Abstract—A central tenet behind accelerators is to partition a program execution into regions with different behavior (e.g., SIMD, Irregular, Compute-Intensive) and then use behavior-specialized architectures [1] for each region. It is unclear whether the gains in efficiency arise from recognizing that a simpler microarchitecture is sufficient for the acceleratable code region or the actual microarchitecture, or a combination of both. Many proposals [2], [3] seem to choose dataflow-based accelerators which encounters challenges with fabric utilization and static power when the available instruction parallelism is below the peak operation parallelism available [4].

In this paper, we develop, Chainsaw, a von-Neumann based accelerator and demonstrate that many of the fundamental overheads (e.g., fetch-decode) can be amortized by adopting the appropriate instruction abstraction. The key insight is the notion of chains, which are compiler fused sequences of instructions. chains adapt to different acceleration behaviors by varying the length of the chains and the types of instructions that are fused into a chain. 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 pipeline 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 a 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.

1. Introduction

While it is clear that using customized hardware accelerators exploiting specific program behaviors is a promising way forward [1], 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 central tenet of the modern accelerator proposals is to split up the program into multiple phases [10], and specialize the architecture for each behavior. Big and small cores [11], [12] adopt this approach, but they tend to use a conventional front-end which dominates overall energy consumption [13]. Other approaches have sought to improve the general-purpose processor (herein referred to as OOO) efficiency by detecting and caching the loops in a smaller L1 cache [14]. 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 [3], [15], [16]. SIMD-based designs amortize the cost of instructions across multiple data parallel operations but restrict the types of instructions that can be bundled and are closely allied to the hardware function unit. A key limitation of such dataflow approaches is designing for programs with varying levels of ILP. A reconfigurable functional-unit fabric [3] 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 [6], [16], [17] can adapt to varied instruction parallelism both across applications and within an application. Section 3 discusses the tradeoffs in dataflow accelerators.

A promising approach to specialization is the notion of “custom or magic instructions” [7], [9], [13]. 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 what property of the dataflow graph should magic instructions convey to the hardware and how do we decouple the hardware from the magic instruction itself.

Our Contribution

In this paper, we explore Chainsaw, 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

978-1-5090-3508-3/16/$31.00 © 2016 Crown
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 [18]. Converse to SIMD or VLIW instructions which express parallelism, chains express the lack thereof. Figure 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 [19] 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 [20]–[22] microarchitectures; 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 minimizes 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 [21], Chainsaw fixes the maximum number of instructions that can be mapped to a lane to minimize the fetch-and-decode costs. Chainsaw performs within 81% of an ideal CGRA accelerator. Chainsaw overhead to the processor core while saving 45% of the energy. Compared to a CGRA with 8× the resources Chains save between 24—54% of the communication power by localizing the communication and 21% of the static power by improving utilization of function units. We make the following contributions:

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

The paper is organized as follows: Section 2 describes related ideas which inspire Chainsaw; Section 3 discusses the challenges with dataflow-based accelerators. Section 4 describes the tradeoff between different algorithms that the compiler prototype and tradeoffs when constructing chains. Section 5 describes the architecture and Section 7 presents the evaluation.
2. Related Work

Instruction set customization

The instruction abstraction plays a key role in permitting the compiler to express more information about the program control and dataflow structure to the hardware; for instance SIMD expresses data parallelism [20], [23], [24] between operations and VLIW expresses instruction level parallelism [20]. A pertinent question is what information about the dataflow structure do we expect instructions to express, can the information be expressed without increasing the size of the instruction, and does it generalize. Dataflow architectures [2], [3], [19] convey information about instruction dependencies and static placement of operations.

Tensilica [8], CCA [9], DSFU [7] have sought to extract commonly observed subgraphs of operations and then implement these operations using custom function units with associated storage elements. This effectively forms CISCy instructions and minimizes the energy overhead of the front-end. While this approach is promising for specific application domains it is not quite clear how it can be generalized especially given the need for specialized function unit. An interesting approach is the separation of compile-time ISA from the hardware ISA, an approach pioneered by Transmeta. Some have applied this idea [18] in a limited context for dynamically fusing pairs of x86 micro ops in a dynamic compiler to reduce the front-end overheads of an OOO.

