Compiling Irregular Software to Specialized Hardware

Richard Townsend

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2019

ABSTRACT

Compiling Irregular Software to Specialized Hardware

Richard Townsend

High-level synthesis (HLS) has simplified the design process for energy-efficient hardware accelerators: a designer specifies an accelerator's behavior in a "high-level" language, and a toolchain synthesizes register-transfer level (RTL) code from this specification. Many HLS systems produce efficient hardware designs for regular algorithms (i.e., those with limited conditionals or regular memory access patterns), but most struggle with irregular algorithms that rely on dynamic, data-dependent memory access patterns (e.g., traversing pointer-based structures like lists, trees, or graphs). HLS tools typically provide imperative, side-effectful languages to the designer, which makes it difficult to correctly specify and optimize complex, memory-bound applications.

In this dissertation, I present an alternative HLS methodology that leverages properties of functional languages to synthesize hardware for irregular algorithms. The main contribution is an optimizing compiler that translates pure functional programs into modular, parallel dataflow networks in hardware. I give an overview of this compiler, explain how its source and target together enable parallelism in the face of irregularity, and present two specific optimizations that further exploit this parallelism. Taken together, this dissertation verifies my thesis that **pure functional programs exhibiting irregular memory access patterns can be compiled into specialized hardware and optimized for parallelism.**

This work extends the scope of modern HLS toolchains. By relying on properties of pure functional languages, our compiler can synthesize hardware from programs containing constructs that commercial HLS tools prohibit, e.g., recursive functions and dynamic memory allocation. Hardware designers may thus use our compiler in conjunction with existing HLS systems to accelerate a wider class of algorithms than before.

Contents

List of Figures

List of Tables

Acknowledgements

My path to a PhD (including writing this dissertation) was filled with bumps, forks with no signposts, and plenty of other obstacles that hindered any form of progress. The following individuals helped me find my way over the past 6 years, showing me how to smooth out the bumps, make decisions at each fork, and deal with the obstacles as they appeared.

I cannot thank Stephen A. Edwards enough for subverting every negative stereotype of a PhD advisor; I wouldn't have made it here without him. Things he didn't do: burden me with unnecessary work, research or administrative; force me to work certain hours/days to fit his schedule or expectations; ignore my attempts at regular communication. Things he did do: provide endless advice and guidance, both in my research and general academic decisions; support my research choices (in my later years, as he said he would) and passion for teaching; talk me out of every research slump I experienced.

I appreciate my co-advisor, Martha A. Kim, for her willingness to advise me and provide feedback as though I was one of her primary students. She guided me through much of the work for my first research paper submission (given that its subject was more in her field than Stephen's), helped design the experimental framework for all my papers, and heavily contributed to the aesthetic presentation of every chart, figure, and graph I have constructed.

I would like to thank the other faculty and staff at Columbia that made this dissertation possible. I thank Luca Carloni, Ronghui Gu, and David F. Bacon for agreeing to serve on my dissertation committee; particular thanks to Luca for also serving on the committees for my candidacy exam and thesis proposal defense. Jessica Rosa smoothed out the administrative burden that comes with managing a PhD career; I had no idea that distributing/defending/depositing a dissertation had so many moving parts, and would not have been able to navigate the process without her. Finally, I attribute much of my academic writing style to the inimitable Janet Kayfetz, whose Academic Writing course was a welcome change to my otherwise Computer Science-focused graduate career. I still reference her purple packet of wisdom regularly.

Three of my PhD peers deserve specific thanks. My desk neighbor, Emilio Cota, wrote and defended his dissertation right before I started working on mine; he was always available to give insight into his process. Andrea Lottarini provided similar dissertation-based advice, and acted as a helpful sounding board for some of the key organizational decisions I made in this dissertation. Finally, Paolo Mantovani has been implicitly aiding me through the entire writing process: I usually had his dissertation in a saved Google Chrome tab, which I referenced weekly to validate my writing process. He also took the time to guide me through the use of hardware synthesis tools to obtain some of the experimental results in this dissertation.

Much thanks to Olin Shivers, whose excellent PhD dissertation served as an organizational template for mine (particularly the introduction and conclusions).

My time at Oberlin College (2009-2013) set me up for success as a graduate student. The Computer Science department filled me with the passion for this subject that got me to this point; much thanks to Ben Kuperman, Alexa Sharp, Tom Wexler, Cynthia Taylor, John Donaldson, Bob Geitz, and Richard Salter. Special thanks to Tom Reid, bowling guru and master of the mental game, who taught me mindfulness in a time when I needed it most.

I want to thank all the families that have supported me throughout the past six years. My barbershop families (Voices of Gotham, the Atlantic Harmony Brigade, and my quartet, Madhattan) helped develop leadership qualities that I used in my academic career, and provided stress outlets through singing and camaraderie. My Oberlin family has grown with me in the city for the past 6 years, and I appreciate everyone in the group for being there for me when things seemed academically darkest.

Finally, my family provided immeasurable emotional support; I would not have survived this experience without their love and advice. I particularly thank my mom, dad, and step-mom (Karen, John, and Suzy) for always being available to talk to me when I needed it.

Chapter 1

Introduction

1.1 My Thesis

Pure functional programs exhibiting irregular memory access patterns can be compiled into specialized hardware and optimized for parallelism.

A growing fraction of the area in modern chips is dedicated to application-specific hardware accelerators. These specialized cores consume less energy to perform a task than a general-purpose processor, and energy consumption is of critical, growing concern. To simplify their design, architects have turned to high-level synthesis (HLS) tools that produce circuits from high-level behavioral specifications. While these tools can produce efficient hardware for "regular" algorithms, they struggle with irregular algorithms that use recursion and dynamic pointer-based data structures like lists and trees.

One major issue with these HLS tools is their use of C-like languages as their source: the mutable memory model and side-effectful nature of these languages prevent standard HLS optimizations from exploiting parallelism in the face of irregularity (i.e., recursion and dynamically allocated memory). In this dissertation, I present an alternative HLS flow that uses a new compiler to synthesize hardware from pure functional programs. The compiler provides new optimizations that enable more parallelism in irregular programs and targets a specific model of computation, patient dataflow networks, that exploits this parallelism in hardware.

1.2 Structure of the Dissertation

This dissertation is organized as follows:

- The rest of this chapter provides background to motivate this dissertation and presents the contributions that support my thesis.[1]
- Chapter 2 surveys a selection of previous work related to these contributions.
- Next, Chapter 3 provides an introduction to functional languages (via a detailed presentation of our compiler's main intermediate representation), a high-level overview of our compiler, and the initial compiler passes that prepare a program for its translation into a dataflow network.
- I then dive into the translation from functional programs to hardware dataflow networks in Chapter 4 and present the novel compositional circuits that implement these networks in Chapter 5. Our networks' semantics are explained across these two chapters, both intuitively (Section 4.2) and formally (Section 5.1).
- The next two chapters describe our novel compiler optimizations. Chapter 6 covers how we optimize a specific class of irregular divide-and-conquer algorithms and presents the partitioned memory system our generated circuits use by default. Chapter 7 presents a "packing" algorithm that transforms recursive types (and the functions operating on them) at compile-time to improve the memory efficiency of our circuits.
- I conclude this dissertation in Chapter 8 with a summary of my work and some potential directions for future research.

[1]Because this work is part of a larger project, I use first-person singular pronouns when describing this dissertation's organization and any experimental evaluation (I am the sole author of the dissertation and ran all the experiments myself); all other pronouns will be first-person plural. I mention others' specific contributions as they are presented in this dissertation.

1.3 High-Level Synthesis and Irregular Algorithms

In the mid-2000s, the landscape of computer architecture experienced a paradigm shift: while semiconductor manufacturers continued to pack more, smaller transistors onto chips (following Moore's Law [92]), it became increasingly difficult to switch them all simultaneously at their highest frequency without experiencing significant increases in power consumption (i.e., Dennard Scaling [38] broke down). In other words, current power constraints dictate that only a small fraction of transistors on a modern chip can be powered simultaneously, giving rise to a phenomenon known as *dark silicon* [14, 46].

Specialized hardware accelerators present a solution to this problem: accelerators are small (compared to general-purpose processors) application-specific circuits that have been carefully designed to execute a task (e.g., web search [105], machine learning [23], and database processing [136]) while providing higher performance and lower energy requirements than a general-purpose processor [128]. Unfortunately, the traditional design process for accelerators makes them hard to adopt: architects must work at the register-transfer level (RTL) of abstraction, implementing high-level algorithms with low-level digital logic constructs like combinational gates and flip-flops. This leads to a tedious, error-prone process that precludes the rapid exploration of design trade-offs (e.g., between computing resources and memory) [4].

The high-level synthesis (HLS) design methodology is a promising alternative [32]: designers use high-level languages to describe their specifications, and a synthesis toolchain generates the RTL code that realizes these specifications in hardware. The majority of existing HLS tools synthesize hardware from C-like software specifications (Nane et al.'s survey [96] presents 33 HLS toolchains; 80% use a C-like input language), and have been shown to produce low-power, high-performing cores [88, 128]. HLS researchers tend to concern themselves with accelerating "regular" algorithms, i.e., computations with predictable memory access patterns. Prevalent HLS testsuites reflect this trend [65, 107]; the majority of benchmark programs provided in these testsuites use statically-sized arrays and matrices to structure data.

Modern HLS tools use loop-based optimizations and memory partitioning schemes to improve the performance of their synthesized accelerators. For example, given a simple array sum loop, an HLS tool could unroll the loop to reveal two array accesses per iteration, partition the array so its odd and even elements are stored in independent on-chip memories, and schedule the instructions to execute simultaneously or in a pipelined fashion (based on available resources). When the loop nests exhibit more convoluted access patterns or loop-carried dependencies, polyhedral analysis [25, 30, 31, 130] can guide code transformations that make the array accesses more amenable to pipelining and partitioning.

Irregular algorithms dealing with dynamically sized, pointer-based structures (e.g., lists, trees, or graphs) stymie these kinds of HLS optimizations. These algorithms appear in many settings and have the potential for parallelization [80], but their use of recursion and dynamic, data-dependent memory operations render traditional HLS optimizations (e.g., polyhedral-based) ineffective or overly conservative. Commercial HLS tools like Xilinx's Vivado [138] prohibit dynamic memory allocation (which is necessary to implement truly dynamic data structures) for this very reason. New synthesis schemes and optimizations are thus required to synthesize these irregular algorithms in hardware.

Although others have recently suggested solutions to these issues [1, 36, 133, 142], they exacerbate the problem by using C-like languages as a specification: C's mutable memory model and direct control over pointers inhibits simple static analysis for memory-based optimizations, and the prevalence of side effects decreases opportunities for parallelization in general.

Pure functional languages are better suited for specifying irregular algorithms in this context. Functional languages in general provide higher-level abstractions (e.g., pattern matching, type inference, algebraic data types) that can improve designer productivity and simplify the correct specification of complex, irregular algorithms [54]. A "pure" language prohibits computations with side effects: an expression will always produce the same result when given the same arguments. Thus, compilers can freely reorder, modify, or parallelize more code in a pure functional language without modifying the underlying semantics [7, 59, 100, 101]. Purity also entails an immutable memory model (mutating a value in memory is a side effect) that admits specialized memory architectures and optimizations catered to irregular algorithms.

Pure functional languages thus have the potential to solve the problems faced by HLS systems handling irregular memory access patterns. This dissertation shows how to realize this potential.

1.4 Contributions

To solve the irregular synthesis problem posed in the previous section, we have designed an optimizing compiler that synthesizes SystemVerilog (RTL) code from the pure functional language Haskell. This dissertation describes the compiler, which comprises the following research contributions (visualized in Figure 1.1):

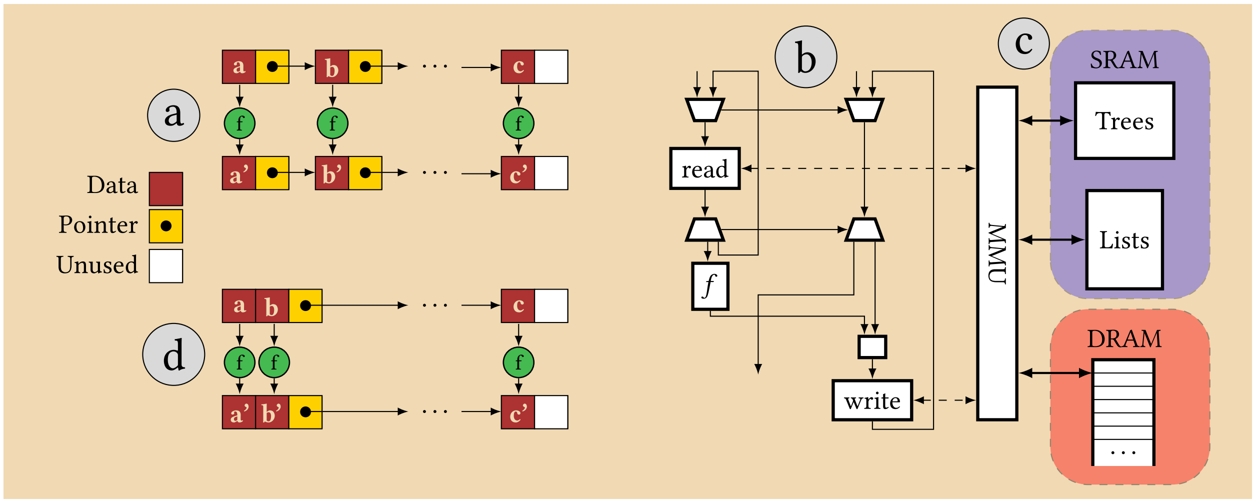


Figure 1.1: A visualization of the research supporting my thesis. I focus on synthesizing hardware from pure functional programs exhibiting irregular memory access patterns, e.g., the function *map f l*, which applies a function *f* to each element of a linked list *l*, storing the results in a new list (a). I translate these programs into modular, parallel dataflow networks in hardware (b), and apply two optimizations that exploit more parallelism in the synthesized circuits: the first improves memory-level parallelism in divide-and-conquer algorithms by combining a novel code transformation with a type-based memory partitioning scheme (c); the second packs more data into recursive types to improve spatial locality (i.e., data-level parallelism) and reduce an irregular program's memory footprint (d).

- **Abstraction-lowering compiler passes.** Our compiler takes in Haskell programs as its source. We use Haskell because its high-level abstractions and pure language model make it easy to correctly implement parallel algorithms operating on recursive data structures [85], e.g., a *map* function that applies a variable-latency operation *f* to each element of a dynamically-sized linked list to produce a new list (Figure 1.1a). To simplify their translation to hardware, we first transform these Haskell programs into a functional intermediate representation (IR) that prohibits constructs with no direct representation in hardware (e.g., recursion, recursively defined data types, anonymous functions). We perform this transformation with a number of abstraction-lowering compiler passes, including a novel algorithm for removing recursion (previously published in the 2015 CODES proceedings [141]) and a simple technique to introduce explicit pointers and memory operations.
- **A translation from functional programs to patient dataflow networks.** This contribution bridges the gap between software and hardware in our compiler. Given a program in our functional IR, we perform a (mostly) syntax-directed translation into abstract dataflow

networks (Figure 1.1b). These dataflow networks are inherently distributed, parallel, and "patient": they can handle long, unpredictable latencies from complex memory systems without any static scheduling or global controller. This property is ideal in our domain, since we target memory-bound irregular algorithms (instead of the more compute-bound scientific algorithms most HLS tools target). A novelty of our approach is how designers "ask for" pipeline parallelism through tail-recursion with *non-strict* functions: our recursive functions can begin execution immediately after their first argument arrives. When such a function calls itself tail-recursively, multiple invocations of the function run in parallel. This work has been published as part of the 2017 CC conference proceedings [125].

- **Compositional dataflow circuits.** After generating these abstract networks, we synthesize them into latency-insensitive [20] circuits that implement a restricted class of Kahn process (dataflow) networks [76]. These circuits may be connected to others with or without buffering, making it easy to consider a variety of designs. For example, buffer-free connections are fast but lead to combinational paths that limit clock speed; inserting buffers breaks long paths at the expense of latency. Our generated circuits retain the "patience" of Kahn's formalism through a valid/ready flow control protocol (i.e., backpressure); local handshaking eliminates any global controller (and thus long signal lines) and enables the insertion and removal of buffering. This accommodates blocks with dynamically varying latency (e.g., memory controllers) and makes it easy to adjust the number of pipeline stages, even in the presence of feedback. We have published work on these circuits in both the 2017 MEMOCODE conference proceedings [44] and a special volume of the TECS journal [43].
- **A framework to accelerate irregular divide-and-conquer algorithms.** We pair a compile-time code transformation with a type-based memory partitioning scheme to optimize recursive divide-and-conquer algorithms operating on recursive data structures. After finding functions that implement such algorithms in a source program, we duplicate the functions and split their input data structures in half ("divide") so each function copy may operate on its half in parallel ("conquer"). Each function becomes an independent circuit in hardware, exploiting task-level parallelism. To prevent a memory-induced bottleneck, we use the rich type information in our functional programs to allocate specific object types (e.g., lists, trees) to dedicated partitions of on-chip memory, and size each partition with a profiling-based heuristic (Figure 1.1c). To avoid local overflow, we back these on-chip partitions with larger, off-chip DRAM, and rely on a cache methodology to determine when to transfer data between on- and off-chip memories.

- **An optimization for recursive data types.** This optimization algorithm modifies the memory layout of recursive data types to reduce the number of high-latency trips to memory and increase data-level parallelism. Modern processors typically rely on caches for a similar purpose (and we use caches in our memory architecture), but the irregular memory access patterns associated with traversing recursive (i.e., pointer-based) structures inhibits the cache's ability to exploit spatial or temporal locality. Our algorithm thus packs recursive types such as lists and trees into cells that hold more data in an effort to improve spatial locality and data-level parallelism at compile-time (Figure 1.1d). This packing algorithm also reduces the total number of pointers in a recursive data structure, which can decrease the circuit's memory footprint and the number of trips made to memory (a common performance bottleneck in irregular algorithms).

The compiler has been implemented with both of the above optimizations included, and I have used it to generate hardware from various Haskell programs exhibiting irregular memory access patterns (due to their use of recursive data structures). This verifies the first part of my thesis: pure functional programs exhibiting irregular memory access patterns can be compiled into specialized hardware.

I have also run experiments that empirically validate the two optimizations described above; the results show that they can improve performance for a variety of Haskell programs realized in hardware. The first optimization exploits task- and memory-level parallelism, while the second exploits data-level parallelism. Taken together, these optimizations verify the second part of my thesis: hardware synthesized from irregular functional programs can be optimized for parallelism.

Chapter 2

Related Work

This chapter presents previous work that most closely relates to my thesis and contributions. I discuss the general problems motivating this dissertation, others' solutions to these problems, and how these solutions differ from or contribute to mine.

2.1 High-Level Synthesis

HLS relates to this work in that both raise the level of abstraction for hardware designers to promote rapid accelerator development and design-space exploration. A typical HLS flow starts with a designer specifying an algorithm in a C-like language; this specification may include hardware-aware constructs like clocks, ports, or timing constraints. The HLS tool analyzes the input program, applies standard compiler optimizations (e.g., common subexpression elimination, dead code removal), and transforms it into a control-flow graph: each node in the graph is an instruction or basic block, and edges indicate the flow of control between instructions (the graph may also include data-dependency information). The tool binds the instructions to hardware resources, and schedules when each instruction will be carried out by its assigned resource. Finally, it produces an RTL circuit specification that respects both the result of resource-binding and scheduling and any of the hardware designer's architectural constraints.

Here, I discuss how others have investigated alternative methods of hardware synthesis. Some use a functional input language to either provide higher-level abstractions to the designer or simplify the verification or optimization of the synthesized circuits. Others retain the imperative approach of typical HLS tools, but propose new hardware architectures to extend the reach of HLS past regular, loops-over-arrays algorithms. Our compiler combines both approaches: we use a functional input language to simplify the design process and reveal new compiler optimizations, and we target irregular algorithms to extend the scope of current HLS techniques.

2.1.1 Functional HLS

Functional programming paradigms have appeared in hardware design research for decades; researchers long ago realized the strong connection between pure functions (those that always produce the same output for a given input) and synchronous digital circuits. These previous works sought to simplify circuit specification via functional constructs (e.g., higher-order functions, algebraic data types) while naturally capturing a circuit's structure and semantics.

Gammie's survey [54] covers much of the historical landscape, focusing on functional languages that take a *structural* approach to digital circuit description: functions represent gate-level constructs (e.g., multiplexers, flip-flops) and operate on streams of data that capture values flowing on wires. Sheeran's μFP language [119] is often touted as the first of these functional hardware description language (HDLs); it leverages higher-order combinators to compose circuit primitives, and prescribes a set of algebraic laws that its specifications fulfill. Due to its focus on these combinators and lack of types, μFP is best for describing simple circuits with highly regular, repetitive structures.

Lava [13, 60, 61] is a family of *embedded* hardware description languages (EHDLs). These EHDLs are Haskell libraries that interpret pure functions as synchronous digital circuits. To capture a notion of time, Lava takes inspiration from the synchronous dataflow language Lustre [63] and provides a special *Signal* data type that defines an infinite sequence of values. Semantically, a *Signal* is a mapping from discrete, global clock cycles to values occurring on a physical vector of wires. Based on the library-defined types used by the programmer, executing a Lava program can either simulate the circuit on specified inputs, verify properties about the circuit, or generate an abstract syntax tree capturing the circuit's structure, which can then be analyzed or fed into other tools for more verification or RTL generation (e.g., to Verilog or VHDL).

Kuper's Cλash project [6, 7] is similar to Lava: it uses Haskell programs for structural circuit specification. However, Cλash has a subtle distinction that brings it closer to our work: instead of solely relying on Haskell's compiler for circuit generation (thus "embedding" the language), Cλash has a dedicated compiler that analyzes the language constructs comprising each Haskell function and synthesizes circuitry for those constructs. Functions thus do not require the special *Signal* type from Lava to be synthesized; a function without a *Signal* is synthesized into a combinational circuit, while the presence of the *Signal* type corresponds to sequential circuitry. While our compiler performs a similar syntax-directed translation to generate hardware, Cλash is still distinct in its use of structural hardware description and its lack of support for user-defined recursive functions and data types (their online tutorial still specifies this restriction [5], although Raa's master's thesis [106] seems to remove it).

Bachrach et al. take a different tack with their Chisel HDL [8] by embedding it in Scala instead of Haskell. Chisel's types capture values flowing on wires (e.g., Bits, Bool, Fix for signed integers); a timing-aware type like Lava and Cλash's *Signal* is not required, as Chisel programs include implicit clock and reset signals where necessary. Instead of Haskell's algebraic data types, Chisel provides an object-oriented model where users can extend base classes to represent collections of data, specific hardware interfaces (e.g., a FIFO input to a circuit), or hierarchical components (similar to Verilog's modules). Functions and classes may be polymorphic, and higher-order functions provide high-level abstractions to simplify the design process.

Unlike the structural approach taken by the above languages, we and others take a *behavioral* approach: designers specify the algorithmic behavior of a circuit (instead of its gate-level structure), and the compiler generates and optimizes the necessary logic to implement that behavior. For example, Kuga et al. [79] synthesize hardware from a subset of Haskell, focusing on the implementation of parallel design patterns like *map*, *zipWith*, and *reduce*. Unlike us, they do not specify whether they can handle recursion or arbitrary algebraic data types.

As a more notable example, the FLaSH compiler of Mycroft and Sharp [95, 117] synthesizes resource-aware hardware circuits from a simple functional language. While their original language (SAFL) was simpler than our compiler's intermediate representation, they later added "channel arguments" to functions to express communication between ports in hardware (SAFL+) [118] and a type-based approach for direct stream processing (SASL) [51, 50]. Their technique for sharing resources (i.e., functions called from multiple places) inspired ours; they place an arbiter at the entry to a shared function, remember which caller gained access, and finally route the result back to the caller. Furthermore, their most recent additions extended the compiler to admit function pipelining and synthesize dataflow networks, bringing them closer to our work. However, their compiler targets hardware with bounded storage requirements: no heaps or stacks are permitted in the synthesized circuits, so they cannot implement recursive data types. My thesis specifically concerns programs with recursive data types (since they elicit irregular memory access patterns); our compiler thus handles a larger class of programs.

The SHard compiler of Saint-Mleux et al. [111] compiles a functional language (Scheme) into a dataflow representation to produce custom hardware. They only implement strict functions: all arguments must arrive at a function before it can begin execution. Our compiler instead leverages a non-strict function policy to reduce execution time and improve throughput by exploiting pipeline parallelism across function calls (see Section 4.1.1 for a specific example). Their treatment of memory is unusual: they only directly support function closures, so data structures such as lists must be coded as closures. Our language uses algebraic data types for data structures, providing a more intuitive approach for the hardware designer.

Bluespec [4] takes an alternative behavioral approach, but still draws inspiration from Haskell to provide a rich type system and inherent parallelism. Designers describe behavior with guarded atomic actions, which are then synthesized into globally scheduled combinational logic blocks. Conversely, our synthesized dataflow networks employ a flow control protocol that effectively acts as a distributed scheduler, eliminating the need for Bluespec's dedicated control logic.

Our translation of a functional language to dataflow networks was inspired by that of Arvind and Nikhil [3], but differs in two important ways. First, they generate dynamic dataflow graphs, i.e., loops and function calls are unrolled on-the-fly as their programs run. We choose a more challenging, higher performance target: physical networks, which means we have to build dataflow graphs with loops that explicitly arbitrate shared resources. Our solution will produce superior results because it avoids general-purpose overhead.

Second, their virtual approach (i.e., using a stored-program implementation) allows them to support unbounded buffers. While this does eliminate the danger of insufficient buffering, it requires the introduction of additional dataflow components to throttle loops and is impossible to implement directly in hardware. Our compiler targets physical hardware with finite buffers and provides a natural throttling mechanism in the form of a flow control protocol.

2.1.2 Irregular HLS

My dissertation shows how to synthesize hardware for irregular algorithms implemented as functional programs; others have instead augmented imperative-based HLS tools to handle these kinds of algorithms. Specifically, four recent works have all proposed novel methods to exploit parallelism in hardware synthesized from irregular C programs (although their definitions of "irregularity" have slight differences). Each leverages the LLVM compiler framework to first translate the input C program into a standardized intermediate representation (IR), which they optimize to generate efficient specialized hardware. They all target loop-based programs and, due to the input language, must grapple with complications caused by a mutable memory model; our compiler instead deals with recursive programs that admit simpler program analysis due to our language's immutable memory model.

Like us, Josipovic et al. [75] describe a synthesis technique that realizes programs as latency-insensitive dataflow networks. Their network building blocks are similar to ours, and they use the same handshaking protocol as us to implement latency-insensitivity. However, their translation process yields inherently sequential networks: their compiler partitions a program's instructions into sequential basic blocks, each basic block is individually translated into a dataflow subnetwork, and special dataflow components are inserted between these subnetworks to implement control flow. They also must correct for potentially out-of-order memory accesses, which can lead to data hazards under C's mutable memory model. Their solution is a complex load-store queue that must be carefully connected to the rest of the network to ensure functional correctness.

The Coarse-Grained Pipelined Accelerator (CGPA) framework of Liu et al. [84] synthesizes novel hardware architectures for C/C++ programs containing complex control flow or irregular memory access patterns. After translating the program to the LLVM IR, their HLS flow implements each loop's instructions with a multi-stage pipeline of hardware "workers" separated by FIFO buffers: sequential workers in one stage supply data to multiple parallel workers in the next, exploiting pipeline parallelism. The sequential workers typically implement irregular data structure traversal, while the parallel workers implement any independent instructions from multiple loop iterations; this decoupling tolerates variable latency (e.g., cache misses may slow down traversal, but the parallel workers can continue executing as long as they have input data in their FIFOs) and enables more parallelism (parallel workers operate independently). Their framework inserts additional LLVM primitives to aid in their analysis, and they impose instruction scheduling constraints to ensure the correctness of their synthesized pipelines. Our synthesized dataflow networks perform dynamic scheduling on their own (no static scheduling is required), and our input language is side-effect free, simplifying our translation to hardware.

Tan et al.'s ElasticFlow HLS tool [122] is similar to CGPA. Given a loop nest with a regular outer loop (i.e., it does not exhibit loop-carried dependencies) and at least one dynamic-bound inner loop, they synthesize a multi-stage pipeline where each inner loop becomes a "loop processing array" (LPA), and all other operations in the loop nest are synthesized into traditional, fixed-latency pipeline stages; the stages are then connected via FIFOs. The LPA architecture is their main contribution: it contains multiple loop processing units (LPUs) that can each execute an inner loop to completion (instead of just some of its instructions), a distributor that dispatches inner loop invocations (one per outer loop iteration) to idle LPUs, and a collector that ensures inner loop results are passed to the next pipeline stage in-order with the help of a reorder buffer (ROB). They use an integer linear programming technique to determine the number of LPUs for a given LPA and the size of each LPA's ROB, maximizing the dynamic throughput of the LPUs under a given hardware area constraint. They improve upon CGPA by handling out-of-order execution for entire loop nests (as opposed to just individual instructions) and achieving higher resource efficiency with special LPUs that can implement one of many inner loop nests.

Zhao et al. [142] present a similar C++-based HLS architectural template, but they specifically focus on decoupling complex data structures (e.g., priority queues and trees) from the algorithms that use them. In their work, a data structure is *complex* if any of its functions exhibit long or variable latency and contain variable-bound loops or memory dependencies. Their templates have four components (like those in ElasticFlow's LPAs), which communicate via latency-insensitive handshaking (like our dataflow networks): mutator function units and accessor function units implement the data structures' mutator and accessor functions; a dispatcher receives function calls from the algorithm and passes them off to the appropriate function units, respecting function dependencies; and a collector receives results from the function units and passes them back to the algorithm. The dispatcher may overlap execution of multiple accessor units, but mutator functions cannot be overlapped with any other since they may modify memory. We rely on an immutable memory model in our work to avoid this restriction; if two writes of different type are available (e.g., writing a tree cell vs. a list cell), we can service them in parallel.

2.2 Hardware Dataflow Networks

Our compiler translates Haskell programs into latency-insensitive dataflow networks in hardware. Dataflow networks are a natural model for parallel, distributed computation: processes in a network (called "actors") execute in parallel and communicate via sequences of tokens passed over unbounded channels. These networks are well-suited to specifying complex hardware designs because of their "patience": process speed has no effect on network function. While the underlying formalism of these networks is well-defined [39, 76, 81, 82], different approaches have been taken to realize these networks in physical hardware.

Tripakis et al. [126] survey a number of these dataflow-to-hardware projects; most focus on statically schedulable models such as SDF that do not support data-dependent actors, e.g., multiplexers and demultiplexers. Carloni et al. and Carmona et al. champion patient dataflow networks following this model with their respective Latency-Insensitive Design [19, 20] and Elastic Circuits projects [21]. Both of these works implement the patience of the abstract dataflow model with a handshaking protocol. Possignolo et al. [104] also consider token/handshaking pipelines for processor design: they start with a synchronous circuit with no handshaking and transform it into a patient dataflow network by introducing four actor circuits (unit-rate, fork, demultiplexer, and merge) based on designer annotations. Although their fork and merge actors can induce deadlock in general (a danger in any dataflow design), they provide a set of design rules that prevent deadlock. They use Colored Petri Nets to model throughput, an augmented form of the model Collins and Carloni used to analyze and optimize their latency-insensitive systems [27].

While many handshaking protocols exist to implement latency-insensitivity, one of the most common ones uses a *valid* bit to indicate that a process is sending a token downstream, and a *ready* bit to indicate that the downstream process can consume that token. This 2-bit protocol is standard in asynchronous systems [115]. Intel's 8008 used a similar protocol in 1972 to wait for slow memory [70], but the protocol likely appeared even earlier. We use this protocol in our networks, specifically taking inspiration from Li et al. [83], but the same system can be found in Cortadella et al. [34, 35], Dimitrakopoulos et al. [40], ARM's AXI4-stream protocol [2], and the FIFOs provided in Altera and Xilinx FPGAs (Field-Programmable Gate Arrays).

Careful implementation of this handshaking protocol is required to prevent combinational cycles, e.g., due to a *valid* bit depending on a *ready* bit and vice versa. ForSyDe [86, 112, 113] avoids these handshaking-induced combinational cycles by always inserting delays on channels. These channels are not user-visible: their system presents the user with a synchronous model of computation (i.e., unit-rate dataflow with no decisions). This makes it difficult for a user to specify variable-rate processes in ForSyDe. Our networks rely on a special pair of buffers and a three-phase evaluation order (data, then *valid*, then *ready*) to prevent combinational cycles, yielding faster designs than the fully-buffered networks of ForSyDe.

The above works apply latency-insensitive design practices to existing hardware systems; others are closer to our work in their use of patient dataflow networks as targets for high-level synthesis. Keinert et al.'s [78] SystemCoDesigner employs behavioral synthesis (Forte's Cynthesizer product) to synthesize hardware for coarse-grained dataflow actors expressed in SystemC with the SysteMoC library [47]. Inter-actor communication is done through FIFOs taken from a library [67]. Janneck et al. synthesize networks from Cal [45]: a rich, functional-inspired language for expressing dataflow process actors and networks. They have a hardware synthesis system for these networks [12, 71, 72], although little has been published about its internals. Thavot et al. [123] instead synthesize hybrid hardware/software systems from Cal; they are unique in that all of their actors are nondeterministic, going against the typical desire to retain determinism across all dataflow actors and the network itself.

