Specialized Macro-Instructions for Von-Neumann Accelerators

by

Amirali Sharifian

B.Sc, Isfahan University of Technology, 2014

Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science in the Department of Computing Science Faculty of Applied Science

© Amirali Sharifian 2016 SIMON FRASER UNIVERSITY Fall 2016

All rights reserved. However, in accordance with the Copyright Act of Canada, this work may be reproduced without authorization under the conditions for “Fair Dealing.” Therefore, limited reproduction of this work for the purposes of private study, research, education, satire, parody, criticism, review and news reporting is likely to be in accordance with the law, particularly if cited appropriately.
# Approval

<table>
<thead>
<tr>
<th>Name:</th>
<th>Amirali Sharifian</th>
</tr>
</thead>
<tbody>
<tr>
<td>Degree:</td>
<td>Master of Science (Computing Science)</td>
</tr>
<tr>
<td>Title:</td>
<td><em>Specialized Macro-Instructions for Von-Neumann Accelerators</em></td>
</tr>
</tbody>
</table>

**Examing Committee:**

<table>
<thead>
<tr>
<th>Chair:</th>
<th>Ryan Shea</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>University Research Associate</td>
</tr>
</tbody>
</table>

**Arrvindh Shriraman**

Senior Supervisor  
Associate Professor  
School of Computing Science

**Martin Ester**

Supervisor  
Professor  
School of Computing Science

**Robert D.Cameron**

Internal Examiner  
Professor  
School of Computing Science

**Date Defended:**  
30 November 2016
Abstract

In the last few decades, Von-Neumann super-scalar processors have been the superior approach for improving general purpose processing and hardware specialization was used as a complementary approach. However, the imminent end of Moore’s law indicates voltage scaling and per-transistor switching power can not scale down with the same pace as what Moore’s law predicts. As a result, there is a new interest in hardware specialization to improve performance, power and energy efficiency on specific tasks. Hardware customization so far has effectively targeted programs with abundant parallelism and simple data access patterns using vector extensions, GPUs, and spatial fabrics. However, programs with diverse control and memory behavior have found it challenging to leverage these accelerators.

Our work is motivated by observing two main approaches in designing hardware accelerators. First many proposals [45, 20] seem to choose dataflow-based accelerators to reduce front-end energy. However, spatial fabrics encounter challenges with fabric utilization and static power when the available instruction parallelism is below the peak operation parallelism available [14]. Second, using custom or magic instructions[19, 10, 23] integrated with the core pipeline reduces both front-end (fetch and decode) and back-end (register access) costs. The problem with this method is the lack of generality across programs because the desired set of custom instructions varies.

This dissertation proposes a Von-Neumann based accelerator, Chainsaw and demonstrates that many of the fundamental overheads (e.g., fetch-decode) can be amortized by adopting the appropriate instruction abstraction. To this end, we use the notion of chains, which are compiler fused sequences of instructions. chains convey the producer-consumer locality between dependent instructions, which the Chainsaw architecture then captures by temporally scheduling such operations on the same execution unit and uses bypass registers to forward the values between dependent operations. Chainsaw is a generic multi-lane architecture (4-stage pipeline per lane) and does not require any specialized compound function units; it can be reloaded enabling it to accelerate multiple program paths. We have developed a complete LLVM-based compiler prototype and simulation infrastructure and demonstrated that an 8-lane Chainsaw is within 73% of the performance of an ideal dataflow architecture while reducing the energy consumption by 45% compared to a 4-way OOO processor.

Keywords: Hardware accelerator; Energy Efficiency; Computer Architecture
Dedicated to Setareh.
Acknowledgements

I would like to thank my senior supervisor, Dr. Arrvindh Shriraman, for his extensive support and being a constant source of motivation during my studies; Snehasish Kumar and Apala Guha, for all the help and advice while working on the Chainsaw project and my friends Dorna Niroomand, Kamyar Khodamoradi, Zeinab Sadeghipour, Sajjad Gholami and all those who were my best friends and companions during my master studies. Last but not the least, I am thankful to my family for their unlimited support during my studies. I especially owe a great debt to my sister, mother, and father without whom none of this would be possible.
# Table of Contents

