fetch and decode cost for each lane. Having INs and the OUTs registers indirectly help to minimize the fetch-decode energy by restricting the number of bits required to encode the register names i.e., instructions within a chain can only refer to 2 INs or 2 OUTs (2 bits for encoding) as opposed to register names (8 bits or more). Each lane starts fetching and decoding pre-loaded instructions from instruction buffer when their Chain Ready bits are set (Section 6.5.2). When a chain finishes executing a chain, the bus routes values in the output registers to the appropriate bank if required, however. Forwarding values from output register using the bus is only needed when the chain has been scheduled on a different lane. Since it can happen that next chain has been scheduled on the same lane. As a result, no more forwarding values is needed, and the lane can run chains back to back.

Finally, the scheduler determines when to schedule each chain in the DFG. Each lane includes a Chain Read bitmap which specifies the scheduled chains ready for execution. The number of entries in this table specifies the maximum number of chains that can be mapped to a single lane; each entry is 10 bits wide. Each chain has two live-in registers. Two 4-bit fields specifying the live-in register ids in the bank are needed to specify register ids. Remember each chain can have
at most two parents and to activating the chain we need to have information about the chain’s parents. Therefore, two flag bits for registering the completion of the parent chains are needed too.

All other structures also need to be proportionately scaled based on the Chain Ready bitmap ($N$), where $N$ is the number of chains; the number of live-in registers is $(2 \times N)$ which the compiler guarantees when forming the chains. The number of entries in the instruction buffer determines the maximum size of each chain; here we set it to the average size of $(\text{Chain}) \times N$. The number of lanes is proportional to the ILP available. For the rest of the chapter, we describe each stage of chains execution more in depth.

### 6.5.1 Instruction Execution

For executing chains each lane fetches and decodes from its instruction buffer. The instruction buffer in each lane is preloaded before *Chainsaw* starts executing instructions; at *Chainsaw* the re-configuration phase involves writing the instructions into the instruction buffer and filling scheduler table base on chains’ dependencies. Chains are statically mapped onto lanes at compile time to ensure that the requirements for the instruction buffer and the live-in register bank do not exceed the resource availability. However, chain activation is carried out dynamically since each lane also support memory operations and delay for memory operations is not deterministic. Therefore, we need to have a dynamic mechanism to improve performance. When a lane becomes available, i.e., there is at least one chain which is scheduled on that lane and the lane has finished running a chain or it hasn’t start running any chain yet, the scheduler task is checking for chains that are ready i.e. chains that have been mapped to the available lane by the compiler and whose live-in values are all available. Scheduler activates one of the ready chains by loading its live-in values.
from the live-in register bank to the $IN0$ and $IN1$ local registers of that lane. The scheduler also signals the fetch/decode unit in the lane to start issuing instructions. During chain execution, if an instruction produces a value that serves as a live-in to another chain, the value is written out to the one of the $OUT0$ or $OUT1$ local registers. When the chain finishes executing, the values in the $OUT$ registers are routed to the appropriate live-in register bank by the bus.

Figure 6.16 shows a simple chained DFG that we use to demonstrate the architecture functions. In this example, we are showing how the compiler fills the scheduler table at configuration phase. Also how the ISA’s fields are set by the compiler. At the end, we show that how the dynamic mechanism works to wake up the lanes.

Figure 6.16: Example DFG execution on Chainsaw.

6.5.2 Chain Scheduling and Wakeup

Table 6.3 shows the scheduler table generated by the compiler for this DFG when mapped to a 2-lane ChainSaw. In this table, every row corresponds to a chain. The fields in a row respectively store the lane mapping, the register bank locations for live-ins, and the children chains. Since, both the number of live-in values and live-out values are limited to two, it is sufficient to specify two children for each chain. For example, the row for $C1$ indicates i) the chain is mapped to execution lane 1, ii) the live-in values are stored at locations in register 0 and register 1 in the Live-in bank, and that $C2$ is its only child chain. The Chain Ready bits specify the chains mapped to the lane and whose dependencies are satisfied and ready for execution. When execution begins, all lanes are available. In the example, only $C0$ is ready to execute. Therefore, $C0$ is scheduled onto Lane0. When it finishes executing, its live-out values are present in register $OUT0$ and $OUT1$, both of which will be consumed by $C1$. The scheduler table entry for $C1$ gives the locations where these values must be routed to and the compiler inserts the appropriate move operations to terminate the chain. The bus routes these values, while $C1$ gets scheduled onto Lane1. Similarly, when $C1$ finishes, $C2$ is scheduled onto Lane1.
Table 6.3: Scheduler Table

<table>
<thead>
<tr>
<th>Chain</th>
<th>Lane</th>
<th>Live-in 0</th>
<th>Live-in 1</th>
<th>Child 0</th>
<th>Child 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>C0</td>
<td>Lane 0</td>
<td>-</td>
<td>-</td>
<td>C1</td>
<td>C1</td>
</tr>
<tr>
<td>C1</td>
<td>Lane 1</td>
<td>Reg 0</td>
<td>Reg 1</td>
<td>C2</td>
<td>-</td>
</tr>
<tr>
<td>C2</td>
<td>Lane 1</td>
<td>Reg 2</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

6.5.3 Instruction Issue

Figure 6.17 shows a chain example containing three instructions. The fetch-decode unit in a lane sequentially fetches, decodes and issues instructions from the instruction buffer. The FU presents a simple 4-stage pipeline with the following stages: fetch, decode, execute, write back register, Figure 6.18 shows pipeline stages for a single lane in Chainsaw. In Figure 6.19 shows overall format of proposed ISA.

Table 6.4 is an example of generated ISA binary for chain C1. The op field gives the operation to be performed. C1 has three operations in the following sequence: cvt(convert), sub(subtraction), and, sext(sign extend). There are five 1-bit fields in each instruction: IN0/1, WR, FWD, L/R, OUT0/1. The FWD indicates whether one of the operands is available through bypassing i.e. the operand is the result of the previous chain operation. This field is clear for the first chain operation and set for subsequent chain operations. IN0/1 indicates whether the instruction must read one live-in value from the IN0 register. For unary operations, this field is meaningful only if the operand is not available through value bypassing. For binary operations, this field is always meaningful because one of the inputs must be a live-in. Therefore, this field is set in the first instruction as the only input is a live-in residing in IN0. This field is clear for the subtraction because, although it consumes a live-in, the value resides in IN1. This field is meaningless for the last instruction because it does not consume any live-ins. L/R, indicates the ordering among the operands. Ordering is meaningless for unary operations. However, for binary operations that are
Figure 6.18: Single lane’s pipeline stages.

<table>
<thead>
<tr>
<th>Op Code</th>
<th>IN 0/1</th>
<th>WR</th>
<th>FWD</th>
<th>L/R Op</th>
<th>OUT 0/1</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVT</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>SUB</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SEXT1</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 6.19: Instruction fields: op=opcode, IN0/1: Operand consumes IN0 or IN1 live in value, WR: does instruction produce live-out, FWD: Forward value to subsequent instruction, L/R Op: operand order (is the IN the left or right operand of the instruction), the other operand is the forwarded value. OUT0/1=write output to register OUT0 or OUT1.

not commutative (e.g. subtraction), this field is necessary. Therefore, the subtraction defines this field whereas the other instructions do not. The WR determines whether the instruction produces a live-out. This flag is only set for the subtraction because it emits a live-out that is consumed by C2. The flag OUT0/1 determines which of the OUT registers the value must be written to. It is set if the destination is OUT0, and clear if the destination is OUT1.