Our dataflow networks depart from each of the aforementioned works. We provide data-dependent actors that can make choices, which cannot be modeled in the typical SDF framework, and include a single nondeterministic actor to share resources and help implement recursion in hardware. Our actors are fairly lightweight due to our use of latency-insensitive buffers, making simple actors like adders and multiplexers practical. Finally, our actors are compositional: each actor becomes a circuit that may be connected to others with or without buffering, and combinational cycles arise only from a completely unbuffered cycle. We formalize our networks, present our actor implementations, and argue for their correctness in Chapter 5.

2.3 Parallelizing Divide-and-Conquer Algorithms

Divide-and-conquer ("DAC") algorithms are intuitively simple to parallelize: after breaking down a task into distinct subtasks, execute the subtasks in parallel before merging the final result. Depending on the implementation of the algorithm, though, it can be difficult for a compiler to automatically find and enable this parallelism. Much work has been done to solve this issue, mostly in purely software-facing frameworks, although a few others have specifically leveraged specialized hardware to parallelize DAC algorithms. Here, I first list some of the techniques the software community has devised; then, I discuss how previous work on memory partitioning can be applied to DAC parallelization, even if that was not the main motivation for the work; finally, I present how others have parallelized DAC algorithms with the help of specialized hardware, which most closely resembles the work I present in Chapter 6.

2.3.1 Software Techniques

Many software techniques rely on the use of specialized language constructs to find and exploit DAC parallelism. Language extensions like Cilk [53] (C++), Satin [127] (Java), and Tapir [114] (LLVM) add extra primitives to express "fork-join" parallelism: *spawn* indicates that a function call can operate in parallel with surrounding statements (or be assigned to a dedicated core), while *sync* specifies where execution must stall in a given function until all spawned processes have terminated. Multiple recursive calls in a DAC function can use *spawn* to execute in parallel, and *sync* can merge their results. Morita et al. [94] take a significantly different tack: they parallelize DAC algorithms on lists, but only if the algorithm is expressed with a pair of sequential functions that perform computation by scanning the list leftwards and rightwards. Their programs are written in a restricted language that forces the user to express DAC functions with this list-scanning paradigm, from which they generate parallel C++ code to run on a distributed system. Collins et al. [28] also generate parallel C code for DAC algorithms. Their Huckleberry tool takes in DAC functions written with a special API, and produces code that distributes data for independent subtasks across multiple cores.

Other software techniques *automatically* parallelize DAC functions by relying on the compiler to find subtasks that may safely execute in parallel. For example, both Gupta et al. [62] and Rugina and Rinard [109] focus on automatically parallelizing recursive DAC algorithms in C programs. As a result of C's mutable memory model, both of these works rely on complex pointer analysis and other data-dependence compiler algorithms to verify that no two subtasks of a DAC function ever write or read the same section of an array simultaneously. Otherwise, if the subtasks were executed simultaneously, a data race could occur and break the program's functionality.

Our technique deviates from these works in two ways. First, it eschews special language constructs to find DAC parallelism; it instead finds this parallelism by analyzing the structures of general Haskell programs. Second, our compiler's immutable memory model prohibits the overwriting of any live data; we can exploit more parallelism than Gupta et al. or Rugina and Rinard by copying shared data to different memory partitions without fear of races.

2.3.2 Memory Partitioning

In general, the HLS community has focused less on DAC function parallelization specifically, and more on how to partition on-chip memory so multiple segments of a (typically statically-sized) data structure can be accessed in parallel. Such on-chip memory partitioning can exploit memory-level parallelism in DAC functions; I discuss some notable work on this subject here.

Most of the previous work on memory partitioning in HLS frameworks has focused on accelerating highly regular, loops-over-arrays programs [25, 29, 31, 90, 130]. If a loop nest accesses an array and does not exhibit loop-carried dependencies, then the loop may be unrolled to reveal more independent accesses per iteration and enable instruction-level parallelism. The memory system exploits this parallelism with banking: the array is distributed across multiple memory banks such that the multiple elements accessed on a given iteration reside in separate banks; this leverages the high memory bandwidth provided by modern FPGAs. If there are data-dependencies in the original loop nest, various linear algebraic transformations may be applied to restructure the array's access pattern into a form that is more amenable to memory banking. These techniques rely on the array residing in contiguous memory and a highly regular access pattern; my dissertation specifically targets programs operating on dynamic data structures that may be distributed throughout the address space and accessed in an irregular fashion.

Others have applied memory partitioning to programs with less regular access patterns. Zhou et al. [143] use a trace-driven technique to exploit memory-level parallelism in loops with non-affine access patterns, i.e., the addresses used to access the array are not affine functions of the loop's iteration variable. Instead of performing static analysis and applying linear algebraic transformations to loops over arrays, they instrument the program to obtain a memory access trace and use important address bits in the trace to guide their banking and array segmentation. Ben-Asher and Rotem [11] present a similar trace-based method that applies to both array and dynamic data structure accesses. For the dynamic data structures, they rely on the assumption that the structures are created with custom memory allocators that always place the data structures at consistent, structure-aligned addresses. This lets them treat any data structure as an array of C structs, simplifying their partitioning algorithm. In our partitioning scheme, we make no assumptions on addresses of the dynamic data structures generated by the input program.

2.3.3 HLS for Divide-and-Conquer

Four specific prior works are closest to ours; they parallelize DAC functions either with specialized hardware support or as part of a full HLS toolchain. Luk et al. [87] parallelize DAC functions with a software/hardware co-design technique: a CPU divides input data into partitions, the partitions are passed to an FPGA-based accelerator that "conquers" the data with a homogeneous network of tightly-coupled functional units, and the results are passed back to the CPU for merging. They use a functional language to present their strategy. Our work is completely hardware based (i.e., our synthesized hardware does not communicate with a general-purpose processor), and we use loosely-coupled, heterogeneous dataflow networks to perform parallel computation.

Two works on exploiting "dynamic parallelism" in HLS flows can be directly applied to DAC functions. Margerm et al. present TAPAS [89], an HLS framework that synthesizes parallel accelerators coded in Chisel from Tapir programs (the LLVM IR extended with instructions for fork-join parallelism). Their synthesized architecture is based on a task-level abstraction: a program becomes a collection of "task units" that can operate in parallel and pass data between each other through shared memory. The key feature of these task units is their ability to spawn new tasks at runtime through a message-passing system. The spawned tasks are implemented with fully pipelined dataflow networks that use a handshaking protocol like ours; buffering every channel in their networks leads to higher pipeline parallelism but increases the latency of their designs. They show that their architecture can parallelize a recursive mergesort algorithm operating on an array, with the recursive calls being spawned into distinct task units.

Chen et al.'s ParallelXL system [22] is similar to TAPAS: they also target dynamically parallel programs for acceleration and use a task-based computational model. They are novel in their use of "continuation-passing style": when a task is spawned, it receives a special continuation argument that indicates where the spawned task's return value will be sent. This model naturally supports recursion: recursive calls spawn tasks whose continuations point to the task that will merge their results. Chen et al. also diverge from TAPAS in their use of "work-stealing", where idle task units can randomly steal work from other units, perform the stolen computation, and send the results back to the original unit. This allows their architecture to exploit parallelism in DAC functions with load-balancing issues. They use multiple caches to exploit memory-level parallelism, assigning one to each task unit (instead of TAPAS's single cache shared among all units). We also use multiple caches, but allow cache sharing across our networks.

Winterstein et al.'s work [134, 135, 139] is by far the closest to our contribution, as they focus on an HLS scheme for parallelizing DAC functions that operates on dynamic data structures. We adopt their cache sizing algorithm (Section 6.2) but operate in a different domain: they analyze loop-based C++ programs, while we deal with recursive Haskell programs. They make copies of a DAC function's subloop if the compiler can determine that memory referenced in one subloop is never referenced in another, after which they give each loop copy and type a dedicated cache. Due to the mutable memory model of C++, their analysis relies on a complex "separation logic" to determine if two subloops do not conflict. Conversely, the recursive functional programs we implement need only trivial dependency checks; we use a type-based scheme to duplicate and assign data to distinct caches, which, combined with our immutable memory model, ensures that function copies (operating on different but structurally identical types) never share caches. We also assign multiple types to each cache to enable larger cache sizes and reduce capacity cache misses.

2.4 Optimizing Recursive Data Structures

The irregular algorithms we target for hardware compilation heavily involve recursive data types like lists and trees. We are thus interested in optimizing their representation in hardware to improve our circuits' performance. While we are the first to consider optimizing recursive types in hardware (to my knowledge), others have considered a similar goal in software. These previous works usually involve "packing" recursive data types (typically, just lists) to hold more data per cell, which entails two major benefits: the additional data in each cell enables data-level parallelism, and fewer cells are required to implement a packed structure.

Shao et al. [116] present a compile-time analysis that uses "refinement types" to pack lists in (functional) ML programs. Although they claim that they can pack k elements into each list cell, their presentation and experiments only use $k = 2$ and rely on list parity: even-length lists are packed into cells of two elements, while odd-length lists add a single extra element to the front of an even-length list. Our experiments explore the impact of higher values of k, and extra elements that cannot be packed into a larger cell may appear anywhere within our transformed data structures (not just at the front). Their code transformation uses a refinement type inference algorithm to determine the parity of lists at compile-time, and transforms functions to have three entry points: one for even lists, one for odd lists, and one for lists of unknown parity. Conversely, our algorithm inserts compiler-generated functions into a program to convert between the original and packed versions of a recursive type, then moves calls to these functions around in a semantics-preserving manner to yield a program that only uses the packed version. Their algorithm produces similar code growth numbers as ours when we pack lists to store two elements per cell; it is unclear whether their work is applicable to more general recursive types like trees, which ours can handle.

Hall's [64] work focuses less on the packing process itself and more on determining where the packed versions of a list should be used. Hall assumes multiple variants of each list function in a standard library: the original operates on "simple" lists (the original type), while others replace one or more of the list types in their type signature with a "compressed" list (identical to how we pack list types to store two elements per cell; we leverage Hall's type in our work). Hall then adds a polymorphic type variable to the original list type throughout the program, which Hindley-Milner type inference [91] may resolve to either a compiler-defined *Simple* type (indicating that the original list type should be used) or a type variable (indicating that the compressed list type may be used). After running the type inference algorithm, the type signatures of each function indicate which kind of list representation may be used for that function. Our algorithm changes all list types to their packed version, generates the packed versions of functions automatically, and extends Hall's focus to other recursive types like trees.

The list compaction methods presented by both Braginsky and Petrank [15] and Platz et al. [103] focus on exploiting spatial locality while providing concurrent operations on lists. They collect list elements into *chunks*, where each chunk is a block of memory containing multiple subsequent list entries. This improves spatial locality, as a single cache line is guaranteed to contain multiple list elements, but insertion and deletion may require splitting or merging chunks, leading to more memory accesses and higher execution time. Furthermore, their implementation requires extras bits to implement concurrent operations, further increasing the size of each list cell. Our algorithm does not introduce any extra instructions to modify a packed structure at runtime, and our structures admit safe, concurrent accesses automatically (i.e., with no additional bits) due to our immutable memory model.

Fegaras and Tolmach [48] present a vector representation for lists: the vectors are implemented as arrays, eliminating all space overhead due to pointers. However, they can only translate list functions into vectorized form if the function expresses a common computation pattern called a "catamorphism"; our algorithm can transform any list function to operate on packed lists. The functionality of their algorithm resembles ours, though: apply a series of semantics-preserving transformations to generate code that operates on vectors instead of lists (we generate code that operates on packed lists instead of unpacked lists). Compared to ours, their technique can completely eliminate inter-cell pointers, but it adds more runtime checks and cannot handle functions that share pointers, e.g., a function that appends two lists together.

Inlining to eliminate inter-object pointers also benefits object-oriented languages. Dolby's algorithm [41] reduces pointer overhead in objects containing other objects by inlining the latter in the definition of the former. Compared to our work, Dolby requires more analysis to ensure inlining is safe, and even if so, he must maintain aliasing information and field references to preserve semantics. Our pure functional setting is far simpler.

A key step in our packing algorithm inlines recursive function calls to produce a structure mimicking the packed types we generate. This inlining step is similar to the "recursion unrolling" algorithm of Rugina and Rinard [110], which inlines calls to recursive C functions implementing divide-and-conquer functions. Their goal is to generate larger, more efficient base cases that operate on more data per recursive call; we instead focus on modifying recursive functions to traverse larger cells in dynamic data structures. Their inlining may introduce multiple, identical conditional statements in a function; a main contribution of their work is a method of detecting that these statements are identical and fusing them into one statement to prevent unnecessary conditional computation. We perform a similar optimization in our algorithm after our inlining step terminates.

Chapter 3

An Overview of Our Compiler

This chapter provides an overview of our compiler, which translates pure functional programs into hardware dataflow networks. The compiler is designed as a sequence of abstraction-lowering program transformations. Most of these transformations are rewriting steps: they take a program written in the compiler's main intermediate representation (IR); modify, remove, or add code in a semantics-preserving manner; and produce a transformed program written in the same IR or a more restricted dialect. The final transformations are direct translations, first from a functional IR to a dataflow IR, then from the dataflow IR to SystemVerilog code.

I first present the compiler's main IR, "Core" (Section 3.1); this presentation introduces functional language concepts and summarizes all the language features that our compiler can handle. I then walk through the compilation of a simple example program, outlining the steps that transform it from Haskell to hardware (Section 3.2). Many of these steps remove Core features that would otherwise complicate optimizations and later translations in the compiler; I finish this chapter by detailing these steps (Section 3.3), which together bridge the gap between Core and its more restricted dialect, "Floh."

3.1 Our Main IR: a Variant of GHC Core

Our compiler uses a strongly typed, pure functional language called "Core" as its main IR. Core is a pared down version of GHC's External Core IR [124]; Figure 3.1 depicts its abstract syntax. The rest of this section describes this syntax, its connection to the lambda calculus [24], and how this connection entails purity.

A Core program begins with a possibly empty sequence of algebraic data type (ADT) definitions (*type-def*). ADTs are a powerful feature of modern functional languages that subsume records, enumerations, and union types. An ADT is named with a type constructor (*Tcon*) and defines one or more variants (*con-def*) that specify how to construct values of this new type. Each

program	::=	*type-def** *var-def*$^+$	
type-def	::=	**data** *Tcon tvar** = *con-def* ($\mid$ *con-def*)*	Type Definition
con-def	::=	*Dcon type**	Variant Definition
var-def	::=	*vid* = *expr*	Variable Definition
expr	::=	*vid*	Variable Identifier
		lit	Integer Literal
		Dcon	Data Constructor (capitalized)
		expr expr	Application
		λ *vid*$^+$ $\to$ *expr*	Lambda
		let (*vid* = *expr*)$^+$ **in** *expr*	Variable binding
		case *expr* **of** (*pattern* $\to$ *expr*)$^+$	Conditional
pattern	::=	*Dcon* (*vid* $\mid$ _)*	Constructor Pattern
		lit	Literal Pattern
		_	Default
type	::=	*Tcon*	Algebraic type constructor (capitalized)
		tvar	Polymorphic type variable
		type $\to$ *type*	Function type
		type type	Type application

Figure 3.1: The abstract syntax of the compiler's main IR: a variant of GHC's Core [124]. We augment this grammar with the regular expression meta-operators * (zero or more), | (choice), and + (one or more). Note that the | token in the *type-def* rule is actual Core syntax, not the choice meta-operator.

variant has a globally unique name called a data constructor (*Dcon*) followed by zero or more type fields. Type fields are either concrete (composed only of type constructors that name other ADTs or primitive types) or polymorphic (containing one or more type variables). Any type variable (*tvar*) used in a variant must appear as an argument to its type definition. If type *T* has a constructor *C* with a type field referring to *T*, then the type, constructor, and type field are all said to be "recursive."

ADTs capture both traditionally "primitive" types and more complex data structures. The familiar Boolean type is built into our standard library as an ADT with two variants, each defining a constant data constructor with no type fields:

```
data Bool = True | False
```

Two other common (polymorphic) examples are singly-linked lists and binary trees. Here, the type variable *a* represents an arbitrary type, allowing for lists of, say, 8-bit integers:

```
data List a = Nil | Cons a (List a)
data Tree a = Leaf | Node (Tree a) a (Tree a)
```

These are both examples of recursive types, e.g., a list is either an empty *Nil* cell or a *Cons* cell containing a value of polymorphic type and a reference to the rest of the list. The polymorphic type variable is resolved to a concrete type based on the data stored in the list, e.g., a list of integers would have type *List Int*, while a list of integer lists would have type *List (List Int)*. The tree type is similar (either empty or carrying a polymorphic value), but its recursive *Node* variant has two recursive fields corresponding to a left and right branch.

Along with Boolean, our standard library provides 8-, 16-, and 32-bit signed and unsigned integers, polymorphic lists, and the polymorphic *Maybe* type that captures optional values:

```
data Maybe a = Nothing | Just a
```

Variable definitions (*var-def*) include functions and comprise the rest of a program. Each binds an expression to a variable name *vid* throughout the program. A program must contain a *main* variable definition; running the program amounts to evaluating the *main* expression.

Core expressions are terms from the typed lambda calculus augmented with a few additional language constructs. The untyped lambda calculus has just three terms: variable names, lambda expressions, and function application. *Lambda expressions* are unnamed (sometimes called "anonymous") functions, e.g, "λ x → x + 1" is a function that takes a single argument, names it "x," and returns the result of incrementing it. Function application is written as left-associative juxtaposition, e.g., "(λ x y→ x + y) 3 5" is the application of a two-argument function to 3 and 5. To evaluate this application expression, we replace any occurrence of the lambda's parameters (x and y) in its body with the two arguments (3 and 5), yielding the simple addition "3 + 5." This form of evaluation via substitution is fundamental to Core's semantics and the notion of purity.

In the formal untyped lambda calculus, named functions like "+" and literal constants like "1" do not exist; in Core, integer literals are primitive expressions and lambda expressions may be bound to variable names, which may then be referenced in other expressions, e.g.,

```
f = λx → x + 1 −−f now refers to the increment function

f 3 −−equivalent to applying the lambda directly
```

The typed lambda calculus simply adds type information to all the terms in an expression. The above function f would assign the type *Int* to x and 1, and the entire function would have type "Int → Int," read as "the type of a function that takes a single Int argument and returns an Int." Similarly, the two-argument addition function would have type "Int → Int → Int." These types are explicit in the typed lambda calculus's syntax; I omit them in Core examples, as they typically clutter the code and are inferred anyway. When helpful, I will include type signatures for variable definitions using the "::" or "type of" operator, e.g., "f :: List a -> Int" means "f has the type of a function that takes a polymorphic *List* argument and returns an *Int*." In general, the number and type of arguments given to each function call must be consistent with that function's type, which is inferred from its definition.

Core extends this basic calculus with four additional expression forms: integer literals (*lit*), data constructors (*Dcon*), *let*, and *case*. A data constructor behaves like a function that creates objects: if type $Tcon$ has a variant defined as $Dcon\ t_1 \ldots t_k$, the expression $Dcon\ e_1 \ldots e_k$, where expression e_i is of type t_i, creates an object of type $Tcon$. Higher-order constructors and functions are prohibited, i.e., a variant cannot have a type field of function type, and functions may not take other functions as arguments or return them as results. This means partial function application is also prohibited.

A *let* expression introduces local variables by binding one or more expressions to names; each name is then in scope in the *let*'s body (the *expr* following the **in** keyword). Local functions may be defined this way, and local names can shadow the same name in outer scopes:

```
g = let f = λx y → x + y −− local function definition
        g = 7 −− shadows the outer g
    in f g 6  −− this g refers to 7; the outer g names the result , 13
```

A *case* expression is a multi-way conditional that selects an expression to evaluate according to a matching pattern. It first evaluates its "scrutinee" expression (the *expr* between the **case** and **of** keywords), then compares the form of the result to a set of one or more alternatives in order (top-to-bottom). Each alternative comprises a pattern and an expression; the first pattern that matches the scrutinee is selected, and the associated expression is evaluated.

A set of patterns may either be literals or constructors (they cannot mix). The wildcard pattern "_" matches anything, and may be used as the whole pattern or to ignore fields of a constructor. For example, the *factorial* function pattern matches on its scrutinee *x*, returning 1 if *x* evaluates to 0 and otherwise multiplying *x* against the result of a recursive call:

```
factorial  = λx → case x of
                  0 → 1
                  _ → x *  factorial  (x−1)
```

When matching against data constructor patterns, the *case* extracts the fields of the data constructor expression associated with its scrutinee; each field is either ignored with the wildcard pattern or bound to a new local variable. For example, the *length* function below scrutinizes its list argument *list*, returns 0 if *list* is empty, and otherwise adds 1 to the result of recursing on the rest of the list. We use the wildcard pattern to ignore the data field of a *Cons*, and the variable pattern *xs* to name its recursive field so we can use it in the alternative's expression:

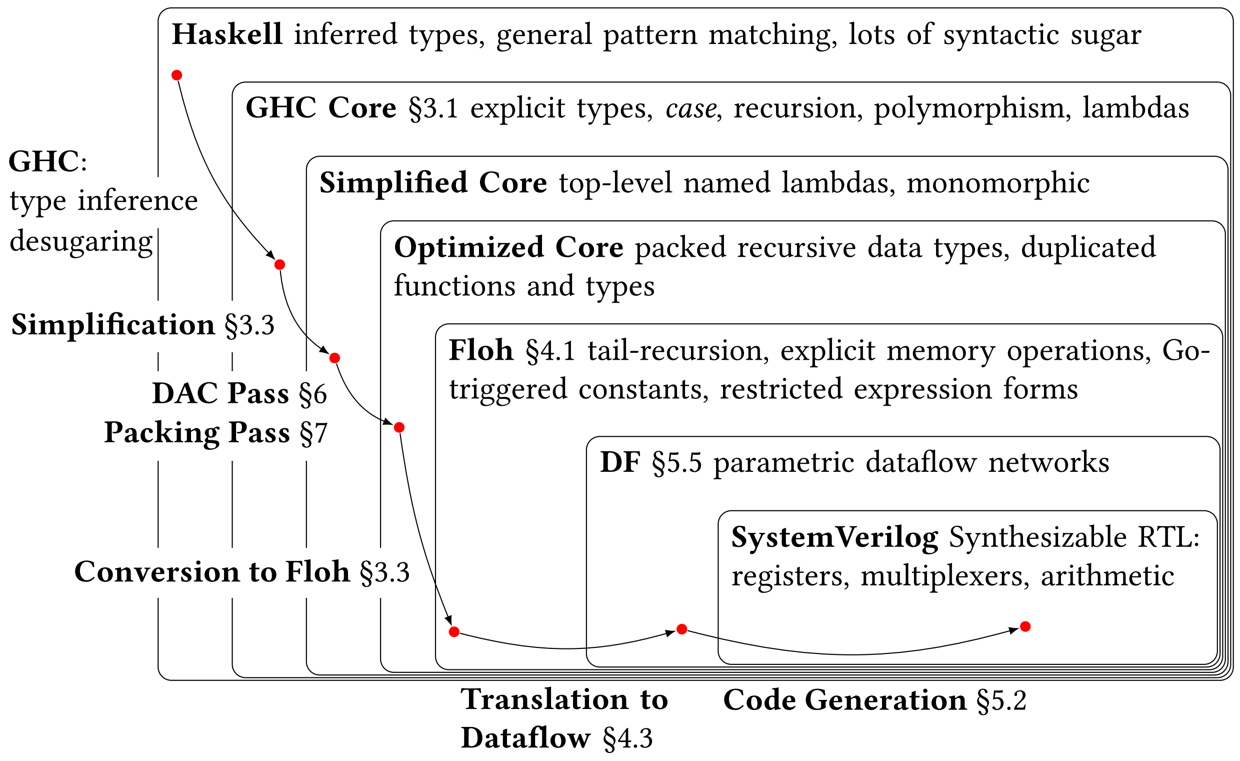
```
length = λ list  → case  list  of
                   Nil       → 0
                   Cons _ xs → 1 + length  xs
```

At the start of this section, I described Core as a pure language; I now define this term and how it affects the language and our compiler. A function is *pure* if it fulfills two conditions:

1. The function always returns the same result when called with the same set of arguments.
2. Evaluating the function has no side effects.

Likewise, an expression is pure if it always evaluates to the same value and has no side effects. A function or expression has a *side effect* if, when evaluated, it modifies some aspect of the program's state (e.g., mutating a global variable).

By defining Core to be a *pure language*, we ensure that every expression and function in a Core program is pure, which has the following implications:



Haskell inferred types, general pattern matching, lots of syntactic sugar
GHC Core §3.1 explicit types, *case*, recursion, polymorphism, lambdas
Simplified Core top-level named lambdas, monomorphic
Optimized Core packed recursive data types, duplicated functions and types
Floh §4.1 tail-recursion, explicit memory operations, Go-triggered constants, restricted expression forms
DF §5.5 parametric dataflow networks
SystemVerilog Synthesizable RTL: registers, multiplexers, arithmetic
GHC: type inference desugaring
Simplification §3.3
DAC Pass §6
Packing Pass §7
Conversion to Floh §3.3
Translation to Dataflow §4.3
Code Generation §5.2

Figure 3.2: Overview of our compilation flow: we rewrite Haskell programs into increasingly simpler representations until we can perform a syntax-directed translation into SystemVerilog.

- All variables are immutable.
- Core expressions are *referentially transparent*: an expression can always be replaced with its corresponding value without affecting the program's result. If a name is bound to an expression, the name and expression can replace one another freely in the name's scope.
- If there are no data dependencies between two expressions, they can be evaluated in parallel without worry of interference or data races.

We rely on these properties throughout the compilation process; purity simplifies our translations, provides opportunities for optimizations specifically catered to the irregular programs my dissertation concerns, and makes our programs inherently parallel.

3.2 The End-to-End Compilation Flow

Figure 3.2 visualizes our compiler as a sequence of abstraction-lowering transformations that convert a Haskell program into a SystemVerilog circuit specification. Here, I apply these transformations to a simple example, only showing the portions affected by each step. As more abstractions are removed, the example will get larger, so I will focus in on smaller portions to avoid overloading the reader. The point is to convey the general compilation process and present the key aspects of transformations applicable to this example. In later chapters (denoted in Figure 3.2), I provide the full treatment of both the transformations and the IRs they generate.

Consider this Haskell program, which computes the length of a list of integers:

```
data List = Nil | Cons Int List

length :: List → Int
length Nil = 0
length (Cons _ xs) = 1 + length xs

main :: Int
main = length (Cons 1 (Cons 2 (Cons 3 (Cons 4 Nil ))))
```

This *length* function is semantically equivalent to the one shown at the end of Section 3.1, but specifically operates on integer lists and uses syntactic sugar: instead of an explicit lambda and *case* expression, a programmer can define a function with multiple bodies, each corresponding to a different input pattern. As in Core, a *main* definition names the result of the program, which here is the length of a four-element list. While Haskell provides strong type inference mechanisms, we present explicit type signatures above the definitions in this example for clarity.

Our compiler's front-end passes this program off to the Glasgow Haskell Compiler (GHC) [100], which parses, typechecks, optimizes, and transforms it into the External Core IR [124]; we pare this IR down to our version of Core (Section 3.1) before linking it with a subset of Haskell's standard libraries, also in Core form. We use an older version of GHC (7.6.3) since later versions removed the ability to dump External Core files. This choice prevents the use of some of Haskell's newer features (thus our omission of some of its standard libraries), but has no bearing on this dissertation's goal to show that irregular functional programs can be compiled into specialized hardware and optimized for parallelism.

The Core version of this example program has the same *main* and *List* definitions, but *length* has been desugared to reveal its underlying implementation with a lambda and *case*:

```
length :: List → Int
length = λ list → case list of
                    Nil        → 0
                    Cons _ xs  → 1 + length xs
```

The compiler next transforms the program to simplify its form. Polymorphic constructs are replaced with specialized, monomorphic forms, all variable names are made unique, and a "lambda lifting" pass names every unbound lambda expression. Here, the lambda lifting pass removes *length*'s lambda and moves its parameter to the other side of the equals sign:

```
length list = case list of ···
```

The program is now ready for two optional optimizations which serve as two of this dissertations major contributions. One optimization applies to divide-and-conquer algorithms (Chapter 6), and thus is not applicable in this example. The other, detailed in Chapter 7, packs recursive types to store more data per cell and modifies functions to operate on these packed types:

```
data PList = PNil | UCons Int PList | PCons Int Int PList

length :: PList → Int
length list = case list of
                PNil         → 0
                UCons _   xs → case xs of
                                 PNil        → 1
                                 UCons _   ys → 2 + length ys
                                 PCons _ y ys → 2 + length (UCons y ys)
                PCons _ _ xs → 2 + length xs

main :: Int
main = length (PCons 1 2 (PCons 3 4 PNil))
```

The *PList* data type comes in three flavors: *PNil* and *UCons* capture the original list's base and recursive variants, while a packed *PCons* contains two integers and a reference. The *length* function can now count two elements at once (when given a *PCons*), and the input to *length* has been packed into two *PCons* cells instead of four *Cons* cells. The packing algorithm also introduces a new, nested *case* expression, which the reader can ignore for now; Chapter 7 will provide the full details to explain where this *case* came from. For the rest of this example, I assume that the packing optimization is turned off.

The next section of the compiler further simplifies Core programs on two fronts: we remove recursive language constructs that are difficult to translate directly into hardware, and add new features that simplify our translation into dataflow. We motivate the removal of general recursion in Section 3.3; here, we simply show how this removal affects our example.

Here is some terminology to help define this recursion removal pass:

- If a function f contains a call to f, that call is *directly recursive*.
- If a function f calls some function g that in turn calls f, then the call to g is *indirectly recursive*. This extends to chains of function calls, e.g., if f calls g, which calls h, which then calls f again, the first call to g is still indirectly recursive.
- A function call is a *tail call* if it is the last expression evaluated in a function's definition.
- A function is *tail-recursive* if it contains one or more directly or indirectly recursive calls, all of which are tail calls.

The *length* function (as it stands) is not tail-recursive: it contains a directly recursive call, but it is not a tail call since the result is passed to *length*'s addition operation. Our recursion removal pass [141] transforms *length* into a collection of tail-recursive functions with an explicit stack:

```
data Stack = K0 | K1 Stack

length :: List → Int
length list = callLength list K0

callLength :: List → Stack → Int
callLength list stack = case list of
                          Nil       → retLength 0 stack
                          Cons _ xs → callLength xs (K1 stack)

retLength :: Int → Stack → Int
retLength arg stack = case stack of
                        K0           → arg
                        K1 nextStack → retLength (1 + arg) nextStack
```

This transformation reimplements the recursive *length* with two new tail-recursive functions, *callLength* and *retLength*. When *length* is called, it simply passes its argument list to *callLength* along with a new data constructor *K0* encoding the "bottom" of a stack data structure. *CallLength* traverses the list in a tail-recursive fashion, “pushing” a *K1* constructor onto the stack for each *Cons* seen.

Once the list has been traversed, *callLength* calls *retLength*, which takes an accumulator argument (starting at 0, the value returned in the base case of the original *length* function), and the stack built up by *callLength*. On each call, *retLength* “pops” the stack by pattern matching on it, adding one to its accumulator for each *K1* popped. Once *retLength* pops the bottom of stack (*K0*), computation has completed and the final accumulator *arg* is returned.

The next two compiler passes add constructs that simplify our eventual translation from a functional IR to abstract dataflow networks: explicit memory operations and pointers, and a special type to handle constants in our dataflow network model. To simplify the current example’s presentation, I focus on how these transformations affect the *callLength* function and its types.

In hardware, we implement recursive data types (here, *List* and *Stack*) with type-specific pointers and a heap, so we first introduce explicit pointer types to replace recursive type fields and read/write functions that convert between pointers and the types they capture. For example, the new type definition of a *List* is

```
data List = Nil | Cons Int ListPointer
```

and *List* objects are stored and recovered from a heap via two functions with type signatures

```
listWrite  :: List → ListPointer
listRead   :: ListPointer → List
```

We then insert calls to these type-specific memory access functions; a read occurs whenever a *case* expression pattern matches on a recursive data type, while a write occurs whenever a new variant of such a type is constructed. This changes *callLength* to

```
callLength :: ListPointer → StackPointer → Int
callLength lp sp = case listRead lp of
                         Nil       → retLength 0 sp
                         Cons _ xs → callLength xs (stackWrite (K1 sp))
```

Constants (here, numeric literals and the *Nil* data constructor) can lead to scheduling difficulties in hardware dataflow networks, so we modify them to act like single-argument functions. Unlike true functions, the argument passed to these “constant functions” should not correspond to any actual data; receiving the argument should simply generate the constant value.