Approval ii  
Abstract iii  
Dedication iv  
Acknowledgements v  
Table of Contents vi  
List of Tables viii  
List of Figures ix  

1 Introduction 1  
1.1 Optimizing processor 2  
1.2 Specialized Hardware 2  
1.3 Contribution 4  
1.4 Dissertation Outline 5  

2 Background and Motivation 6  
2.1 Related Work 6  
2.1.1 Instruction set customization 6  
2.1.2 Efficient General-purpose Processors 8  
2.1.3 Dataflow accelerators 9  
2.2 Dataflow Graph Execution 10  
2.2.1 Fabric Utilization and Static Power 10  
2.2.2 Reconfiguration costs 12  
2.2.3 Data Movement Energy: 13  

3 Chains 15  
3.1 Chain Instructions 15  
3.1.1 Chain benefits 16  
3.1.2 Chain Distribution 20  


## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Characteristics of CGRA execution (4x4 and 8x8). Performance (cycles), Idle Cycles (cumulative function unit idle cycles). ILP: dataflow instruction parallelism</td>
<td>13</td>
</tr>
<tr>
<td>4.1</td>
<td>Scheduler Table</td>
<td>27</td>
</tr>
<tr>
<td>4.2</td>
<td>Instruction fields</td>
<td>28</td>
</tr>
<tr>
<td>4.3</td>
<td>Dynamic Execution</td>
<td>29</td>
</tr>
<tr>
<td>5.1</td>
<td>$\Sigma$Coverage top five traces</td>
<td>31</td>
</tr>
<tr>
<td>5.2</td>
<td>System parameters</td>
<td>35</td>
</tr>
</tbody>
</table>
List of Figures

Figure 1.1  *Chainsaw* overview. Our compiler constructs control-free superblocks [6, 22] 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 [19, 18, 10] chains only fuse only sequences of operations and do not require specialized compound function units. 4

Figure 2.1 Array of FUs [19]. 10

Figure 2.2 DFG example. 11

Figure 2.3 Execution on a spatial fabric. 12

Figure 2.4 Execution on a von neumann architecture. 12

Figure 3.1 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). 16

Figure 3.2 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. 17

Figure 3.3 Strategy *MaxSize* merges chains greedily to reduce data movement. Register writes decreased to 4. Critical path length increased to 14 ops. 18

Figure 3.4 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. 20

Figure 3.5 Relative proportion of inter-chain and intra-chain dependencies. Fusing ops into chains localizes communication. 20
Figure 3.6  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. ........................................... 21

Figure 3.7  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. 22

Figure 3.8  # of chains generated by the MaxILP algorithm and the MaxSize algorithms. MaxSize typically creates fewer chains enabling. .......................... 23

Figure 3.9  Average chain lengths generated by MaxSize algorithm. .......................... 23

Figure 4.1  Block diagram of the Chainsaw accelerator. It is composed of lanes. Each lane has a INT and FPU with forwarding registers, two input/output registers (INs and OUTs), a live-in register bank, and an instruction buffer. ........... 25

Figure 4.2  Example DFG execution on Chainsaw. .................................................. 26

Figure 4.3  Chain’s instructions example. ................................................................. 27

Figure 4.4  Single lane’s pipeline stages. ................................................................. 28

Figure 4.5  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. .................................................. 28

Figure 5.1  Chainsaw Simulator overall modules ..................................................... 33

Figure 5.2  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. .......................... 37

Figure 5.3  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. ........... 39

Figure 5.4  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. .......................... 40
Figure 5.5  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.
Chapter 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:

\[ P = Q f C V^2 + V I_{\text{leakage}} \]

\( Q \) 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 [53]. 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.
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 [50].

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 [36, 33], registers [46] and scratch pad memory [24, 13] which only focuses on data acquisition. For instance, [29] uses optimized memory and memory interface with a simple in-order Von-Neumann core.

However, at [4] 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.

1.2 Specialized Hardware

Altering general purpose process 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 [40], and specialize the architecture for each behavior. Heterogeneous multicore systems (Big and small cores) [35, 43]) adopt this approach, but they tend to use a conventional front-end which dominates overall energy consumption [23]. Other approaches have sought to improve the
general-purpose processor (here is referred to as \textit{OOO}) efficiency by detecting and caching the loops in a smaller \textit{\mu}op cache \cite{25}. 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 \cite{20, 39, 54}. 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 (\textit{ILP}). A reconfigurable functional-unit fabric \cite{20} 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\cite{55, 54, 22} can adapt to varied instruction parallelism both across applications and within an application. Section 2.2 discusses the tradeoffs in dataflow accelerators.