Table 6.4: Instruction fields

<table>
<thead>
<tr>
<th>Op</th>
<th>IN 0/1</th>
<th>WR</th>
<th>FWD</th>
<th>L/R</th>
<th>OUT 0/1</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVT</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>SUB</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SEXT1</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>0</td>
</tr>
</tbody>
</table>

6.5.4 Dynamic execution

Table 6.5 shows the execution sequence of chains belonging to Figure 6.17, in their respective lanes. For this example we assume a latency of one cycle for each operation, however, in the real example, these latencies can vary depending on the type of operations and also depends on memory response time for MEM operations. Based on Table 6.3 scheduler realised that C0 is ready to run at cycle 0. Therefore, it wakes up lane 1 and from Cycle 1, lane 1 starts to executing scheduled chain. At the cycle 3, lane 1 finishes executing C0. For the next cycle, scheduler updates the values. After finishing C0, C1 becomes free and ready to run. Therefore, for the next cycle scheduler wakes up lane 2. This procedure keeps continuing until scheduler realized that all the chains they have executed, and there is no other chain remained.

The scheduler is not limited by the order of the scheduled chains, i.e., since scheduler has the capability to search all the chain’s table so that it can pick chains in an order different from the initial order. Therefore, chains can run out of order and performance gets to improve.
6.6 Framework and Evaluation

6.6.1 Profiling

In order to extract desired DFGs from benchmarks, we use following steps. First, the applications are profiled using gprof which identifies the critical functions and the function call hierarchy. Based on the gprof profile, we identify top-level functions that consume the largest amount of execution time. However, it can be possible that some of the top-level functions are functions which only do read or write so that we exclude those functions from the rest. We then inline all functions called by this identified function in a bottom–up recursive manner. The generated LLVM–based infrastructure identifies paths[16] in the function. In the next step we enumerate paths are profiled using large representative inputs (eg. ref for SPEC benchmarks). At this step, we also filter out some of the paths which don’t have desire characteristics in order to run on an accelerator. These Paths include unacceleratable features such as external library calls or memory allocation.

We profile the workloads to understand how much “coverage” is provided by each path. We define the coverage of a path as the number of operations in the path times frequency of execution of that path, represented as a fraction of the whole routine. Table 6.6 summarizes the coverage of the top five ranked paths in each workload. On average the coverage provided by the top five traces is 69% (median 88%). The five highest ranked paths by coverage are selected for chain extraction.

<table>
<thead>
<tr>
<th>∑Cov.</th>
<th>Avg</th>
<th>Workloads</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-25</td>
<td>19%</td>
<td>sjeng, 401.bzip2</td>
</tr>
<tr>
<td>25-50</td>
<td>40%</td>
<td>blackscholes, bodytrack</td>
</tr>
<tr>
<td>50-75</td>
<td>64%</td>
<td>fluidanimate, freqmine, art</td>
</tr>
<tr>
<td>75-100</td>
<td>92%</td>
<td>h264ref, mcf, mcf, dwt53, namd, parser, soplex, gcc, gzip, equake, sphinx3, povray, hmmer, lbm</td>
</tr>
</tbody>
</table>

Table 6.6: ∑Coverage top five traces
6.6.2 Chain Extraction

In order to extract desired chains from the selected functions, we need to focus on basic blocks. But at the basic block granularity, there are no control dependences since basic blocks are terminated by branches. As a result, the blocks themselves are quite small in size, as our tool chain shows on average, there are 11 operations at each block and 150 at maximum for namd. However, the Chainsaw architecture relies on the extraction of longer chains to reduce energy overheads by internalizing communication. To extract larger chains, we use an approach inspired by dynamic just-in-time compilers. Therefore, we convert all branches in the previously described outlined functions to control flow assertions. The Chainsaw incorporates store buffers to export an atomic view of Chainsaw invocation. The average number of stores for all paths is 3 and the maximum is 25. 97% of profiled paths had less than 16 stores.

The dataflow graph of the control free outlined function is constructed by examining each LLVM instruction and its operand. Finally, we construct chains via the decomposition algorithms described in Section 6.4 to feed Chainsaw simulator. In order to compare Chainsaw simulator with CGRA and OOO cores we use unmodified dataflow graph as the basis of the timing simulation for both of the base lines. Our goal from comparing Chainsaw with CGRA and OOO cores is proof of the concept that using dataflow accelerators can not always be beneficial. Therefore, we chose an optimistic CGRA scheduling strategy which doesn’t take into account any constraints exist in CGRA accelerator.

6.6.3 Code Generation

The chained dataflow graph is derived from decomposing the original dataflow graph. After decomposing the original dataflow graph, the compiler needs to add complimentary information to the binary so that Chainsaw simulator can schedule the chains, keep track of chains’ dependencies and also communicate with memory. Therefore, the compiler adds the following information to the program binary:

- Markers to indicate regions that carry Chain information
- Start addresses and lengths of chains in the region
- The dependencies among the chains belonging to the region
- The stripped chain instructions (13 bits).

6.6.4 Simulation

We have developed a detailed cycle-accurate simulator\(^1\) that models the host core, the Chainsaw accelerator, and spatial fabrics of parameterizable size. The host OOO core pipeline is modeled

\(^1\)The following six workloads crafty, freqmine, sar-back, sar-pfa, streamc, swaptio. were not supported on our simulator.
using MacSim [107]. MacSim is a heterogeneous architecture simulator, which is trace-driven and cycle-level. It thoroughly models architectural behaviors, including detailed pipeline stages, multithreading, and memory systems. However, we modified the Macsim simulator memory system and replaced the existence one with Ruby [133]. Ruby is a component of the GEMS framework which implements a detailed simulation model for the memory system. It models inclusive/exclusive cache hierarchies with various replacement policies, coherence protocol implementations, interconnection networks. We assume that Chainsaw accelerator communicates with the OOO core via the L1 cache.

**Chainsaw simulator:** Chainsaw simulator has been designed in a modular fashion. It has four major components which they communicate with each other during simulation I) Graph processor, II) Scheduler III) Execution Unit and IV) Memory system. Figure 6.20 shows an overview of the simulator.
Graph: The graph module decomposes the original DFG using either of algorithms described at chapter 6.2. The output of this part is a collection of chains. The input for the graph module is original DFG which contains LLVM IR, instruction ID, memory addresses and dependencies between each instruction. The simulator has a configuration file using that user can set decomposing algorithm for decomposing the original DFG. Chainsaw simulator uses three different decomposing algorithms, basic decomposition, MAXILP and MAXSIZE. Because of modular nature of the design, each module have to guaranty set of specific properties. The graph module guarantees two important features. First, it guarantees that there is no loop in the generated chain graph. Having loop in the chain graph means the scheduler cannot schedule the chains on the lanes in way which Chainsaw can finish the execution. Second, it guarantees that the number of live-ins and live-outs for each chain is limited by $N$. $N$ is a configurable variable in the configuration file. The output of the graph module is chained DFG which has all the information about the dependencies between chains.

Scheduler: The scheduler is responsible for tracking status of the both lanes and chains. The input for the scheduler module is the decomposed DFG. Scheduler gets the chained DFG and schedules the chains on a set of lanes. The scheduler considers multiple factors for scheduling the chains. Factors affecting the scheduling process consists number of lanes, number of chains, number of available instructions for each lane and size of each chain. Based on these variables the scheduler module schedules chains on each lane. In the end, the scheduler has a table just like table 6.3. The scheduler keeps communicating with lanes during simulation. At each cycle of the simulation, the scheduler is responsible for monitoring dependencies of each chain. When a lane finishes running a chain, it sets a flag indicating the output of that chain is ready. Therefore, the scheduler needs to update the scheduling table and set all the chains that are dependent on the finished chain. Whenever the scheduler finds that there is a chain which all of the inputs are ready, then the scheduler marks the chain as a ready chain. However, since all the chains are pre-scheduled, the scheduler should keep track of each lane status. At each cycle, if the scheduler can find an idle lane which has a ready to run scheduled chain, it sends a signal to the lane and makes it activated. The scheduler keeps updating the table while it finds out all the chains they have finished. At this point scheduler send the finish signal to the main core.