To that end, we introduce a special, single-valued type called "*Go*" whose sole purpose is to trigger constant generation. A single *Go* object is passed around the whole program as an additional argument (in hardware, it's supplied by the environment), each constant expression of type *T* in the program becomes a function call of type *Go* → *T*, constant constructors are given a *Go* type field, and constant patterns are modified to ignore this new field with a wildcard:

```
data Stack  = K0 Go |  K1 StackPointer
data List   = Nil  Go |  Cons Int  ListPointer

callLength  ::  ListPointer  →  StackPointer  →  Go →  Int
callLength  lp  sp  g  =  case  listRead  lp  of
                            Nil   _      →  retLength   (0  g)   sp                       g
                            Cons _ xs  →  callLength  xs      (stackWrite  (K1 sp))  g
```

This example only requires one more transformation to convert it into the more restricted Core dialect, Floh: lifting subexpressions. In Floh, all function arguments, data constructor arguments, and case scrutinees must be simple variable expressions. We thus lift any non-variable subexpressions into local *let* bindings, yielding the Floh version of *callLength*:

```
callLength  ::  ListPointer  →  StackPointer  →  Go →  Int
callLength  lp  sp  g  =  let  t0  =  listRead  lp  in
                          case  t0  of
                            Nil   _      →  let  t1  = 0  g  in
                                            retLength  t1  sp  g
                            Cons _ xs  →  let  t2  = K1 sp  in
                                            let  t3  =  stackWrite  t2  in
                                            callLength  xs  t3  g
```

From a Floh program, the compiler next performs a mostly syntax-directed translation to produce a dataflow network. A *dataflow network* is composed of computational *actors* that execute in parallel and communicate via sequences of *tokens* passed over unbounded FIFO *channels*. We use dataflow networks to bridge the gap between Floh and hardware because they are inherently distributed, parallel, and latency-insensitive: they schedule themselves dynamically with a light-weight protocol that handles the long latencies we expect from modern memory systems.

Figure 3.3 depicts the dataflow network generated from *callLength*. When a list pointer token arrives on input channel *lp*, a *merge* actor passes it off to the *listRead* actor and reports its input selection to two multiplexers, which steer the other two inputs (*sp* and *g*) into the network. The

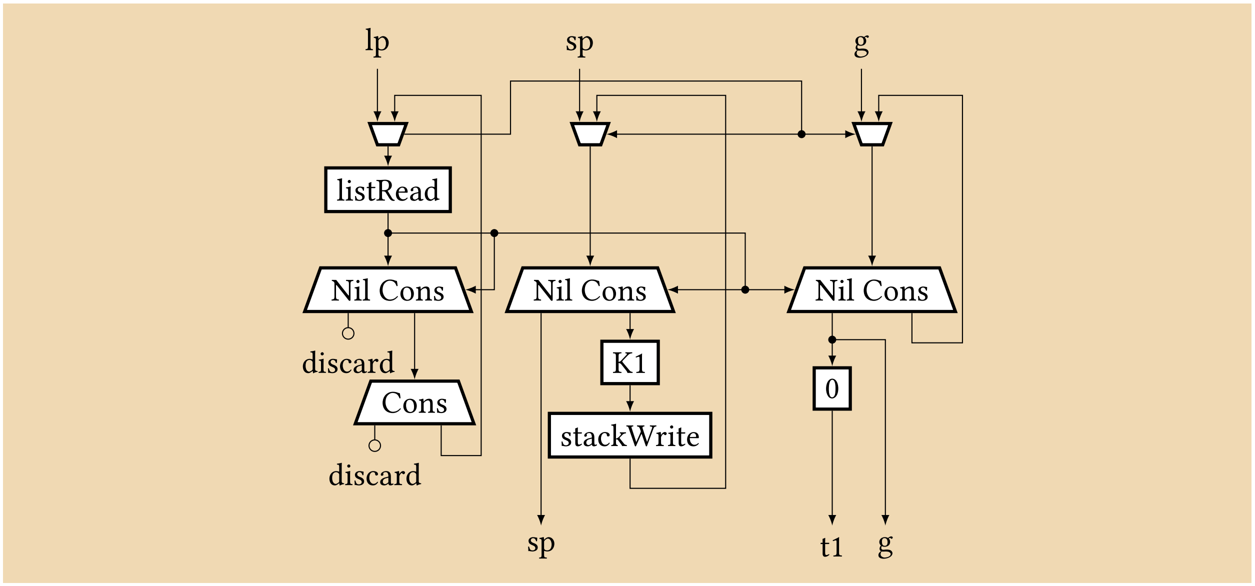


Figure 3.3: A dataflow graph for the *callLength* function. This walks an input list, pushing *K1* onto the stack for each element traversed. Upon reaching the end of the list, the stack pointer, an accumulator, and a *Go* value are passed to the subnetwork implementing *retLength*.

list cell output by *listRead* is forked to the select input of three demultiplexers (and the data input of the leftmost one). If the cell is *Nil* (the base case), the stack pointer and *Go* token are sent to the network implementing *retLength*, along with a *Go*-triggered constant (the three outputs at the bottom of Figure 3.3).

If *listRead* instead produces a *Cons*, the network uses a *destruct* actor to extracts the *Cons*'s pointer field (discarding its data field), pushes a *K1* onto the stack to obtain a new stack pointer, and passes these two new tokens along with the *Go* token back into the network along feedback loops. Once the list pointer field arrives at the merge actor, the above process repeats.

This dataflow network is internally represented with another IR, DF, which serves as the input to the compiler's final translation into SystemVerilog (it can also be dumped by the compiler for debugging or design-space exploration). Each dataflow actor becomes a small block of logic, augmented with a handshaking protocol to retain the network's patience. Channels are either the two-place buffers of Cao et al. [18] (which are implemented with the handshaking protocol in mind) or direct wires.

The handshaking protocol uses two extra bits on each channel. A *valid* bit is bundled with data, indicating if a token is present on the *data* wires. A downstream block sends a *ready* bit upstream to indicate it is able to consume a token being proffered by the upstream block. A token

```
/* demux */
logic [1:0] onehot;
always_comb
  if (( select [0] && in [0]))
    unique case ( select [1:1])
      1'd0    : onehot = 2'd1;
      1'd1    : onehot = 2'd2;
      default: onehot = 2'bx;
    endcase
  else onehot = 2'd0;
assign nilOut   = {in [65:1], onehot [0]};
assign consOut  = {in [65:1], onehot [1]};
assign select_r = | (onehot & {consOut_r, nilOut_r });
assign in_r = select_r ;
assign nilOut_r = 1;

/* destruct */
assign x  = {consOut [33:2], consOut [0]};
assign xs = {consOut [65:34], consOut [0]};
assign consOut_r = &({xs [0], x [0]} & {xs_r , x_r });
```

Figure 3.4: SystemVerilog for a two output demultiplexer, with one output feeding into a destruct actor that dismantles a *Cons* cell into its respective fields.

is transferred from the upstream block to the downstream block (or "consumed") when both *valid* and *ready* are asserted.

Memory access actors are an exception in our translation; they become channels that route memory requests and results between the network and either a robust, cycle-accurate memory simulator (Section 6.3) or a collection of tiny, simply managed on-chip memories (Section 5.6.5). The user can decide which to use via a compiler flag.

As with any compiler, the final code generated is much larger than the original program. I thus only show the generated code in Figure 3.4 for the leftmost demultiplexer and the destruct actor that dismantles a *Cons* cell into its constituent fields. To represent, say, an 8-bit channel *c*, I use a nine-bit vector *c* for data (c[8:1]) and *valid* (c[0]), and a wire named *c_r* for *ready*. The *List* type is realized as a 66-bit vector: 1 bit for *valid*, 1 tag bit to indicate if the vector encodes a *Nil* or a *Cons*, 32 bits for a *Cons*'s integer data, and 32 bits for a *Cons*'s pointer.

The two-output demultiplexer copies its input data (*in*) to all outputs (*nilOut* and *consOut*). If both the *in* port and *select* port have valid tokens, a one-hot decoder uses the value of the *select* token to indicate that exactly one of the output ports has a valid token. Both inputs are consumed if the selected output is ready. Note that the *nilOut_r* signal is always high; this means that any *Nil* tokens are always consumed but not passed to any actor downstream.

When a *Cons* token arrives at the demultiplexer, it is passed to the downstream destruct actor on the *consOut* wire. The contents of the token are split into two signals, *x* and *xs*, which are both valid if the input is valid. The input token is consumed when both outputs are valid and ready.

Given the SystemVerilog specification of a circuit, we can simulate it for performance measurements or, if it doesn't use memory or uses the tiny on-chip memories mentioned earlier, synthesize it using Intel's Quartus software for area or timing estimates. Most of the experimental evaluation in this dissertation is done via cycle-accurate simulation, which is sufficient to support my thesis's claim that hardware synthesized from irregular functional programs can be optimized for parallelism.

3.3 Lowering Core

The Core IR provides various abstractions that simplify the specification of irregular algorithms, but make direct translation to hardware difficult. Our compiler thus performs a number of independent, abstraction-lowering passes that together transform Core into its more restricted dialect, Floh, which admits a (mostly) syntax-directed translation from a functional language to dataflow networks. This section describes these "lowering" passes in detail, following the order of their application in the compiler.

3.3.1 Removing Polymorphism

We first remove polymorphic constructs from a Core program. This pass specializes and renames each polymorphic function, type, and data constructor based on their concrete type arguments, which are explicit in the code (but omitted from the syntax presented in Figure 3.1). Our implementation follows the description of the MLton compiler's monomorphise pass [49].

The pass first walks over the program to construct a symbol table mapping each polymorphic construct's name to a "type map." Every time we see a given polymorphic construct applied to a new set of concrete types during this walk, we create an entry in its type map that associates those types with a fresh name. The name will refer to a monomorphic version of the construct, specialized to the corresponding concrete type arguments.

For example, say we have a program that calls a polymorphic *length* function on a list of booleans *bools* and a list of integers *ints* (explicit types are included to aid this discussion):

```
data List a = Nil | Cons a (List a)

length :: List a → Int
length = ···

main = let bools :: List Bool = Cons True (Cons False Nil)
           ints  :: List Int  = Cons 1 (Cons 2 Nil)
       in length bools + length ints
```

The symbol table would have entries for the *List* type, *length* function, and both of *List*'s data constructors. The *List* entry's type map would associate the concrete types *Bool* and *Int* with new names *List_Bool* and *List_Int*; the other constructs would have similar type maps.

After populating the symbol table, we walk over the program again, replacing names of polymorphic constructs with the monomorphic ones found in its type map. This can reveal new sets of concrete types applied to a polymorphic construct, which adds new entries to the symbol table; the process then repeats. The process is complete once no new entries are found.

Finally, a monomorphised version of each polymorphic definition is created for each entry in its type map, with all names changed to capture the monomorphised versions; this translates our polymorphic *length* example into

```
data List_Int  = Nil_Int  | Cons_Int Int  List_Int
data List_Bool = Nil_Bool | Cons_Bool Bool List_Bool

length_Int :: List_Int → Int
length_Int = ···

length_Bool :: List_Bool → Bool
length_Bool = ···

main = let bools :: List_Bool = Cons_Bool True (Cons_Bool False Nil_Bool)
           ints  :: List_Int  = Cons_Int 1 (Cons_Int 2 Nil_Int)
       in length_Bool bools + length_Int ints
```

3.3.2 Making Names Unique

All variable identifiers in the program are made globally unique (type names and data constructor names are already globally unique); this is a classical compiler technique that simplifies further program analyses. This pass only changes the program if top-level identifiers are shadowed with *let* expressions or if two identical names are in different scopes, e.g.,

```
x = let y = 2 --shadows the global y
        z = 3 in 1 + y + z
y = let z = 4 --z already used in x's definition
    in z
```

becomes

```
x  = let y1 = 2
         z1 = 3 in 1 + y1 + z1
y2 = let z2 = 4
     in z2
```

The compiler runs this pass whenever new names are introduced due to other program transformations, so all subsequent passes may assume that names are globally unique.

3.3.3 Lambda Lifting

Our lambda lifting [74] pass, implemented by Lizzie Paquette, eliminates anonymous functions from a Core program by lifting any unnamed lambda expressions into the top-level, assigning them fresh names, and using the names in place of the original expression. It also lifts locally defined functions into the global scope. Any free variables used by the local/anonymous functions (i.e., variables that are not named as that function's arguments or defined in its body) are added as additional parameters in the process.

```
sum = λn → case n of
             1 → 1
             _ → let f = λx → n + x in
                 f (sum ((λy → y − 1) n))
```

The contrived example above contains both a local function f and an anonymous function. The local function is given an additional argument to capture its free variable n, the anonymous function is bound to a fresh name g, and both functions are lifted into global definitions:

```
f = λx n → n + x

g = λy → y − 1

sum = λn → case n of
                1 → 1
                _ → f (sum (g n)) n
```

Since lambda expressions only occur at the top-level now, we eliminate them from the syntax and instead write a function definition's arguments next to its name:

```
f x n = n + x

g y = y − 1

sum n = case n of
            1 → 1
            _ → f (sum (g n)) n
```

3.3.4 Removing Recursion

This is the most complex lowering pass, and the algorithm it implements was a main contribution in the first paper describing our compiler [141]. The transformations involved have strong implications on the final hardware produced, so I present the full motivation and details of the recursion removal algorithm here. Kuangya Zhai implemented and described the original algorithm; I modified the implementation to handle corner cases that have cropped up since then, and largely reuse Zhai's presentation from [141] here.

First, I use an example to explain why general recursive (i.e., not tail-recursive) functions can pose problems for hardware translation, presenting the concepts leveraged by the algorithm. I then present the actual algorithm, using small snippets of code to help explain how it works.

Illustrative Example: Fibonacci

The example below implements the familiar recursive Fibonacci number function: it compares the integer argument n with constants 1 and 2 to determine whether it has reached a base case, for which it returns 1, or needs to recurse on $n-1$ and $n-2$. After applying our pass, this function is realized with a trio of tail-recursive functions that use an explicit stack, together implementing the general recursion here.

```
fib n = case n of 1 → 1
                  2 → 1
                  _ → fib (n−1) + fib (n−2)
```

Translating this recursive function into hardware is difficult because of the two recursive function calls. The usual technique of inlining calls (e.g., typical for HLS tools) would attempt to generate an infinitely large circuit unless we limited the recursion depth. Interpreting the structure of this program literally would produce a circuit with multiple combinational loops (multiple ones from each recursive call), but it would likely oscillate unpredictably. Inserting registers in the feedback loops would prevent the oscillation, but since this is not simply tail-recursion, it is not obvious how to arbitrate between the two call sites or how to "remember" the remaining computation that should occur after a recursive call returns. Instead, our compiler restructures this program into a semantically equivalent form that is straightforward to translate into hardware using the technique I present in Section 4.3.

Since Core is a pure language, the evaluation of *fib (n−1)* and *fib (n−2)* can occur in any order without changing the function's result. To avoid arbitration circuitry to decide which call to evaluate first, we impose a particular order on them by transforming the function into continuation-passing style [52, 120], or CPS. In CPS, each function is given an extra "continuation" argument (traditionally named "k") that captures what to do with the result of the function. A continuation is a single-argument function; when a CPS function computes its result, it applies its continuation to that result.

Many functional compilers rewrite entire programs into CPS form for control-flow analysis; we only use it on recursive functions to order multiple recursive calls. Specifically, we use a CPS helper function *call* to do *fib*'s actual work, and modify *fib* to invoke *call* with a continuation that returns the result to the outside, non-CPS world:

```
call  n k = case n of 1 → k 1
                      2 → k 1
                      _ →  call  (n−1) (λn1 →
                           call  (n−2) (λn2 →
                          k (n1 + n2)))
fib  n =  call  n (λx → x)
```

The structure of *call* now represents the control flow explicitly: recurse on *(n-1)*, name the result *n1*, recurse on *(n-2)*, name its result *n2*, compute *n1 + n2*, and finally pass the result to the continuation *k*. As a specific example, consider evaluating *fib 3*:

```
fib  3 =  call  3  (λx → x)
       =  call  2  (λn1 →  call  1  (λn2 → (λx → x)  (n1 + n2)))
       =           (λn1 →  call  1  (λn2 → (λx → x)  (n1 + n2)))  1
       =β                  call  1  (λn2 → (λx → x)  (1  + n2))
       =                            (λn2 → (λx → x)  (1  + n2))  1
       =β                                  (λx → x)  (1  + 1)
       =β                                  2
```

The first call to *fib* simply passes the argument 3 to *call* with the identity continuation. Since 3 doesn't match 1 or 2, this first call evaluates to the expression *call 2 (λ n1 → . . .)*. Evaluating this call applies the whole continuation *(λ n1 → . . .)* to 1; every instance of *n*1 in the continuation's body is replaced with the argument 1 (this process is called "β-reduction", referenced with the β subscripts). The rest of the steps follow similarly (either evaluating a tail-recursive call or performing β-reduction).

The CPS transformation has scheduled the two original *fib* calls (now captured in the *call* function) and transformed the program to use only tail-recursion, which we implement in hardware as buffered feedback loops in a dataflow network. Two issues remain before we can directly translate this function into hardware, though: the second continuation *(λ n2 → . . .)* references *n1*, which is defined by the first continuation, not the second; and more seriously, lambda expressions (our continuations) are being passed as arguments, which our IR's semantics prohibits.

We perform lambda lifting again (Section 3.3.3) to address these issues. First, any variables that are not defined within their continuation (here, *n* and *k* in the first continuation, *n*1 and *k* in the second) are added and passed as additional arguments to that continuation. For example, the expression *(λn2 → k (n1 + n2))* becomes *((λn1 k n2 → k (n1 + n2)) n1 k)*.

```
call  n k = case n of 1 → k 1
                      2 → k 1
                      _ → call  (n−1) ((λ n  k n1 →
                          call  (n−2) ((λ n1 k n2 →
                           k (n1 + n2))  n1 k))
                                         n  k)
fib  n =  call  n (λx → x)
```

Second, each lambda expression is extracted and named as a top-level fuction:

```
call  n k = case n of 1 → k 1
                      2 → k 1
                      _ → call  (n−1) (k1 n k)
k1 n   k n1 =  call  (n–2) (k2 n1 k)
k2 n1 k n2 = k  (n1 + n2)
k0        x   = x
fib  n =  call  n k0
```

Here, *k0* is the identity function, *k1* evaluates the second recursive call, and *k2* produces a result (to pass to the next continuation) by adding *n1* to *n2*. Each of these continuations is passed as a partially applied function, e.g., *k1* takes three arguments, but is only given *n* and *k* when passed to *call*. The third argument is passed to *k1* when it is called, either in one of *call*'s base cases or in the body of *k2*. With the lambda lifting step complete, *fib 3* would be evaluated as:

```
fib  3 =  call  3  k0
       =  call  2  (k1  3  k0)
       = (k1  3  k0)  1
       =  call  1  (k2  1  k0)
       = (k2  1  k0)  1
       = k0  (1  +  1)
       = 2
```

Partially-applied functions do not have a clear hardware representation, so we eliminate them via defunctionalization [37]. An algebraic data type *Cont* encodes the continuations (one variant for each), and a helper function *ret* "applies" a continuation *k* to a result *r* (using a *case* expression to identify the continuation):

```
data Cont = K0 | K1 Int Cont | K2 Int Cont

 call n k = case n of 1 → ret k 1
                      2 → ret k 1
                      _ → call (n−1) (K1 n k)
ret k r = case k of K1 n  k′ → call (n−2) (K2 r k′)
                    K2 n1 k′ → ret k′ (n1 + r)
                    K0       → r
fib n = call n K0
```

This is now much closer to a hardware implementation. No partially applied or higher-order functions remain, and the *k* argument functions like a top-of-stack: creating a continuation effectively pushes onto the stack; scrutinizing the continuation in *ret* pops the stack (with *k′* serving as the new top-of-stack).

Each recursive function transformed in this way gets its own dedicated *Cont* type, and our eventual translation to hardware realizes such recursive types with type-specific pointers and a heap. This means that two recursive functions could use independent memories for their respective *Cont* types to improve memory-level parallelism. We leverage this idea in our specialized memory system to exploit potential parallelism (discussed in Section 6.2).

The Actual Algorithm

The previous example introduced the concepts underlying our recursion removal procedure; I now present the actual procedure, which applies to more general cases and eschews the introduction of constructs that would then have to be removed (e.g., higher-order functions). It starts from any collection of functions, which may contain recursion of any form, and produces an equivalent collection of functions that are at most tail-recursive. This procedure assumes that a lambda-lifting pass has occurred (e.g., that all functions called are named directly) and that no higher-order functions are present (including partially applied functions).

Our procedure operates by identifying groups of mutually recursive functions, merging each group into a single recursive function, explicitly scheduling the recursive calls, splitting apart each function at recursive call sites, inserting continuation control with tail-recursive helper functions, and encoding continuations with a stack-like data type. I detail these steps below.

Combining Mutually Recursive Functions We begin by combining mutually recursive functions into a single function. We build a static call graph of all the functions in the program; each strongly connected component (SCC) is a group of mutually recursive functions to be merged.

Each function in an SCC can have different argument and return types, so to merge the functions, we need to merge their types. Consider two mutually recursive functions f and g that return variables *ff* and *gg* of respective types T and U:

```
f :: X → T
f x = ... g a ... ff

g :: Y → U
g y = ... f b ... gg
```

To merge f and g's return types, we define an algebraic data type that can hold either T or U,

```
data Ret = Ret_f T | Ret_g U
```

introduce variants of f and g that return this new type by wrapping their results in the appropriate data constructor,

```
f′ :: X → Ret
f′ x = ... g a ... (Ret_f ff)

g′ :: Y → Ret
g′ y = ... f b ... (Ret_g gg)
```

and re-implement f and g to call their variants and extract the wrapped results

```
f :: X → T
f x = case f′ x of Ret_f r → r
g :: Y → U
g y = case g′ y of Ret_g r → r
```

Inlining f and g in the definitions of their variants leaves only f' and g', which are still mutually recursive but each return the same type.

```
f′ :: X → Ret
f′ x = ... (case g′ a of Ret_g r → r) ... (Ret_f ff)
g′ :: Y → Ret
g′ y = ... (case f′ b of Ret_f r → r) ... (Ret_g gg)
```

We next unify the argument types of f' and g'. Again, we introduce an algebraic type that can represent either:

```
data Arg = Arg_f X | Arg_g Y
```

and merge the bodies of f' and g' into a new function *fg*, which uses a *case* expression to determine which body to evaluate (based on the input of type *Arg*). We also replace calls to f' and g' by wrapping the calls' arguments in the appropriate *Arg* variant and passing the argument to an *fg* call instead of f' or g'. We make similar modifications to f and g.

```
fg :: Arg → Ret
fg a = case a of
  Arg_f x → ... (case fg (Arg_g a) of Ret_g r → r) ... (Ret_f ff)
  Arg_g y → ... (case fg (Arg_f b) of Ret_f r → r) ... (Ret_g gg)
f :: X → T
f x = case fg (Arg_f x) of Ret_f r → r
g :: Y → U
g y = case fg (Arg_g y) of Ret_g r → r
```

After these transformations, each group of n mutually recursive functions is fused into a single recursive function with n non-recursive "wrapper" functions that interface with the new, fused function. This procedure resembles Danvy's defunctionalization [37], which introduces an *apply* function that takes a function identifier as the first argument; our *fg* function is the *apply*, and the *Arg* variant is the function identifier.

Sequencing Recursive Call Sites To prepare for the final transformation into CPS, we rewrite all recursive functions so that each recursive call appears only in a *let* with a single binding. This effectively imposes a linear order on all the recursive calls, and the body of each *let* binding is exactly the code to be executed after the call returns, i.e., its continuation. We will later slice the function at these points.

Our algorithm lifts out the scrutinee of any *case* expression and the arguments of any function call and binds each such subexpression to a new temporary. Next, our algorithm flattens groups of nested *let* expressions to yield a simple sequence of *let*s. To illustrate, consider the following recursive function *f*, which contains several directly recursive calls along with calls to *h* and *g*.

```
f arg = ···
  case h arg of
    _ → let a = f (g (f b)) in c ···
```

We first bind the result of *f*'s inner call to a new temporary *t1*:

```
f arg = ···
  case h arg of
    _ → let a = f (let t1 = f b in g t1) in c ···
```

Next, we lift the scrutinee of the *case* expression and the argument to the outer *f* call and bind them to new temporaries (*t3* and *t2*, respectively).

```
f arg = ···
  let t3 = h arg in
  case t3 of
    _ → let a = (let t2 = (let t1 = f b in g t1)
                 in f t2)
        in c ···
```

Finally, we use the equivalence

$$\textbf{let}\ v1 = (\textbf{let}\ v2 = e2\ \textbf{in}\ e1)\ \textbf{in}\ e$$
$$==\ \ \textbf{let}\ v2 = e2\ \textbf{in}\ \textbf{let}\ v1 = e1\ \textbf{in}\ e$$

to flatten all nested *let* expressions, giving

```
f arg = ···
  let t3 = h arg in
  case t3 of
    _ → let t1 = f b in
        let t2 = g t1 in
        let a = f t2 in c ···
```

Dividing Functions into Continuations After the last step, recursive calls are in a linear sequence and located in the local bindings of simple *let* expressions. Our ultimate goal is to replace these recursive calls with tail-recursive calls that manipulate continuations on a stack.

Rather than use lambda expressions, which we would ultimately have to eliminate in a hardware implementation, we directly introduce both an algebraic data type "*Cont*" that represents partially applied continuations (each is missing the value returned by the recursive call), and a continuation-handling function "*ret*" that takes a continuation and a result from a recursive call and evaluates each continuation in a *case* expression. We create dedicated *Cont* and *ret* definitions for each recursive function in the program.

There is always a single continuation implying a return of the result to the environment. We call this *K0*, giving us

```
data Cont = K0

ret k r = case k of K0 → r
```

The original recursive function (here called "*g*") is renamed *call* and given an additional argument *k* to represent the continuation that will receive the result. We then reuse *g* to name a wrapper function that calls the new entry point with *K0*:

```
call x1 ··· xn k = ···

g x1 ··· xn = call x1 ··· xn K0
```

The bodies of the *let* expressions with recursive calls are exactly the continuations for those calls, so we divide up the function by applying the following steps to each of these *let* expressions:

1. Replace the whole *let* expression with the recursive call it named.
2. Add a variant to the *Cont* type that captures all the free variables in the body of the *let* (effectively performing lambda lifting).
3. Pass this new variant as an argument to the recursive call, including any free variables as its fields.
4. Add the body of the *let* to a new branch of the *case* in the *ret* function. This branch matches on the newly introduced *Cont* variant.

For example, if from the last step we have

```
call  ···  = let  v1 = ···  in
             ···
             let  vn = ···  in
             let  z  =  call  a1  ···  am in ···  v1  ···  vn ···
```

we turn it into the fragment

```
call  ···  = let  v1 = ···  in
             ···
             let  vn = ···  in
             call  a1  ···  am (K1 v1  ···  vn)
```

and extend the *Cont* type and *ret* function (assuming variable vi has type Ti, for $1 \leq i \leq n$):

```
type Cont = K0
            K1 T1 ···  Tn
            ···

ret  k  r  = case k of
   K0 → r
   K1 v1  ···  vn → let  z  =  r  in  ···  v1  ···  vn ···
```

Once this is completed, each recursive function g is converted into a simple wrapper function that interfaces between external callers of g and a pair of mutually tail-recursive functions $call_g$ and ret_g that manipulate a function-specific stack type $Cont_g$. This has eliminated (non-tail) recursion from all functions in the program; we next deal with recursive data types.

3.3.5 Tagging Memory Operations

A software program executing on a general-purpose processor usually requires up to four segments of memory: *code* to store the program's instructions, *data* for global or static variables, *stack* to store local variables and implement function calls, and *heap* for dynamically-allocated data (e.g., "malloced" data in C, objects in Java, thunks in Haskell). Since our compiler targets specialized hardware, we may organize and interact with memory in whatever way we choose.

Our hardware dataflow networks use a simple memory model: variants of recursive data types and their fields are the only things written to or read from memory (all other live values flow through the dataflow network directly or reside in registers). Values of non-recursive type are thus stored in memory only if they exist as a field in a recursive type's variant, e.g., the *Int* field of a *Cons* cell.

We encode non-recursive types as statically-sized (i.e., bounded) bit vectors in hardware; unfortunately, a recursive type cannot be bounded at compile time, in general. We thus model recursive data types with type-specific pointers and a heap. Here, we use a compiler pass to introduce abstract pointer types and memory access functions that indicate where memory operations will need to occur.

This pass begins by introducing three new polymorphic definitions: a *Pointer* data type that holds a value of recursive type, a *write* function that wraps values in the new *Pointer* type, and a *read* function that unwraps them:

```
data Pointer a = Pointer a

write :: a → Pointer a
write value = Pointer value

read :: Pointer a → a
read pointer = case pointer of
                 Pointer value → value
```

The pass then performs three simple transformations:

1. If type *T* is recursive, replace any type field *T* found in a variant definition with *Pointer T*.
2. Replace every data constructor expression *e* of recursive type with *write e*.
3. If expression *e* is a *case* scrutinee (i.e., **case** *e* **of** . . .) of recursive type, replace *e* with *read e*.

After this pass, we run the monomorphise pass again to specialize the *Pointer* type and *read* and *write* functions. Once complete, all recursion in type definitions is captured with type-specialized *Pointer* types, and type-specific *read* and *write* functions denote the only code locations where values of recursive type can be inspected and generated, respectively. These invariants both simplify our translation to dataflow networks and provide optimization opportunities for memory operations at the functional language level, i.e., since all memory operations are strongly typed and pointers cannot be generated by the user, memory-focused compiler analyses are made much simpler than, say, those required in C-like languages where one pointer may point to different types of data during execution, including garbage values.

The above implementations of the abstract *Pointer*, *read*, and *write* constructs are ignored by the translation to dataflow (they are used to maintain semantic correctness throughout the lowering compiler passes). Instead, the translation modifies pointers to carry actual addresses, and type-specific *read* and *write* functions are treated as language primitives, i.e., they cannot be implemented directly and are treated specially by the compiler.

For example, the translation would realize a binary tree of integers with types

```
data Btree  = Branch Bptr  Int  Bptr  |  Leaf
data Bptr   = Bptr  Int
```

where *Btree* is an algebraic type with two variants: *Branch*, with pointers to two *Btrees* and an integer; and *Leaf* representing an empty tree. A *Bptr* carries an address that points to a *Btree* object on the heap.

Such *Btree* objects are stored and recovered from a heap via two functions with type signatures

```
treeWrite  ::  Btree  →  Bptr
treeRead   ::  Bptr   →  Btree
```

TreeWrite takes a *Btree* object, writes it to the heap, and returns a *Bptr* that, when given to *treeRead*, returns the written object.

Providing memory operations in a parallel programming language usually introduces data races and nondeterminism; we avoid these problems with a simple but profound limitation: only memory write functions can create pointers (e.g., only *treeWrite* may construct *Bptr* objects). This restriction, paired with a heap following the standard heap discipline (i.e., live data is never overwritten), ensures that our IR remains deterministic with explicit memory operations. Thus, given any object x with type-specific memory operations *read* and *write*, $read(write(x)) = x$ always holds. We give more details on this treatment in Section 4.2 and Section 4.3.

3.3.6 Simplifying *Case*

The Floh dialect of Core maintains a number of syntactic invariants that simplify its translation to dataflow networks. The next three compiler passes modify Core programs so they adhere to these invariants, thus finishing the translation from Core to Floh.

This pass imposes a restriction on *case* expressions: a *case* can only pattern match on data constructor expressions (or variables bound to those expressions), and if it is matching on a value of type *T*, every constructor of type *T* must be matched explicitly. We thus remove literal pattern matching and default patterns with this pass.

We first remove any *case*s pattern matching on literals (here called a "literal *case*"), replacing them with equality checks and matchings on booleans. A literal *case* always matches on a finite number of integer literals explicitly, then has a final default pattern to handle all other integers, i.e., we perform the following transformation:

$$\begin{aligned}
&\textbf{case } e \textbf{ of } (lit_1 \rightarrow e_1)\ldots(lit_k \rightarrow e_k)\ (\_ \rightarrow e_{default}) \\
= &\textbf{ case } e == lit_1 \textbf{ of } (True \rightarrow e_1)\ (False \rightarrow \ldots \\
&\quad \textbf{case } e == lit_k \textbf{ of } (True \rightarrow e_k)\ (False \rightarrow e_{default})\ \ldots)
\end{aligned}$$

A *case* matching on a single default pattern is unnecessary, as it always evaluates to the default alternative's expression. We replace the whole *case* with that expression:

$$\textbf{case } e \textbf{ of } (\_ \rightarrow e_{default}) = \ e_{default}$$

The only other situation breaking our desired invariant is a *case* that matches on some data constructors but uses a default pattern to handle all others. We add an extra alternative for each data constructor that wasn't matched, and have them all evaluate to the expression used by the original default pattern, e.g., for a *case* matching on a scrutinee whose type has $n > k$ variants (here we use C_i to represent the ith data constructor pattern):

$$\begin{aligned}
&\textbf{case } e \textbf{ of } (C_1 \rightarrow e_1)\ldots(C_k \rightarrow e_k)\ (\_ \rightarrow e_{default}) \\
&= (C_1 \rightarrow e_1)\ldots(C_k \rightarrow e_k)\ (C_{k+1} \rightarrow e_{default})\ldots(C_n \rightarrow e_{default})
\end{aligned}$$

3.3.7 Adding *Go*

In our dataflow network semantics, computation can only occur when an actor in the network is provided with one or more inputs, after which the actor "fires," consuming its inputs and producing some number of outputs. As will be shown in Section 4.3, each expression in a program is translated into a collection of dataflow actors, so each expression should have some notion of "input" to respect the actor firing semantics.