In recent years, hardware specialization has become a promising paradigm for continued 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 \cite{41}, 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” \cite{23, 10, 19}. 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 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 \textit{what property of the dataflow graph should magic instructions convey to the hardware} and \textit{how do we decouple the hardware from the magic instruction itself}. 
Figure 1.1: Chainsaw overview. Our compiler constructs control-free superblocks [6, 22] 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 [19, 18, 10] chains only fuse only sequences of operations and do not require specialized compound function units.

1.3 Contribution

In this dissertation, we explore Chainsaw [49], 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 [27]. In contrast to SIMD or VLIW instructions which express parallelism, chains express the lack thereof. Figure 1.1 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 [47] 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 [17, 5, 30]; 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 [5], 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 $8 \times$ 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.

Our contributions are as follow:

- 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 ($\text{MaxSize}$) and ILP ($\text{MaxILP}$) 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.

1.4 Dissertation Outline

The dissertation is organized as follow: Chapter 2 describes related ideas which inspire Chainsaw. Chapter 3 describes chain abstraction and the tradeoff between different algorithms that the compiler prototype and tradeoffs when constructing chains. Chapter 4 describes Chainsaw architecture. The Chainsaw framework is explained at Chapter 5 and chapter 5.6 presents the evaluation. Conclusions and future work are outlined in chapter 6.
Chapter 2

Background and Motivation

2.1 Related Work

*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*.

2.1.1 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 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 $\mu$ops. 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 [17, 51, 31] 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 [26]. 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 [47, 20, 45]. 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 [18] 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 doesn’t 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 [10] and DSFU [19] 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 [27] 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.

2.1.2 Efficient General-purpose Processors

Clustering of execution resources [30, 17, 28] 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 [43, 35, 3] 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 [25] 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 [42] 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 [16, 5]. A key benefit of this approach is that it generalizes how compound function units [10, 39, 22] 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 [5], Chainsaw develops a compiler framework to carefully fuse and organize the instructions to guarantee dependent instructions are scheduled on a same lane.

### 2.1.3 Dataflow accelerators

Many recent proposals in hardware accelerators [20, 9, 45, 44] have been inspired by past work in dataflow architectures [52, 47]. These approaches have sought to switch between the Von-Neumann execution on the general-purpose processor and the dataflow-based accelerator, Figure 2.1, in a fine-grained manner [39]. 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 [38], 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 [38]. 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).

**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 [47]. 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.
2.2 Dataflow Graph Execution

We motivate the chain abstraction for instructions using the DFG in Figure 2.2. 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 2.3 and Figure 2.4 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.

2.2.1 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

![Array of FUs](image-url)
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 2.3 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 [21].

Table 2.1 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.
### 2.2.2 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 2.1 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.
Table 2.1: Characteristics of CGRA execution (4x4 and 8x8). Performance (cycles), Idle Cycles (cumulative function unit idle cycles). ILP: dataflow instruction parallelism