Execution unit: The execution unit simulates execution lanes. Number of lanes is a user-defined variable which user can set the number inside the configuration file. Execution unit also has an MSHR to keep track of memory requests. As we described at chapter 6.5 each lane is a four stage in-order single issue core. Therefore, each lane has Fetch, Decode, Execute, Write-Back stages beside an instruction buffer. Each lane also has a status flag. The scheduler sets and unsets the flag base on the scheduled chain status. Base on the status flag, lane starts fetching from the instruction buffer. Execution time depends on the instruction. If the instruction is a memory type instruction, lane generates a request containing the type of memory operation, memory address,
and lane ID and send it to In-buffer. In-buffer serves memory requests at the end of each cycle and communicates with memory system using provided interface.

**Memory System:** Chainsaw simulator has an aggressive non-blocking interface with memory. To accurately model the host-accelerator interaction via the memory system, we capture a window of memory accesses before the accelerator invocation and warm up the caches. The memory accesses for host execution, and the accelerator is collected using Intel Pin [127]. All memory operations from the host are collected in a trace. Memory operations at the IR level may not translate to an x86 instruction in the binary for the accelerated path. Thus, IR level memory operations are marked in the binary during acceleration extraction in LLVM; the pin tool recognizes these accesses during tracing and dumps them to a separate accelerator trace. Each memory operation in the accelerator trace contains a unique identifier which maps it to a particular node in the dataflow graph. The Chainsaw and CGRA simulations use this trace to issue memory operations with addresses consistent with the host core.

**CGRA simulation:** We model a CGRA, a spatial homogeneous fabric accelerator similar to [77, 152]. To model the CGRA, we traverse the activity of the dataflow graph cycle-by-cycle, generating any requisite memory operations in a cycle and stalling the appropriate operations as necessary. At each cycle, each node at CGRA model can have three different statuses. It can be IDLE which means there is an operation scheduled on that node, but the input of the operation which coming from the node’s parents in the DFG are not ready. FREE is another status which means the node has finished the scheduled instruction. And the last status is MEM-IDLE which means there is a memory operation scheduled on that node, and the node has sent a request to the memory subsystem, and it’s waiting for the response.

Table 6.7 shows the characteristics of the architectures that we model.

**Power model:** To model the power consumption we need to take into account two factors: i) precise activities and ii) accurate power characterization of different components of the lane’s design. We adopt an event-based power model similar to Aladdin [181].

For modeling the memory system we considered a commercial register file and static RAM (SRAM) memory compiler.

### 6.6.5 Synthesis and Area Overhead

We designed the Chainsaw pipeline based on the RISC-V 4-stage IMAFD pipeline using our custom instruction encoding. For synthesis, we used the Synopsys design compiler (Vision Z-2007.03-SP5) 45nm technology library. To tease out the impact of the main design tradeoffs we fix the design parameters of a single lane. The primary parameter that influences the complexity or overhead of a lane is the number of instruction buffer entries supported; our evaluation assumed 16 instructions per lane. Given that each instruction requires 13 bits (see Section 6.5) the entire instruction buffer
Host Core

<table>
<thead>
<tr>
<th>Core</th>
<th>2 GHz, 4-way OOO, 96 entry ROB, 4 INT, 4 FPU, INT RF (64 entries), FP RF (64 entries)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>32 entry load queue, 32 entry store queue</td>
</tr>
<tr>
<td>L1</td>
<td>64K 4-way D-Cache, 3 cycles</td>
</tr>
<tr>
<td>LLC</td>
<td>4M shared 16 way, 8 tile NUCA, ring, avg. 25 cycles. Directory MESI coherence.</td>
</tr>
<tr>
<td>Memory</td>
<td>200 cycles.</td>
</tr>
</tbody>
</table>

Accelerators

<table>
<thead>
<tr>
<th>CGRA8</th>
<th>8 × 8 function units or</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chainsaw</td>
<td>Lane sizes:1,2,4,8,16 #instructions/lane:256,128,64,32,16.</td>
</tr>
</tbody>
</table>

Energy Parameters (Static and Dynamic)

| CGRA8 Lane | Mcpat [121]; ARM A9 2Ghz template. |
| Chainsaw | CGRA Network (650 fJ/switch), Function units (510 fJ/INT,1500 fJ/FP) |
| Chainsaw Comm. | Instruction buffer (16 entries, 120 fJ/read, 220 fJ/write), Decode (100 fJ/instruction) |
| Chainsaw | Pipeline forwarding (250 fJ), Live-in Registers (Read: 180 fJ and Write: 250 fJ), Bus (1100 fJ/access) |

Table 6.7: System parameters

in each lane requires 26 bytes and is single ported since the lanes are single issue. We picked this parameter to minimize the fetch overhead. The largest components in the lane design are the register banks which directly correlate with how many chains we would like to support and the number of chain dependencies which are the only communication that requires registers (see Figure 6.12). The sample configuration we synthesized and simulated supports 8 chains; given the maximum fan–in for many workloads is 1–2, we assumed a total of 2 × 8 registers per lane (i.e., 16 x 32 bit = 64 bytes). The register banks are dual ported to supply the chain registers in a single cycle. The scheduler, unlike OOO, consists of only one wakeup component; it does not require any tag matches since the compiler explicitly encodes the dependent children; each entry in the chain wakeup flag is 9 bits (1 bit for ready and two 4-bit live–in register ids). The chain wakeup matrix has as many entries as the number of chains per lane; (8 × 9 bits). Overall, the per lane overhead ≃ 100 bytes (64 bytes for the 16 entry register bank, 26 bytes for the 8 entry instruction buffer, and 9 bytes for the chain wakeup flag). Overall, we found that the area for a 16 lane design (≃ 1.6 KB) is 0.21 mm² (including the functional units).
6.6.6 Evaluation

For the rest of this chapter we evaluate the performance and the power of the Chainsaw described in chapter 6.5. The evaluation is performed using the infrastructure described in Section 6.6.

6.6.7 Performance Comparison

To understand the performance characteristics of the Chainsaw architecture, we compare Chainsaw8 and Chainsaw16 to a 4-wide OOO processor, an IDEAL-CGRA i.e. an unbounded CGRA that is only limited by the application\(^2\) ILP. Figure 6.21 shows the performance of the OOO, Chainsaw8 and Chainsaw16 normalized with respect to the IDEAL-CGRA. Higher bars are better, i.e., performance is closer to an IDEAL-CGRA.

For 10 of the 20 workloads (gzip, art, gcc, namd, soplex, hmmer, h264ref, lbm, bodytrack, and, fluidanimate), we find that performance of OOO < Chainsaw8 < Chainsaw16. In these benchmarks, the unchained DFG ILP (4–23, see Table 6.2) is greater than the width of the OOO core. Therefore OOO is unable to exploit all the available ILP. The chained DFG ILP in the range 3–16, thus performance improves as the Chainsaw architecture can exploit more ILP. Of these, there are three workloads (gcc, hmmer, and, lbm) with ILP in the range 6–16, for which Chainsaw16 performs significantly better than Chainsaw8 due to increased hardware resources.

For 6 workloads (mcf, equake, parser, bzip2, sphinx3, and, dwt53) the performance of the Chainsaw8 was the same as Chainsaw16. For all these workloads the chained DFG ILP is less than 8. Thus Chainsaw16 is over-provisioned. In 3 of these workloads (mcf, equake and dwt53) the performance of the OOO is better than Chainsaw architectures. The chained ILP for each of these is less than the unchained ILP (see Figure 6.12) as well as the average memory-level parallelism being lower than 3. For the remaining workloads, the Chainsaw is able to exploit more MLP than OOO and improve performance.