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 paper, we use the abstraction of chains which fuses only sequences of operations which minimizes the amount of state that needs to be maintained.

Efficient General-purpose Processors

Clustering of execution resources [20], [22], [25] seeks to scale up execution resources, localize communication between instructions, and minimize the cost of issuing instructions. These works minimized instruction-instruction communication by steering instructions to individual clusters of execution resources. In the past, these approaches have been pursued largely because of wire delays challenging pipeline design. Today, similarly with wire energy dominating overall instruction execution, clustering approaches can help improve energy efficiency. Another line of work [11], [12], [26] has used heterogeneous backends and schedule low-ILP or moderate ILP code regions on an inorder backend to save energy. A key limitation of past work is that they primarily focused on clustering backend resources while minimizing changes to frontend which expends a large fraction of the processor’s energy. Loop-accelerators [14] recognized that the key to improving energy efficiency is to disable the frontend 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. Other work on increasing the front-end efficiency [27] 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 [21], [28]. A key benefit of this approach is that it generalizes how compound function units [6], [9], [15] 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 [21], Chainsaw develops a compiler framework to carefully fuse and organize the instructions to guarantee

Dataflow accelerators

Many recent proposals in hardware accelerators [2], [3], [29], [30] have been inspired by past work in dataflow architectures [19], [31]. These approaches have sought to switch between the von neumann execution on the general-purpose processor and the dataflow-based accelerator in a fine-grained manner [15]. Dataflow accelerators statically map at compile-time the program dependence graph to a fabric of homogeneous or heterogeneous function units to completely eliminate the overhead of fetch-and-decode. 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. Dataflow accelerators are also by design more optimal when plenty of ILP is available in the program region; when there is only moderate levels of ILP 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 [19]. 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.
3. Background and Motivation

We motivate the chain abstraction for instructions using the DFG in Figure 2a. 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 2b and Figure 2c 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.

![Example DFG](image)

![Execution on a spatial fabric](image)

![Execution on a von neumann architecture](image)

Figure 2: Execution of DFG on spatial fabric leads to higher idleness in the spatial fabrics if the ILP in the application does not match the peak ILP of the spatial fabric and consequently static power will limit the overall power efficiency.

**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, however the amount of reduction depends on the initiation interval. Unlike processor pipelines which are seeking to overlap pipeline stages, pipelining a dataflow graph requires the loop around the dataflow graph to be pipelineable 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 2b has to wait for 20 to complete. Otherwise, a customized number of pipeline latches are needed between 25 and 27, clearly challenging [32].

**TABLE 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>Arg ILP</th>
<th>CGRA 4x4</th>
<th>CGRA 8x8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Perf Cycles</td>
<td>Idle Cycles</td>
<td>Avg Cycles</td>
</tr>
<tr>
<td>grasp</td>
<td>59</td>
<td>5.1</td>
<td>1041</td>
</tr>
<tr>
<td>art</td>
<td>54</td>
<td>3.9</td>
<td>2076</td>
</tr>
<tr>
<td>mcf</td>
<td>29</td>
<td>3.6</td>
<td>441</td>
</tr>
<tr>
<td>equake</td>
<td>24</td>
<td>2.6</td>
<td>1106</td>
</tr>
<tr>
<td>parser</td>
<td>38</td>
<td>4.7</td>
<td>1236</td>
</tr>
<tr>
<td>bzip2</td>
<td>321</td>
<td>12.4</td>
<td>17856</td>
</tr>
<tr>
<td>gcc</td>
<td>126</td>
<td>10.5</td>
<td>1782</td>
</tr>
<tr>
<td>mcf</td>
<td>30</td>
<td>3.3</td>
<td>354</td>
</tr>
<tr>
<td>namd</td>
<td>78</td>
<td>6</td>
<td>1158</td>
</tr>
<tr>
<td>soplex</td>
<td>39</td>
<td>3.25</td>
<td>1522</td>
</tr>
<tr>
<td>povray</td>
<td>95</td>
<td>6.78</td>
<td>1307</td>
</tr>
<tr>
<td>hmer</td>
<td>147</td>
<td>13.3</td>
<td>7992</td>
</tr>
<tr>
<td>sjeng</td>
<td>99</td>
<td>5.8</td>
<td>4302</td>
</tr>
<tr>
<td>h264ref</td>
<td>64</td>
<td>5.3</td>
<td>3420</td>
</tr>
<tr>
<td>bmem</td>
<td>207</td>
<td>23</td>
<td>1847</td>
</tr>
<tr>
<td>sphinx3</td>
<td>44</td>
<td>3.6</td>
<td>1350</td>
</tr>
<tr>
<td>blacksc</td>
<td>273</td>
<td>5.5</td>
<td>18257</td>
</tr>
<tr>
<td>bodytra</td>
<td>81</td>
<td>4.7</td>
<td>3780</td>
</tr>
<tr>
<td>dw53</td>
<td>33</td>
<td>3.6</td>
<td>1024</td>
</tr>
<tr>
<td>fluidan</td>
<td>70</td>
<td>4.1</td>
<td>2860</td>
</tr>
</tbody>
</table>