<table>
<thead>
<tr>
<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>Perf. Cycles</td>
<td>Idle Cycles</td>
</tr>
<tr>
<td>gzip</td>
<td>59</td>
<td>5.1</td>
<td>117</td>
</tr>
<tr>
<td>art</td>
<td>54</td>
<td>3.9</td>
<td>166</td>
</tr>
<tr>
<td>mcf</td>
<td>29</td>
<td>3.6</td>
<td>93</td>
</tr>
<tr>
<td>equake</td>
<td>24</td>
<td>2.6</td>
<td>119</td>
</tr>
<tr>
<td>parser</td>
<td>38</td>
<td>4.7</td>
<td>102</td>
</tr>
<tr>
<td>bzip2</td>
<td>321</td>
<td>12.4</td>
<td>680</td>
</tr>
<tr>
<td>gcc</td>
<td>126</td>
<td>10.5</td>
<td>271</td>
</tr>
<tr>
<td>mcf</td>
<td>30</td>
<td>3.3</td>
<td>79</td>
</tr>
<tr>
<td>namd</td>
<td>78</td>
<td>6.0</td>
<td>161</td>
</tr>
<tr>
<td>soplex</td>
<td>39</td>
<td>3.25</td>
<td>128</td>
</tr>
<tr>
<td>povray</td>
<td>95</td>
<td>6.78</td>
<td>191</td>
</tr>
<tr>
<td>hmemr</td>
<td>147</td>
<td>13.3</td>
<td>499</td>
</tr>
<tr>
<td>sjeng</td>
<td>99</td>
<td>5.8</td>
<td>219</td>
</tr>
<tr>
<td>h264ref</td>
<td>64</td>
<td>5.3</td>
<td>175</td>
</tr>
<tr>
<td>lbm</td>
<td>207</td>
<td>23</td>
<td>412</td>
</tr>
<tr>
<td>sphinx3</td>
<td>44</td>
<td>3.6</td>
<td>91</td>
</tr>
<tr>
<td>blacksc.</td>
<td>273</td>
<td>5.5</td>
<td>368</td>
</tr>
<tr>
<td>bodytra.</td>
<td>81</td>
<td>4.7</td>
<td>173</td>
</tr>
<tr>
<td>dwt53</td>
<td>33</td>
<td>3.6</td>
<td>102</td>
</tr>
<tr>
<td>fluidan.</td>
<td>70</td>
<td>4.1</td>
<td>184</td>
</tr>
</tbody>
</table>

2.2.3 Data Movement Energy:

Dataflow accelerators at the instruction granularity maintain a one-to-one mapping between operations and function unit [21] 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 [11].

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 [27]. Coarse-grain instructions can potentially reduce frontend overhead and exploit locality between operations [23]. 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.
Chapter 3

Chains

In this chapter, we propose Chains as a new instruction abstraction for representing computation to the hardware. Chains are fused sequences of operations where at least one of the instruction’s operands is produced by the previous instruction. Also, the outputs of chains become visible to other instructions only when the chain in entirety is completed regardless of when the values are produced, to minimize dependence tracking and chain wakeup costs.

3.1 Chain Instructions

Figure 3.1 shows a possible decomposition of the dataflow graph in Figure 2.2 into chains; there are five chains, C0–C4. This decomposition has been produced using Dilworth chain decomposition algorithm [15]. 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.

Figure 3.1 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 C4, C3 and C1 are scheduled on one functional unit in that order, while C0 and C2 are scheduled on the other unit in that order. Note that 25 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?

### 3.1.1 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 3.1. Although instruction $i_{13}$ is ready to run at the very outset, it cannot execute until the chains $C_0, C_2, C_3,$ and, $C_4$ have finished.
Therefore, to recover lost ILP, we break chains at inter-chain dependencies as shown in Figure 3.2. 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 3.2. 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 3.1.2. 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.

![MaxILP Decomposition Schedule](image)

Figure 3.2: 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, 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 1 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 MaxSize. For example, the chain decomposition in Figure 3.2 is converted to the chain decomposition in Figure 3.3. The overall latency increases to 14 cycles, but the number of external dependencies reduces to four.
Figure 3.4 shows the chained graph for a frequently executed region in 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 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 3.5 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 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 3.6 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; ≃35% of the operations have only one op.

Figure 3.7 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

![MaxSize Decomposition and Schedule Diagram](image.png)

Figure 3.3: Strategy MaxSize merges chains greedily to reduce data movement. Register writes decreased to 4. Critical path length increased to 14 ops.
Algorithm 1: Algorithm for the MaxSize strategy.

**Input**: Dilworth decomposed graph

**Output**: MaxSize decomposed graph

```
foreach edge do
    srcNode = source(edge);
    tgtNode = target(edge);
    srcChain = chain(srcNode);
    tgtChain = chain(tgtNode);
    if srcChain == tgtChain then
        continue;
    end
    tmpChain = concat(srcChain, tgtChain);
    if liveIns(tmpChain) > 2 then
        continue;
    end
    if liveOuts(tmpChain) > 2 then
        continue;
    end
    remove srcChain from dfg;
    remove tgtChain from dfg;
    add tmpChain to dfg;
    if dfg has cycle then
        remove tmpChain from dfg;
        destroy tmpChain;
        add srcChain to dfg;
        add tgtChain to dfg;
    end
end
```

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× 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.
Figure 3.4: 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.