For the remaining 4 workloads (mcf, povray, sjeng, blackscholes) we see an interesting pattern where the performance of the OOO is higher than Chainsaw8 but lower than Chainsaw16. In blackscholes, the OOO is marginally better than Chainsaw8 as the CPI is 8% less on average while the ILP is \(\simeq 4\). However, Chainsaw16 is significantly better than both the OOO and Chainsaw8 (upto 21%). Chainsaw16 is able to enqueue more long latency floating point operations in parallel than Chainsaw8. The number of idle cycles, cycle count of ready chains blocked due to resource contention, is \(3.4 \times\) for Chainsaw8 when compared to Chainsaw16. For the remaining 3 workloads, we see a common pattern of wide (>8) issue potential in the chain graph with long latency memory operations. The performance of the OOO with respect to Chainsaw8 is marginally better as the average ILP is only 3.4. The performance of Chainsaw16 is significantly better (average 9%) as it is able to overlap the long latency memory operations.

\(^2\)We do not include the CGRA8x8 in this comparison as the fabric size restricts the number of operations. Of the 20 workloads we study, only 9 execute without reconfiguration (< 64 ops, see Table 6.2).
Figure 6.21: Performance of various architectures normalized to the IDEAL performance. We don’t plot CGRA8 on this plot since in all applications the CGRA8 attains the performance of IDEAL. Higher is better.

To summarize, in 8 of 20 workloads the performance of the Chainsaw architectures is within 90% of an unbounded dataflow. Across all workloads, the performance of Chainsaw16 is 81% and Chainsaw8 is 73% of an unbounded CGRA. Chainsaw architectures outperform an OOO core by 20.3% on average (17 out of 20 applications) with most significant improvements for gcc and hmmer.

6.6.8 Energy Comparison

Chainsaw and CGRA architectures represent two accelerator designs with different tradeoffs on various components of the total energy consumption. We first discuss dynamic power components, followed by static power. CGRA8 statically maps ops to the FUs, whereas Chainsaw incurs fetch-decode overhead per instruction. Chainsaw also incurs energy costs on chain activation and chain completion. On the other hand, Chainsaw attempts to minimize data movement, while CGRA8 moves data for every producer-consumer instruction pair. With regard to static power, CGRA8 is expected to have significant static power costs due to larger fabric size which leads to more idle cycles. Chainsaw improves utilization and limits static power.

**Dynamic Energy:** Figure 6.22 shows the dynamic energy consumed by CGRA8 and Chainsaw8 normalized to functional unit energy. Across workloads the energy of the OOO (numbers above each bar) is $\approx 5 \times$ the functional unit energy. The least is found in blackscholes (50% floating point operations) and the most in gzip (only INT ops, no memory). The dynamic energy of the consumption of the CGRA8 is $\approx 3.3 \times$ the functional unit energy where the least is blackscholes. The most however is gcc where the link energy dominates due to the high connectivity of the dataflow graph. Similarly for the Chainsaw8, the average is $\approx 2.8 \times$. The net dynamic energy consumption is reduced via internalizing communication within chains.
Communication Costs: Figure 6.23 shows the total energy expended by Chainsaw in communication, normalized to CGRA8 communication energy. Each benchmark is represented by a stacked bar, which shows the relative proportions of energy spent in the bus, pipelining forwarding, and register \((INs/OUt s)\) communication by Chainsaw. The CGRA8 communication energy includes the fabric overhead and the latches at each PE. The reduction in communication cost is a motivation for the Chainsaw accelerator, because fetch-decode, which is the other component of dynamic power, is an essential component of von neumann style architectures. The chart shows that Chainsaw improves communication cost significantly, on average 38%. The variation in energy reduction is well illustrated by Figure 6.10.

Although Chainsaw reduces bus events as far as possible, bus communication still consumes majority of the power. The dynamic energy cost of registers is related to the bus cost as each inter-chain dependence scheduled on a different lane triggers a bus access as well as a register access. Note that a bus access is \(3.5 \times\) as expensive as pipeline forwarding. An inter-lane register write
Figure 6.23: Communication Energy Breakdown (Chainsaw8). Localized computation in CHAINSAW8 reduces communication costs significantly compared to dataflow architectures. The dominant energy component in both CHAINSAW8 and CGRA8 is the network transfers required between producer-consumer operations.

and then read is 44% more than forwarding. On average, forwarding events are 21% more frequent, with the highest occurrence in hmmer ($2.2 \times$). For 3 out of 20 workloads (equake, blackscholes, fluidanimate), the bus events are more frequent (average 30%) due to small chain formation (average size $< 2$ ops) coupled with greedy scheduling. The scheduling strategy is tuned for performance which seeks to schedule chains on free lanes to extract maximal ILP.

Overall, the dynamic communication energy is 38% lower for Chainsaw8 compared to CGRA8 due to conversion of link transfers to internal pipeline forwarding.

**Static Power:** Figure 6.24 shows the static power consumption normalized to the static power consumption of the CGRA8. The static power component is broken down into two components: the IDLE power and the FREE power. In lane or PE based execution such as Chainsaw and CGRA, a particular hardware resource might be inactive due to either the instruction assigned to the PE/lane has not been woken up yet since the producer instructions have not completed (IDLE), or the PE/Lane may have completed all the instructions assigned to it (FREE). The CGRA fabric for
Figure 6.24: Static Power. CGRA8 and CHAINSAW8 normalized to CGRA8. IDLE: the static power expended while waiting for scheduled operations to be ready to run. FREE indicates the static power due to over-provisioning resources compared to the available ILP i.e., the PE or Lane does not have any instruction scheduled to execute.

Instruction-granularity accelerators [79] is scaled based on the number of operations to be accelerated while the Chainsaw is scaled based on the instruction parallelism available. Hence, in many cases the CGRA is excessively provisioned and is underutilized unless there is data parallelism to be exploited. This leads to an interesting case where the static power in the CGRA8 (64 units) is dominated by the FREE power; the FREE power may be curtailed by power-gating the PEs or the execution lanes. Chainsaw8 improves overall utilization and consumes much less FREE power but may introduce contention for the lanes or PEs by mapping multiple operations onto the same PE; this leads to an overall increase in IDLE power due to chain operations being stalled due to other unrelated chains occupying the lane. Note that both CGRA and Chainsaw will suffer from IDLE power since they statically map the operations to the resources.

Chainsaw8 reduces overall static energy by \( \simeq 21\% \). From Figure 6.24, we see 11 of the 20 applications reduce energy by 20% to 40%. In equake, povray, hmerm, sjeng and lmb we see little to no reduction in overall static energy consumption. In all these applications a large fraction of the operations are long latency operations (FP or memory) thus increasing the IDLE’ness of the Chainsaw8. For 15 out of 20 workloads, there is more IDLE’ness in the Chainsaw architecture. In the remaining 5 workloads (namd, sjeng, blacksholes, bodytrack, fluidanimate), the IDLE’ness of the CGRA8 is more than the Chainsaw as even an unconstrained mapping leaves resources available due to the number of operations.
6.6.9 Chainsaw vs. SIMD

Most modern processors include ISA extensions for vector operations like SSE/AVX, altivec or NEON, which are designed to accelerate single thread performance by exploiting data-level parallelism (DLP). These SIMD operations provide energy efficiency by reducing front-end cost of the processor. In the other word they provide energy efficiency by reducing per-instruction overheads, and performance by explicitly defining parallel operations. From energy perspective, Chainsaw can’t do better than SIMD. Even though, in Chainsaw we have reduced fetch and decode cost by reducing the size of ISA, but still for each instruction we need to fetch and decode instruction from an instruction buffer. As a result in cases which we need to apply single instruction on multiple data, SIMD can save more energy by reducing number of needed fetch and decode.