Constant expressions, though, do not have any inputs at this stage in the compiler. To avoid the need for "source" actors that generate constant data without being prompted (which lead to scheduling headaches), this compiler pass introduces a special type called "*Go*," defined as **data** *Go* = *Go*, and re-implements each numeric literal in a program with a function that takes a single *Go* argument and produces the appropriate constant. An object of the *Go* type functions as a trigger: it does not carry a value, similar to the *void* type in C-like languages and the *unit* type in many functional languages. As a result, *Go*-triggered literal functions ignore their sole argument (and are the only kind of function in Floh that may do so).

The goal of this pass is to transform all constant expressions into single-argument functions that simply take a *Go*-valued argument and produce the original constant value. As a first step, we inline all top-level constant definitions to prevent parallelism-hindering sharing: our translation into dataflow creates a single sub-circuit for each function, which is then shared among its callers, so inlining will yield more opportunities for these simple constant-generating functions to operate in parallel.

Consider the below example, which is contrived to capture all the effects of this pass. The initial inlining step transforms it from

```
--top-level constant definitions
nil  = Nil
zero = 0

v :: List → Int
v l = case l of Nil      → zero
                Cons x _ → f zero x

f :: Int → Int → Int
f n x = zero

main = v nil
```

into

```
v :: List → Int
v l = case l of Nil       → 0     --zero inlined
                Cons x _  → f 0 x --zero inlined

f :: Int → Int → Int
f n x = 0 --zero inlined

main = v Nil   --nil inlined
```

Next, each constant data constructor expression is given a *Go* field; each constant literal becomes a function call that, when given a *Go*-valued argument, produces the original constant value; and the special *main* definition (that names the program's result) is redefined to bind a *Go* value to a variable. A program cannot produce a *Go* object except in the *main* definition, and this single *Go* object is passed around the program as an additional function argument. Specifically, any function that produces a constant value, either in its own definition or via a call to another function, will take an additional *Go* argument.

Continuing with the previous example, *Nil* is redefined to have a *Go* field, each use of 0 becomes a call to a unique function (to prevent sharing), *v* and *f* receive a *Go*-valued argument to pass to their (no longer constant) subexpressions, and *main* produces a single *Go* value to thread through the program:

```
data List = Nil Go | Cons Int List

zero1 go = 0
zero2 go = 0
zero3 go = 0

v :: Go → List → Int
v go l = case l of Nil _     → zero1 go
                   Cons x _  → f go (zero2 go) x

f :: Go → Int → Int → Int
f go n x = zero3 go

main = let go = Go in v go (Nil go)
```

All constant literals have now been abstracted into single-argument functions, and no constant data constructors exist anywhere in the program. Furthermore, *main* is the only remaining top-level variable definition; if other top-level variable definitions exist, we inline them at their use sites, leaving only function and data type definitions at the top-level.

While our *Go* machinery clutters our programs, it simplifies our eventual translation to dataflow. The *Go* type is an algebraic type like any other and the *Go*-valued variables behave like all other variables; our translation does not need any special rules for triggering literals.

3.3.8 Lifting Expressions

Our dataflow translation requires one more invariant that we now introduce in our programs: all arguments to function calls and data constructors and all case scrutinees must be simple variable expressions. Any non-variable subexpressions in these code locations are lifted and named with local *let* expressions. This pass simply traverses the program and, at each of the listed expression forms, lifts any non-variable subexpressions into freshly named *let* expressions.

As an example, this pass would transform

```
buildList :: Go → Int → Int → List
buildList go x y l = case x > y of
                       True _ → Nil go
                       False _ → Cons x ( buildList go (x+1) y)
```

into

```
buildList :: Go → Int → Int → List
buildList go x y l = let t1 = x > y in
                     case t1 of
                       True _ → Nil go
                       False _ → let t2 = x + 1 in
                                 let t3 = buildList go t2 y
                                 in Cons x t3
```

After this pass, the program has been transformed into the Floh dialect of Core, which we discuss in detail in Section 4.1. We perform one simple optimization to improve memory-level parallelism before hitting the actual translation to dataflow in our compiler.

3.3.9 Optimizing Reads

Memory accesses that appear in the same scope have a high chance of being parallelized in our dataflow networks. We partition on-chip memory by type (e.g., lists might be in a distinct memory from trees), so simultaneous memory accesses for different types can be serviced in parallel. However, the form of an expression can hinder this parallelism, e.g., if *a*, *b*, *c* are each pointers of different types, then

```
case x of
  X a b c → let a′ = read a in case a′ of
    A′ ··· → let b′ = read b in case b′ of
     B′ ··· → let c′ = read c in case c′ of
       C′ ··· → ···
```

will incur three serialized memory accesses, since the result of each is needed before the subsequent *case* expression can evaluate its scrutinee. However, we have access to all the different pointers after the top-level *case* scrutinizes *x*. We can thus "lift" the second two reads into the same scope as the first:

```
case x of
  X a b c → let a′ = read a
                b′ = read b
                c′ = read c
            in case a′ of
        A′ ··· → case b′ of
         B′ ··· → case c′ of
           C′ ··· → ···
```

enabling potential parallelism among the three reads to disjoint memories.

This pass searches each function for independent, potentially speculative read operations (i.e. accesses *a* and *b* might not both occur on every execution path in the function, but *a* is not a field in the object pointed to by *b* and vice versa), and lifts them all into the same scope to enable more memory-level parallelism. We only perform this optimization on reads, since speculative writes generate additional garbage.

Chapter 4

From Functional Programs to Dataflow Networks

The compiler passes discussed in Section 3.3 transform a Core program into a more restricted dialect called Floh. In this chapter, I present the next compilation step, which forms one of my major contributions: a largely syntax-directed translation of Floh programs into abstract dataflow networks that exhibit pipeline and other forms of parallelism. As another contribution, I informally describe a technique for implementing these abstract networks in hardware with limited, bounded buffering. In the next chapter, I formalize our network model and show that our hardware implementation upholds this formalism. Taken together, these two chapters support the first half of my thesis: functional programs exhibiting irregular memory access patterns can be compiled into specialized hardware.

We specifically selected functional programs and dataflow networks as the endpoints of our compilation process to reveal new opportunities for parallelism in irregular algorithms. We target dataflow networks because they are modular, inherently parallel, naturally "patient" about the long, varying latencies associated with today's memories, and they can yield high-speed hardware implementations [18]. We start from what is effectively a pure functional language to provide inherent parallelism and high-level abstractions to the designer, making it simple to correctly express and reason about irregular algorithms [85]. These abstractions also present optimization opportunities in our compiler that may otherwise be infeasible due to side effects or direct control over pointers; we discuss these optimizations in Chapter 6 and Chapter 7.

To simplify the analysis of programs with irregular memory accesses, our Floh IR uses an immutable memory model. In particular, we maintain referential transparency while admitting the potential for data duplication across multiple memories (to enable parallel computation), all without having to maintain coherence. We assume the presence of automatic garbage collection, which we have not yet implemented, but it can be done: Bacon et al. [9] show that real-time garbage collection is practical in hardware, incurring only modest increases in logic and memory at high clock frequencies.

A novelty of our approach is how designers "ask for" pipeline parallelism through tail-recursion with non-strict functions. A tail-recursive call can begin execution immediately after its first argument arrives. When such a call occurs, multiple invocations of the function run in parallel.

The rest of this chapter is structured as follows, drawing from our earlier publication of this work [125]. Section 4.1 describes our translation's starting point, the Floh IR; Section 4.2 introduces our target, an abstract dataflow model with unbounded buffers. Our translation operates in two steps: Section 4.3 presents the translation from Floh to dataflow networks; Section 4.4 explains how to practically implement such networks in hardware (I cover the full details of our hardware implementation in Chapter 5). Section 4.5 presents some experimental results, which show that our compiler-generated networks exploit pipeline parallelism and cope with varying memory latency.

4.1 A Restricted Dialect of Core: Floh

Our synthesis process begins from the Floh ("Functional Language On Hardware") intermediate language, a restricted dialect of our compiler's initial Core IR (Section 3.1). The compiler uses the same data structure to represent Floh and Core programs internally; Floh programs simply have a more limited syntax to simplify its translation into dataflow. By retaining the core components of Core, we attain a simple but rich IR with inherent parallelism.

Most of Floh's syntactic restrictions were presented in Section 3.3; this list summarizes them:

1. Functions are all defined at the top-level (Section 3.3.7) and may only contain tail-recursion (Section 3.3.4).
2. Recursive data types are implemented with type-specific pointers and a heap (Section 3.3.5).
3. A *case*'s scrutinee is bound to a data constructor expression, the *case* provides exactly one pattern for each variant of that constructor's data type, and every argument of a pattern's data constructor is explicitly named or ignored with a wildcard pattern (Section 3.3.6).
4. Constant literals are implemented with *Go*-triggered functions, and all data constructors have at least one type field (e.g., previously constant constructors now have a single *Go* type field) (Section 3.3.7).
5. Function call arguments, data constructor arguments, and *case* scrutinees are simple variable expressions (Section 3.3.8).

Some additional syntactic restrictions are imposed on Floh programs. Functions must use all of their specified parameters somewhere in their body (primitive literal functions are the exception). *Let*-bound variables are visible only within the *let's* body; no definition in a given *let* can refer to another variable defined by that *let*. This restriction provides a simple source of parallelism: since definitions within a *let* have no inter-dependencies, we can evaluate their expressions in parallel. We insist that each variable bound in a *let* be referenced at least once in the *let's* body. This simplifies the translation process and prevents the definition of unused variables.

Unlike Haskell, Floh uses different strictness policies for data constructors and function calls to balance simplicity with performance. Data constructors and non-recursive functions are *strict*: they evaluate all their arguments before producing a result, which simplifies the memory system's semantics, prevents potential deadlock in the targeted dataflow networks, and eliminates the bookkeeping overhead of Haskell's lazy evaluation scheme. Tail-recursive functions in Floh are only *strict in their first argument*: a tail-recursive call may begin evaluation after the first argument is available, but will not return a final result until all other arguments have arrived, even if some are unused (e.g., if an argument is used in a *case* alternative that was not selected). Starting before all arguments are available facilitates pipeline parallelism; insisting on ultimately having all the arguments simplifies the translation. We elaborate on our non-strict functions in Section 4.3.

4.1.1 An Example: Map

As a running example for this chapter, Figure 4.1 shows how the classical *map* function can be coded in Floh. It takes a list of integers and produces a second list by applying some function *f* to each element of the first list.

In Haskell, we code *map* recursively:

```
map list = case list of
             Nil       → Nil
             Cons x xs → Cons (f x) (map xs)
```

When the list is empty, the result is empty. Otherwise, *map* splits the list into its head (*x*) and tail (*xs*), recurses on the tail, and prepends the result of the call *f x* to the result of the recursive call.

This function operates in two phases. In the first phase, it traverses the source list and pushes each element (*x*) on the stack. In the second phase, it pops each element from the stack, applies *f*, and prepends this new list cell to the result list.

Our compiler performs the passes described in Section 3.3 to translate the recursive Haskell function into the tail-recursive Floh program in Figure 4.1. It transforms recursive functions into

```
data ListPtr  = ListPtr  Int
data List     = Cons Int  ListPtr  |  Nil  Go

data ContPtr      = ContPtr  Int
data Continuation = K1 Int  ContPtr  |  K0 Go

map g lp  =  let  k0 = K0 g          in
             let  sp  = stackWrite  k0 in
             call  g  lp  sp

call  g  lp  sp =  let  le  =  listRead  lp  in
                   case  le  of
                     Cons x xs  →  let  nc   = K1 x  sp        in
                                   let  nsp = stackWrite  nc  in
                                   call  g  xs  nsp
                     Nil  _     →  let  nil  = Nil  g          in
                                   let  lpn  =  listWrite  nil  in
                                   ret  sp  lpn

ret  sp  lp  =  let  se  = stackRead  sp  in
                case  se  of
                  K1 x  nsp  →  let  fx  =  f  x              in
                                let  nle  = Cons fx  lp       in
                                let  nlp  =  listWrite  nle  in
                                ret  nsp  nlp
                  K0 _       →  lp
```

Figure 4.1: The *map* function implemented in Floh. The *call* function walks the input list and pushes each element on a stack of continuations (replacing function activation records) encoded with a list-like data type; the *ret* function pops each element *x* from the stack, applies *f* to it, and prepends the result to the returned list.

continuation-passing style, performs lambda lifting to name each continuation as a global function, creates a recursively defined continuation type (*Continuation*) to encode these new functions (*K1* and *K0*), and finally builds a pair of functions that operate on the new type to handle the calls (*call*) and continuations (*ret*) of the *map* function. Since the *Continuation* type is recursive in Core, Floh implements it with type-specific pointers and a heap (as described in Section 3.3.5). In Floh, a transformed function's continuations behave like stack activation records, so we use stack terminology (i.e., "push" and "pop") to refer to their interactions with heap memory.

In Figure 4.1, the *map* function receives a list pointer (*lp*) and a *Go* object (*g*) as arguments, pushes an initial terminal continuation (*K0*) on the stack to obtain an initial stack pointer (*sp*), and then starts *call*. The *call* function reads a cell of the input list and either pushes its contents in a *K1* continuation onto the stack before tail-recursing or writes an empty list to the heap and invokes *ret*. The *ret* function pops a continuation off the stack and either applies *f*, prepends a new list cell to the result list, writes it to the heap, and tail-recurses, or returns the final list pointer. If *f* is a high-latency, pipelined function and the continuations are stored in fast, on-chip memory, *ret's* non-strictness can exploit pipeline parallelism across tail-recursive calls: each call's first argument (*nsp*) will be available before the second (*nlp*, which depends on *f x*), so we can recurse multiple times and fill *f's* pipeline with data from the popped continuations.

4.2 Dataflow Networks

We translate a Floh program into an idealized dataflow network with unbounded buffers, which we ultimately convert into hardware with finite buffers. This intermediate step enables the exploration of alternative hardware implementations (e.g., trading area for clock speed) without complicating the translation from the higher-level language. Here, I give an informal introduction to our abstract network model; a formal treatment is provided in Chapter 5.

We use a dataflow representation to bridge the gap between a functional software language and hardware because it is inherently distributed, parallel, and "patient": infinitely buffered dataflow networks can handle long, unpredictable latencies from complex, hierarchical memory systems without requiring any kind of costly global synchronization. Modeling hardware with streams [54, 141] does not accommodate delays as readily.

A dataflow network consists of a collection of *actors* connected via unbounded point-to-point FIFO *channels* that convey typed, data-carrying *tokens*. All tokens on a particular channel have

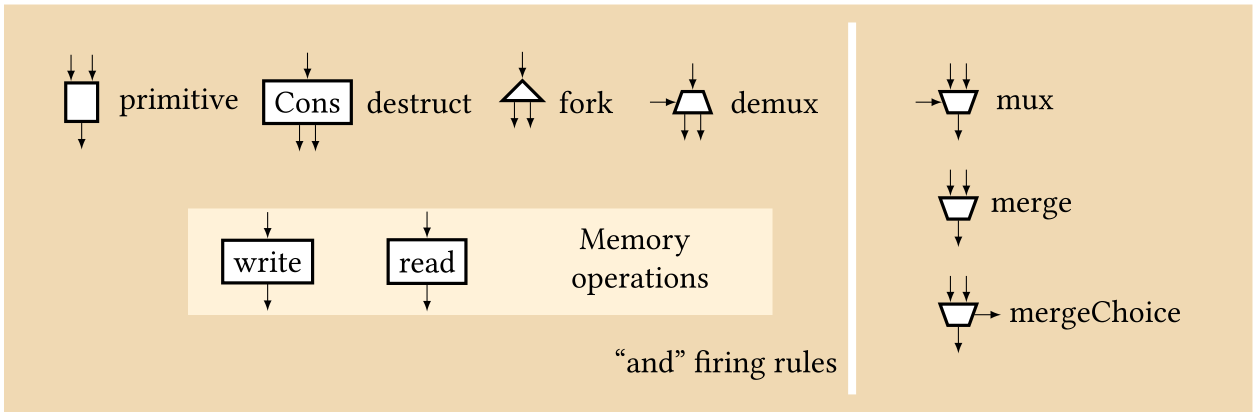


Figure 4.2: Our menagerie of dataflow actors. Those left of the line require data on every input channel to fire.

the same Floh type. When an actor *fires*, it consumes one or more tokens from at least one of its input channels, performs computation on their contents, and produces tokens on zero or more output channels. An *enabled* actor has sufficient input tokens to fire.

The *state* of a dataflow network consists of the tokens on each channel. At any point, this state may evolve by firing any or all enabled actors (i.e., those with sufficient tokens on their input channels). The choice of which actors actually fire is nondeterministic.

At this level, our networks resemble Kahn Process Networks (KPNs) [76]: a KPN comprises a set of deterministic actors that communicate via tokens passed along unbounded FIFOs. Since we use a nondeterministic merge (arbiter) actor in our networks, we do not exactly follow the KPN model and thus cannot rely on Kahn's proof of determinism. However, the networks produced by our compiler are intuitively deterministic: we use nondeterministic merges around pure (i.e., side-effect free) blocks and "correct" for the nondeterminism by splitting merged streams according to the nondeterministic choices (see Section 4.3, Section 5.1.4, and Section 5.2.4 for further explanation). We formalize our network behavior (with the help of the KPN model) in Chapter 5.

Figure 4.2 lists the types of actors in our networks; each has its own firing policy. Because each channel is an unbounded FIFO that can always accept another token, an actor's ability to fire solely depends on the presence of tokens on its input channels.

Each actor on the left part of Figure 4.2 has "and" firing rules: each requires exactly one token per input to fire. A *primitive* actor models a constant, simple arithmetic or Boolean function, or data constructor, and produces a single output token when it fires. A *destruct* actor takes a constructed object (e.g., *Cons*) and produces an output token for each of the object's fields on a dedicated output channel. A *fork* actor consumes a single input token and copies it to each of its output channels.

A *demux* actor routes an input token (from the top) to one of its output channels depending on the value of a “choice” token (from the side). The “choice” token is a data constructor (just like the input to a destruct) of some type *T*, and the demux has an output channel for each variant of *T*. It routes its input token to the output channel corresponding to the variant received on the side channel.

Memory *read* and *write* actors behave like primitive actors with single inputs, but deserve special mention anyway. As in Floh, our dataflow networks assume an immutable, garbage-collected memory model. As such, a write actor takes a data token as input and generates an address token as output; a read actor does the opposite. Together, these actors maintain the deterministic memory operation invariant discussed in Section 3.3.5.

Our translation treats read and write actors abstractly: as independent and non-interfering, but their eventual implementation is more subtle. The memory system must ensure that it never generates an address token from a write before it is prepared to respond when that address is passed to a read. The system should also partition memory into multiple regions to improve memory-level parallelism. In Chapter 6, I present such a system that handles these concerns.

The actors on the right half of Figure 4.2 only require tokens on a subset of their inputs to fire. A *mux* actor is the opposite of the demux: it takes a choice token (from the side) and a token on the input corresponding to the choice (on the top), and transfers the input token to the output.

A *merge* actor is an arbiter: it consumes a token from one of its inputs (if tokens are available on more than one input channel, this selection is nondeterministic) and routes it to its output. *MergeChoice* actors have an additional choice output (drawn on the right) that generates a token indicating which input channel provided the selected token. This choice output often drives a demux that, together with a mergeChoice, manages access to a shared resource, e.g., a non-primitive function with multiple callers. In the rest of this chapter, I usually say “merge” to refer to either merge or mergeChoice actors since their only difference is the extra output reporting their selected input (otherwise, they have identical behavior). Where necessary, I make the distinction for clarity.

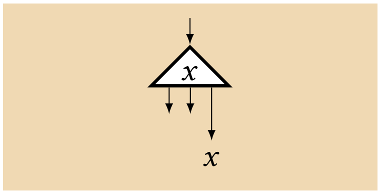


Figure 4.3: Translating a reference to a variable x: a connection is made to the fork actor that distributes its value.

4.3 Translation from Floh to Dataflow

In this section, I describe the translation procedure that transforms a Floh program into a dataflow network. Overall, each function definition is transformed into a subgraph of the network with at least one input channel per argument and one output channel. When one function calls another, additional inputs and outputs are added as described below.

Running a program amounts to supplying a single token to each argument input channel of a distinguished "main" function and waiting for a single output token to be produced in response. The "main" function typically takes a single *Go*-valued argument (representing the *Go* object bound in a Floh program's *main* definition), but may take other values from the environment.

4.3.1 Translating Expressions

The dataflow subgraph we generate for each Floh expression behaves like a function: the subgraph has a non-zero number of input ports, one per live variable in the expression, and a single output channel (tail-recursive calls are the exception). Delivering a single token on each input channel of the subgraph will trigger the evaluation of the expression, which will produce a single token on the output.

Our translation maintains an invariant that each live variable has its own fork actor; each reference to a variable in an expression adds an output port and channel from that variable's fork, as shown in Figure 4.3 (this may produce unary fork actors, which we optimize away). This implicitly assumes every reference to a variable in an expression will be consumed, which Floh's syntax and our translation rules enforce.

A call to a data constructor or a built-in, constant-generating, or memory access function is converted into a single primitive actor. As shown in Figure 4.4, we add a new connection from each argument's fork (each argument is necessarily a variable) to the appropriate input port. The result channel from such an expression is the output channel from the primitive actor. Although constant-generating and abstract memory access functions are defined in the input program, their definitions are ignored by our translation; the memory access function calls are

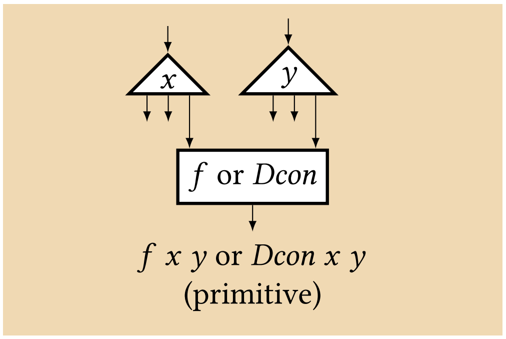


Figure 4.4: Translating a data constructor or a primitive, constant-generating, or memory access function call: each argument (one or more variables) is taken from a new connection to that variable's fork actor and the result is the output of the primitive actor.

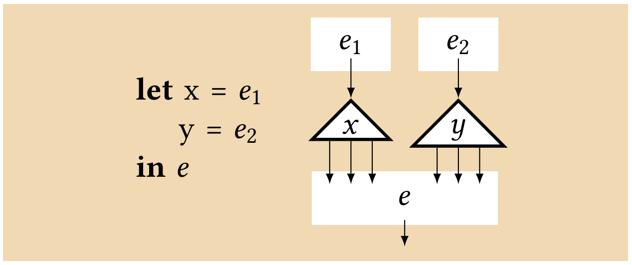


Figure 4.5: Translating a *let* construct: Each of the newly-bound variables is evaluated and connected to fork actors that make their values available to the body.

realized with the dedicated read and write actors from Section 4.2, while constant-generating functions are implemented as primitive actors that take a single *Go*-valued argument and produce the appropriate constant.

Translating a *let* construct, depicted in Figure 4.5, consists of translating the expression for each new variable, connecting the output of each to a new fork, and then translating the body of the *let* to produce the final result.

Our translations of *case* expressions and general function calls (i.e., those not implemented with primitive actors), especially tail-recursive calls, are context-dependent; I describe them in the next sections.

4.3.2 Translating Simple Functions and Cases

We divide Floh functions into two groups for translation: *simple* and *clustered*. A simple function has no tail-recursive calls; a group of one or more mutually (tail-) recursive functions is a *cluster*.

Simple functions are easily pipelined; function clusters often exhibit pipelines internally, but pipelining external calls to clusters is difficult because the subgraph for a cluster may return results from multiple calls out of order. While externally pipelining clusters would be possible by tagging tokens and adding reorder buffers, we have not yet attempted to do so. Instead, we internally pipeline them by implementing Floh's non-strict semantics for tail-recursive calls; we discuss how the translation implements this and why the same translation scheme does not work

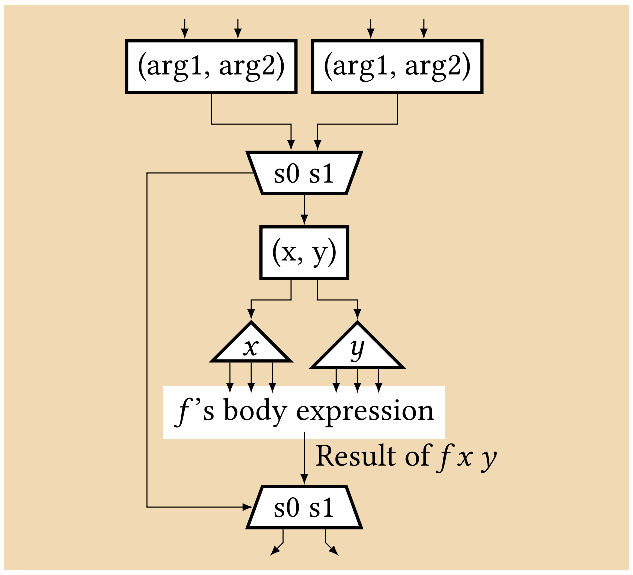


Figure 4.6: Translating a simple two-argument function f with two external call sites, *s0* and *s1*. Data constructor actors impose strictness by bundling a caller's arguments in a tuple; a destruct actor dismantles the tuple back into the constituent arguments. A mergeChoice actor selects which caller's tuple will access the function, while a demux routes the result to the caller.

for simple functions in Section 4.3.3.

Figure 4.6 shows how each simple function becomes a collection of actors surrounding the translation of the function's body. To implement Floh's strict semantics for simple functions, a primitive data constructor actor bundles a caller's arguments in a tuple; every argument must arrive before the actor can output its tuple token. The tuple token then goes through a merge that generates a choice token indicating which call site provided the (now bundled) arguments. A destruct actor extracts the arguments from the selected tuple, and the choice token is sent to a demux that routes the result of the body expression back to the appropriate caller. Each argument extracted by the destruct actor is fed into a dedicated fork that distributes the argument wherever it is used within the expression. Each additional call site for a simple function adds another tuple construction actor, a merge input for the arguments, and an output to the demux.

Although nondeterministic merge actors break Kahn's semantics (and thus prevent a simple proof of determinism), they let us avoid a global scheduler to arbitrate access to shared functions. Such a scheduler would be inefficient, as Kahn's semantics would prevent it from doing any kind of dynamic load balancing across the shared resources.

The merge actors also enable pipeline parallelism across multiple callers. As soon as a caller's tuple arrives on one of the merge actor's inputs, the merge can pass it into the function's body (arbitrating if multiple tuples arrive simultaneously), even if the tokens corresponding to another caller are still flowing through the function. The FIFO channels connecting dataflow actors prevent out-of-order execution among these multiple function calls, and the choice token passed

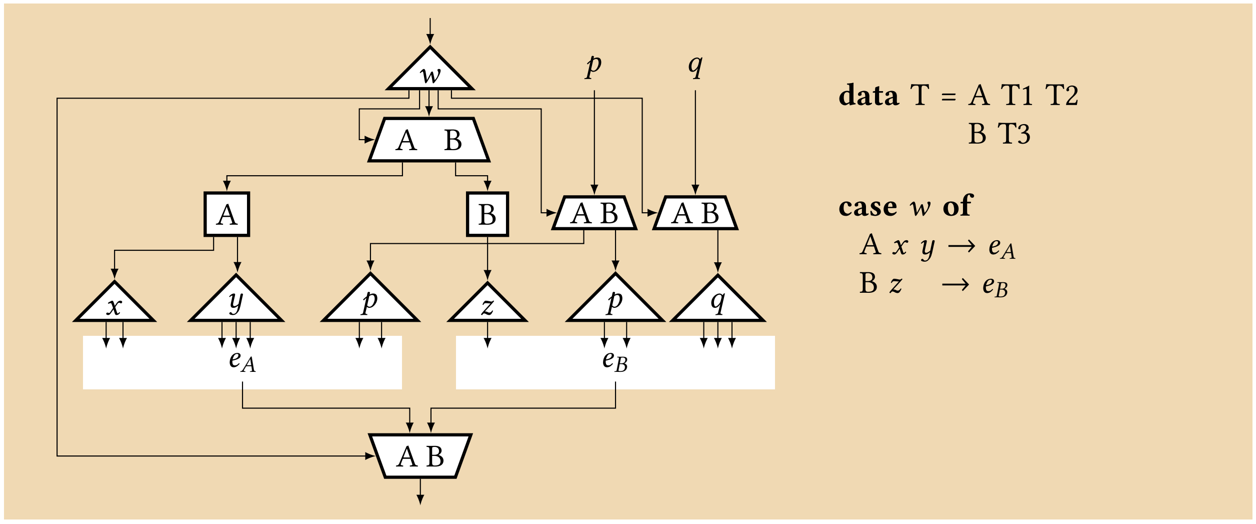


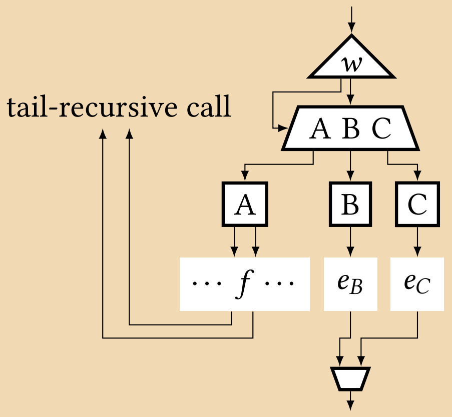
Figure 4.7: Translating a *case* construct: a demux actor routes input w to the destruct actor corresponding to w*'s* data constructor. Each destruct splits the token into fields: x and y for A or z for B. The data constructor from w also serves as a choice token that drives both the mux that selects the *case's* result and the demuxes that steer the values of live variables p and q to the alternatives that use them. The omitted demux output for q means e_A does not reference that variable.

from the merge to the demux ensures that results are passed back to the appropriate callers.

Figure 4.7 illustrates how a *case* construct is translated in general (some *cases* within clustered functions require special treatment). The *case* is implemented with a demux actor and a set of destruct actors, one for each of the *case*'s data constructor patterns. A data constructor token (the *case's* argument) is forked to both inputs of the demux, which routes it to the destruct actor matching its data constructor. The destruct then forks that constructor's fields out (if they are referenced) to its alternative expression as newly bound variables. If none of the fields of the matched data constructors are used in any alternatives (e.g., a *case* matching on a Boolean), the demux and destruct actors are unnecessary; the data constructor token is simply fed into a set of demuxes and a mux, explained below.

The data constructor token is used to steer local variables to different alternatives. Its fork distributes this token to a demux for each free variable that is live in some alternative. If the token encodes an alternative that does not need a given free variable, that variable's demux simply consumes its inputs without producing an output; the demux for q in Figure 4.7 does this when alternative A is selected. This ensures that no extraneous tokens are produced, and that all produced tokens will be consumed.

The output of the *case* is selected by a mux according to which alternative was evaluated. By



```
f a b = ··· case w of
  A x y → ··· f ···
  B z   → ··· e_B
  C v   → ··· e_C
```

Figure 4.8: Translating a *case* construct containing tail-recursive calls. Values produced by the alternatives are collected at a merge actor; arguments for intra-cluster tail calls are fed to each function's internal call site machinery. Not shown are the demuxes for live variables, which are treated the same as in Figure 4.7.

definition, a simple function may not contain a tail-recursive call, so every alternative expression will produce a value. This invariant does not hold within clusters, necessitating an alternate translation scheme.

4.3.3 Translating Clustered Functions and Cases

A function containing a tail-recursive call—a clustered function—presents a wrinkle in our translation scheme. Unlike all the expressions presented so far, a tail-recursive call within a cluster does not generate a subgraph with an output channel; it induces a cycle in the network that feeds arguments to a function within the same cluster. These cycles necessitate a different approach for translating calls within and to a cluster. Before presenting this new scheme, we first discuss how to deal with tail-recursive calls produced by *case* constructs.

Our original translation of *case* constructs assumed an output channel for every alternative; *case* alternatives ending in tail-recursive calls (which can only occur in a cluster) violate this assumption, since they induce cycles instead of providing a new output channel. Note that these types of *case*s cannot occur within *let*-bound expressions, even in a cluster; any recursive call within such a *case* would require more computation after the call returned, i.e., such a call is not tail-recursive.