Table 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 hence 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...
We illustrate the counter intuitive case introduced by the 7
(we are looking for:
This captures the locality between back-to-back dependent
4. Our proposal: Chains
instructions feasible we need to group operations without
consumers and producers. To make coarse-grain
instruction. Also, the outputs of chains become visible to
other instructions only when the chain in entirety is com-
completed regardless of when the values are actually produced,
to minimize dependence tracking and chain wakeup costs.

Figure 3a shows a possible decomposition of the dataflow
graph in Figure 2a into chains; there are five chains, C0—C4.
This decomposition has been produced using the Dilworth
chain decomposition algorithm [34]. 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 3a 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 a ISA abstraction between the com-
piler 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 are restricted.
One of the operands for each internal instruction will be
produced by the predecessor instruction in the chain.

Listing 1: Algorithm for the MaxSize strategy.

for each edge {
    src_node = source(edge);
    tgt_node = target(edge);
    src_chain = chain(src_node);
    tgt_chain = chain(tgt_node);
    if (src_chain == tgt_chain)
        continue;
    tmp_chain = concat(src_chain, tgt_chain);
    if (live_ins(tmp_chain) > 2)
        continue;
    if (live_outs(tmp_chain) > 2)
        continue;
    remove src_chain from dfg;
    remove tgt_chain from dfg;
    add tmp_chain to dfg;
    if (dfg has cycle) {
        remove tmp_chain from dfg;
        destroy tmp_chain;
        add src_chain to dfg;
        add tgt_chain to dfg;
    } else {
        destroy src_chain;
        destroy tgt_chain;
    }
}
Chains provide the following benefits: 1) they localize a large fraction of communication between dependent instructions and eliminate many register 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. In this section, we explore questions that are critical to the success of chains: 1) Are chains potentially beneficial? and, 2) Is chain management hardware-friendly?

4.1. Are chains potentially beneficial?

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 3a. Although instruction (13) is ready to run at the very outset, it cannot execute until the chains C0, C2, C3, and, C4 have finished.

Therefore, to recover lost ILP, we break chains at inter-chain dependencies as shown in Figure 3b. Thus, for example, 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 3b. 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 4.2. Breaking at every live-in limits 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.

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. Listing 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 3b is converted to the chain decomposition in Figure 3c. The overall latency increases to 14 cycles, but the number of external dependencies reduces to four.

Figure 3d 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 4a 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 4b 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 i.e. 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 5 shows the average ILP in dataflow graphs for both the strategies as compared to the ILP in the unchained state. Since the MaxILP algorithm forcefully breaks the chain when any internal instruction
has an external dependency it attains as much ILP as the original unchained dataflow graph. The MaxSize algorithm may potentially lose ILP, when an internal operation in the chain is delayed from waking up a remote chain. The loss in ILP may also manifest as an increase in the critical path when compared to the ideal unchained dataflow graph. Our model here assumes every instruction in the critical path has the same latency to eliminate effects of memory operations. Overall, we find that in 6 applications chains increase the overall performance won’t necessarily suffer. In summary, chains have a high potential for improving energy-efficiency.