Chainsaw has comparable performance in compare to SIMD. SIMD performance comes from explicitly defining parallel operations. While there exist DLP in the program Chainsaw is also able to exploit the parallelism between operations and run them in parallel. As a result, we can expect the same performance between Chainsaw and SIMD.
Chapter 7

Framework Release

This chapter details the release of software developed to conduct the research presented herein. All code is distributed via repositories on Github under the Simon Fraser University Architecture Group organization.

7.1 µIR Simulator

µIR-sim is a new simulation environment that improves software and hardware integration, and simulation accuracy compared to functional simulation. This framework integrates the hardware development process into the software stack from the beginning, allowing features to be incrementally implemented and evaluated as workloads evolve. Under this environment, the hardware description is the actual specification. This reduces the burden of maintaining consistency between the specification written usually in a higher language such as C/C++ and the actual hardware design described in a language such as Verilog. Moving to µIR-sim will allow us to have a unified hardware-software specification, and invite more contributions to modify different layers of the stack.

Moreover, this integration provides more accurate performance feedback, i.e. clock cycles, compared to the traditional functional model of a hardware accelerator. This is because µIR-sim is based on an open-source hardware simulator called Verilator [186], which compiles Verilog designs down to C++ classes for cycle-accurate simulation.

Language: Python, C++ and Chisel
Release: github.com/sfu-arch/muir-sim
License: BSD 3-Clause License

7.1.1 µIR-sim Design

µIR-sim uses Verilator to integrate µIR generated accelerators into overall system design and provides flexibility in the hardware language used to implement these designs. For example, one could
use OpenCL, C/C++ or Chisel3 – we mainly rely on Chisel3 – to describe an accelerator design that would eventually be compiled down to Verilog, since it is the standard input language for FPGA/ASIC tools. To manage runtime process we use Direct Programming Interface (DPI), supported by Verilator. DPI is part of the Verilog standard and a mechanism to support foreign programming languages.

We leverage these features available in Verilator to interface hardware designs from upper layers in the microR stack such as drivers, runtime, etc (Code is borrowed from Tsim). In fact, we have developed all the glue layers to make this happen, including:

**DPI module:** The dpi_module.cc is in charge of loading the shared library libHW.so that contains the hardware accelerator and the Verilator execution function. As stated earlier, Verilator is used to compile the hardware accelerator from Verilog to C++. Additionally, the DPI module provides an API that can be used by drivers to manage the accelerator by writing/reading registers and terminate (exit) the simulation.

**Verilator execution function:** This function is called tsim.cc and it is used by Verilator to instantiate the accelerator, generate clock and reset signals, and dump simulation waveforms when it is enabled. The tsim.cc also contains function pointers to DPI functions which are implemented in the DPI module dpi_module.cc. This adds greater flexibility because the behavior of DPI functions can be modified by upper layers in the stack.

**Hardware DPI modules:** Normally, a hardware accelerator interface can be simplified in two main components, one for control and another for data. The control interface is driven by a host CPU, whereas the data interface is connected to either external memories (DRAM) or internal memories in the form of scratchpads or caches. There are two hardware modules written in Verilog implementing these two interfaces called VTAHostDPI.v and VTAMemDPI.v. Accelerators implemented in Verilog can use these modules directly but we also provide Chisel3 wrappers BlackBox for accelerators described in this language.

**Toy accelerator example:** To showcase the interaction between all of these components, we implemented an Add-by-one accelerator, in both Chisel3 and Verilog, together with a software driver called test_driver.cc. Also, we provide cmake scripts for building everything automatically and a config.json file for managing accelerator and simulation options.

Finally, the following snippet shows how a python wrapper design simulation, based on the toy example, can invoke the simulation:
```python
import numpy as np
import platform
dsim

def test01(a, b):
    return a * b

if platform.system() == 'Linux':
    hw_lib_path = './hardware/chisel/build/libhw.so'
elif platform.system() == 'Darwin':
    hw_lib_path = './hardware/chisel/build/libhw.dylib'

events = dsim.sim(pts = [ ], vars= [5, 3], numRet=1,
                   numEvents=1, hwlib = hw_lib_path)

print("Cycle: " + str(events[0]))

if events[1] == test01(5, 3):
    print("Success!
Ret: " + str(events[1]))
else:
    print("Failed!
Ret: " + str(events[1]))
```

7.2 µIR Library

µIR-lib is a library of hardware components for auto generating highly conﬁgurable parallel dataflow accelerator. The µIR library repository contains code that is used to implement library modular hardware components to build hardware accelerators. Hardware generation is done using Chisel, a hardware construction language embedded in Scala. The µIR-Lib code base is itself factored into several Scala packages. Some of these packages provide Scala utilities for generator conﬁguration, while others contain the actual Chisel RTL generators themselves. µIR-lib provides the implementation of the following hardware units:

- A set of highly conﬁgurable and parameterizable computation nodes.
- A set of control units to support arbitrary control path.
- A collection conﬁgurable Memory structures like Cache, Scratchpad memory, and such.

Language: Chisel, C++, Python
Release: github.com/sfu-arch/muir-sim
License: BSD 3-Clause License.
A set of standard flexible set of junctions and interfaces to connect different pieces of the design.

7.2.1 \(\mu\)IR-lib components

The \(\mu\)IR library repository contains code that is used to implement library modular hardware components to build hardware accelerators. Hardware generation is done using Chisel, a hardware construction language embedded in Scala. The IR-Lib code base is itself factored into several Scala packages. These packages are all found within the \texttt{src/main/scala} directory. Some of these packages provide Scala utilities for generator configuration, while others contain the actual Chisel RTL generators themselves. Here is a brief description of what can be found in each package:

- **Accel**: This RTL package contains all the accelerator code used to wrap a dataflow scala file.

- **Node**: This RTL package resembles LLVM IR instruction nodes in our design so that we can target arbitrary dataflow of computation nodes from our input software IR. All these nodes use the Handshaking interface from our interface package to talk with other nodes. The logic within each node can vary from a simple ALU operation like addition to more complicated operations like memory address calculation.

- **Control**: In this package, we have contains our control logic implementation for supporting arbitrary dataflow between IR-lib’s nodes and loop nodes.

- **Concurrent**: This utility package provides Scala interfaces for configuring a generator via a dynamically-scoped parameterization library.

- **DNN**: This RTL package contains our computation nodes to implement application-specific accelerators for dnn workloads. For instance, we have different implementation of Systolic arrays in this branch. Another type of computation nodes which exist in this package is our Typed node computation nodes.

- **FPU**: This RTL package provides wrappers around Floating point operations to be integrated with IR-lib design. At this moment, there are two different wrappers in this package. One is a warper for Berkeley’s hard floating point unit. Second, is a wrapper for embedding Alter’s IP cores in our design during the FPGA mapping process.

- **Interfaces**: This package contains all the different definition of our interfaces between nodes. These interfaces connect different types of the node. For instance, for connecting two compute node Databundle is provided, alternatively for connecting control signals, Control-Bundle is provided.

- **Junctions**: This RTL package includes the implementation of generic 1: N, N:1, and M: N connections. All the memory operations in the task are time-multiplexed over the junction.
There are different implementation of junction in this package. For instance, one possible implementation of junctions is a static tree network or a local bus. We have parameterized the junction implementations in IR-lib so that the designer can control the physical network that the junction is lowered into.

- **Memory**: This RTL package contains our different memory unit implementation of memory units. These units vary from the different implementation of Caches to Scratchpad memories. There is also a different implementation of IR-lib memory controllers for a different type of data like Scalar, Tensor2D.

### 7.2.2 µIR AWS

µIR-AWS contains µIR hardware development kit (HDK) and associated software drivers to run µIR accelerators on Amazon F1 instances. We have developed AWS compatible wrappers that connects µIR accelerators to AWS FPGAs.