Figure 4.8 illustrates our solution to this problem; two of the *case*'s alternatives return results while a third yields a tail-recursive call. A demux still examines and routes the algebraic data type

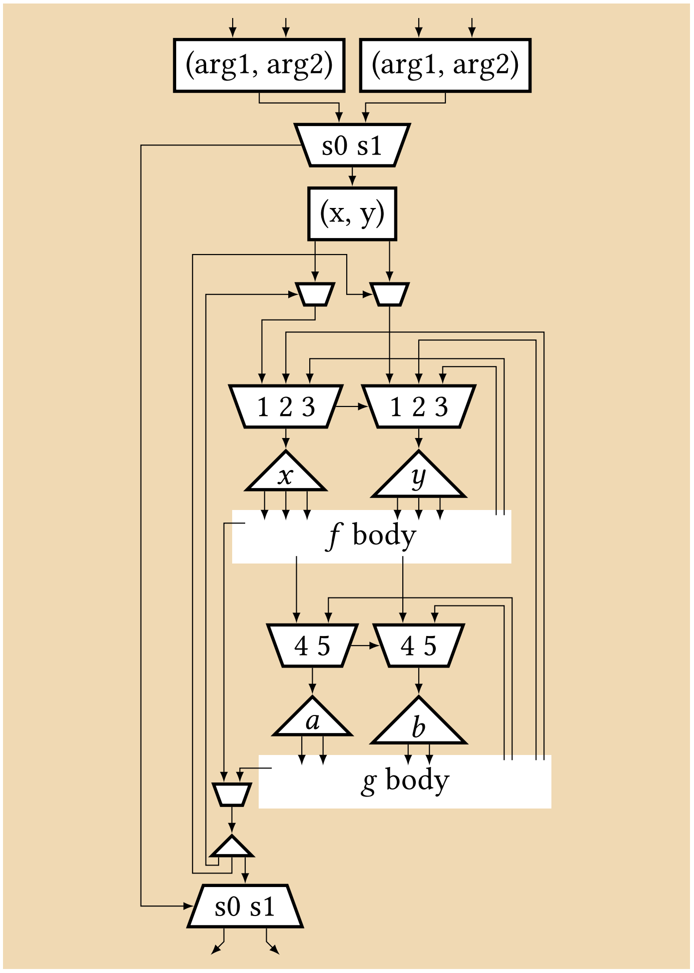


Figure 4.9: Translating function clusters. Functions f and g comprise the cluster since they call one another recursively. Any values produced by members of a cluster are merged together to form the cluster's output channel; using a mux instead could lead to deadlock. We omit local demuxes for clustered functions for the same reason. A layer of mux actors below the argument tuple's destruct actor act as a "lock" by preventing multiple external calls from overlapping within a cluster; the presence of a token on the cluster's output channel triggers the "unlocking" of these muxes, allowing another external call to access the cluster.

to destruct actors that dismantle it into fields, and the data type token still steers live variables (not shown in Figure 4.8), but alternatives ending in a tail-recursive call do not produce a value and thus are not assigned a dedicated output channel. A more significant change is the replacement of the *case*'s mux with a merge; we use a merge actor because tail-recursive calls make it difficult to determine where a result will ultimately come from.

We translate a cluster of functions as a whole since they tail call each other (by definition). Each cluster is assumed to have only one entry point (i.e., we do not handle clusters where more than one member of the cluster is called from the outside); we add a destruct, merge, and tuple constructor actors for the arguments to the sole entry point and a demux for the cluster's result as in the simple function case. Functions within the cluster are translated differently, however.

Figure 4.9 shows how we translate a cluster of two functions, f and g, that recursively tail call each other and themselves. Each function within a cluster receives its arguments via merge and mux actors, which manage the intra-cluster calls to the function; the first (leftmost) argument goes through a merge that generates a choice token indicating which call site provided the argument. Every other argument comes from a mux that uses the choice token to select the input channel corresponding to the same call site. As with simple functions, each argument—the output of either the merge or one of the mux actors—is fed into a dedicated fork that distributes it across the function's body. Unlike simple functions, additional (tail) call sites to a clustered function adds another input to each of the merge and mux actors for the arguments, not an additional tuple constructor actor. As another change, the results of each function are passed to a single merge actor for the cluster (i.e., rather than the per-function demuxes used in our translation of simple functions). Again, our use of a merge actor is motivated by the presence of tail-recursion.

The other substantial difference in translating a cluster is a layer of "locking" mux actors that block multiple external calls from accessing the cluster (seen below the tuple destruct actor in Figure 4.9). The interior of a function cluster does not behave like a simple pipeline: intra-cluster tail calls turn into data-dependent feedback paths. If we allowed *n* external calls to access a cluster, the network for the cluster would still produce *n* result tokens, but not in any pre-determined order. Rather than adding tags to every token and a reorder buffer to guarantee in-order delivery of results, we instead opt to limit each cluster to one external call at a time.

The locking layer of mux actors allows exactly one external function call to execute within the cluster at any given time. These actors accept one token on each (top) input channel and block any additional inputs until the cluster signals it has produced its result, which is indicated by duplicating the result token with the fork near the bottom of Figure 4.9 and passing it as an "unlock" token to the muxes.

Here, we make an important choice that separates us from similar dataflow translations: tail-recursive function calls are not strict. In particular, the actors comprising a clustered function's body may start firing before every function argument is available; once the first argument from a given recursive call site passes through the merge actor, the other arguments from that call site can arrive in any order, allowing computation to proceed in a data-dependent manner and enabling pipeline parallelism across multiple calls. Since our translation does not reorder arguments, the programmer can enable parallelism by ordering function arguments appropriately.

Our asymmetric handling of arguments is key to this non-strict policy: if each argument had its own merge, for example, each might make a different choice when faced with simultaneous calls, effectively permuting the arguments among multiple recursive call sites. We avoid this problem with a single merge actor that dictates which call site to service.

In an earlier iteration of our compiler, simple function calls were also non-strict [125], but it turns out that this may cause a subtle form of deadlock. Consider the Floh example below comprised of a simple function *f* and two tail-recursive functions *f1* and *f2* that all call some other simple function *g*:

```
f  x = ··· g (f1 ···) (f2 ···)
f1 y = ··· g (f1 ···) (f1 ···)
f2 z = ··· g (f2 ···) (f2 ···)
```

If *f* is called first, it will eventually call *f1* and *f2*, triggering their simultaneous execution. Assume *f1* produces a result before *f2* is done, i.e., *f2* still needs to call *g* again before it can produce its result. The result from *f1* will be returned to *f*, which will then pass it as the first argument to *g*. If *g* were non-strict, this first argument would pass through a merge actor which would then inform the mux for *g's* second argument to wait for a token from the same call site (in *f*). But this second argument is the result of the initial call to *f2*, which will not arrive until *f2* gets access to *g* (since it has not finished executing). This will never happen, and thus a deadlock has formed.

We thus impose a strict policy on simple function calls to prevent this form of deadlock. If a simple function only has a single caller, though, this situation cannot occur, so we optimize away all of the actors surrounding the function body's subnetwork (they are only needed to arbitrate between multiple callers).

4.3.4 Putting It All Together: Translating the Map Example

Figure 4.10 shows the dataflow network our procedure generates for the *map* example introduced in Figure 4.1. As described in Section 4.1.1, this walks an input list and pushes each value on a stack, then repeatedly pops the stack, removing each element, applying the function *f*, and prepending the result to a new list.

Call and *ret* contain tail-recursive calls but are not mutually recursive, so each is treated as a cluster. Thus, each has a layer of "locking" muxes on their inputs to ensure they will not accept another outside call until they have generated an output. Since each function has only one caller, the strictness-inducing tuple constructor and destruct actors are all optimized away by the compiler.

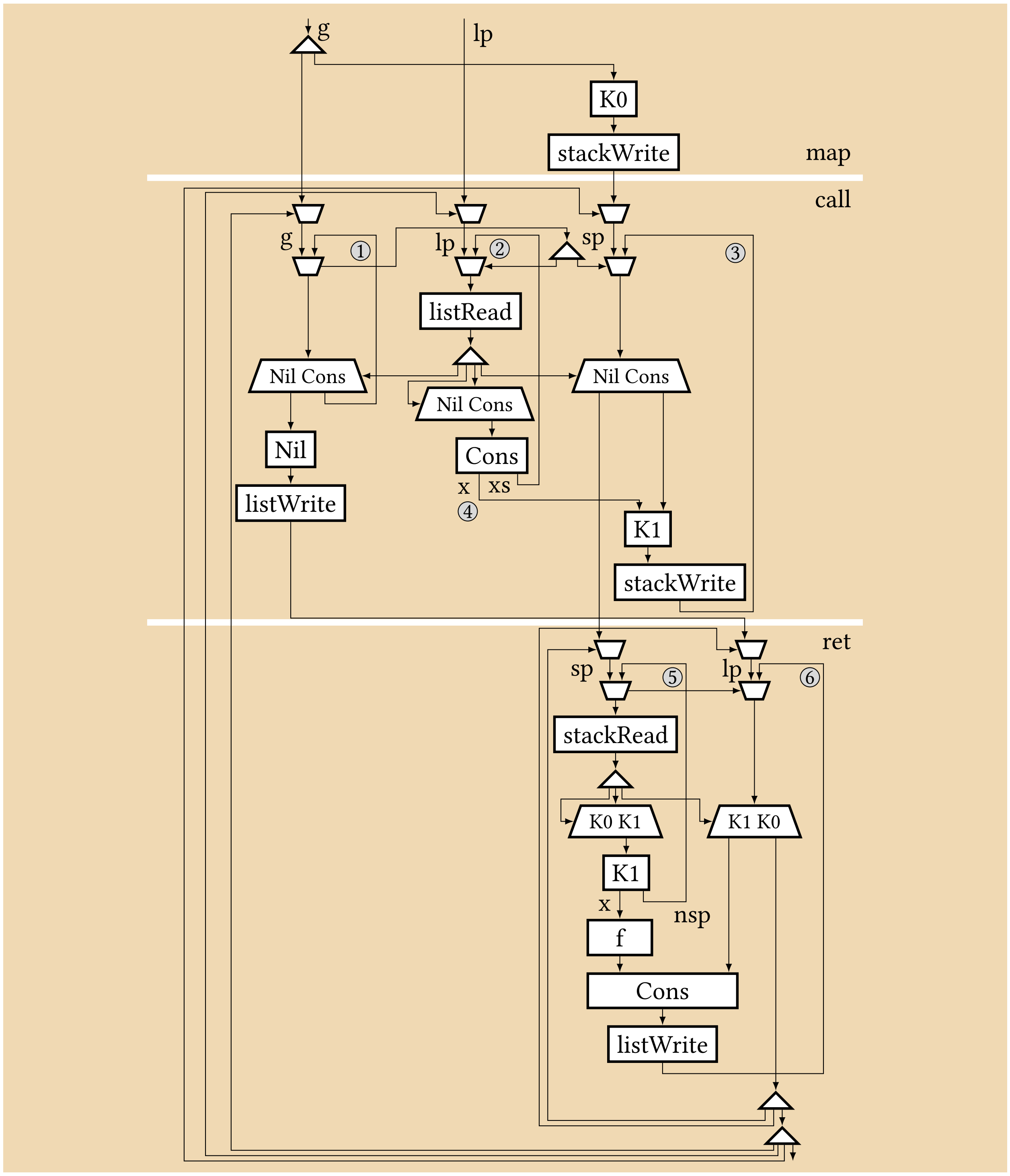


Figure 4.10: A dataflow graph for *map* from Figure 4.1. This initializes the stack (*map*); walks the input list, pushing each element on the stack (*call*); then pops each element off the stack, applies *f*, and places the result at the head of the new list (*ret*). Tail calls to *call* and *ret* are not strict, decoupling loops 1, 2, and 3, and more importantly, loops 5 and 6, to enable pipelining.

This example illustrates how tail-recursion coupled with non-strict functions and buffering enables pipeline parallelism. The tail-recursive call in the *call* function induces three separate loops—1, 2, and 3—which operate largely independently. In particular, loop 2, which reads the input list, can race ahead (since it only has to wait for loop 1, which has no long-latency operations on it), producing data tokens on channel 4. These tokens are eventually consumed by loop 3, which places them on the stack as a series of "K1" objects. Although fast, loop 2 is a bit wasteful: it waits and releases a *Go* token when the end of the list is reached, triggering the creation of the result list.

A strict implementation of the *call* function would force all three loops to operate in lock-step, i.e., the next element of the list could not be read before the stack was pushed.

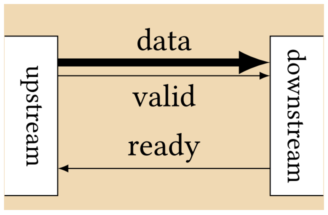
Pipelining is even more effective in the *ret* function. Loop 5 pops data off the stack so that f can be applied to it. If f is a long-latency function, loop 6 will be slow because it will have to wait for f to complete, write the new list element, and recurse. But the tail-recursive call to *ret* is non-strict so loop 5 can race ahead, perhaps even filling f*'s* pipeline to greatly improve parallelism.

In Section 4.5, I quantify how well our technique exposes parallelism in this example and others.

4.4 Dataflow Networks in Hardware

We take a structural, distributed approach to implementing dataflow networks in hardware: each actor becomes a small block of combinational logic, these blocks use a handshaking protocol to implement latency-insensitivity, and each buffer is bounded as a finite bank of flip-flops. A large, central memory could simulate unbounded buffers, but such an approach would likely require additional throttling mechanisms, such as Arvind and Nikhil [3] found. Our approach thus maintains the parallelism of the dataflow network model while enabling high-speed hardware, since far-flung parts of the circuit do not need to communicate with a global memory or each other.

Bounded buffers complicate actor firing rules, which must also take into account the availability of space downstream. Since this "availability information" flows upstream, a naïve translation may generate an excessively slow circuit due to long combinational paths or a broken circuit plagued with combinational cycles. Cao et al. [18] present a solution to these issues by implementing each channel as a two-token buffer. This technique breaks any potential cycle or long



valid	**ready**	**Meaning**
0	–	No token to transfer
1	1	Token transferred
1	0	Token valid, but not consumed (i.e., held upstream)

Figure 4.11: A point-to-point link and its flow control protocol, after Cao et al. [18]. Data and *valid* bits flow downstream, with *valid* indicating the data lines carry a token; *ready* bits flow upstream, indicating that downstream is willing to consume a token.

combinational path with a flip-flop but hinders throughput, as tokens can only cross up to one actor per cycle. Since some actors do more work than others, grouping simple actors into a single cycle would reduce latency without affecting clock speed.

We adopt a variant of Cao et al. in which channels are either two-place buffers or direct wires. Having two choices allows us to control the work per clock cycle by "fusing" multiple actors together. Our circuits use the flow control protocol shown in Figure 4.11, which presents a danger of a combinational cycle (and hence deadlock) if the *valid* signal depends on the *ready* signal and vice versa. Below, I discuss our implementation technique for avoiding these cycles.

4.4.1 Evaluation Order

Establishing a fixed, constructive evaluation order for *valid* and *ready* signals prevents deadlock in the flow control logic. We choose a three-phase evaluation order starting from the buffers of Cao et al. [18]: the *valid* and *ready* outputs from every buffer are defined at the beginning of each cycle and do not depend on any inputs. In the second phase, *valid* bits propagate downstream, unaffected by *ready* bits. Finally, *ready* bits are propagated upstream and may depend on *valid* bits. For this arrangement to work, the *valid* outputs of an actor may never depend on its *ready* inputs in the same cycle, which turns out to be a delicate property to guarantee.

4.4.2 Stateless Actors

Under our *valid*-then-*ready* evaluation order, actors that produce a single token when fired have fairly straightforward flow control logic. Primitives actors are simple: the output is valid if all the inputs are valid; the inputs are ready if the output is valid and ready. A demux is similar: the chosen output is valid if both the inputs are valid; the inputs are ready if the chosen output is valid and ready. A mux is slightly more complicated: the output is valid if the choice (side) input and the chosen input are valid; the choice input and chosen input are ready if the output is valid and ready. The merge is still more complicated: we currently use a priority-based arbitration scheme in which the output is valid if at least one input is valid; the leftmost valid input is ready if the output is valid and ready.

4.4.3 Stateful Actors

Actors such as fork that generate multiple tokens when they fire present a challenge to our evaluation scheme. Tokens could be erroneously duplicated if we used the obvious rules for fork, i.e., all the outputs are valid if the input is valid and the input is ready if all the outputs are ready. Under these rules, if one of the outputs was not ready when the fork fired, that output would not consume the token but the others would; a duplicate token would then be presented to all the outputs again on the next cycle, even though some already consumed it. It might seem possible to address this by making the outputs valid only if all the outputs are ready, but this violates our evaluation order policy and can cause deadlock.

Our solution, credited to Andrea Lottarini, is to add a few bits of state to actors that can generate multiple tokens: fork, destruct, and mergeChoice. Specifically, each output is given a state bit that indicates whether a token has been generated from that output in the current "round" of firing. When sufficient tokens are available on the inputs to produce outputs, an output's bit is set if its channel is not ready. If any output bits are set, the input is not consumed; it is proffered again on the next cycle, but the actor only produces tokens on the outputs that have an unset state bit. Once tokens have been generated on all the outputs in a given round (i.e., all state bits are set), all the bits are reset and the next round can begin in the next cycle.

With this policy, a state bit can disable a *valid* output, preventing erroneous token duplication, but a *valid* signal never immediately depends on a *ready* signal, satisfying our scheduling criteria.

4.4.4 Inserting Buffers

As stated above, we implement certain channels in our dataflow graph with the two-token buffers described in Cao et al. [18] and the rest as simple wires, leaving them unbuffered. We insert buffers primarily to avoid deadlock, although additional ones are often desirable to balance both per-cycle computation and pipelined paths.

Choosing appropriate buffer sizes (i.e., the number of buffers to place on a channel) can range from straightforward to undecidable. Unfortunately, the optimal buffer insertion problem for a dataflow network with arbitrary topology is undecidable: Buck [17] showed that a very simple subset of actors such as ours (which include data-dependent mux and demux) is Turing complete, rendering undecidable the question of whether arbitrary networks of our actors and buffers can run without deadlocking. However, a dataflow network generated from a syntax-directed translation of a structured program (e.g., Dennis [39]) never requires the accumulation of an unbounded number of tokens to run, so inserting buffers according to a simple policy suffices to prevent these networks from deadlocking. We have a number of such policies explained below; numerous authors have proposed other approaches [10, 55, 56, 57, 58, 66, 93, 97, 121].

Before applying one of our several buffering heuristics to prevent deadlock, we buffer the "locking" layer of a function cluster for correctness. The muxes in this layer each receive a single buffer on their select input; each buffer is initialized with a *Go* token, which "unlocks" the muxes for the first call to the cluster. The output token produced by the cluster is converted into a *Go* token (to match the type the muxes expect on their select input), forked to these buffers, and passed to the muxes to unlock them for the next call. Without these initialized buffers, the muxes would stay locked and prevent any callers from accessing the cluster.

We provide three different heuristics to determine where to place additional buffers for deadlock prevention; the user may select which heuristic to apply. Each heuristic has two goals: buffer each cycle in the dataflow network (to eliminate combinational cycles) and prevent "reconvergent deadlock." Multiple paths in a network are *reconvergent* if they share the same source and destination actors but no others. These paths necessarily originate at multi-output actors and terminate at multi-input actors. Two such paths are *mismatched* if exactly one is buffered.

As an example of reconvergent deadlock, consider the dataflow graph shown in Figure 4.12, but without buffers 3 and 4; two mismatched reconvergent paths originate at the fork and terminate at g. If a token is in buffer 1 when the fork first fires, f will consume both inputs and produce a token in buffer 2. However, g cannot consume the fork's right output token yet, so the fork does not consume its input token. In the next cycle, buffer 2 and the right output of the fork will both supply a token to g, which would normally enable it to fire, but f is blocked because it does not have a new token from above. This will in turn block g, creating a deadlock.

Our first heuristic is the simplest: place one buffer on every channel. This buffers every cycle in the network and prevents mismatched reconvergent paths by definition, but is highly inefficient; the excessive buffering hinders throughput and increases overall network latency. We only use this heuristic to test our networks for functional correctness.

The second heuristic targets channels corresponding to a function's inputs and outputs; it

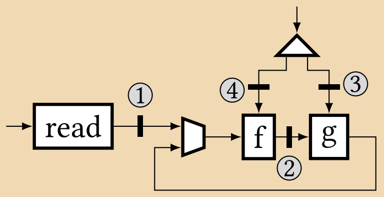


Figure 4.12: An example of our buffer allocation scheme (using our third heuristic). We insert buffers to break cycles in the network (e.g., 2) and prevent reconvergent deadlock (e.g., 3).

assigns less buffers than the first heuristic (improving network performance) but is not formally guaranteed to prevent reconvergent deadlock (although none of our networks using this heuristic have deadlocked in practice). Simple functions receive buffers on the inputs of their argument-bundling data constructor actors and the outputs of the final demux routing the function's results to its callers. Clustered functions get buffers on the inputs of the merge and mux actors implementing non-strict tail-recursive calls (below the layer of "locking" mux actors). Buffering clustered functions' inputs eliminates unbuffered cycles in our networks, since cycles only arise due to our translation of tail-recursive calls. The other buffering in this heuristic intuitively prevents reconvergent deadlock: reconvergent paths often contain a function's input or output channel, so buffering these channels should preclude mismatched paths. We use this second heuristic to buffer all the example networks in the experiments of Section 6.3 and Section 7.3, and it is the default heuristic used in the compiler's current implementation.

The second heuristic relies on our specific translation scheme, which dictates where cycles can occur and where mismatched reconvergent paths tend to appear. Our final heuristic instead focuses on general network topology; the rest of this section describes this third heuristic, and we use it to buffer the networks used in the experiments of Section 4.5 and Section 5.6.

Figure 4.12 shows part of a network after buffer insertion using our third heuristic. The first part of this heuristic ensures that cycles have been broken with buffers with the following process: find the shortest unbuffered cycle in the graph (we modify Dijkstra's shortest-paths algorithm to solve this); find an actor on the cycle with the largest number of outputs; place a buffer on the output that belongs to the cycle. We repeat this process until all cycles are buffered. This heuristic targets actors with multiple outputs to prevent throughput degradation on the other outputs that are not part of the cycle. In Figure 4.12, buffer 2 was inserted to break the cycle.

The second part of the heuristic finds and buffers mismatched reconvergent paths after breaking cycles (since the latter may buffer some mismatched paths implicitly). Although finding all such paths is tractable in DAGs [108], it is NP-hard in general, requiring a heuristic solution.

We leverage an approximation algorithm for counting reconvergent paths [131] between nodes i and j. We convert our network into a weighted graph by assigning a 0 to unbuffered edges and a 1 to buffered edges. We find a shortest path on this graph between i and j (terminate if no such path exists), remove it from the graph, and repeat. The set of removed paths comprises a set of reconvergent paths from i to j. Let out_i and in_j be the out-degree of i and in-degree of j, respectively; then $\min(out_i, in_j)$ is an upper bound on the number of searches required. Using Dijkstra's algorithm on a graph with m edges and n nodes, this algorithm runs in $O(\min(out_i, in_j)(m + n \log n))$ time.

Our third heuristic applies this algorithm to each pair (i, j) of multi-output (i) and multi-input (j) actors, keeps any mismatched sets of paths, and returns the unbuffered members of each set. We walk over this set of unbuffered paths, selecting the first edge from each (unless a selected edge from a previous path is also on the current path), and assign a buffer to each edge in the resulting set, updating their weights in the graph. Let $d = \min(out_{max}, in_{max})$, where out_{max} (in_{max}) is the largest out-degree (in-degree) of any node in our set of p node pairs. Then this heuristic algorithm runs in $O(pd(m + n \log n))$ time.

Although this scheme successfully prevents reconvergent deadlock, its overly conservative nature yields unnecessary buffers. As shown in Figure 4.12, the heuristic will first allocate buffer 3 to balance the reconvergent paths terminating at g. The placement of this buffer means that the reconvergent paths from the fork to f are now mismatched, which will cause the heuristic to place buffer 4. However, this buffer is unnecessary, as buffers 2 and 3 together prevent the deadlock. These excessive buffers may improve performance by providing implicit pipelining, but some topologies are hurt by these buffers, since they can increase overall completion time without increasing throughput.

This final heuristic algorithm adds some nondeterminism to our translation: permuting the heuristic's input can lead to different buffer allocations for the same topology. However, this only affects the performance of the resulting circuit, not its correctness; we present an argument for this correctness in the face of arbitrary buffering in the next chapter.

4.5 Experimental Evaluation

To evaluate the quality of the dataflow networks produced by this compilation pass, I simulated several examples. I analyze the impact of non-strict evaluation on network performance, the importance of argument order, and sensitivity to memory latency.

All experiments in this section used an earlier version of the compiler that had a slightly different translation scheme (described in [125]). The specific differences are as follows:

- *Case* constructs were directly implemented with a special *case* dataflow actor. We realized this actor's behavior could be implemented with other actors we had already created (a demux and several destruct actors), and thus removed it from our translation.
- A special *lock* actor prevented multiple external callers from sharing a cluster simultaneously. As with the case actor, we determined that the same behavior could be achieved with a layer of "locking" muxes.
- All non-primitive function calls used a non-strict evaluation policy.

While the first two differences have negligible effects on the experimental results (the underlying implementations of the *case* and *lock* actors are comparable to the sets of actors now used), our use of strictness for simple function calls could inhibit pipeline parallelism across our networks and thus reduce performance. While we currently apply strictness to all simple functions to prevent the deadlock issue discussed in Section 4.3.3, none of the examples in the following experiments experience this form of deadlock, even without the strictness rule applied. This suggests that a heuristic could be developed to determine where strictness is truly required; the following results thus give an idea of how our compiler may generate networks when such a heuristic is applied.

4.5.1 Methodology

Simulator To evaluate the performance of our generated dataflow networks, I wrote a simulator that executes a network on a set of inputs and reports both the final output token (to compare against the output of the original Floh program) and the number of clock cycles required.

In one mode, my simulator runs a cycle-accurate model of a hardware implementation that employs the finite buffers of Cao et al. [18]. In particular, it models their single cycle latency.

In the other mode, my simulator calculates a lower bound on the number of cycles hardware would take by assuming ideal buffering. Here, buffers are modeled as unbounded, but in each cycle, each actor is limited to firing at most once. This captures an ideal buffering assignment where every buffer is "big enough" to never cause backpressure; the resulting cycle count provides a lower bound on the cycles a particular network requires to process the input.

Test Programs I compiled six recursive Haskell programs into dataflow networks for evaluation. Append, Filter, and Map each traverse a list and perform computation on each element: Append prepends each element to a new list, Map applies a function *f*, and Filter applies a Boolean-producing function *g* whose result determines if an element is kept or discarded. I assume *f* and *g* are both fully pipelined and each have a latency of 10 cycles. Treemap functions the same as Map but operates on a tree.

The DFS and Mergesort tests are more complicated. DFS applies depth-first search to a binary tree, producing a preordering of its elements. At each tree node it recurses on the left and right subtrees, appends the results, and prepends the node's element to the final list.

Mergesort is the well-known sorting algorithm with four tail-recursive functions: *evens* and *odds* together partition a list into its even- and odd-indexed elements, *merge* combines two sorted lists into a single sorted list, and *mergeSort* drives the other three functions. The translated dataflow graph consists of seven clusters: the *call* and *ret* components of *mergeSort* are mutually recursive and thus share a cluster, while the components of the other functions each have their own cluster. This application represents the types of real-world programs our work targets: memory-intensive algorithms implemented with multiple interactive recursive functions.

Input Data Each dataflow network is fed a *Go* token that triggers the construction of an input data structure. This structure is either a 100 element list or 100 element balanced binary tree, according to the test program. This structure is then processed by the rest of the network.

4.5.2 Strict vs. Non-strict Tail Recursive Calls

My first experiment measures the performance impact of our non-strict function evaluation policy intended to enable pipelining. I generated each test's dataflow network under three different policies: non-strict function calls with infinite FIFOs on each channel, non-strict with finite buffering, and strict with finite buffering. I also varied the order of function arguments under each policy; a "good" ordering implies that the first argument routinely arrives before the others (like the first argument to *call* and *ret* in Figure 4.10), which enables the function to start being

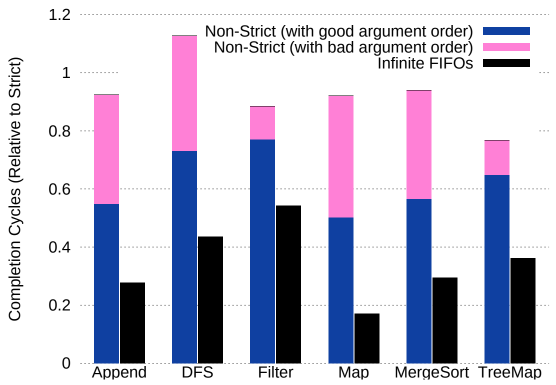


Figure 4.13: Non-strict evaluation is generally superior to strict. When combined with a good function argument ordering, the finite, non-strict implementations yield a 1.3–2× speedup over strict.

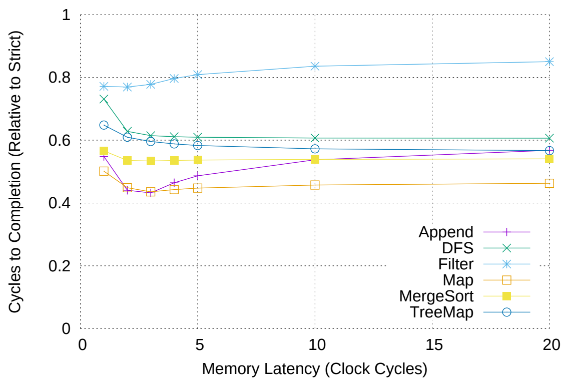


Figure 4.14: Mitigating increasing memory latency with non-strict function evaluation.

evaluated; a "bad" ordering entails one or more arguments arriving at muxes before the first arrives at its merge, meaning the function waits longer to start than absolutely necessary. These orderings are dictated by the Floh program's syntax, i.e., neither our translation nor our buffering scheme affects this ordering.

Figure 4.13 shows the fraction of cycles each test took relative to a strict policy with the same argument order. For reference, the baseline for Append (i.e., under a strict policy) was 1107 and 1308 cycles with good and bad argument orders; Mergesort was the longest at 14324 and 16973 cycles.

Figure 4.13 shows non-strictness with proper argument ordering yields faster completion times, which I attribute to the successful exploitation of pipelining. Under "bad" ordering, non-strict does slightly better than strict in most cases (the anomalous performance loss in DFS is due to our heuristic making poor buffering choices); combining non-strictness with effective ordering leads to approximate speedups from 1.3× (DFS, Filter, Treemap) to 2× (Append, Map, Mergesort). The infinite FIFO policy gives roughly twice the performance with non-strict functions under a good ordering, suggesting improved buffering can substantially improve performance.

4.5.3 Sensitivity to Memory Latency

Above, I modeled memory optimistically, taking only a single cycle; in reality, memory is rarely this fast. I conducted additional experiments to see how well our generated networks coped with higher memory latencies.

Figure 4.14 shows how long it took each program to run under increasing memory latency. Again, I used strict functions as the baseline and calculated the improvement under a non-strict policy (under "good" argument ordering throughout).

The initially negative slopes in Figure 4.14 show that our non-strict evaluation policy does do an increasingly better job with small memory latencies (e.g., under 5 cycles) than a strict policy does, but the differences become negligible after that, i.e., while the non-strict policy is consistently better, its advantage levels off at a constant improvement factor. After inspecting the execution traces for these workloads, I attribute these results to unbalanced buffer capacities along reconvergent paths in our networks.

To explain, consider two reconvergent paths originating at some actor *n*, such that one path has more buffers (i.e., has higher capacity) than the other. Depending on the frequency of *n*'s firing, the smaller capacity path can fill up before the longer path, preventing *n* from filling the longer path's pipeline. If the buffer capacity on the short channel matched the long channel's pipeline length, *n* could continue firing and potentially fill up both channels' pipelines, yielding higher throughput and lower completion times. Although our buffering heuristic can find these reconvergent paths, determining how to distribute buffers along these paths remains a difficult problem that requires further study.

4.5.4 Sensitivity to Function Latency

I also conducted an experiment designed to illuminate how our networks deal with varying function latency. Specifically, the function applied to each element in Map, Filter, and Treemap may take longer than 10 cycles to execute.

I varied this function's latency in these three tests from 1 to 50, keeping the memory latency at a single cycle. Not surprisingly, the resulting trends are nearly identical to those seen in Figure 4.14: after a slight widening of the gap between non-strict and strict, non-strict completion cycles rise before plateauing off. This further supports the previous conclusions that our buffer allocation scheme is not mature enough to fill up long pipelines, be they functional or memory-related.

Chapter 5

Realizing Dataflow Networks in Hardware

The previous chapter introduced the dataflow network model targeted by our compiler. I described the behavior of these networks informally, as the main purpose was to show how a functional program could be translated into an abstract dataflow network.

In this chapter, I formally define the semantics of our dataflow networks, provide the hardware implementations for the actors comprising our networks, and show that these implementations maintain the formal semantics. The hardware implementations and the argument for their correctness were primariliy developed by Stephen A. Edwards. I also present the final IR used by the compiler, DF; this IR is the textual output of the previous chapter's translation. DF enables design space exploration at the dataflow level, and serves as the input to the compiler's final code generation phase.