4.2. Chain Distribution

Figure 6b 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 (equake, blackscholes, sar-pfa, and swaptions), all other applications have 50% or more chains with size greater than 2. Figure 6a shows the number of chains per application. The MaxILP approach produces significantly more chains than the MaxSize approach as described previously in § 4. There are 30 chains per workload on average, 10 workloads have fewer than 20 chains. Only 482.sphinx3 has more than 60 chains (avg. size 3). In conclusion, the number of chains per workloads is a tractable for frequently executed regions.

5. Architecture

Figure 7 describes the overall Chainsaw design. Chainsaw is a multi-lane design consisting of lanes, each of which contains a 4-stage single-issue pipeline which fetches and decodes instructions from an instruction buffer. Each chain is scheduled in entirety on a single lane to exploit the intra-chain locality within the lane’s pipeline registers. The instruction buffer stores the execution sequence for multiple chains and chains are executed out-of-order as they are activated. 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-outputs values into the register once they complete all the operations 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).

They help indirectly 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). When a chain finishes, a bus routes values in
the output registers to the appropriate bank if required.
Finally, a scheduler determines when to schedule each chain
in the DFG. Each lane includes a Chain Read bitmap
which specifies the chains ready for execution. The number
of entries in this table specifies the maximum number of
chains that can be mapped to that lane; each entry is 10 bits
wide; two 4-bit fields specifying the live-in register ids in
the bank and two flag bits for registering the completion
of the parent chains. 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 determine the maximum size of each chain; here we set
it to the average size of \((\text{Chain}\times N)\). The number of lanes
is proportional to the ILP available.

**Execution**

The instruction buffers in each lane are preloaded before
Chainsaw starts execution; the reconfiguration involves writing
the instructions into the instruction buffer. Chains are
statically mapped onto lanes to ensure that the requirements
for the instruction buffer and the live-in register bank does
not exceed availability. However, chain activation is carried
out dynamically to improve latency. When a lane becomes available,
the scheduler checks 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. It activates one of the ready chains by loading its live-in values from the live-in register bank to the IN0 and IN1 registers. The scheduler also signals the fetch/decode unit in the lane to start issuing instructions. During chain execution, if an instruction produces a value that serves as a live-in to another chain, the value is written out to the one of the OUT0 or OUT1 registers. When the chain finishes executing, the values in the OUT registers are routed to the appropriate
live-in register bank by the bus. Figure 8 shows a simple
chained DFG that we use to demonstrate the architecture
functions. Figure 8 also shows the execution sequence of
chains belonging to this DFG, in their respective lanes. We
assume a latency of one cycle for each operation. C0 executes
first followed by C1 and C2.

**Chain Scheduling and Wakeup**

Figure 8 shows the scheduler table in 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 lane 1, ii) the live-in values are stored at locations at 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 Lane 0. 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 Lane 1. Similarly, when C1 finishes,
C2 is scheduled onto Lane 1.

**Instruction Issue**

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 register. Figure 8 shows
the instruction buffer fields for chain C1. The \textit{op} field gives the
operation to be performed. C1 has three operations
in the following sequence: \texttt{cvt} (convert), \texttt{sub} (subtraction),
and, \texttt{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.
6. Framework

We have built a LLVM-based toolchain for profiling, extracting chains and generating code for Chainsaw.

Profiling. 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. We then inline all functions called by this identified function in a bottom-up recursive manner. This LLVM-based infrastructure identifies paths [5] in the function. Enumerated paths are profiled using large representative inputs (eg. ref for SPEC benchmarks). Paths which include “unacceleratable” characteristics (such as external library calls, memory allocation) are pruned from the set.

We profile our workloads to understand how much “coverage” is provided by each path. Coverage of a path is calculated as the number of operations in the path times frequency of execution, represented as a fraction of the whole routine. Table 2 summarizes the coverage of the top five ranked paths in each workload. On average the coverage provided by the top five traces is 69% (median 88%). The five highest ranked paths by coverage are selected for chain extraction.