### 7.2.3 µIR RISC-V

µIR-Lib uses api-config-chipsalliance project to have a centralized system for configuring the overall accelerators parameters. Using this library in our design enables us to integrate µIR accelerators with RISCV faster. We use cake pattern design for integrating µIR configuration parameters with host core parameters. µIR-RISCV contains µIR hardware development kit (HDK) and associated software drivers to run µIR accelerators on RocketChip, an open-source implementation of RISC-V cores. We have developed RISC-V compatible wrappers that connect µIR accelerators to RISCV Cores.

**Language**: Chisel, C++, Python

**Release**: github.com/sfu-arch/muir-lib

**License**: BSD 3-Clause License.

### 7.3 µIR Generator

µIR-Generator is a tool to generator hardware accelerators from software programs. µIR-Generator uses µIR as an intermediate representation to generate hardware accelerators. Currently, µIR-Generator supports C/C++ and Cilk programs. The input to our generator is a C/C++ or Cilk program, and the output of a Scala file that contains the accelerator’s design in Chisel. However, the generated Chisel design can not be compiled as standalone and needs to be linked with µIR-lib. To simulate and test the design, someone can use µIR-Simulator project. To extract SoC shell, the designer still needs to use µIR-lib.
7.4 Chainsaw Sim

Chainsaw-sim simulates our Chainsaw accelerator to execute chain instructions. The repository contains software compiler that decomposes program traces, forms chain instructions and schedules them on execution lanes. The input to the simulator is the decomposed and scheduled instructions, simulating the traces. Chainsaw-sim simulates 3-stage pipeline in-order cores that fetch, decode and execute the instructions which the compiler generates in the first step. To simulate the host core, we have connected Chainsaw-sim to Macsim [107] simulator. We also modified MacSim memory model and replaced it with Ruby [133] to accurately simulate the memory behavior of the program traces.
Chapter 8

Summary and Future Work

In this chapter, I review my finds on the automation of designing hardware accelerators using intermediate representation (Section 8.1), outline future directions and further challenges (Section 8.2), and eventually, I reflect on my research works and comment on them (Section 8.3).

8.1 Summary

With the end of Dennard scaling and the slow death of Moore’s Law, the period of free performance improvements on conventional CPU architectures is coming to a close. As it does, computer architects and computer scientists alike are looking to more specialized hardware architectures to continue improving runtimes, throughput, and energy efficiency for performance critical applications.

While reconﬁgurable architectures like FPGAs are a natural ﬁt for these specialized hardware designs, their low-level programming models have historically limited their adoption. One of the commonly-accepted solutions for closing this abstraction gap between hardware and software languages is to raise the level of abstraction in the design process. FPGA vendors like Xilinx and Intel and other companies that offer FPGAs as part of their cloud services like Amazon are aware of this programmability challenges. To address this, they usually offer two mainstream frameworks to program their FPGAs. For instance, Amazon AWS provides a Software Development Kit (SKD) and Hardware Development Kit (HDK) for their customers to program the F1 FPGA instances. FPGA HDK is for experienced hardware designers who want to design their fully customized accelerators and have full control over their design. As Amazon has stated in their programming guide [5]: "A fully custom hardware accelerator may take months to develop and developer must be familiar with".

Alternatively, Amazon offers FPGA SDK for non-experienced designers to compile their C/C++ and OpenCL codes into the FPGA as kernels, and use OpenCL APIs to pass data to FPGA. Usually, Software developers with no FPGA experience will ﬁnd a familiar development experience that supercharges cloud applications. Nonetheless, SDK framework comes with limitations; the primary
limitation is the ad-hoc mixture between hardware and software, which has made it difficult to adopt when implementing optimized hardware designs.

In this thesis, I proposed an alternative; decouple the representation used for accelerator microarchitecture and hardware optimizations from the functional behavior specification. I implemented four pieces to make this goal happen:

1. A well-defined system abstraction level, to design hardware accelerators. We proposed $\mu IR$, a new intermediate representation for representing the accelerator microarchitecture structure. $\mu IR$ is a structural graph that explicitly specifies the accelerator’s microarchitecture components and orchestrates data movement between the different components. I designed $\mu IR$ as a hierarchical graph. The first level of $\mu IR$ captures the system-level behavior of the accelerator. At this level, each task node represents an asynchronous execution block concurrent with other nodes in the graph. The edges between nodes represent the memory modules across these tasks. A task block is analogous to a closure (unlike a function call) in software that takes arguments and produces a set of values after running to completion. Representing the accelerator as a pipeline of asynchronous task blocks has two benefits i) it avoids centralized control stores and stall signals in the hardware ii) it can implement arbitrary heterogeneous parallel patterns (including nested loops and recursion). The $\mu IR$ abstraction makes the design more effortless for both localized and global transformations to optimize the accelerator microarchitecture. The multi-stage approach encourages a clear demarcation of behavior optimizations (e.g., loop unrolling), microarchitecture optimizations, e.g., memory banking), and RTL optimizations (e.g., FPGA-specific SRAMs).

2. A well-known component of a particular abstraction level to optimize the accelerator’s design. I created an optimization framework, $\mu opt$, that decouples microarchitecture optimizations from the lower RTL. $\mu opt$ helps designers realize optimizations as an iterative pass of the $\mu IR$, without having to modify RTL. $\mu opt$ optimization passes can be automatically applied to different accelerators, and multiple passes can be stacked for a specific accelerator.

3. Having a precise semantics to move from a higher level of abstraction to a lower level. I developed a synthesizer to synthesize hardware accelerator structural description ($\mu IR$) from behavior description (software). I showed that using C/C++ programming languages, I could define accelerators behavior in software language and benefit from their high-level abstractions such as support for control and defining loops to extract fine-grain parallelism. I showed C/C++ languages can not always exploit all the parallelism exist in a program, and to build high-quality accelerators, I need a way to expose coarse-grain parallelism to the backend compiler as well. To this end, I used Cilk programming model to capture high-level parallelism behavior.

4. A new abstraction to represent fused instructions. I observed that using the dataflow model suffers from two weaknesses when they are underutilization. One is the cost of high static
power when there is no functional units reuse. Second, the dynamic cost that logic pays to transfer data between PEs can be a bit portion of the overall accelerator. Hence, I proposed a new Von Neumann style accelerator to address these two challenges. Using the Von Neumann style, at runtime, the architecture reuses function units in the design and reduces the static power in the circuit. Another benefit is that using Von Neumann accelerators localizes data reuse in the core. Therefore, the core does not need to spend the expensive cost of data transfer between PEs.

I conclude that µIR is an attractive intermediate representation for not only building hardware accelerators from imperative software languages like C/C++ but also it can be a backend for software DSLs like Halide and Tensorflow. µIR is extensible that can support complex data types, not only than scalar and vector operations. As a showcase, in section 4.6.3, I showed a designer could embed domain hints in µIR and benefit from specific domain optimizations. Finally, µIR allows the designers to capture their ideas as a set of optimization passes on µIR graph, and it will enable designers to reuse their optimizations on different applications without refactoring source codes.

8.2 Future Directions and Further Challenges

Although this dissertation has mostly focused on the design of µIR abstraction and how the compiler is organized, I believe that µIR abstraction opens up new opportunities for new dataflow execution model, debugging of hardware accelerator and designing domain specific accelerators that I did not explore them in this dissertation.

The current µIR compiler has a number of areas where it can be further expanded and improved. In below, I look at some of these ideas.

8.2.1 How it is possible to debug µIR?

One of the crucial advantage of µIR compare to traditional RTL design techniques is the allowance for faster development of hardware accelerators than traditional RTL design. However, verifying the correctness and functionality of HLS-produced designs remains a significant challenge. Once a bug is exposed, designers must trace signals, co-relate them with RTL, and walk-back simulation cycles in a repetitive manner. The challenge is further exacerbated when the hardware machine model bears no resemblance to a software program.