As a running example for this chapter, Figure 5.1 illustrates a dataflow network we can implement with our actors, i.e., our actors may be used for manual hardware design outside of our

```
gcd(a, b) =
  if a = b
    a
  else if a < b
    gcd(a, b − a)
  else
    gcd(a − b, b)
```

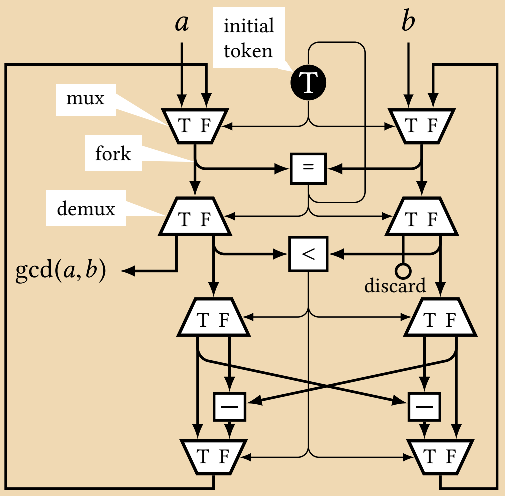


Figure 5.1: A recursive definition of Euclid's greatest common divisor algorithm and a dataflow network implementing it. The tail-recursion is implemented with feedback loops.

compiler. This example uses Euclid's algorithm to compute the greatest common divisor of pairs of tokens arriving on the two input channels. For example, if channel *a* receives tokens 100 and 56 and channel *b* receives 45, 49, and 3, the output will be $5 = \gcd(100, 45)$ followed by $7 = \gcd(56, 49)$. The 3 on *b* is ignored because no mating token ever arrives on *a*.

In this network, an initial "T" token is fed to the top row of multiplexers, instructing each to steer a token from an input to the primitive equality actor ("="). Because the channels from the multiplexers fork (we just use diverging channels for forks here instead of the triangles from the previous chapter), the first row of demultiplexers also receive copies of these tokens. If the tokens are equal (the base case for the algorithm), the equality actor emits a T, causing the demultiplexers to emit the *a* token as the result and discard the *b* token. The T from the comparator is also fed back to both input multiplexers, prompting them to accept a new pair of tokens from the inputs.

In the recursive case, the tokens differ, prompting the demultiplexers to send copies of the tokens to the primitive less-than actor (<) and to the second row of demultiplexers. The output of the less-than actor flows to the second demuxes and bottom multiplexers, which together control whether *a* is subtracted from *b* or *b* is subtracted from *a*. Since the equality actor emitted an F, the outputs from the bottom multiplexers are fed around and flow back through the top multiplexers and the process repeats.

Given a DF program specifying this network (a fragment of which is shown in Figure 5.11), our compiler synthesizes it into hardware by transforming each actor into a small block of logic and replacing each channel with a mixture of point-to-point connections, buffers, and fork circuitry.

In the rest of this chapter, I present:

- the formal semantics of our unbounded dataflow networks (Section 5.1);
- circuits for a small, rich family of data-dependent dataflow actors, which can be composed without buffering yet are safe from spurious combinational cycles (Section 5.2);
- a novel way to implement a nondeterministic merge actor that reports its choices, allowing it to safely manage shared resources (Section 5.2.4);
- an approach to breaking long combinational paths and loops that uses two distinct types of buffers: one for the data network and one for backpressure (Section 5.3);
- a typed "assembly language" for describing our networks with polymorphic actors and algebraic data types that we can compile into SystemVerilog (Section 5.5); and
- experiments that show how buffering may be added to explore the design space without changing functionality (Section 5.6).

5.1 Specifications: Kahn Networks

Our goal is a hardware implementation of a dataflow network. Here, we describe our specifications: a restricted class of Kahn networks with unbounded buffers. These specifications are deliberately more abstract than our implementations to allow buffers to be added and removed (e.g., to adjust pipeline depth) as part of the implementation process.

This section is largely review: Kahn [76] provides the framework, Lee and Matsikoudis [81] show how to model firing rules, and our model of nondeterministic merge is due to Broy [16].

5.1.1 Kahn Networks

A Kahn network consists of Kahn processes that pass around tokens. Kahn networks are deterministic because the processes are continuous, meaning that supplying additional input tokens can only produce additional output tokens; a Kahn process cannot "change its mind" once it has decided to emit a token. Below, we formalize such networks.

A Kahn network passes around *tokens* drawn from a set Σ. This set is typically finite and often includes 32-bit binary integers, but its structure is irrelevant for the semantics we present in this section; see Section 5.5 for how we construct sets of tokens in practice. Our networks do not necessarily terminate, so we consider both finite and infinite sequences of tokens flowing on channels and write $S = \Sigma^* \cup \Sigma^\omega$ for the set of such sequences. Note that the empty sequence ϵ is included in this set ($\epsilon \in \Sigma^*$). Juxtaposition will denote concatenation, e.g., for two tokens $x, y \in \Sigma$, xy represents the two-token sequence consisting of x followed by y. We also use juxtaposition for concatenation of sequences: if $a \in \Sigma^*$ is a finite sequence and $b \in S$, ab is the sequence a followed by b.

We use prefix ordering on sequences. We write $a \sqsubseteq b$ if a is a prefix of b or a is equal to b. It follows that $\sqsubseteq$ is a partial order. Technically $a \sqsubseteq b$ iff $a = b$ or $\exists c \in S$ s.t. $ac = b$. We extend this ordering elementwise to n-tuples of sequences (written in bold): if $a_1, \dots, a_n, b_1, \dots, b_n \in S$, $\mathbf{a} = (a_1, \dots, a_n) \in S^n$, and $\mathbf{b} = (b_1, \dots, b_n) \in S^n$, we write $\mathbf{a} \sqsubseteq \mathbf{b}$ iff $a_1 \sqsubseteq b_1, \dots, a_n \sqsubseteq b_n$. Juxtaposition of n-tuples of sequences denotes elementwise concatenation: $\mathbf{ab} = (a_1 b_1, \dots, a_n b_n)$.

A *Kahn process* is a continuous function $P : S^n \to S^m$ that takes a tuple of n input sequences and produces a tuple of m output sequences. Continuity means P is monotonic, so $\mathbf{a} \sqsubseteq \mathbf{b}$ implies $P(\mathbf{a}) \sqsubseteq P(\mathbf{b})$. Equivalently, providing P with additional tokens may produce more output tokens, but tokens that have already been produced cannot be changed or rescinded. Continuity also means a process cannot produce an output only after an infinite time. See Lee and Matsikoudis [81] for a formal discussion of continuity.

As an example, consider a process that adds two input sequences to produce an output sequence. Assume integer-valued tokens, i.e., $\Sigma = \mathbb{Z}$. This process computes the pairwise sum of the two sequences up to the end of the shorter sequence, i.e.,

$$P(x_1 x_2 \cdots x_n, y_1 y_2 \cdots y_m) = w_1 w_2 \cdots w_{\min(m,n)} \tag{5.1}$$

where $w_i = x_i + y_i$ and sequence lengths m and n may be zero, finite, or infinite.

A Kahn network is a collection of Kahn processes whose input sequences are supplied through channels, each of which is either supplied by the environment or the output of some process. A *Kahn network* $N = (\mathbf{P}, e, M)$ is a triple consisting of a vector of r Kahn processes $\mathbf{P} = \{P_1, \ldots, P_r\}$, a number $e \in \{0, 1, \ldots\}$ of input channels from the environment, and a "wiring matrix" function $M : \{1, \ldots, r\} \times \{1, \ldots\} \to \{1, \ldots, e\} \cup (\{1, \ldots, r\} \times \{1, \ldots\})$ that maps each process input (a process number and input index) to either one of the e environment channels or the output of some process (a process number and output index). The M function is pure bookkeeping: it merely encodes the connectivity of the network (i.e., a graph) by specifying the source of each input to each process.

Let $c_{i,j}$ be output j from process i, c_k be environmental channel k, m_i be the number of outputs from process i, and let

$$\mathbf{c} = (\underbrace{c_1, \ldots, c_e}_{e \text{ inputs}}, \underbrace{c_{1,1}, \ldots, c_{1,m_1}}_{\text{process 1 outputs}}, \underbrace{c_{2,1}, \ldots, c_{2,m_2}}_{\text{process 2 outputs}}, \ldots, \underbrace{c_{r,1}, \ldots, c_{r,m_r}}_{\text{process } r \text{ outputs}})$$

be the vector of all channels in the system. The *behavior* of a Kahn network for input $(c_1, \ldots, c_e)$ is the least $\mathbf{c}$ satisfying

$$\begin{array}{rcl} (c_{1,1}, \ldots, c_{1,m_1}) & = & P_1\Big(c_{M(1,1)}, \ldots, c_{M(1,n_1)}\Big) \\ & \vdots & \\ (c_{r,1}, \ldots, c_{r,m_r}) & = & P_r\Big(c_{M(r,1)}, \ldots, c_{M(r,n_r)}\Big) \end{array} \tag{5.2}$$

where n_k is the number of inputs on the kth process and $c_{M(k,l)}$ is the channel feeding the lth input of the kth process: either an environment channel (i.e., $1 \le M(k,l) \le e$) or a specific output channel of a specific process (i.e., $M(k,l) = i,j$, where $k, i \in \{1 \ldots r\}$, $1 \le l \le n_k$, and $1 \le j \le m_i$).

Channels may "fork": each channel has a single source (either a process output or the environment) but may have multiple receivers. I.e., $M(k_1, l_1) = M(k_2, l_2)$ may hold for some $(k_1, l_1) \neq (k_2, l_2)$.

Kahn showed [76] that his networks are *deterministic*: there is exactly one least $\mathbf{c}$ that satisfies (5.2) for each tuple of input sequences provided the P_i are continuous.

5.1.2 Dataflow Actors

The Kahn formalism describes our networks; we follow Lee and Matsikoudis's formalism for actors [81] for describing processes. Actors react to input tokens according to firing rules; a rule is a tuple of empty or singleton token sequences. When an actor's input matches a rule, the actor consumes the matched tokens and produces a single token on certain outputs. Lee and Matsikoudis use sequences in their firing rules and reactions; we use only singletons because we target hardware.

Formally, an n-input, m-output *dataflow actor* is a pair (R, f) where $R \subset (\Sigma \cup \epsilon)^n$ are the *firing rules*, $f : R \to (\Sigma \cup \epsilon)^m$ is the *firing function*, and for any $\mathbf{a}, \mathbf{b} \in R$ with $\mathbf{a} \neq \mathbf{b}$, there is no $\mathbf{c}$ such that $\mathbf{a} \sqsubseteq \mathbf{c}$ and $\mathbf{b} \sqsubseteq \mathbf{c}$. This "no-common-prefix" constraint on R ensures the actor behaves as a continuous process: in particular, once an actor can fire on a given rule it cannot fire on another, even if additional tokens arrive.

The process for an actor simply fires repeatedly according to its firing rules. Formally, the Kahn process P for the dataflow actor (R, f) is

$$P(\mathbf{s}) = \begin{cases} f(\mathbf{r})P(\mathbf{t}) & \text{when } \exists \mathbf{r} \in R \text{ such that } \mathbf{s} = \mathbf{rt}; \\ \epsilon^m & \text{otherwise,} \end{cases} \tag{5.3}$$

where ϵ^m is the m-tuple of empty sequences and juxtaposition represents the pointwise concatenation of sequences. Lee and Matsikoudis [81] showed that P is a continuous function (and thus a Kahn process) provided the firing rules R obey the no-common-prefix rule described above. Note that (5.3) matches the usual recursive definition of the *map* function familiar to functional programmers (e.g., the Haskell definition we gave in Section 4.1.1).

Our networks allow channels with "initial tokens" to break deadlocks in loops. For example, the top multiplexers in Figure 5.1 would deadlock without the initial token provided. We model such tokens by allowing processes to emit initial tokens before entering their periodic firing behavior. A *dataflow actor with initial output* is a triple $(R, f, \mathbf{i})$ where R and f are as before and

$\mathbf{i} : (\Sigma^*)^m$ is the initial output from the actor. The Kahn process for such an actor is

$$P'(\mathbf{s}) = \mathbf{i}P(\mathbf{s}). \tag{5.4}$$

5.1.3 Unit-rate, Mux, and Demux Actors

We construct our networks from three stateless actors. The first, a *unit-rate* actor, waits for a single token on each of its inputs before producing a single output token on each of its outputs. Using this definition, the primitive, destruct, write, and read actors from Section 4.2 are all formally unit-rate.

For example, a two-input process that adds its two integer token inputs is a unit-rate actor. Again, let $\Sigma = \mathbb{Z}$. The actor (R, f) has

$$\begin{aligned} R &= \{(x,y) : x, y \in \mathbb{Z}\} \\ f\big((x,y)\big) &= (x+y), \end{aligned} \tag{5.5}$$

i.e., the actor can fire on any pair of integer tokens (R) and, given such a pair of tokens x and y, the actor produces a single token whose value is $x+y$ (f). It is easy to show that this R follows the no-common-prefix rule. Furthermore, an inductive argument shows that applying (5.3) to the R and f in (5.5) gives the P function in (5.1). In Figure 5.1, the equality (=), less-than (<), and subtractor (−) actors are each unit-rate.

Our second building block is the *mux* actor (Figure 5.1 uses four), which consumes a token from its control input to determine from which of its inputs to consume a further token that it then emits on its output channel. For example, a two-way mux actor that takes a 0 or a 1 on its select input has

$$\begin{aligned} R &= \{(0,x,\epsilon) : x \in \Sigma\} \cup \{(1,\epsilon,y) : y \in \Sigma\} \\ f\big((0,x,\epsilon)\big) &= (x) \\ f\big((1,\epsilon,y)\big) &= (y). \end{aligned} \tag{5.6}$$

Our third fundamental type of actor is the *demux* (Figure 5.1 uses four): each input token is routed to an output channel based on a select input token. For a two-output demux,

$$\begin{aligned} R &= \Big\{(x,y) : x \in \{0,1\}, y \in \Sigma\Big\} \\ f\Big((0,y)\Big) &= (y,\epsilon) \\ f\Big((1,y)\Big) &= (\epsilon,y). \end{aligned} \tag{5.7}$$

5.1.4 Nondeterministic Merge

A nondeterministic merge process produces its output sequence by interleaving two or more input sequences. That is, each token of each input sequence appears exactly once and in order in the output sequence, but successive tokens from an input sequence are not necessarily successive in the output. Practically, nondeterministic merge processes expose the timing of a dataflow system implementation by interleaving sequences according to *when* tokens arrive at their inputs, and thus are used to improve performance or break a deadlock by avoiding the need to wait. For example, in the dataflow networks produced by our compiler (Section 4.3), we use nondeterministic merge processes to share resources as shown in Figure 5.5; Section 5.2.4 explains this in detail.

By this definition, a nondeterministic merge process is not a Kahn process (it does not compute a function), so to introduce them into our framework we use a mathematical trick due to Broy [16]: each nondeterministic merge process is one of many different Kahn processes, one for each possible interleaving. The behavior of a system, then, is the set of behaviors produced by the system under every choice of interleaving.

Equivalently, each nondeterministic merge process can be thought of as just a (deterministic) mux actor whose control input is fed from a nondeterministic environment that directs the interleaving of the mux's data inputs. One way to implement a nondeterministic merge process is to have, say, an arbiter decide how to interleave input sequences, and treat the decisions of that arbiter as the control "input" from the environment. While we provide such merge processes, we also provide a merge process with an explicit control stream generated not by the environment, but by the merge process itself (i.e., the mergeChoice node of Section 4.2). Knowing how a nondeterministic merge interleaved streams is helpful for knowing how to later deinterleave them, which I discuss more in Section 5.2.4.

5.2 Hardware Dataflow Actors

In this section and the next, I describe how we implement our dataflow networks in hardware (recapping some of the ideas introduced in Section 4.4 for cohesion). Our compiler's code generation phase uses these implementations to perform a syntax-directed translation from an abstract dataflow specification (e.g., written in our DF IR) to a SystemVerilog circuit description.

Our circuits facilitate design space exploration because inserting or removing buffering does not affect what is computed, but removing buffering can introduce deadlock. Multiple actors may be chained together directly to avoid latency (with the danger of increasing the critical path) or buffers may be added to break frequency-robbing critical paths with pipelining.

To implement a dataflow network, each dataflow actor becomes a block of logic with handshaking communication ports, one for each input and output. Each channel in the network becomes a small communication network of wires potentially augmented with fork and buffering circuity (Section 5.3); each cycle must be buffered to prevent combinational cycles, and the user (or our compiler) may freely buffer channels to modify the frequency, area, and latency of the synthesized circuit.

In general, the datapath of each actor implements its firing function f and the flow control logic implements its firing rules R. Although we do not have formal proofs that show each circuit faithfully implements its specification, we argue for the correctness of our circuits in Section 5.4.

Paired with our hardware implementations of channels, the limited, core group of actors presented here are rich enough to implement any program written in our Floh IR (Section 4.1). Our framework could support additional actor types, as long as they follow the Kahn rule of blocking on exactly one input at a time to avoid nondeterministic behavior.

5.2.1 Communication and Combinational Cycles

Actors, buffers, and forks in our implementation communicate through unidirectional point-to-point links. We use the bundled-data protocol with handshaking inspired by Carloni's latency-insensitive design [19] and Carmona et al's elastic circuits [21] shown in Figure 4.11. This is a bundled data protocol in which the *valid* bit indicates a token is present on the *data* wires. The downstream block sends the *ready* signal upstream to indicate it is able to consume a token being proffered by the upstream block. A token is transferred from the upstream block to the downstream block in each cycle in which both *valid* and *ready* are asserted.

We chose this protocol because it is fast (able to transfer one token per clock cycle indefinitely), patient (both transmitter and receiver can wait indefinitely with no loss of data), and simple (to reduce overhead). We are unaware of other protocols that meet these criteria.

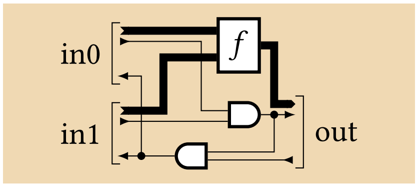
This seemingly simple protocol poses a potentially perilous problem: combinational cycles inadvertently induced by the *ready* signals, which flow backwards through the network. For example, it would be easy to produce a cycle if a *valid* signal depended instantaneously on a *ready* at an output port while a *ready* instantaneously depended on a *valid* at an input port.

We avoid combinational cycles by insisting each cycle in the dataflow network have at least one data and one control buffer (see Section 5.3) and by insisting no block has a combinational path from a *ready* to a *valid* signal. The data buffer rule eliminates combinational cycles in the data/*valid* network; the control buffer rule similarly breaks cycles in the *ready* network; and prohibiting combinational paths from *ready* to *valid* means no combinational cycle can include a signal that crosses between the two networks. Intuitively, these rules mean the flow control network can be scheduled statically: the *valid* network can be computed first (it is acyclic, with inputs from data buffers and the environment) followed by the *ready* network, which may take inputs from the *valid* network, control buffers, and the environment.

We provide a fragment of synthesizable RTL SystemVerilog for each block of logic implementing an actor. To represent, say, an 8-bit channel *c*, we use a nine-bit vector *c* for data (c[8:1]) and *valid* (c[0]), and a wire named *c_r* for *ready*. In our schematics, we label all wires of this port just with the name *c*. We also provide a DF specification (Section 5.5) for each block; given that DF specification, the compiler produces the logic block shown.

5.2.2 Unit-Rate Actors

Figure 5.2 shows how we implement single-output unit-rate actors such as a two-input adder. First, note that the datapath of this actor is just the combinational function f; our technique merely adds two AND gates for flow control to the *valid* and *ready* networks. The flow control logic waits for a valid token on both inputs before asserting the output is valid. The actor indicates it is willing to consume both its inputs when they are both valid and the downstream is also ready to consume the output token. Additional inputs can be added to the circuit of Figure 5.2 by widening the AND gate for the *valid*s; the *ready* logic remains the same but fans out more widely. An n-output actor can be implemented by n single-output actors with their inputs connected in parallel, although doing so requires forks feeding the input channels. For example, the destruct



```
out = op_add Int < in0 in1;
```

```
assign out =
  { f(in0[W:1], in1[W:1]),          // f(in0, in1)
    in0[0] && in1[0] };             // out valid

assign in0_r = out_r && out[0]; // in0, in1 ready =
assign in1_r = in0_r;             // out valid and ready
```

Figure 5.2: A unit-rate actor that computes a combinational function f of two W-bit inputs (bit 0 of each port carries the "valid" flag) once both arrive. Depicted is the circuit, a sample DF specification of the actor (assuming f is a primitive addition function), and the corresponding SystemVerilog.

actor from Section 4.2 is implemented as a fork distributing a token for a data constructor with n fields to n unit-rate actors; the ith unit-rate actor extracts the ith field from the input token.

5.2.3 Mux and Demux Actors

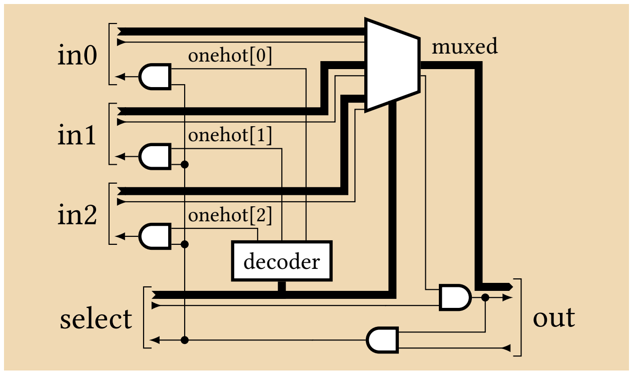
Mux and demux are not unit-rate actors because they use the value of a select token to determine on which input or output to communicate. A mux uses the value of a selection token to route a token on one selected input to the output. The select token and the token on the selected input must be valid to produce a valid output; input and select tokens are consumed when the output is ready.

The demux is complementary: it directs an input token to a single output depending on the value of a select token. Both the select and input tokens must be valid before a token is proffered on the selected output; that output must be ready before the two tokens are consumed.

Figure 5.3 shows a three-input mux that routes W-bit tokens. The datapath is a multiplexer that routes one of the inputs to the output depending on the value of the *select* input. A standard one-hot decoder also takes the *select* input and transforms it into a vector that indicates which input should be consumed. We implement the decoder (along with the multiplexer) in the "case" block of the SystemVerilog code: when the *select* value is 0, 1, or 2, the decoder sends 001, 010, or 100 (3'd1, 3'd2, 3'd4), respectively, to the *onehot* vector to select one of the inputs.

The output *valid* bit is the AND of the *valid* from the selected input and the *valid* bit of *select*. *Select* and the selected input are ready when the output is valid and ready.

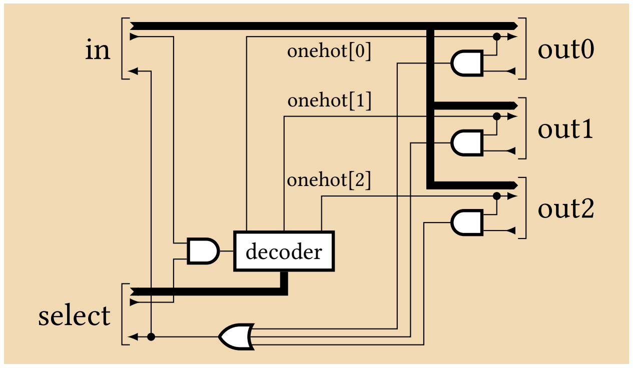
Figure 5.4 shows a three-output demultiplexer. The datapath is simply fanout that sends the input to all outputs. If both the *in* port and the *select* port have valid tokens, the one-hot decoder



```
out = mux Tri Int < select in0 in1 in2 ;
```

```
logic  [2:0]  onehot;  // One per input
logic  [W:0] muxed;
always_comb
 unique case ( select  [2:1])
  2′d0    :  {onehot,  muxed} = {3′d1,  in0 };
  2′d1    :  {onehot,  muxed} = {3′d2,  in1 };
  2′d2    :  {onehot,  muxed} = {3′d4,  in2 };
  default:  {onehot,  muxed} =
                  {3′bx,  {{W{1′bx}},  1′d0 }};
 endcase
assign out  =
   {  muxed[W:1], muxed[0] && select[0]  };
assign  select_r  = out[0]  && out_r;
assign {in2_r , in1_r , in0_r}  =
              select_r ?  onehot  :  3′d0;
```

Figure 5.3: A three-input W-bit multiplexer; select is 2 bits.



```
out0 out1 out2 = demux Tri Int < select in ;
```

```
logic  [2:0]  onehot;    // One per output
always_comb
 if  ( select  [0]  && in [0])  // Inputs valid?
   unique case ( select  [2:1])
     2′d0      :  onehot  =  3′d1;
     2′d1      :  onehot  =  3′d2;
     2′d2      :  onehot  =  3′d4;
     default  :  onehot  =  3′bx;
   endcase
 else  onehot  =  3′d0;
assign out0  =  {in [W:1],  onehot [0]};
assign out1  =  {in [W:1],  onehot [1]};
assign out2  =  {in [W:1],  onehot [2]};
assign  select_r =
   |  (onehot & {out2_r,  out1_r,  out0_r });
assign in_r  =  select_r ;
```

Figure 5.4: A demultiplexer with a two-bit select input and three W-bit outputs.

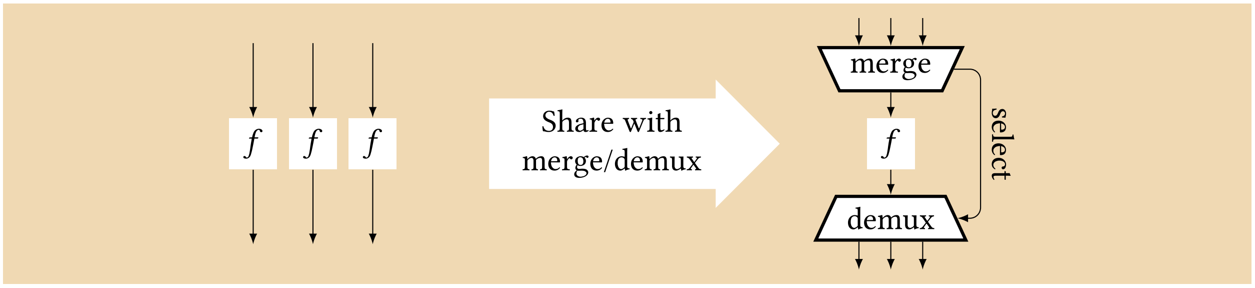


Figure 5.5: A merge used to share a unit-rate subnetwork.

uses the value of the *select* token to indicate which one of the output ports is given a valid token. Both *in* and *select* tokens are consumed if that selected output is ready.

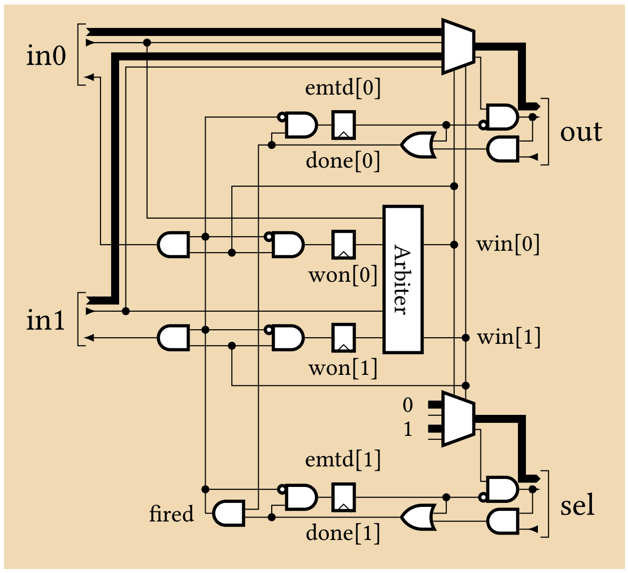
5.2.4 Merge Actors

Our implementation of the nondeterministic merge actor is novel: as mentioned in Section 5.1.4 it is essentially a mux actor whose select "input" is electrically an output. In our formalism, a merge actor is a mux with a nondeterministic select input; in our implementation, the *merge actor itself* generates the tokens on the select channel rather than receiving them.

Figure 5.5 illustrates how we use our merge actor to share a stateless block or subnetwork f that produces one output token per input; if f maintained state across tokens, sharing would change f's functionality. The merge nondeterministically chooses a token from one of its three inputs to route to the shared f and reports its choice in the form of a token on the select channel. When f produces its result, the demux routes the result to the output corresponding to the chosen input. As seen in Section 4.3, our compiler uses this merge to enable dynamic load balancing across the users (callers) of a shared resource (function) and to simplify the mapping of recursive design specifications onto our networks.

Figure 5.6 shows a two-input merge actor, which has two possible behaviors. If only one data input (*in0* or *in1*) provides a token, it is routed onto the *out* port. If both data inputs provide tokens in a given cycle, only one is routed onto *out* by the arbiter, whose implementation can vary (thus our modeling of the merge as nondeterministic). In both cases, the merge reports which input was selected via the *sel* port.

As I described in Section 5.1.4, Broy [16] models nondeterministic merge as a mux driven by a nondeterministic input that controls how the input streams are interleaved; our merge actor emits this selection sequence.



```
sel  out  =  mergeChoice Bool Int  <  in0  in1 ;
```

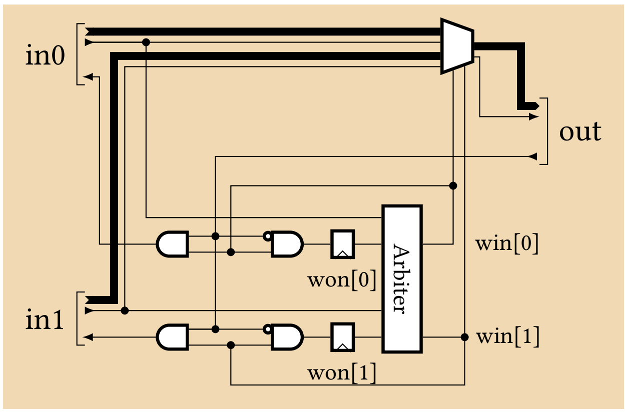
```
logic  [1:0]  won, win;  // 2 input arb. bits
assign win = | won ? won  :  // Decided
             in0 [0]  ? 2'd1 :  // in0 wins
             in1 [0]  ? 2'd2 :  // in1 wins
                        2'd0;   // No winner
initial  won = 2'd0;  // No winner initially
always_ff @(posedge clk)
     won <= fired  ?  2'd0  :  win;
logic  [1:0]  emtd, done;    // One per output
assign done = emtd |
        ({ sel [0], out [0]}  & { sel_r , out_r });
initial  emtd = 2'd0;  // Nothing yet emitted
always_ff @(posedge clk)
     emtd <= fired  ?  2'd0  :  done;
assign  fired  =  & done;
assign {in1_r , in0_r}  =  fired  ? win :  2'd0;
assign out  =  win[0]  && !emtd[0] ? in0  :
               win[1]  && !emtd[0] ? in1  :
                   { {W{1'bx}},  1'd0 }  ;
assign  sel  =  win[0]  && !emtd[1] ? 2'b01  :
                win[1]  && !emtd[1] ? 2'b11  :
                                      2'bx0  ;
```

Figure 5.6: A two-input nondeterministic merge that reports its arbitration decisions on the 1-bit output *sel*.

The circuit in Figure 5.6 is complex because it generates tokens on two output channels when it fires and needs to cope with only one channel being ready. The naïve approach of insisting both outputs be ready for the actor to fire leads to circuits with combinational cycles; our circuit avoids this by allowing firing across multiple cycles and thus needs state. The fork described in Section 5.3.2 is similar.

An n-input merge has $n+2$ state bits: n one-hot "*won*" bits that indicate which input won the arbitration and two *emtd* bits that indicate the *out* and *sel* outputs have already emitted tokens in this firing and should not emit any more. All of these bits reset to 0 between firings.

Our merge actor is built around an arbiter that, with a simple priority-encoding scheme, selects a valid input and declares it the winner through the one-hot *win* vector. This vector controls the multiplexers that route the winning input to *out* and its identity to *sel*. While our generated



```
out = merge Int < in0 in1;
```

```
logic [1:0] won, win; // 2 input arb. bits
assign win = | won ? won :  // Decided
             in0[0] ? 2'd1 : // in0 wins
             in1[0] ? 2'd2 : // in1 wins
                      2'd0;  // No winner
initial won = 2'd0; // No winner initially
always_ff @(posedge clk)
    won <= out_r ? 2'd0 : win;
assign {in1_r, in0_r} = out_r ? win : 2'd0;
assign out = win[0] ? in0 :
             win[1] ? in1 :
                 { {W{1'bx}}, 1'd0 } ;
```

Figure 5.7: A two-input nondeterministic merge that does not report its selection.

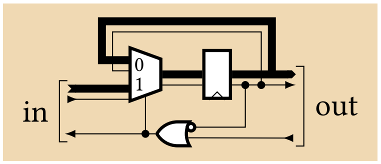
circuits are technically deterministic at the cycle level, changing the buffering on the channels may affect the behavior of the merge actor because it responds to the cycle-level timing behavior of input sequences. As such, it is effectively nondeterministic at the level of the Kahn network.

If both *out* and *sel* are ready and valid and the *emtd* bits are at 0, then both *done* signals become true, *fired* becomes true, the winning input's *ready* is asserted, all the *won* and *emtd* registers stay at 0, and the arbiter can handle another token in the next cycle.

When an output is not ready, it sets the *emtd* bit for the other output, suppressing that output's *valid* signal in the next cycle. Furthermore, because *fired* is not asserted, the *win* vector will be loaded into the *won* register. In the next cycle, since the *won* register is non-zero, the arbiter will maintain the identity of the winner and ignore any new input tokens.

In cycles after the initial arbitration, the *win* vector holds its value and maintains a valid token on the output that has not yet been consumed. When both outputs have finally been consumed (i.e., when each is either emitted or ready), *fired* will be asserted, the winning input token is finally consumed, and the merge actor's state resets to fire again in the next cycle.