Figure 3.5: Relative proportion of inter-chain and intra-chain dependencies. Fusing ops into chains localizes communication.

3.1.2 Chain Distribution

Figure 3.9 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.equake, blackscholes, sar-pfa and swaptions), all other applications have 50% or more chains with size greater than 2. Figure 3.8 shows the number of chains per application. The MaxILP approach produces significantly more chains than the MaxSize approach as described previously in section 3. There are 30 chains per workload on average, 10
Figure 3.6: 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. 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.
Figure 3.7: 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 3.8: # of chains generated by the MaxILP algorithm and the MaxSize algorithms. MaxSize typically creates fewer chains enabling.

Figure 3.9: Average chain lengths generated by MaxSize algorithm.
Chapter 4

Chainsaw architecture

This chapter describes overall architecture of Chainsaw. We describe what is the lane and how they execute individual chains. How the lane’s instruction buffer fills. How scheduler find free chains and activate the lanes. And finally how Chainsaw can run chains.

4.1 Overall Architecture

Figure 4.1 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 base 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 base on the chain abstraction. Therefore, in the scheduling phase, each chain scheduled entirety 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 opportunistically 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 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 4.1.2). 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 (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.

4.1.1 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 reconfiguration 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 \textit{IN}0 and \textit{IN}1 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 \textit{OUT}0 or \textit{OUT}1 local registers. When the chain finishes executing, the values in the \textit{OUT} registers are routed to the appropriate live-in register bank by the bus.

Figure 4.2 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 4.2: Example DFG execution on *Chainsaw*.](image-url)
4.1.2 Chain Scheduling and Wakeup

Table 4.1 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>
<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>

4.1.3 Instruction Issue

![Figure 4.3: Chain’s instructions example.](image)

Figure 4.3 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 4.4: Single lane’s pipeline stages.

Figure 4.4 shows pipeline stages for a single lane in Chainsaw. In Figure 4.5 shows overall format of proposed ISA.

Table 4.2 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 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.
### 4.1.4 Dynamic execution

Table 4.3 shows the execution sequence of chains belonging to Figure 4.3, 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 depends on the type of operations and also depends on memory response time for $MEM$ operations. Base on Table 4.1 scheduler realised that $C_0$ 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 $C_0$. For the next cycle, scheduler updates the values. After finishing $C_0$, $C_1$ 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.

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Lane 1</th>
<th>Lane 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>$C_0$</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>$C_0$</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>$C_0$</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>-</td>
<td>$C_1$</td>
</tr>
<tr>
<td>6</td>
<td>-</td>
<td>$C_1$</td>
</tr>
<tr>
<td>7</td>
<td>-</td>
<td>$C_1$</td>
</tr>
<tr>
<td>8</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>-</td>
<td>$C_2$</td>
</tr>
</tbody>
</table>

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.
Chapter 5

Framework and Evaluation

This chapter describes the simulation infrastructure used to evaluate the power and the performance of Chainsaw. The infrastructure contains:

- An LLVM–based toolchain for profiling, extracting the chains and finally generating code for Chainsaw.
- A cycle accurate C++ simulator.
- Hardware synthesis and area overhead.

5.1 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\([6]\) 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 5.1 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 5.1: \( \Sigma \) Coverage top five traces

<table>
<thead>
<tr>
<th>( \Sigma_5 ) 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>

5.2 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 3 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.

5.3 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).

5.4 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 using MacSim [2]. MacSim is a heterogeneous architecture simulator, which is trace-driven and cycle-level. It thoroughly models architectural behaviors, including detailed pipeline stages, multi-threading, and memory systems. However, we modified the Macsim simulator memory system and replaced the existence one with Ruby [37]. 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.

5.4.1 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 5.1 shows an overview of the simulator.

Graph

The graph module decomposes the original DFG using either of algorithms described at chapter 2.1. 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 can not 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

---

\(^1\)The following six workloads crafty, freqmine, sar-back, sar-pfa, streamc, swaptio. were not supported on our simulator.
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. Base on these variables the scheduler module schedules chains on each lane. In the end, the scheduler has a table just like table 4.1. 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 the 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 4 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 [34]. 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.
5.4.2 CGRA simulation

We model a CGRA, a spatial homogeneous fabric accelerator similar to [20, 45]. 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 5.2 shows the characteristics of the architectures that we model.