I find that the core challenges in debugging µIR are: i) how to understand the execution flow and causal relationship of events in the microarchitecture from bit-level signals I observe in the hardware, ii) how to understand what software specification and µIR knobs led to the particular microarchitecture. To address these challenges, I believe the implementation of an iterative debugging framework designed to integrate with µIR can help to investigate these bugs. This new debugging extension to µIR can help the user to understand the underlying reason and cause that the hardware implementation of a program does not result in the same outputs as the software. The core hardware in the debug extension can be a new debug core, auto-synthesized in conjunction
with the core accelerator’s microarchitecture. This extension can be a separate hardware module that serves to monitor the signals as the core accelerators are running and log them in memory for post-processing. There are three key main benefits in this approach i) the debugger hardware is automatically generated as the components based on the internal representation of the microarchitecture within the HLS compiler, ii) the monitored components are iteratively refined based on the post-processing of the logs that indicate the potential site of the bugs, and iii) I correlate the information in the logs with software run on the CPU (the golden model) and semi-automate the process of narrowing the bug.

8.2.2 How else can \(\mu IR\) support more efficient computation kernels?

In section 4.6.3, I demonstrated \(\mu IR\) is not only limited to scalar operations, and it can support higher-order ops, such as Matrix, Vector and mix of these types at microarchitecture level and it showed the potential benefit of supporting a library of BLAS operations in \(\mu IR\). By supporting such higher-order ops, the high-level compilers can focus on optimizing at higher-level and relying on the backend to support such operations efficiently. Supporting higher-order ops in the backend helps the domain compilers optimize the input program at a high level and lower the optimized program to these function calls.

To support higher-order ops in \(\mu IR\) two areas can be more explored. The first one is how to autogenerate the higher-order ops efficiently. Recently, there are works in autogenerating matrix multiplication based on a systolic array architecture [70, 33]. These works focus on generating different systolic arrays base on specific parameters. Currently, these are limited in stand-alone accelerators, and there is no integration of these frameworks with other HLS tools. The integration of these techniques with \(\mu IR\) framework and exposing the APIs to the compiler can be an interesting work to be expanded. Essentially, \(\mu IR\) can be used as a host language to include different hardware DSL languages, to generate different pieces of Hardware, and use \(\mu IR\) to glue these DSLs together.

The second area is how to make a pipeline of these specialized kernels. To pipeline these kernels, these questions need to be answered: 1) The kernels can have different data shapes (e.g. Matrix versus Vector), how to connect these kernels? 2) how to schedule and tile the kernels? 3) how to stream or buffer the data between these kernels when the producer and consumer have a different rate. 4) how to make these kernels scalable and satisfy: data access pattern, resource constraint and resource utilization. Autogenerating of memory system is another framework is another piece that needs to be added to \(\mu IR\) and improve by defining design space.

8.2.3 How else can \(\mu IR\) support better scheduling?

In section 3.2, I showed \(\mu IR\) supports dynamically scheduled dataflow execution model. I briefly explained the execution model and how \(\mu IR\) supports arbitrary control flow in section 3.2.7. However, I did not investigate in-depth on efficiently supporting arbitrary control flow. One of the powerful ideas that dynamic dataflow can support is the speculative execution model.
```cpp
for (int i = 0; d < x; i++){
    d = a[i] + b[i];
    c[i] = d;
}
```

**Figure 8.1:** Limitations of static scheduling

```
Iter (1): a[0] = 50.0; b[0] = 30.0
Iter (2): a[1] = 40.0; b[1] = 40.0
```

**Figure 8.2:** Example of nonspeculative versus speculative execution. In nonspeculative execution, the compiler needs to serialize the loop iterations and can not pipeline the loop because the loop termination condition is dependent on data calculated on each iteration. In contrast, in speculative execution, the loop can be pipelined independently from existing data dependency.

Figure 8.1 shows an example that the loop header has data dependency with existing computation inside the loop body. The standard HLS compilers that try to statically schedule the circuit, in this case, takes a conservative decision and serialize the loop iterations. I have shown an example of runtime of the conservative schedule. However, in speculative execution the loop body can be pipelined with each iteration, independent from the loop header and at the time that the loop terminator operation has been resolved the execution can either commit or discard the results. In this way, the speculative execution can fully utilize the pipeline and keep the pipeline busy in the presence of data dependency.

µIR uses dynamic dataflow model, and as I mentioned, this is a good match execution model with speculative execution. Currently, µIR dynamic dataflow model is conservative at the loop boundaries and in the existence of data dependency between loop iterations, it takes a conservative decision and waits until the loop dependencies are resolved and then continue the next iteration. However, because of the nature of dynamic dataflow, this conservative design decision can be relaxed, and the dataflow can start pipelining dependent iteration. Implementation of speculative execution and combine it with dynamic dataflow model, supported in µIR, can show exploit more parallelism in the existence of irregularity and data dependencies.
8.3 Reflection

In this section, I share some observations and opinions concerning hardware accelerators design. These opinions are based on five years of academic research on accelerator design and workloads, many conversations with system designers, and two years working in industry. These opinions are based on my experiences, and they are just that: opinions. Although these opinions are not supported by hard evidence, they may provide insight to those interested in accelerator design abstractions. Some of these opinions may be obvious and some may be controversial, but they reflect my current thoughts on accelerator design abstractions; I reserve the right to change my mind about them later.

8.3.1 FPGAs Have the Wrong Abstraction

In section 2, I revisited the existing hardware abstractions and techniques that are used to program reconfigurable architectures. I gave a brief overview of FPGA architectures and compared different frameworks. In this section, I like first to go back and redefine reconfigurable devices. Next, I would like to explain why programming FPGAs are still challenging after decades of research while we could solve the same problem in general purpose processors. In the processors, we defined ISA as a contract between software and hardware and then started building layers of abstraction on top of ISA. We built compilers for each level that allowed us to search in the space and lower the optimized higher abstraction to a lower abstraction.

8.3.2 What is an FPGA?

Unfortunately, I found that the community does not provide a consensus definition of FPGAs that all the researches and industry agree on that. Nevertheless, I believe three major definitions can cover most of the viewpoints:

Definition 1: An FPGA is a collection of transistors that the designer can wire up together to make any arbitrary circuit. This definition is analogous to defining FPGA as a breadboard at the nanometer scale. In this definition, having an FPGA is like taping out a chip, but the designer only needs to buy one chip to build lots of different designs and take an efficiency penalty in exchange.

This answer is the basic definition of FPGA that can answer a non-expert architect person. It is neither right nor a solid metaphor for how people use FPGAs in practice. It is not right because when the synthesizer program an FPGA does not rewire the FPGA, the synthesizer generates config bits for a 2D grid of lookup tables connected by a routing network some arithmetic units and memories thrown in for good measure. Nevertheless, FPGAs can fake an arbitrary circuit. Still, if the design is not optimized and meant to target FPGA design, the output is not an efficient design, and it is more like a software circuit emulator that fakes a design. This answer does not work metaphorically because it oversimplifies the way FPGA is used in reality. The next two definitions will provide a more comprehensive definition of FPGAs:
**Definition 2:** An FPGA is a more economical choice for making a custom chip to prototype and low-volume production. For example, if the goal is to build a router, the company can avoid the immense cost of taping out a new chip with shipping an off-the-shelf FPGA programmed with the designed functionality. As another example, if a research group is experimenting with designing CPUs, the group can use an FPGA as a prototype: they can build a real, bootable system around it for testing and snazzy demos before they ship the design off to a fab. Circuit emulation is the classic, mainstream use case for FPGAs, and it is one of the reasons they exist in the first place. The point of an FPGA is to take a hardware design, in the form of HDL code, and program cheap hardware that behaves like the ASIC the designer would eventually produce instead. It is unlikely to take the same Verilog code and make it work on an FPGA and real silicon, but at least it is in the same abstraction.