While having a nondeterministic merge that reports which input token won the arbitration is useful for resource sharing (e.g., as in Figure 5.5) the select output is not always needed. We also provide a merge actor with no additional select output (shown in Figure 5.7), whose implementation is much simpler than that in Figure 5.6: the *emtd*, *fired*, and *done* signals are removed, *out* solely depends on which input was selected by the arbiter, and the winning input's ready is asserted when *out* is ready.



```
out = dbuf Int < in;
```

```
initial out = { {W{1'bx}}, 1'b0};   // Start empty
assign in_r = out_r || !out[0];     // Will we have space?

always_ff @(posedge clk)
  if (in_r) out <= in;              // If so, save the token
```

Figure 5.8: A data (pipeline) buffer, after Cao et al. [18]. This breaks combinational paths in the data/valid network.

5.3 Channels in Hardware

In our specifications, a channel is an abstract mechanism that conveys a sequence of tokens generated by a process or supplied by the environment to one or more processes. Our technique allows such channels to be implemented in a variety of ways, providing various speed/area tradeoffs.

A point-to-point channel can be implemented with a direct connection, as shown in Figure 4.11. Such a link is the fastest and consumes the fewest resources but may produce a long combinational path that limits clock frequency. It also couples the two process' firings.

Adding a buffer to a point-to-point link decouples the firing of the upstream and downstream actors. Such buffering is mandatory on loops in the network and on channels with initial tokens (see Section 5.1.2). Buffering can also improve performance by breaking long combinational paths in the generated circuit, effectively pipelining them to improve throughput and clock frequency.

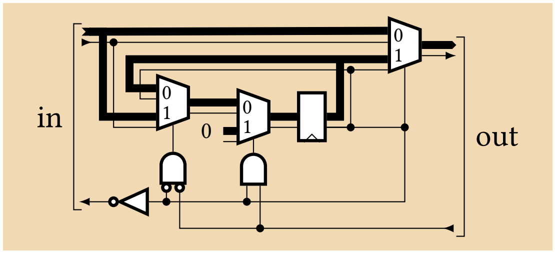
We provide *fork* blocks for implementing channels with fanout. The datapath of a fork is trivial (simply wires that fan out); the flow control logic (i.e., for *valid* and *ready*) turns out to be fairly complicated to avoid a combinational path from *ready* to *valid*.

Choosing an optimal channel implementation is outside the scope of this dissertation. However, we can correctly implement any channel in our specification as a single-source, feed-forward network comprised of forks, buffers, and point-to-point connections.

Below, I describe how we implement buffers and forks.

5.3.1 Data and Control Buffers

We provide two buffer types, based on the designs of Cao et al. [18]: a data buffer breaks a combinational path (or cycle) on the data/valid network; a control buffer does so on the ready network. Each type of buffer can hold a single data token, but their implementations differ.



```
out = rbuf Int < in;
```

```
initial buffer = {{W{1'bx}}, 1b' 0}; // Empty
always_ff @(posedge clk)
  if (out_r && buffer[0])               // Will send?
      buffer <= { {W{1'bx}}, 1'd0}; // then clear
  else if (! out_r && !buffer [0])      // Must hold?
      buffer <= in;                     // then save
assign out = buffer[0] ? buffer : in;
assign in_r = ! buffer [0];
```

Figure 5.9: A control buffer, after Cao et al. [18]. This breaks combinational paths in the (upstream) *ready* network.

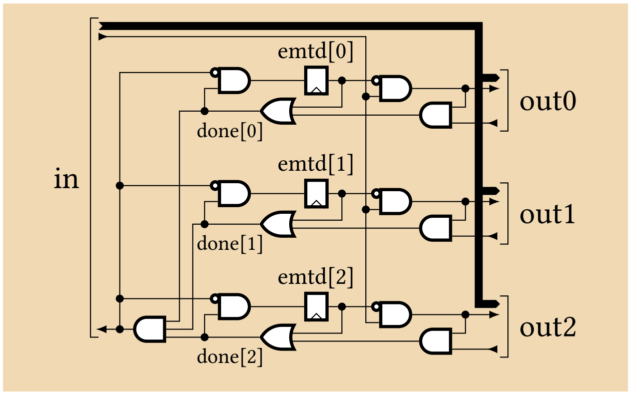
A data buffer (Figure 5.8) is a traditional pipeline register: it breaks the combinational path on data/*valid* signals, stores a single data token, and adds a clock cycle of latency. The downstream *ready* signal acts like a latch enable when the buffer holds a valid token; an upstream token is always latched when the buffer is empty. Note that a data buffer's *ready* path is combinational.

The control buffer in Figure 5.9 performs the more challenging task of breaking the combinational path on the *ready* network. By design, the upstream *ready* signal (*in_r*) depends only on a flip-flop output—the *valid* bit of a "spill buffer." Complementary to a data buffer, a control buffer induces a cycle of latency on the *ready* network, but not necessarily any on the data/*valid* network.

The control buffer intercepts and stores any valid token that the downstream cannot accept. The buffer in Figure 5.9 starts empty: its *valid* bit is false, any valid token flows directly from *in* to *out*, and *in_r* is asserted. If *out_r* remains true, the buffer remains empty (holds its previous contents); however, if *out_r* goes false, the downstream will not consume any valid token, so instead any valid token on *in* is "spilled" into the buffer.

When the buffer holds a token, no token is accepted from upstream because *in_r* is false, the buffered token is proffered downstream, and *out_r* controls whether the token will continue to be held or advanced in the next cycle.

Connecting control and data buffers back-to-back in either order breaks any combinational path that would pass through them, allowing them to be chained arbitrarily without reducing peak clock frequency. Back-to-back, these buffers behave like the latency-insensitive relay stations of Li et al. [83].



```
out0 out1 out2 = fork Int < in;
```

```
logic  [2:0]  emtd, done;  // Per output
initial  emtd = 3'd0;      // Start clear
assign out0  =  {in[W:1],  in[0]&& !emtd[0]};
assign out1  =  {in[W:1],  in[0]&& !emtd[1]};
assign out2  =  {in[W:1],  in[0]&& !emtd[2]};
assign done =
   emtd |  ({out2[0],  out1[0],  out0[0]}  &
              {out2_r,   out1_r,   out0_r});
assign in_r  =  & done;

always_ff @(posedge clk)
   emtd <= in_r  ?  3'd0  :  done;
```

Figure 5.10: A three-way fork. An output port's *emtd* flip-flop is set when the input token has been consumed on that port. All are reset after a token is consumed on every port.

5.3.2 Forks

To implement channels with fanout, we use "fork" circuits that handle the flow control logic (i.e., *valid* and *ready* signals) without introducing combinational cycles when blocks are connected.

The obvious way to implement fork—a block that waits for all its downstream actors to be ready before firing—would introduce combinational cycles when composed because such a policy requires a combinational path from *ready* to *valid*. A block that considered a downstream block's *ready* inputs to control whether its outputs were valid would not be compositional.

Our implementation of fork avoids combinational cycles by introducing a limited amount of state. By using one flip-flop per output, a valid token may pass through a fork and be consumed downstream *before* it is consumed upstream. The amount of state in a fork block depends only on how much it fans out and not on the width of the data tokens (unlike control and data buffers).

Figure 5.10 illustrates our solution, which uses one flip-flop per output in a vector called *emtd* (for "emitted"). Each *emtd* bit indicates whether the downstream consumer previously consumed the current token. If an output's *emtd* bit is set, the fork suppresses that output's *valid* signal to avoid sending a second copy of the token to the consumer.

Initially all *emtd* bits are zero. If there is no input token, the state is unchanged. If an input token arrives it is proffered on all downstream ports. If all consumers are ready, *done* is all ones, the upstream *ready* is asserted, and the *emtd* flip-flops remain cleared.

If any consumer is not ready, the input token is not consumed (the upstream *ready* is not asserted) and the ready consumers' *emtd* bits are set to one to prevent any further tokens being proffered on the outputs before the current token is consumed.

When some *emtd* bits are set, the upstream *ready* is not asserted, the input token is held, and an output token is proffered on output channels whose *emtd* bits are zero. Each output's *done* bit is asserted if the proffered token was consumed in this or a previous cycle. Once all *done* bits are true, the *emtd* bits are reset, the upstream token is consumed, and the process repeats.

5.4 The Argument for Correctness

In this section, we argue that our circuits faithfully implement the specifications in Section 5.1 in that anything the hardware implementation can do is permitted by the specification. However, the reverse is not true: the hardware may deadlock because of (finite) buffer overflow where the specification would proceed. Specifically, the sequence of tokens that can be observed passing through any channel in a hardware implementation is a prefix of (but often equal to) the sequence of tokens that the Kahn fixed point semantics implies would pass through the channel.

Pingali and Arvind [102] show this in the Kahn formalism (our argument is for the hardware realization of this formalism). To impose demand-driven scheduling on a Kahn network, they introduce *demand streams* that run opposite each communication channel. Such streams model buffer capacity: each process is forced to consume (and thus potentially wait for) an incoming demand token before it sends a token on the corresponding output channel. Similarly, after a process consumes a normal input token, it immediately emits a corresponding demand token. Pingali and Arvind show that a network augmented with such streams produces on each channel a prefix of the stream of tokens that would be produced in the original network. Geilen and Basten [56] present an alternative proof.

Our argument for circuit correctness relies on our hardware behaving according to the formal notion of an actor firing. According to (5.3), when a process finds tokens on its input sequences that match a firing rule $\mathbf{r}$, it produces tokens on its output sequences according to its firing function f, and then advances past ("consumes") the tokens identified in the firing rule by recursing on the tuple of sequences $\mathbf{t}$, which skips the tokens in the firing rule $\mathbf{r}$.

We use the *valid* signal to indicate the "next" token in sequence; a block indicates it is willing to consume a token when it asserts the *ready* signal on the port. An upstream block must continue to provide the same valid token until the downstream block is ready.

We argue that each block maintains the following inductive invariant between clock cycles: on each port, if *valid* is true, the data wires carry the token value that appears "next" in the sequence given by the underlying Kahn network behavior; the "previous" token value was consumed during the last cycle in which both *valid* and *ready* were true (or no such token existed because the circuit was reset). Furthermore, once *valid* has been asserted, it must stay asserted until the next cycle in which *ready* is asserted. Thus, *valid* indicates the correct next token value is present; when it is accompanied by *ready* the token has been consumed. A corollary is that observing the data values on a port in cycles where both *valid* and *ready* were true gives a prefix of the sequence on that port. For the sequence of data values $\ldots, c_{t-1}, c_t, c_{t+1}, \ldots$, we might observe clock cycles with the following data (here, "X" represents garbage data):

data:	$\cdots$	c_{t-1}	c_{t-1}	c_{t-1}	X	c_t	c_t	c_{t+1}	$\cdots$
valid:	$\cdots$	1	1	1	0	1	1	1	$\cdots$
ready:	$\cdots$	0	0	1	X	0	1	0	$\cdots$

The highlighted columns are token-transfer cycles. The environmental inputs must also follow this protocol and hold valid data until it is ready to be consumed.

We also assume that when the circuit is reset, any and all initial tokens on the channels required by the specification are residing in the appropriate data or control buffers.

The **unit-rate actor** (Figure 5.2) preserves the invariant. If all its inputs are valid, each carries the value of the next token on their respective sequences, the function block computes the next token in sequence on the output, which is made valid. If, additionally, the output is ready, the inputs are also made ready, indicating the input tokens have been consumed.

For the **multiplexer** (Figure 5.3), if the *select* input is valid, it must carry the next token in sequence. The value of the *select* token routes the data/*valid* signals from the appropriate input port to the *muxed* signal internally. If *muxed* is valid, the output carries the proper value (next token in sequence) and is set valid. If, additionally, the output is ready, both the *select* input and the selected input are made ready and no others.

For the **demultiplexer** (Figure 5.4), only if both the input and *select* inputs have a valid token is the decoder activated and the appropriate output made valid. If, furthermore, that output is also ready, only then are both the input and *select* inputs marked ready.

The **nondeterministic merge** block (Figure 5.6) must ensure that once it decides what the next tokens on its output should be, these values persist. When the *emtd* and *won* registers are all zero, the arbiter decides which one, if any, of the valid inputs wins the arbitration. This causes correct, valid tokens to appear on both the output and select ports. If both ports are ready, *fired* is asserted, the winning input is also marked ready, and the *emtd* and *won* registers stay zero. If only one output port is ready, *fired* will remain low, the ready output port will set its *emtd* bit and the *won* register will record the arbitration winner. In future cycles the token, if any, from the winning input port will continue to be routed to the output port and the corresponding value will be sent on select, but the *emtd* register will suppress the *valid* signal on the already-ready port. Note that the environment must sustain the valid token on the winning input port. If *ready* is asserted on the non-emitted port, *fired* will be asserted, the winning input will be made ready, the registers cleared, and the process repeats.

Only buffers can hold tokens. Consider when the **data buffer** (Figure 5.8) is empty. The output is invalid and the *ready* output is asserted. If a valid token is proffered, the token will be stored in the buffer at the end of the cycle, consistent with the invariant. When the data buffer is full, *valid* is asserted. If the downstream *ready* is false, the upstream *ready* is false and the register will hold the token. When the downstream *ready* is true, the upstream *ready* will be asserted and the token in the buffer will be overwritten. If a valid token was proffered, the buffer will store it.

If the **control buffer** (Figure 5.9) is empty, the input token/*valid* signal is simply copied to the output and the upstream *ready* is asserted. If the downstream does not assert *ready*, the valid upstream token, if any, will be stored in the buffer for the next cycle. If the buffer is full, the upstream *ready* is false and the valid token is proffered on the output. If the downstream *ready* is true, the buffer will be emptied in the next cycle.

The **fork** block (Figure 5.10) relies on the upstream block sustaining a valid token until it is consumed. When the *emtd* register is zero, a valid token on the input becomes a valid token on each of the outputs. Any output that is also ready asserts its respective *done* signal. If all the *done* signals are set, the upstream *ready* is asserted and the *emtd* registers are all reset. Otherwise, each ready output sets its *emtd* bit in the next cycle. These bits suppress the *valid* signal on each of the outputs that had already asserted *ready* with the current input token. Each *done* bit becomes true if its *emtd* bit is true or if a valid token has been consumed by a *ready* on the output. When all the *done* bits are true, the block consumes the current input token and resets all the *emtd* bits.

5.5 Our Dataflow IR: DF

To bridge the gap between our abstract dataflow networks and their hardware implementations, we developed an additional IR for our compiler: a typed dataflow assembly language called DF. Similar to a software assembly language DF admits a one-to-one translation scheme: each line instantiates one dataflow actor by generating the SystemVerilog code presented earlier (modifying the number of input and outputs, and setting their bitwidths appropriately). A DF program describes a dataflow network, which our compiler type-checks before generating the corresponding SystemVerilog following our procedure (Section 5.2).

Compared to coding dataflow networks directly in SystemVerilog, DF provides the usual advantages of a higher-level language: it makes good designs easier to express and prohibits many bad designs. It is much more succinct than the equivalent SystemVerilog would be, making it easier to write and read. It provides higher-level types: signed and unsigned binary vectors plus algebraic data types provide both abstraction to more succinctly express ideas and opportunities to catch design errors early. The network is checked for types and structure: each actor specifies constraints on the types of its input and output ports, each channel is required to have exactly one writer and one reader, and the types of the ports on a channel must match.

We use DF as a textual IR to both simplify debugging and give a more modular compilation flow than a pure binary representation would. Although we would have preferred to express our actors as a SystemVerilog library, SystemVerilog's polymorphism does not permit modules with a varying number of inputs or outputs (e.g., merge and fork). Furthermore, although the 2012 SystemVerilog standard includes tagged unions for implementing algebraic data types, none of the SystemVerilog tools we use (e.g., Quartus and Verilator) supports them.

Figure 5.11 shows a complete DF program that expresses a fragment of the GCD example from Figure 5.1. It starts with channel type definitions (DF has no built-in types) and type definitions for the various actors, then instantiates the actors (on the right-hand side of Figure 5.11).

Figure 5.12 shows the syntax for DF. A program consists of channel type definitions, actor type definitions, and actor instances. I describe each below.

5.5.1 Channel Type Definitions

Each channel in a DF specification has a type indicating the type of tokens it conveys. A channel type must be defined with the *data* keyword before it can be used in a DF program, and its name

```
// Type definitions
data Int  signed 32  ;
data Bool = F | T;
// Actor type definitions
source   a     :       > a                ;
sink     a     : a     >                  ;
fork     a     : a     > a+               ;
op_eq    a     : a  a  > Bool             ;
demux    a  b  : a  b  > b∧(variants  a)  ;
mux      a  b  : a  b∧(variants  a)  > b  ;
 initbuf  a  (b  :  a)  :  a          > a  ;
rbuf     a     : a     > a                ;
```

```
// Actor instances
aIn         = source  Int        <;
bIn         = source  Int        <;
aFB         = source  Int        <;
bFB         = source  Int        <;
e1b         = rbuf  Bool         < e1;
c           =  initbuf  Bool T   < e1b;
c1  c2      = fork  Bool         < c;
a           = mux Bool Int       < c1  aIn  aFB;
a1  a2      = fork  Int          < a;
b           = mux Bool Int       < c2  bIn  bFB;
b1  b2      = fork  Int          < b;
e           = op_eq Int          < a1  b1;
e1  e2  e3  = fork  Bool         < e;
 result  aF = demux Bool Int     < e2  a2;
_        bF = demux Bool Int     < e3  b2;
            = sink  Int          <  result ;
            = sink  Int          < aF;
            = sink  Int          < bF;
```

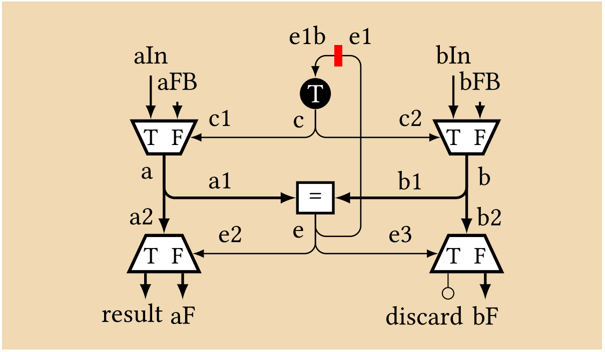


Figure 5.11: A DF program describing the topology and channel types for a portion of GCD from Figure 5.1.

must begin with an uppercase letter. Our compiler implements all types as fixed-width bit vectors in SystemVerilog.

DF's channel types are either primitive integers or algebraic data types. An integer type definition names either a binary (**unsigned**) or two's complement (**signed**) fixed-width bit vector:

```
data Int  signed 32;         // 32-bit signed (two's complement) integer
data Char unsigned 8;        // 8-bit unsigned integer
data Uint14  unsigned 14;  // 14-bit unsigned integer
```

DF's algebraic types mirror those from our Core IR (Section 3.1). An algebraic type in DF consists of one or more variants; each variant has a name ("tag") starting with an uppercase letter and a payload of zero or more data fields of specific types:

program	::=	[*typedef* $\mid$ *actordef* $\mid$ *instance*]*	
typedef	::=	**data** *type-id* [**signed** $\mid$ **unsigned**] *int-lit* **;**	Primitive sized integer type
	$\mid$	**data** *type-id* = *tag-id type-id** [$\boldsymbol{\mid}$ *tag-id type-id**]* **;**	Algebraic type definition
actordef	::=	*actor-id* [*var-id* $\mid$ **(** *var-id* **:** *type* **)**]* **:** *type** > *type** **;**	Actor type definition
instance	::=	*channel-id** = *actor-id* [*type-id* $\mid$ *int-lit*]* < *channel-id** **;**	Actor instance
type	::=	*type-id*	Named type
	$\mid$	*tag-id*	Tag name (variant)
	$\mid$	*var-id*	Type variable/function name
	$\mid$	*type* **+**	One or more
	$\mid$	*type* $\wedge$ *type*	Prescribed number of
	$\mid$	*type type*	Type function application
	$\mid$	**(** *type* **)**	Grouping

Type (*type-id*) and tag (*tag-id*) names start with an uppercase letter.
Actor (*actor-id*), channel (*channel-id*), and type variable (*var-id*) names start with a lowercase letter or an underscore (_).
Integer literals (*int-lit*) may be negative.

Figure 5.12: Syntax of DF. Brackets [], bar |, and asterisk * are meta-symbols denoting grouping, choice, and zero-or-more. Bold characters, including parentheses **()**, bar **|**, and caret ∧, are tokens.

```
data Bool = F | T;           // A succinct Boolean type
data Pair  = Pair  Int  Int ;  // A pair of integers
```

Whereas an integer token carries a numeric value, a token of an algebraic type carries one variant and its associated payload data, which may be other algebraic types and integers. Our compiler encodes algebraic types as a tagged union: a tag field indicates the variant followed by enough bits to hold the largest payload.

As in Floh (Section 4.1), hierarchy is allowed in algebraic types, but not recursion; recursive types may be expressed with pointers encoded as integers:

```
data Tree  = Branch Uint14  Uint14  // Binary tree with 14-bit pointers.
           |  Intleaf   Int          // Leaves may be 32-bit integers
           |  Boolleaf  Bool         // or Booleans
```

5.5.2 Actor Instances and Type Definitions

An actor instance adds a new actor to the network and consists of a list of output channels, an actor name, a list of zero or more arguments to be passed to that actor's type definition, and a list of input channels:

output-channel … = *actor-name type/literal-argument* … < *input-channel* … ;

Unlike other network specification languages such as Verilog, DF does not need to name each actor instance; the names of an instance's channels uniquely identify that instance since connecting two actors to exactly the same inputs and outputs is nonsensical (and illegal).

An actor type definition specifies the type and number of ports for an actor by providing a unique actor name, a list of parameters, and lists of input and output port types:

actor-name parameter … : *input-port-type* … > *output-port-type* … ;

Our actors support parametric polymorphism by taking zero or more named parameters, which are typically used to specify the type and number of ports for that actor. A parameter with just a name (e.g., "a") is a type variable that ranges over channel types. A parameter may also be constrained to constants of a particular channel type (e.g., "a : Int") or variant tags of an algebraic type (e.g., "a : tag Bool"). Actor instances provide concrete type and constant arguments to resolve any parametricity in a definition.

We use a regular-expression-like syntax inspired by Hosoya et al.'s type system [68, 69] to specify the number and type of an actor's input and output ports. A type name (e.g., "Bool") denotes a single port of that type, while a type variable (e.g., "a") represents a single port of polymorphic type. The $\wedge$ and + operators let us assign types to multiple ports at once: an expression of the form $t \wedge n$ means n ports of type t (where n is an integer expression), and the postfix + operator denotes one or more ports of a given type (e.g., "Bool+"). To avoid ambiguity, the + operator may only be used once in the input channels and once in the output channels.

Below are some actor type definition examples. A *source* is an input connection from the environment that supplies tokens of a given type. It adds input ports to the SystemVerilog module generated for the network. An *op_eq* is a two-input polymorphic comparator (Section 5.2.2) that emits True when its inputs are equal and False otherwise. An *op_add* takes two objects of the same type, sums their bits, and returns an object of the same type. A *buf* is a data/control buffer pair that can buffer any type of token (Section 5.3.1). An *initbuf* is a *buf* that starts with an initial token whose value is the second parameter. A *fork* is a polymorphic single-input actor that can have one or more outputs (Section 5.3.2).

```
source   a          :      > a;     // Actor with a single output of any type
op_eq    a          :  a a > Bool;  // Two inputs of the same type and a Bool output
op_add   a          :  a a > a;     // Two inputs and an output, all the same type
buf      a          :  a   > a;     // Single input and single output ports are of the same type
 initbuf a  (b : a) :  a   > a;     // Second parameter is a constant of type a
fork     a          :  a   > a+;    // One input; one or more outputs, all the same type
```

We also provide three built-in type functions that actor definitions may use to help define channel types. The *variants* function returns the number of variants of a channel type. This is used with the ∧ operator to specify a number of ports, e.g., for a multiplexer, which uses the variant sent to it on a *select* input to choose an input. Below, because the multiplexer's *select* input takes the three-variant type *Tri*, the mux has three inputs.

```
data Int  signed 32;
data Tri  = One |  Two |  Three;
mux a b  :  a  b∧(variants  a)  > b;
output  = mux Tri  Int  <  select  input1  input2  input3;
```

The *tag* and *variant_fields* functions work in tandem to let us define a *variant* actor that assembles the payload fields of an algebraic type object to produce an instance of that type (corresponding to a primitive data constructor node from Section 4.2). Specifically, the *tag* function specifies that a parameter should be a variant of a given type, and passing that parameter to the *variant_fields* function yields a list of channel types corresponding to that variant's type fields. For example, the following fragment constructs both variants of an *OptPair* type:

```
data Int  signed 32;
data OptPair = Pair  Int  Int  |  Null;
source  a  :  > a;
 variant  a  (b  :  tag  a)  :  (  variant_fields   b)  > a;
 i1  = source  Int  <  ;
 i2  = source  Int  <  ;
p  =  variant  OptPair  Pair  <  i1  i2;  // Pair is a variant of OptPair, payload of two Ints
n  =  variant  OptPair  Null  <  ;        // Null is a variant of OptPair, no payload
```

The *destruct* actor works in reverse, extracting the payload from the given variant:

```
destruct a (b: tag a) : a > ( variant_fields b);
o1 o2 = destruct OptPair Pair < p; // o1 and o2 will be Ints
```

5.5.3 Checking DF Specifications

The back-end of our compiler takes in a DF program (produced by the Floh-to-dataflow translation of Section 4.3) and performs two main tasks: it verifies that the DF program is correctly typed and translates it into SystemVerilog. Section 5.2 and Section 5.3 describe the translation to SystemVerilog; here I discuss the rules our compiler uses to check types.

The first set of checks concerns name usage. A DF program has four global namespaces: type names, tag names, actor names, and channel names. Type and tag names must start with an uppercase letter, while actor and channel names start with a lowercase letter or an underscore. Each type, tag, and actor defined must have a globally unique name within its respective namespace (e.g., we can define a type and a tag of the same name).

To enforce the point-to-point nature of communication channels, the DF compiler requires that each named channel appear exactly twice in a network: once as an output, and once as an input. It may be possible to relax this requirement and allow a channel to be connected as an input to multiple actors; we explicitly specify fork actors instead.

Once all names have been checked, the compiler first validates each data type definition individually by checking that each variant's payload only carries defined types, followed by an overall check that no type is recursively defined.

To validate an actor type definition, the compiler ensures that both its parameters and channel types follow various rules. Each parameter name must be unique, but only for the actor being defined. Constraints on parameter types can only refer to earlier parameters for that actor (e.g., "a : b" is only valid if "b" was defined earlier in the parameter list), and such constraints must resolve to either a channel type or the tags of a type.

The channel types for an actor type definition must be consistent. Each must resolve to either a named type, a type variable, or the + or $\wedge$ operators applied to one of these. At most one + operator may appear among the inputs and at most one may appear in the outputs. The right argument of each $\wedge$ operator must resolve to an integer. The *variants* and *tag* functions may only be applied to a type, while the *variant_fields* may only be applied to a tag.

The compiler checks each actor instance in two steps: first, actual arguments are bound to each of the actor's parameters; second, types are assigned to each channel (both inputs and outputs). Binding arguments to parameters is done in the usual way: the *n*th argument is bound to the *n*th parameter provided its type is consistent, i.e., a parameter that is a type variable only accepts a concrete type (name), a parameter constrained to a channel type must be passed a literal of that type, and a parameter constrained to be a tag of a particular channel type must be given such a tag.

If binding arguments to parameters succeeds, the second step is a matching process that assigns types to input and output channels. Parameters are used to resolve the type (and number, in the case of the $\wedge$ operator) of expressions without a + operator, but the presence of an expression with a + operator complicates things. With a + operator, our procedure assigns the channels before the + to the channels at the beginning of the input or output list, the channels after the + to those at the end of the list, and the remainder to the range denoted by the +. Multiple + operators in a list of channels would introduce ambiguity, so we prohibit them. Multiple $\wedge$ operators, however, are allowed because they prescribe a specific number of channels when resolved.

Finally, we verify that the type assigned to each channel when it appears as an output is the same as the type assigned to the channel when it appears as an input.

5.6 Experimental Evaluation

A designer can use our blocks to implement a dataflow network and adjust buffering to affect area and performance without changing functionality, although insufficient buffering may introduce deadlock. To verify this, I created dataflow networks (both manually with DF and automatically with Haskell programs fed into our compiler), buffered them both randomly and manually, simulated the resulting circuits to check that each functioned correctly, and calculated the circuits' highest clock rate when synthesized on an FPGA.

I both simulated the function of each circuit with Verilator 3.874 to verify that the circuits operated correctly and were free of combinational cycles and synthesized each circuit using Intel's Quartus 15.0, targeting a modest-performance Cyclone V 5CGXFC7C7F23C8 FPGA with 56480 ALMs (Adaptive Logic Modules), to estimate the maximum operating frequency and resource usage of the design.

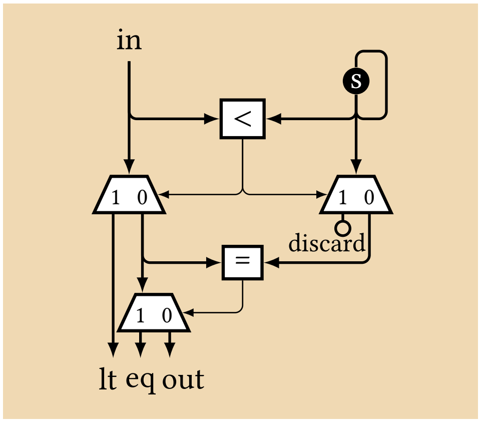


Figure 5.13: The splitter component of a Conveyor. This network partitions tokens arriving on the input stream *in* by comparing each input value against a "split" value (the initial token *s*). Each input token is sent out on one of three ports depending on whether it is less than (*lt*), equal to (*eq*), or greater than (*out*) the split value.

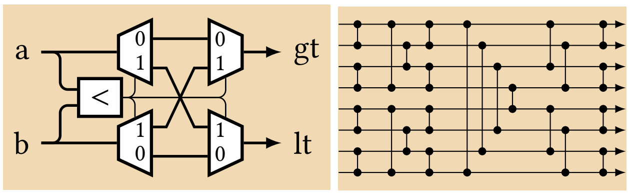


Figure 5.14: Bitonic sorting: the network on the left routes the larger of two input tokens to the top output and the smaller to the bottom. These are the vertical lines in the eight-element bitonic sorting network on the right (after Cormen et al. [33]).

5.6.1 Experimental Networks

I experimented with the five applications described below. I manually coded the first three networks in our dataflow language; the last two networks were synthesized from small Haskell programs using the lowering passes and translation described in Section 3.3 and Section 4.3.

GCD This is Figure 5.1's network made to compute gcd(100,2) with 32-bit integers.

Conveyor This network performs range partitioning [137]. The design chains n splitters (Figure 5.13) to partition an input stream into $2n+1$ output streams (e.g., 10 splitters yield a 21-way Conveyor design). Each splitter routes tokens to its outputs depending on how each token compares to the splitter's value. I fed the Conveyor an input sequence of 32-bit numbers $(1,\ldots,10000)$ and set the ith splitter value to $10000/(i+1)$. To limit I/O pins, I merged the Conveyor's outputs with a chain of merge actors to produce a single (nonsensical) 32-bit output stream.

Bitonic Sorting Network (BSN) This sorts a fixed number of values with two-input comparators that operate in parallel. Figure 5.14 shows the dataflow network for a comparator and an eight-input BSN. Each comparator takes in a pair of tokens and routes the smaller to its lower output and the larger to its upper, either by passing the tokens straight through or swapping

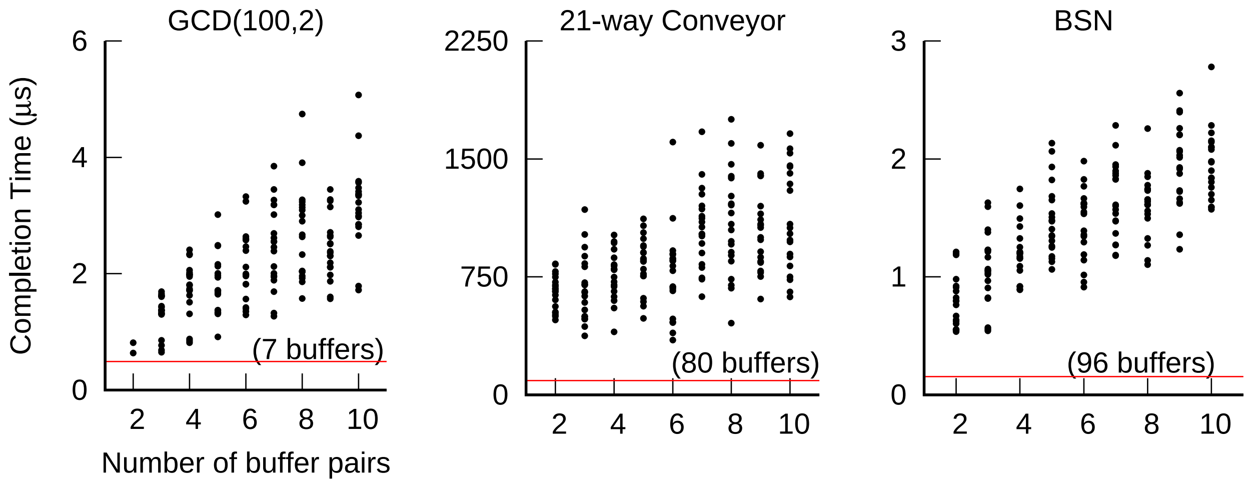


Figure 5.15: Completion times under random buffer placement. Horizontal lines labeled with a buffer count indicate the completion time of the best manual design.

them. I executed this network on ten sets of eight 8-bit values and merged the sorted numbers with an 8-input adder (again, to limit I/O pins).