<table>
<thead>
<tr>
<th>System parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Host Core</strong></td>
</tr>
<tr>
<td>2 GHz, 4-way OOO, 96 entry ROB, 4 INT, 4 FPU, INT RF (64 entries), FP RF (64 entries)</td>
</tr>
<tr>
<td>32 entry load queue, 32 entry store queue</td>
</tr>
<tr>
<td><strong>L1</strong></td>
</tr>
<tr>
<td>64K 4-way D-Cache, 3 cycles</td>
</tr>
<tr>
<td><strong>LLC</strong></td>
</tr>
<tr>
<td><strong>Memory</strong></td>
</tr>
<tr>
<td><strong>Accelerators</strong></td>
</tr>
<tr>
<td><strong>CGRA8</strong></td>
</tr>
<tr>
<td>8 × 8 function units or Lane sizes: 1, 2, 4, 8, 16 #instruction-s/lane: 256, 128, 64, 32, 16.</td>
</tr>
<tr>
<td><strong>Energy Parameters (Static and Dynamic)</strong></td>
</tr>
<tr>
<td><strong>OOO</strong></td>
</tr>
<tr>
<td>Mcpat [32]; ARM A9 2Ghz template.</td>
</tr>
<tr>
<td><strong>CGRA8</strong></td>
</tr>
<tr>
<td>CGRA Network (650 fJ/switch), Function units (510 fJ/INT, 1500fJ/FP)</td>
</tr>
<tr>
<td><strong>Chainsaw</strong> Lane</td>
</tr>
<tr>
<td>Instruction buffer (16 entries, 120 fJ/read, 220 fJ/write), Decode (100 fJ/instruction)</td>
</tr>
<tr>
<td><strong>Chainsaw Comm.</strong></td>
</tr>
<tr>
<td>Pipeline forwarding (250 fJ), Live-in Registers (Read: 180fJ and Write: 250fJ), Bus (1100fJ/access)</td>
</tr>
</tbody>
</table>

5.4.3 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 [48].

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

5.5 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 4) the entire instruction buffer 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 3.7). 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 $2 \times 8$ registers per lane (i.e., $16 \times 32$ 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; $(8 \times 9)$ bits. Overall, the per lane overhead $\simeq 100$ 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 ($\simeq 1.6$ KB) is 0.21 mm$^2$ (including the functional units).

### 5.6 Evaluation

For the rest of this chapter we evaluate the performance and the power of the Chainsaw described in chapter 4. The evaluation is performed using the infrastructure described in Section 5.

### 5.7 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 5.2 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, ibm, bodytrack, and, fluidanimate), we find that performance of OOO $<$ Chainsaw8 $<$ Chainsaw16. In these benchmarks, the unchained DFG ILP (4–23 , see Table2.1) 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

---

$^2$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 Table2.1).
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 3.7) 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 $\approx 4$. 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.

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.
5.8 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.

5.8.1 Dynamic Energy

Figure 5.3 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 \( \simeq 5 \times \) 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 \( \simeq 3.3 \times \) 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 Chainsaw, the average is \( \simeq 2.8 \times \). The net dynamic energy consumption is reduced via internalizing communication within chains.

5.8.2 Communication Costs

Figure 5.4 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, pipeling forwarding, and register (INs/OUTs) 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 3.5.

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 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...
Figure 5.3: 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.

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.

5.8.3 Static Power:

Figure 5.5 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 instruction-granularity accelerators [21] is scaled based on the number of operations to be accelerated while the Chainsaw is
Figure 5.4: 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.

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. *Chainsaw*8 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.

*Chainsaw*8 reduces overall static energy by $\approx 21\%$. From Figure 5.5, we see 11 of the 20 applications reduce energy by 20% to 40%. In equake, povray, hmmer, sjeng and lbm 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 *Chainsaw*8. For 15 out of 20 workloads, there is more IDLE’ness in the *Chainsaw* architecture. In
Figure 5.5: 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.

the remaining 5 workloads (namd, sjeng, blackscholes, 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.