<table>
<thead>
<tr>
<th>Coverage</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>

Chain Extraction. At the basic block granularity, there are no control dependences (basic blocks are terminated by branches). While chains can be derived from basic blocks, the blocks themselves are quite small in size (11 operations on avg, max 150 for namd). 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. All branches in the previously described outlined functions are converted 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, 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. Chains are formed via the decomposition algorithms described in Section 4. The unmodified dataflow graph also serves as the basis of the CGRA timing simulation. We choose an optimistic CGRA schedule with no constraints.

Code Generation. The chain dataflow graph is derived from decomposing the original dataflow graph. The compiler adds the following information to the program binary: 1) markers to indicate regions that carry Chain information, 2) start addresses and lengths of chains in the region, 3) the dependencies among the chains belonging to the region, and 4) the stripped chain instructions (13 bits).

7. Evaluation

Simulation. We have developed a detailed cycle accurate simulator that models the host core, the Chainsaw accelerator, and spatial fabrics of parameterizable size. The host OOO core pipeline is modelled using MacSim [35]. We assume that Chainsaw is an accelerator that communicates with the OOO core via the L1 cache. We model a CGRA, a spatial homogeneous fabric accelerator similar to [2], [3]. The memory hierarchy is modelled using Ruby [36]. We assume an aggressive non-blocking interface to memory. To accurately model the host-accelerator interaction via the memory system, we capture a window of memory accesses prior to the accelerator invocation and warm up the caches. The memory accesses for host execution and the accelerator are collected a priori using Intel Pin [37]. 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.

We model all operations of the Chainsaw architecture as described in Section 5. 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. To model the power consumption, we adopt an event-based power model similar to Aladdin [38]. Table 3 shows the characteristics of the architectures that we model.

<table>
<thead>
<tr>
<th>TABLE 3: 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>
</tbody>
</table>

| **L1**                      |
| 64K 4-way D-Cache, 3 cycles |

| **LLC**                     |

| **Memory**                  |
| Energy Parameters (Static and Dynamic) |
| CGRA8                        |
| 8×8 8 function units or Lane sizes:1,2,4,8,16 #instruction-s/ lane:256,128,64,32,16. |

| **Accelerators**            |

| **OOO**                     |
| Mcap [39]; ARM A9 20hz template. |
| CGRA Network (650 fJ/switch), Function units (510 fJ/INT,1500fJ/FP) |
| Instruction buffer (16 entries, 120 fJ/read, 220 fJ/write), Decode (100 fJ/ instruction) |
| Pipeline forwarding (250 fJ), Live-in Registers (Read: 180fJ and Write: 250fJ), Bus (1100fJ/access) |

1. The following six workloads crafty, freqmine, sar-back, sar-pfa, streamc, swaprio. were not supported on our simulator.
Synthesis and Area Overhead. We designed the Chainsaw pipeline based on the RISC-V 4-stage IMAFD pipeline using our own 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 5) 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 5). The sample configuration we synthesized and simulated supports 8 chains; given the maximum fan–in for many workloads is 1–2, we assumed a total of 2×8 registers per lane (i.e., 16×32 bit = 64 bytes). The register banks are dual ported to supply the chain registers in a single cycle. The scheduler, unlike OOO, consists of only one wakeup component; it does not require any tag matches since the compiler explicitly encodes the dependent children; each entry in the chain wakeup flag is 9 bits (1 bit for ready and two 4–bit live–in register ids). The chain wakeup matrix has as many entries as the number of chains per lane; (8×9 bits). Overall, the per lane overhead ≃ 100 bytes (64 bytes for the 16 entry register bank, 26 bytes for the 8 entry instruction buffer, and 9 bytes for the chain wakeup flag). Overall, we found that the area for a 16 lane design (≃ 1.6 KB) is 0.21 mm² (including the functional units).

7.1. 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 application² ILP. Figure 9 shows the performance of the OOO, Chainsaw8 and Chainsaw16 normalized with respect to the IDEAL-CGRA. Higher bars are better, i.e performance is closer to an IDEAL-CGRA.