**Definition 3:** An FPGA is a pseudo-general-purpose computational accelerator. Like a GPGPU, an FPGA is a fit for offloading a certain kind of computation. While FPGA is harder to program than a CPU, but for the right workload, it can be worth the effort; an efficient FPGA implementation can offer orders-of-magnitude performance and energy advantages over a CPU baseline. This definition is an entirely different definition of FPGAs that we provided previously. This definition is more in line with emerging FPGA cloud services that current companies such as Amazon, Microsoft and Intel are providing to their customers. Unlike circuit emulation, computational acceleration is an emerging use case for FPGAs. Furthermore, the computational use case critically does not depend on FPGAs’ relationship to real ASICs: the Verilog code designers write for FPGA-based acceleration. They need not bear any similarity to the kind of Verilog that would go into a proper tapeout. These two use cases differ sharply in their implications for programming, compilers, and abstractions. In this thesis, I focus on the latter use case, which I call computation FPGA programming. Our thesis was that the current approach to programming computation FPGAs, which borrows the traditional programming model from circuit emulation, is not the right thing. Verilog and VHDL are the right things if the goal is to prototype an ASIC. In this thesis, we tried to promote the idea that we should rethink the entire stack when the goal is computation.

### 8.3.3 A Sudden Analogy

If we go back and look at the early 2000s, there was a time when Graphic Processing Units (GPUs) were designed only for graphical processing. After emerging of deep learning applications, people realized they could violate GPUs use case and use them as an accelerator for lots of computationally intensive kernels that are far from the graphic application; From that point, GPU designers started building a more general kind of machine, for which 3D rendering, the initial intent, was just one application. Computational FPGAs, third definition, are following the same trajectory. The idea is to go one step further of the traditional use case of these devices for circuit emulation and start exploiting computational patterns that are amenable to circuit-like execution.
To permit GPUs to become mainstream data-parallel accelerators, designers had to rethink what kind of abstraction could express the GPU programming model to the programmers. The initial abstraction for GPUs was an exotic, intensely domain-specific description of a visual effect. The realization of general-purpose programming on GPUs unlocked their potential for accelerating data-parallel applications [21]. This realization let GPUs evolve from targeting a single application domain to targeting an entire computational domain. I believe we are in the midst of a similar transition with computational FPGAs:

<table>
<thead>
<tr>
<th>GPU</th>
<th>FPGA</th>
</tr>
</thead>
<tbody>
<tr>
<td>massive, mostly regular data parallelism</td>
<td>irregular parallelism with static structure</td>
</tr>
</tbody>
</table>

At the time of writing this thesis, the community has not settled on an abstraction that can express the fundamental computation pattern that FPGAs are considered to be good at, irregular parallelism with data reuse, and mostly-static data flow. Like GPUs, FPGAs need a hardware abstraction that embodies this computational pattern. What’s missing here is an ISA-like abstraction for the software that FPGAs run.

**GPU: SIMT ISA**

**FPGA:** ?

### 8.3.4 RTL Is Not an ISA

A right ISA should directly expose the underlying hardware to a higher-level abstraction. This abstraction, like the assembly language, need not be convenient to program in. However, it should be extremely fast to compile and yield predictable results, the same as the assembly. For having a suitable ISA, the ISA should satisfy two features: 1) It has to express low-level hardware abstraction and 2) It has to have a high-level programming abstraction. HDLs as the primitive abstraction for programming FPGAs, satisfy neither of these two conditions. By way of contradiction, let us imagine what it would look like if HDLs satisfied these features:

*Verilog is a low-level abstraction for FPGA hardware resources which is to an FPGA as an ISA is to a CPU.* It may not be convenient to program in, but it is a good target for compilers from higher-level languages because it directly describes what goes on in the hardware. Furthermore, it is the last resort’s programming language when the designer needs to suck out the last few percentage points of performance.

Indeed, existing HLS frameworks are using HDLs as an ISA for today’s computational FPGAs. Compilers from higher-level languages emit Verilog as their output and the major FPGA vendors’ toolchains take Verilog as input. We argued in this dissertation that this technique is limited by the ad-hoc, often underspecified mix of software and hardware abstractions. This, in turn, results in difficulties in HLS compiler implementations. As a result, when using high-level synthesis tools, programmers must keep in mind that, despite the software programming abstractions, they must employ hardware, not software, optimization techniques when using HLS tools. Such HLS tools rely on hardware pragmas for hardware-specific information, the addition, changing, or removing
various pragmas can become more of a guess-and-check process. Such limitations restrict the designer’s ability to explore more complex design spaces. For most domain experts, the combination of these factors makes it challenging to write code which produces fully optimized designs.

8.3.5 The Right Abstraction?

While there exist proposed higher-level abstractions to contain computation patterns, such as Spatial [109], the limitation of these abstractions is that these new abstractions are far from existing software abstractions. Thus, this dissimilarity makes programmers rewrite their algorithm in these new languages to target FPGAs. This is against the general desire of software programmers that see FPGAs as an alternative to CPU that can compile their code for FPGAs without any changes and improve performance. I believe, we still do not know what abstraction should replace RTL for computational FPGAs. Practically, replacing Verilog may be impossible as long as the FPGA vendors keep their lower-level abstractions secret and their sub-RTL toolchains proprietary. However, building higher-level abstraction on top of RTL is the direction that probably should be taken to make programming computation FPGAs more practical.

In this dissertation, our work was a practice to build higher-level abstraction on top of RTL by considering the fact that the new abstraction should be close enough to the existing software abstraction that the translation phase from software abstraction to the new hardware abstraction becomes possible. This approach decouples behavioral specification from structural specification. It allows software programmers to target FPGAs without the need of restructuring or reoptimizing their code for a new target, while hardware designers can still express their ideas as a set of graph transformations on top of the new intermediate representation. We achieved this goal by making implicit software abstractions like dataflow graph and control-flow graph explicit in the hardware.

While this approach works well with the existing programs, but the question that arises with this technique is: at which level the software abstraction is representative enough for making explicit hardware. In $\mu$IR case, we picked software abstraction at the lowest abstraction, which was LLVM IR. To make this goal happen we had to answer the following questions:

1. How to build the dataflow graph that can face uncertain behavior of the memory.
2. How to build control structures that can exploit both fine grain and coarse grain parallelism.
3. What are the necessities to support arbitrary control-flow graph.
4. What is the right memory hierarchy for a custom accelerator to benefit from data reuse.

We showed that by building $\mu$IR abstraction, we could cover a wide range of software applications from arbitrary C code to domain-specific codes written in Tensorflow. By moving toward domain-specific application, however, maybe $\mu$IR is too low level and comes with the cost of generality that it is not always necessary while we are building specialized hardware accelerators. In the domain-specific applications, the hardware abstraction does not always need to answer all these
four questions, and some of them can be eliminated. As a result, the hardware implementation can be simplified and become specialized for a domain. The domain abstraction guides the hardware abstraction and the right hardware abstraction for a specific domain.

Maybe the right direction to continue building hardware accelerators is to provide a framework that can host different DSLs, a hardware library for DSLs. Nevertheless, it is impossible to neglect the role of the general-purpose abstraction such as $\mu IR$ that can still deliver efficiency and acceleration on FPGAs while they may not be and will not be the best-specialized hardware accelerator for an application.
Bibliography


[81] Shantanu Gupta, Shuguang Feng, Amin Ansari, Scott Mahlke, and David August. Bundled execution of recurring traces for energy-efficient general purpose processing. In PROC of the 44th MICRO, 2011.


[214] Xilinx. Xilinx unveils versal: The first in a new category of platforms delivering rapid innovation with software programmability and scalable AI inference.