5.9 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 6

Conclusion

In this chapter the primary findings are summarized and describe future work motivated by these findings.

6.1 Conclusions

The processor industry is based on an economy of scale i.e. chips need to be produced in high volumes to offset the steep costs of design and manufacture [1]. Therefore, only processors or accelerators that have wide applicability over a variety of programs actually make it to the production phase. At the same time, due to the approaching end of Moore’s Law, hardware customization is being viewed as a promising approach for continued increases in energy-efficiency. Economies of scale are at odds with processor customization as such approaches reduce the applicability of customizations. Therefore, most of the customized hardware that has made it to the market optimize a specific type of code which is still widely applicable. For example, GPUs and vector extensions accelerate data–level parallelism. Instruction-set extensions such as AESNI accelerate cryptographic functions which are ubiquitous. Spatial fabrics [7] accelerate high-ILP programs. The unifying feature of these accelerators is the abundance of parallel work with simple data access patterns that can be mined statically. The efficiency arises from the fact that the hardware does not spend resources on mining instructions to execute, and each instruction/configuration encodes a large amount of work.

There is a large body of software that does not have these characteristics. Such software may have complex data access patterns, control-flow intensive code regions which manifest as low ILP. The common theme in these codes is their diversity which current accelerator approaches are not suitable for. At the same time, designing a variety of accelerators [18, 54, 22] to optimize these codes cannot be supported from an economic standpoint. The major focus of this dissertation is an investigation into designing an economically viable accelerator for these other codes. The requirements for such an accelerator are:
1. preserving the basic ISA and eschewing the path of designing special operators so as to maintain wide applicability

2. maintaining a Von-Neumann style instruction fetching-based architecture to avoid the tension between configuration costs and idle power costs in spatial fabrics when variable ILP is present.

6.2 Looking Forward

Making register data transfer a first-order concern

We focus on data transfer energy as it is known to be a significant fraction of overall energy consumption in a core[23]. We did away with the automatic routing of intermediate values to the global register file and stored them in local bypass registers instead. We believe that research on other hardware mechanisms to capture data with different characteristics is possible. Traditional processors mandate data to be either routed to registers or the rest of the memory hierarchy. However, research processors such as Elm[12] explored several other data storage mechanisms in hardware.

The different data characteristics that may be addressed in hardware are several in number e.g. read-only/write-once, short-lived heap data, stack arrays and small arrays. The challenge here is to identify data characteristics that are ubiquitous and exploit them in hardware. While we coupled bypass registers to the datapath at the lowest level, different storage structures may need to be integrated at different levels.

Additionally, programming language research is necessary to easily expose the nature of data to compilers. Programming languages traditionally expressed the lifetime of data (e.g. stack or heap), however garbage collection has essentially precluded that. Newer programming languages (e.g. Python, Scala) provide a richer semantic description of data as opposed to the traditional implementation-oriented description. We hope compiler research will leverage such semantic description to extract and exploit data characteristics that are relevant to hardware.

Making programmable and generalizable accelerators

Additional concerns in accelerator research are how programmable and generalizable a particular accelerator is. Exposing different hardware mechanisms in the accelerator to software is a design choice. In this dissertation, we chose to expose bypass registers to software, so as to statically mine chains from frequently executed code regions and to map sequentially dependent operations to the same lane. We used static scheduling with chain triggering being the only dynamic component. Partitioning concerns between hardware and software is an important design choice because software tends to be conservative (which may reduce performance and indirectly reduce efficiency), and hardware directly reduces efficiency (more complex circuitry).

Static scheduling has been extensively studied (eg. VLIW and EPIC architectures) with limited success. However, it may regain importance in the accelerator context where the code regions being executed are fairly constrained themselves. Compiler-based mapping of instructions to
different components in the accelerator datapath still needs much research. One area where hardware could be significantly better than software is dynamic fusion of operations mapped to the same FU using asynchronous or trigger-based execution. A similar approach is macro op fusion in RISC-V [8]. Dynamic operation fusion can retain many of the benefits of custom instructions without a priori implementing any custom instructions. Finally, these techniques make the approach more generalizable, an important consideration which will lend to the widespread adoption of accelerators.
Bibliography


[22] 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.