For 10 of the 20 workloads (gzip, art, gcc, namd, soplex, hmmer, h264ref, lbm, bodytrack, and, fluidanimate), we find that performance of OOO < Chainsaw8 < Chainsaw16. In these benchmarks, the unchained DFG ILP (4–23 , see Table1) 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 is able to exploit more ILP. Of these three workloads (gcc, hmmer, and, lbm) with ILP in the range 6–16, for which Chainsaw16 performs significantly better than Chainsaw8 due to increased hardware resources.

For 6 workloads (mcf, equake, parser, bzip2, sphinx3, and, dwt53) the performance of the Chainsaw8 was the same as Chainsaw16. For all these workloads the chained DFG ILP is less than 8, thus Chainsaw16 is over-provisioned. In 3 of these workloads (mcf, equake and dwt53) the performance of the OOO is better than Chainsaw architectures. The chained ILP for each of these is less than the unchained ILP (see Figure 5) 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 ≃ 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

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 Table1).
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.

### 7.2. Energy Comparison

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

**Dynamic Energy** Figure 10a shows the dynamic energy consumed by CGRA8 and Chainsaw8 normalized to functional unit energy. Across workloads the energy of the OOO (numbers above each bar) is $\approx 5 \times$ the functional unit energy. The least is found in blackscholes (50% floating point operations) and the most in gzip (only INT ops, no memory). The dynamic energy of the consumption of the CGRA8 is $\approx 3.3 \times$ the functional unit energy where the least is blackscholes. The most however is gcc where the link energy dominates due to the high connectivity of the dataflow graph. Similarly for the Chainsaw8, the average is $\approx 2.8 \times$. The net dynamic energy consumption is reduced via internalizing communication within chains.

**Communication Costs** Figure 10b shows the total energy expended by Chainsaw in communication, normalized to CGRA8 communication energy. Each benchmark is represented by a stacked bar, which shows the relative proportions of energy spent in the bus, pipelining forwarding, and register (INs/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 4a.

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×). For 3 out of 20 workloads (equake, blackscholes, fluidanimate), the bus events are more frequent (average 30%) due to small chain formation (average size < 2 ops) coupled with greedy scheduling. The scheduling strategy is tuned for performance which seeks to schedule chains on free lanes to extract maximal ILP.

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

![Graph showing static power consumption normalized to CGRA8.]

Figure 11: Static Power. CGRA8 and CHAINSAW8 normalized to CGRA8. IDLE: the static power expended while waiting for scheduled operations to be ready to run. FREE indicates the static power due to over-provisioning resources compared to the available ILP i.e., the PE or Lane does not have any instruction scheduled to execute.

Figure 11 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 [32] is scaled based on the number of operations to be accelerated while the Chainsaw is scaled based on the instruction parallelism available. Hence, in many cases the CGRA is excessively provisioned and is underutilized unless there is data parallelism to be exploited. This leads to an interesting case where the static power in the CGRA8 (64 units) is dominated by the FREE power; the FREE power may be curtailed by power-gating the PEs or the execution lanes. Chainsaw8 improves overall utilization and consumes much less FREE power but may introduce contention for the lanes or PEs by mapping multiple operations onto the same PE; this leads to an overall increase in IDLE power due to chain operations being stalled due to other unrelated chains occupying the lane. Note that both CGRA and Chainsaw will suffer from IDLE power since they statically map the operations to the resources.

Chainsaw8 reduces overall static energy by \(\approx 21\%\). From Figure 11, we see 11 of the 20 applications reduce energy by 20% to 40%. In equake, povray, hmrmer, 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. For 15 out of 20 workloads, there is more IDLE’ness in the Chainsaw architecture. In 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.

8. Conclusions

In this paper, we presented a new instruction abstraction, Chains, to exploit the producer-consumer locality between instructions. Chains present an effective strategy to localize communication between dependent instructions and save communication energy. Chains also enable the accelerator design to amortize the front-end costs. We have developed an end-to-end compiler prototype based on LLVM that extracts hot paths from applications, decomposes the dataflow graph into chains and prepares them for the Chainsaw architecture. Chainsaw is a lane-based architecture that leverages chains to achieve dynamic energy efficiency proportional to CGRA architectures, while minimizing idle and communication energy.

References