Mergesort and Treesort These are recursive sorting algorithms that use memories (on-chip BRAMs) to store their data structures (lists and trees) and the continuation objects that implement their recursion. I fed each network a list of 20 32-bit integers. I limited the input size because the circuits generate a large number of intermediate structures, and our synthesizable on-chip memories are not currently garbage collected.

5.6.2 Random Buffer Allocation

I first employed random buffer allocation, not because it produces efficient designs, but to show that buffers may be added arbitrarily without affecting functionality thereby facilitating design space exploration. Given unbuffered GCD, BSN, and 21-way Conveyor networks, I assigned data buffers to store the initial tokens from their specifications (GCD and Conveyor) and placed a control buffer on the same channels to break a combinational cycle.

I next assigned between two and ten control/data buffer pairs on randomly chosen channels, discarding any implementation that produced premature deadlock or left a combinational cycle. All remaining implementations computed the same result. Figure 5.15 shows the completion time of each of these implementations in microseconds: cycles divided by maximum frequency (MHz).

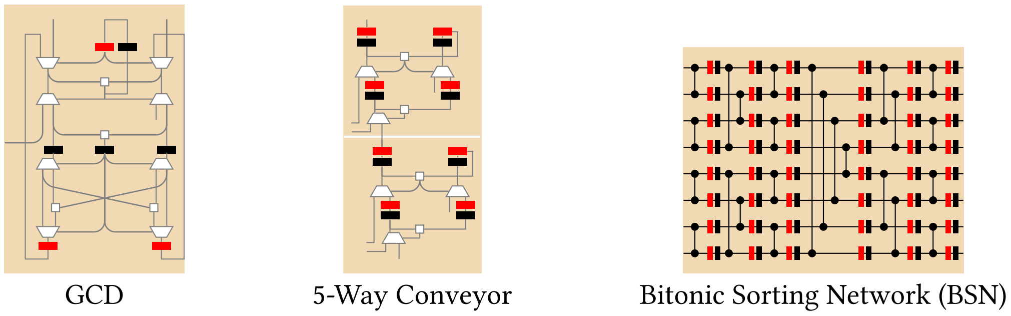


Figure 5.16: Buffering our networks. Red bars represent control buffers; black for data. Each cycle requires both buffers.

5.6.3 Manual Buffer Allocation

In Figure 5.15, I also plot the completion time for the best design I could devise manually (the horizontal lines). Naturally, these are much better than what random buffering produced.

Figure 5.16 depicts my best manual buffer placements. Each black bar represents a data buffer; red represents a control buffer. Due to its redundant nature, I only depict two representative splitters (of ten) in the Conveyor.

I manually implemented 6-stage and 20-stage BSN and Conveyor networks, respectively. Each stage adds only a single additional cycle to the total execution (since no stalls occur) while substantially increasing the clock frequency (103 MHz for BSN; 109 MHz for Conveyor) to reduce completion time.

I found separating data and control buffers improved performance for the GCD example. I initially placed the two buffer types together at the bottom of the network, but found splitting and moving them as shown in Figure 5.16 improved the frequency from 79 MHz to 109 MHz.

5.6.4 Pipelining the Conveyor

Over Conveyors with 4 to 64 splitters, I experimented with the three pipelining strategies shown in Figure 5.17: two splitters per stage, one splitter per stage, and two stages per splitter.

The graphs show that the two stages/splitter design provides the best overall performance on our workload, followed closely by the one stage/splitter. For one stage/two splitters, the barely noticeable reduction in the number of cycles required is swamped by the 40% reduction in maximum clock frequency.

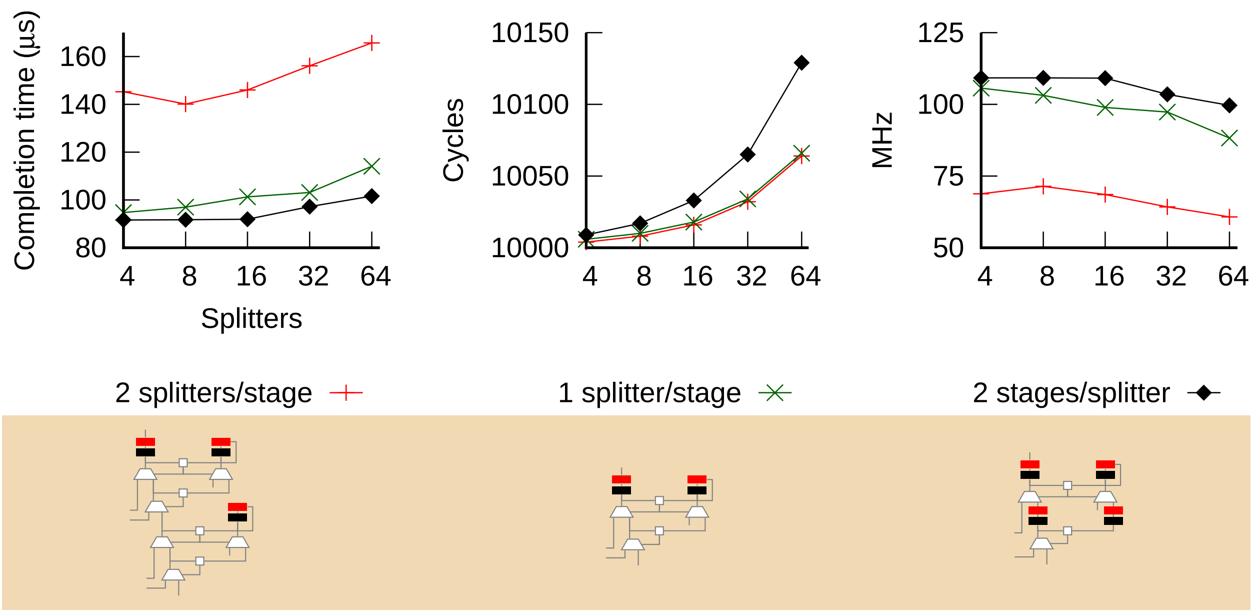


Figure 5.17: Three Conveyor pipelining strategies: doubling the number of stages has only a small effect on total completion time.

5.6.5 Memory in Dataflow Networks

My goal is to use our dataflow networks to implement realistic algorithms with irregular memory access patterns. Mergesort and Treesort both meet these criteria: sorting is a ubiquitous problem and these algorithms employ pointer-based data structures.

We can incorporate memories in our networks with block RAM ("BRAM") actors along with actors that maintain an address pointer (one per BRAM) and route memory requests and results to and from the rest of the network. I employed a separate BRAM per object type, allowing me to tailor its width to the size of the object.

I modified the translation of Section 4.3 to insert memory actors into the Mergesort and Treesort networks and translate them to SystemVerilog. Each BRAM actor becomes a bit vector array with 8-bit addresses, which we access with a basic memory model: given an address and an optional write enable signal with data, the array produces the data at that address before writing in new data if the write enable signal is high. I place two data/control buffers around each BRAM to impose a two-cycle latency per Quartus's recommendations.

The Treesort circuit operates at a higher frequency but uses more memory because its (two-pointer) tree objects are wider than (one-pointer) list objects: it completes in 94.8 μs, operates at 54 MHz, and uses 3330 ALMs and 8.7 kB of memory, while Mergesort takes 82.9 μs, running

	Frequency (MHz)			Area (ALMs)			Registers		
Application	**Manual**	**DF**	**Ratio**	**Manual**	**DF**	**Ratio**	**Manual**	**DF**	**Ratio**
GCD	139	109	0.784	107	234	2.19	67	161	2.4
Conveyor (1 stage)	223	115	0.515	68	61	0.9	165	79	0.48
Bitonic sort	194	103	0.531	327	1212	3.71	434	1944	4.45
Mergesort	137	52	0.38	206	3160	15.3	354	5564	15.7

Table 5.1: Comparing manually-coded RTL SystemVerilog with that generated from DF.

at 52 MHz with 3160 ALMs and 5.9 kB. These are meant to demonstrate that our formalism readily accommodates memory and are not high-performance sorters.

5.6.6 Overhead of Our Method

Table 5.1 lists some statistics on a subset of my designs; I compare the output of our compiler to RTL SystemVerilog coded by Martha Barker (another member of our research group). The numbers I report here come from Quartus running as described earlier.

These measurements attempt to quantify the costs of using our distributed buffering strategy, but are not so simple because they often compare different designs due to the differences between dataflow and combinational logic. The GCD example mostly reflects the cost of additional buffers. While the manual implementation can operate with just two 32-bit data buffers, the DF version requires four: two on the data network, and two on the control network. The number of buffers is directly correlated to registers used, which is shown in the 2.4 ratio between manual and dataflow. This also affects the area due to the extra logic required for the handshake protocols.

Both versions of the Conveyor example use almost the same area, but this time extra buffers were placed around the inputs and outputs of the manual implementation, which is why in this case the dataflow has half as many registers as the manual implementation.

The Bitonic sort example is much larger and slower than the manual implementation because of the insertion of more buffers than necessary and the additional flow control logic to handle when only certain outputs are ready to accept data. The manual implementation does not support backpressure on the outputs.

The Mergesort example diverges most widely, but this is because the two implementations take a very different approach to implementing the algorithm. The manual implementation assumes the data arrives in an array and uses random access to that array; the DF implementation is synthesized from a Haskell program that recursively sorts two linked lists. The different speeds, areas, and registers of the two implementations reflect the overhead of handling lists and recursion.

Chapter 6

Optimizing Irregular Divide-and-Conquer Algorithms

High-level synthesis tools rely on compiler optimizations to improve the performance, area, and/or energy usage of synthesized circuits. Most of the standard HLS optimizations cater to regular-loops-on-arrays programs, and struggle to improve the circuitry implementing pointer-based data structures. New optimizations and synthesis frameworks have been proposed to handle such structures, but most are realized as add-ons to commercial synthesis tools like Xilinx's Vivado, which cannot synthesize specifications containing recursive functions or dynamic memory allocation [138].

In this chapter and the next, I narrow this gap in the HLS research community with two compiler optimizations that together support the second part of my thesis: specialized hardware synthesized from irregular functional programs can be optimized for parallelism. Both optimizations leverage properties of functional languages to cleanly support irregularity.

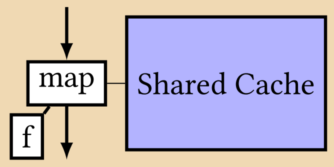
The first optimization, presented in this chapter, is applicable to recursive divide-and-conquer algorithms that operate on irregular pointer-based data structures, an important class of algorithms that challenge standard HLS techniques. Our optimization improves performance of these memory-bound algorithms by partitioning on-chip caches to increase memory bandwidth and transforming programs to duplicate computational resources; the duplicates can operate in parallel by exploiting the additional bandwidth. Others have proposed HLS architectures for pointer-based data structures [84, 122, 142], and even considered divide-and-conquer algorithms [134], but none have proposed our safe and easy-to-apply fusion of memory architecture synthesis and parallelizing program transformations.

Figure 6.1 illustrates how we transform a divide-and-conquer algorithm to improve its performance. In the unoptimized implementation (Figure 6.1a), a recursive function *map* applies a function f to each element of a tree. When translated to hardware, our recursion removal compiler pass introduces a recursive type to model *map*'s stack (Section 3.3.4); both this stack type and the recursive *Tree* type are held in a single cache backed by off-chip DRAM.

In Figure 6.1b, our technique has split the cache into four independent blocks (total capacity

(a)

```
map :: Tree → Tree
map tree = case tree of
  Leaf → Leaf
  Node l x r →
    Node (map l) (f x) (map r)
```



(b)

```
map_S :: Tree → Tree
map_S tree = case tree of
  Leaf → Leaf
  Node l x r →
    Node (map l) (f x) (fromC (map_C (toC r)))

map_C :: TreeC → TreeC
map_C tree = case tree of
  Leaf_C → Leaf_C
  Node_C l x r →
    Node_C (map_C l) (f_C x) (map_C r)
```

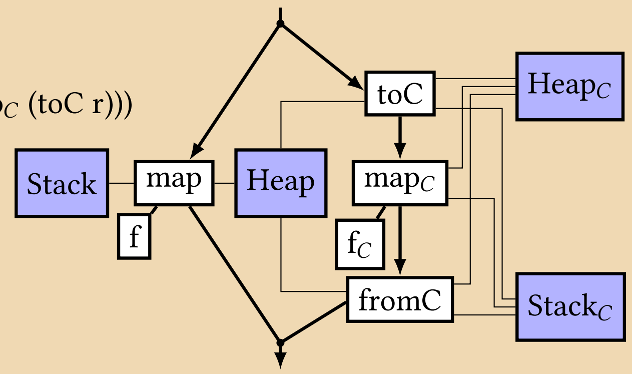


Figure 6.1: (a) A divide-and-conquer function synthesized into a single block connected to a monolithic on-chip cache. (b) Our technique duplicates such functions and partitions the cache to enable task- and memory-level parallelism.

is unchanged) to improve memory bandwidth and duplicated computational resources to run in parallel (copies *map*$_C$ and *f*$_C$). The parallel copies also exploit the increased bandwidth. While the cache can be split evenly, our technique often finds a better partition after profiling.

In this optimized form, a top-level function *map*$_S$ splits the input *Tree* in half: one half is stored in the *Heap* cache and passed to the original *map*, while *toC* converts the other half into a distinct but structurally identical type *TreeC*; the transformed half is stored in the separate *Heap*$_C$ cache and passed to a distinct function copy *map*$_C$. Due to the recursion removal pass, both *map* and *map*$_C$ also have their own stack types, which are assigned to distinct caches. When *map*$_C$ terminates, *fromC* converts a *TreeC* object back into a *Tree* to be combined with *map*'s output as the result from *map*$_S$. If the input *Tree* is well-balanced, *map* and *map*$_C$ can operate largely in parallel, leading to performance improvements in the synthesized circuit.

Our compiler's use of pure functional programs benefit divide-and-conquer algorithms. The strong type system lets us specialize memories to particular types and easily verify that memory operations in separate tasks do not interfere. For example, in Figure 6.1b, *map*$_C$ and *f*$_C$ operate exclusively on the copy *TreeC* of the tree type *Tree* and thus do not compete with *map* and *f* for memory bandwidth. Types similarly motivate the *toC* and *fromC* conversion functions and make it easy to argue that our technique produces correct results.

The purity of our language entails an immutable memory model, which benefits parallel computation. Immutability eliminates the danger of races when tasks run in parallel because shared data cannot have write-after-read hazards. Furthermore, copying data never results in a missed update (e.g., our caches never have to snoop). Many consider these problems central to parallelizing divide-and-conquer algorithms [62, 109, 134].

I present the following technical contributions over the rest of this chapter:

- a code transformation algorithm that enables more parallelism in functional divide-and-conquer programs operating on pointer-based data structures (Section 6.1);
- a type-based on-chip memory partitioning scheme that exploits this parallelism (Section 6.2); and
- experimental results indicating that our methodology can enable up to 1.8× more memory-level parallelism and exploit that parallelism to attain speedups of 1.4× to 1.9× across five representative divide-and-conquer programs (Section 6.3).

6.1 Transforming Code

Our code transformation enables more parallelism in programs expressed with our compiler's Core IR (Section 3.1). It identifies recursive functions to parallelize, duplicates and transforms them and their memory-resident types to enable parallel execution, and inserts type conversion functions that move data structures between caches for memory-level parallelism. As seen in Figure 3.2, the input to this transformation is a simplified Core program, i.e., it has no polymorphism, variable names are globally unique, and lambda lifting has removed local and anonymous functions; general recursion in functions and types still remains.

In the rest of this section, I use *italics* to refer to general functions and types, and `typewriter` font to specify actual functions and types from concrete examples.

6.1.1 Finding Divide-and-Conquer Functions

The algorithm starts by identifying recursive divide-and-conquer ("DAC") functions operating on recursive types. A function f is DAC if it has at least one argument of recursive type and its

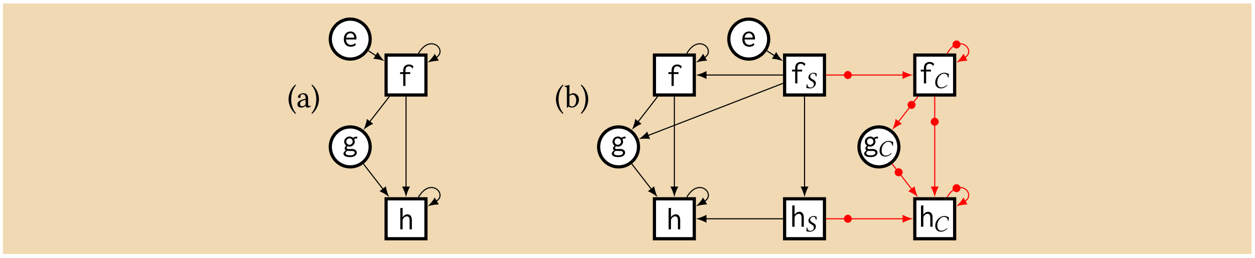


Figure 6.2: A call graph (a) before and (b) after transformation. Divide-and-conquer (DAC) functions `f` and `h` are copied to form $\mathtt{f}_C$ and $\mathtt{h}_C$; $\mathtt{f}_S$ and $\mathtt{h}_S$ are additional versions that split the work between the originals and their copies. Non-DAC functions called from DAC functions are also copied ($\mathtt{g}_C$). "Copied" types flow along dotted red arrows.

body contains an expression of the form $(g \ldots (f \ldots) \ldots (f \ldots) \ldots)$ where g is some function or data constructor distinct from f. In Figure 6.1a, `map` and `Node` are f and g.

Many recursive divide-and-conquer functions contain this simple expression form: two (or more) recursive calls to f operate on distinct halves (or smaller divisions) of an input data structure, and a combining function g merges the results of these calls to produce a final structure. This form is not only commonplace, it also presents a parallelism opportunity; the pairs of recursive calls are not sequentially dependent on one another and can thus be executed in parallel.

The recursive calls in a DAC may operate on shared data, but this does not pose the danger of a data race because of our immutable data model. Immutability guarantees that even if the functions share data, data can be copied without fear of one task missing an update from the other because updates are not possible.

6.1.2 Task-Level Parallelism: Copying Functions

To enable task-level parallelism, we copy every function involved in a divide-and-conquer algorithm, create an additional "splitting" version of each DAC function to run the original and copied versions in parallel, and duplicate and rename types to prevent sharing between tasks. Figure 6.2 shows this procedure applied to an example. DAC functions are rectangles in the call graph, while non-DAC functions are circles. Each copied function has a $_C$ subscript; splitting functions have an $_S$.

From the static call graph of the program, we first determine R, the set of all functions reachable from DAC functions (which includes the recursive DAC functions themselves). In Figure 6.2, `f` and `h` are DAC, so $R = \{$`f`, `g`, `h`$\}$.

Next, we create a new function f_C for every $f \in R$: we copy f's body and, in the copied body, replace every call to a function g (including any recursive calls) with a call to g_C. In Figure 6.2, this creates $\texttt{f}_C$, $\texttt{g}_C$, and $\texttt{h}_C$, each rewritten to call themselves as shown. In Figure 6.1, this creates $\texttt{map}_C$ (which still operates on the original `Tree` type at this point) and $\texttt{f}_C$.

Next, we create a splitting function f_S for every DAC function f by copying f's body and replacing both the second recursive call to f with a call to f_C and any calls to DAC functions $h \neq f$ with calls to h_S. In Figure 6.2, this adds $\texttt{f}_S$ and $\texttt{h}_S$ with their calls as shown. Note that the call of the non-DAC function `g` from `f` is copied unchanged. In Figure 6.1 this step adds $\texttt{map}_S$ without the conversion functions `toC` and `fromC`.

Finally, if a function $e \notin R$ calls a DAC function f, we make it instead call the splitting function f_S. E.g., `e` calls $\texttt{f}_S$ in Figure 6.2b.

6.1.3 Memory-Level Parallelism: Copying Types

Above, we copied functions to enable task-level parallelism between a DAC function f and its copy f_C; we now copy types to enable memory-level parallelism between these functions while ensuring type correctness. Each recursive type is stored in exactly one cache, so f and f_C must operate on different recursive types to avoid cache-sharing bottlenecks. We can prevent these bottlenecks by creating a copy T_C of any recursive type T passed to or returned by a function in R (the set of functions reachable from a DAC function) and modify f_C to only use T_C. However, this can break type correctness, e.g., if some function `f` uses a non-recursive tuple type to carry pairs of `T` values:

```
data Tuple = Tuple T T
```

then its copy, $\texttt{f}_C$, cannot use the same `Tuple` type to carry its $\texttt{T}_C$ values.

We thus make a set Γ of all recursive types passed to or returned by a function $f \in R$, then add to Γ the types of all expressions in R that have a variant with a type field that is already in Γ and repeat until we reach a fixed point. For example, if some $f \in R$ returns a value of recursive type `T` and some expression in R has the above `Tuple` type, Γ will initially contain `T`; `Tuple` will then be added to Γ since $\texttt{T} \in \Gamma$ is a type field for one of its variants.

Next, we duplicate each type in Γ to obtain a new set Γ_C (each duplicated type also has duplicated data constructors). As with function calls in the copied functions, we modify the type definitions in Γ_C such that if a type $T_C \in \Gamma_C$ has a type field $S \in \Gamma$, we replace it with its copied type

$S_C \in \Gamma_C$. Thus, no type defined in Γ refers to any defined in Γ_C and vice versa. This step produces the `Tree`$_C$ type from Figure 6.1b.

6.1.4 New Types and Conversion Functions

Finally, we introduce the Γ_C types into our function copies. First, we replace any data constructor for a type $T \in \Gamma$ in a function copy f_C with a data constructor for the corresponding type $T_C \in \Gamma_C$. In Figure 6.1, this introduces `Leaf`$_C$ and `Node`$_C$ into the body of `map`$_C$. Next, we introduce conversion functions that correct the types passed to and returned by the call of f_C in each f_S. For each recursive type $T \in \Gamma$, we create two recursive functions: *toC* converts an object of type T to an equivalent object of type T_C, while *fromC* converts an object of type T_C back to type T. Each pair of conversion functions is uniquely named (e.g., we may have `list_toC` and `tree_toC` to produce `List` and `Tree` copies). We then add these conversion functions to the call of f_C in f_S: any argument x of type T passed to f_C is wrapped in a *toC* call, and if f_C returns a value of type T_C then any call to f_C is wrapped in a *fromC* call. E.g., if x is of type T then $(f_C \ \ldots \ x \ \ldots)$ in f_S becomes $(f_C \ldots (toC\, x) \ldots)$; if f_C returns a type T_C, $(f_C \ \ldots)$ becomes $(fromC(f_C \ \ldots))$. In Figure 6.1, this final step of the code transformation introduces the calls to `toC` and `fromC` in `map`$_S$.

6.2 Partitioning On-Chip Memory

I now present this chapter's second contribution: a type-based scheme for partitioning on-chip memory into multiple, independent caches to increase memory parallelism. This scheme produces a different memory system than the actor-based one described in Section 5.6.5; a compiler flag lets the user decide which memory system to use.

We apply this partitioning technique after the compiler has generated a dataflow network from the program (in DF form; see Section 5.5). By design, our memory partitioning dovetails with our code transformation to provide additional memory bandwidth to the parallel tasks; I confirm this experimentally in Section 6.3.

The first step of our technique converts each (type-specific) read or write actor in the dataflow network into a pair of point-to-point links that interface the dataflow network to the on-chip memory system. One link passes data to write (or an address to read) from the network out to the memory system, the other returns the resulting address (or data) back into the network. We treat each pair of links as a single bundle called a "channel", and partition memory such that each channel is connected to exactly one on-chip cache.

Our code transformation makes a copy f_C of each function f reachable from a DAC function, and the compiler generates independent hardware blocks for both f and f_C. The f block can have two sets of memory channels: heap channels that access data of any recursive type found in Γ, and stack channels that access the explicit stack types introduced for f (if it is recursive). The block for f_C connects to similar but distinct memory channels for types in Γ_C and the stack for f_C. It is not uncommon for the two blocks to generate simultaneous requests for all four types.

We thus partition on-chip memory into four caches. The stack and heap channels from f are connected to two independent caches; those from f_C are connected to two others. Partitioning stack and heap channels maintains the pipeline parallelism inherent in our generated dataflow networks (Section 4.3), while partitioning f's channels from f_C's exploits memory-level parallelism between the two sets. While we could go further and create a distinct cache for each memory channel type, this would likely decrease performance: we maintain a fixed on-chip cache memory budget, so the caches would be smaller and their miss rates would increase. Restricting the cache count to four also drastically shrinks the search space of our cache sizing algorithm (discussed below), increasing its runtime efficiency in practice.

While this scheme has so far connected each memory channel in functions reachable from DAC functions to caches, channels from other functions are still unconnected. We again partition these channels into stack and heap sets, but then split each set randomly in half and associate each half with one of the two heap or stack caches created previously. While assigning the "extra" types randomly would seem unwise, I verified experimentally (see Section 6.3.2) that doing so has only a negligible effect on performance, so we did not try any further optimizations.

We determine cache sizes through profiling. We divide a power-of-two amount of on-chip memory (1 MB in our experiments) into four equal-sized caches, connect them to the dataflow circuit, and simulate with a representative input to obtain a trace of memory accesses. We feed this trace to a variant of Winterstein et al.'s cache sizing algorithm [135], which searches for cache partitions with the maximum aggregate hit rate under the constraints that the caches' total capacity is within the on-chip budget and that every cache size is a power of two. This narrows the search space and conserves resources. Our algorithm generally preferred 1/2–1/4–1/8–1/8 to uniform (1/4–1/4–1/4–1/4) partitioning. Winterstein et al. assumed direct-mapped caches; our caches are two-way set associative, so we check both ways for a cached address and model an LRU policy with timestamps.

6.3 Experimental Evaluation

I compiled and simulated five algorithms, evaluating both how much memory parallelism our code transformation could expose by itself and how effectively our memory partitioning could exploit it to improve performance. Our techniques exposed up to 1.8× more memory-level parallelism that usually translated into similar performance improvements, provided we both duplicated functional units and partitioned the cache. Not surprisingly, performance improvements also depended on balanced task-level parallelism.

I implemented our code transformation as an optional pass in the compiler, situated between the lambda lifting (Section 3.3.3) and recursion removal (Section 3.3.4) passes. After applying the code transformation, the compiler performs the other, previously discussed translations to synthesize the final program into a dataflow network expressed in SystemVerilog. I then used Verilator 3.874 to convert our SystemVerilog specification into a C++ behavioral model for simulation. An additional run by our compiler produces an application-specific C++ testbench that interfaces the Verilator-generated code with a memory simulator that I wrote.

My cycle-accurate memory simulator implements an immutable heap model; this prevents race conditions, as only new objects may be written. The dataflow network presents write data to the memory system, which chooses a new address, writes the data, and only then returns the address to the network. Reads operate in the usual manner: the dataflow network passes an address to the memory system, which responds with the data at that address. The network itself never creates or modifies addresses. The memory simulator does not model garbage collection, but because others have done it efficiently in hardware [9], our results would not be affected greatly by the addition of garbage collection.

The memory simulator operates in two modes (Table 6.1). The “oracle” mode measures potential memory parallelism independent of cache or latency effects, but is physically unrealizable. This mode is similar to the simulated memories used in our initial dataflow network experiments from Section 4.5. The “realistic” mode is designed to estimate more realistic performance improvements and considers multiple on-chip caches that operate on 32-bit words, all backed by a single unbounded off-chip DRAM.

As test cases, I selected five memory-dominated divide-and-conquer algorithms using pointer-based structures, ranging from simple (sorting) to complex (K-means clustering). These are the kinds of algorithms our technique is designed to improve.

Mergesort sorts a list of integers using divide-and-conquer. It uses one helper function to split

Table 6.1: Simulated Memory Parameters

	Oracle	**Realistic**	
		On-chip Cache	**Off-chip DRAM**
Latency	1 cycle	1	50
Word length	*n* bits†	32	32
Capacity	∞	1 MB‡	∞

† For an *n*-bit object. ‡ Total capacity; may be partitioned.
Cache is 2-way associative, write-back, LRU, with a 16-entry miss queue.

the input list in half and another to merge the results after recursing. Note that this is a different implementation than the Mergesort evaluated in Section 4.5: the single splitting function used here takes a list as input, splits it in half recursively, and returns a tuple of the two halves.

Treesort and RBsort sort a list of integers by transforming it into a binary search tree then flattening the tree into a sorted list via an in-order tree traversal. The flattening function recurses on the left and right children of the root, then inserts the root's data between the two resulting lists, calling an *append* function twice. Treesort naïvely constructs a binary tree; RBsort constructs a balanced red-black tree. Both start from an empty tree then insert elements one at a time.

RBmap constructs a red-black tree from a list of integers then calls a variant of the *map* function from Figure 6.1 (i.e., the *Node* variant has another field indicating its color) on it ten times, each time applying a function that adds a constant to each node's data. Normally, each call of *map* would have to convert the tree into a parallel form and then back, but I manually rewrote the generated code so that a conversion happens once before all the *map* operations and once after. Automating this optimization constitutes future work.

Kdfilter implements Kanungo et al.'s K-means clustering algorithm [77], which is used in machine learning, image processing, and other fields. A function first recursively constructs a K-d tree from a list of 2D points, using *mergeSort* on each recursive call to balance the remaining points across the tree. The final tree is then passed to the filtering algorithm: a DAC function that frequently calls another version of *mergeSort* to sort smaller lists of points. The K-d tree construction, *mergeSort*, and filtering functions are each DAC and transformed by our technique.

Each test first generates a list of 16384 random integers (except Kdfilter, which generates a list of random 2D points), then passes the list to the functions described above. I omit the time taken to generate the inputs when reporting performance numbers, since this portion of each test is identical in both the original and transformed versions. I also ran each experiment on

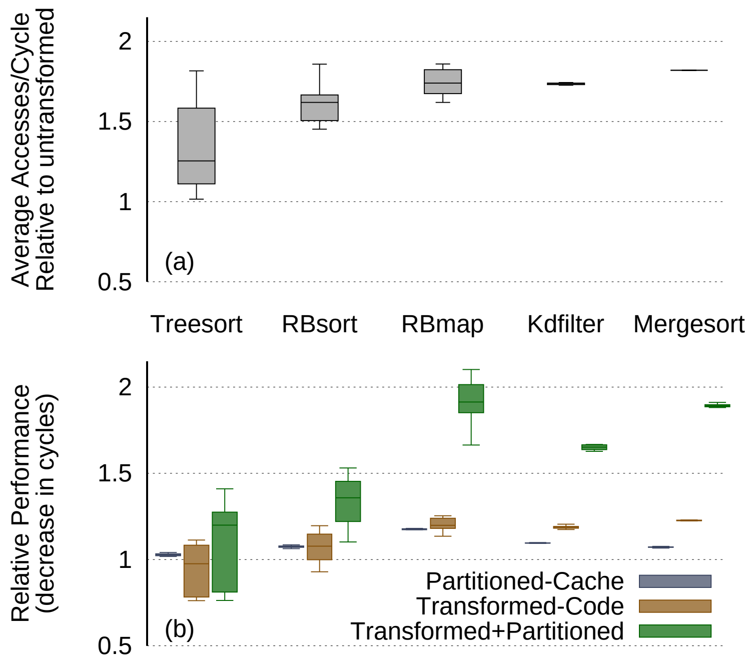


Figure 6.3: (a) Our code transformation consistently increases the number of memory accesses per cycle—a proxy for memory-level parallelism (oracle memory model). (b) Under the realistic memory model, cache partitioning and the code transformation each produce modest improvements; their combination is best because parallel tasks can exploit extra memory bandwidth.

input lists of sizes 256, 512, 1024, . . . , 8192 but found input size had little effect on the results, so I present results only for the largest inputs. Each test's four cache sizes, shown in Table 6.2, were determined by simulating the test on an input size of 4096 and passing the resulting trace to our implementation of Winterstein et al.'s cache sizing algorithm. The shorter trace ensured that the exhaustive algorithm terminated in a reasonable amount of time.

6.3.1 Exposing Memory-Level Parallelism

I estimated how much memory-level parallelism (MLP) our technique exposes by generating circuits from the example programs with and without our code transformation applied and comparing their average number of parallel memory accesses. To measure MLP independent of memory latency and caching, I ran these experiments under our oracle memory model, which performs one memory access per cycle per type; multiple accesses to the same type must queue.

For each example, I applied 20 randomly generated inputs to both the original network and the network produced after applying our code transformation. For each run, I calculated the average number of memory accesses per cycle (the “access rate”) by counting every (single-cycle) memory access made by an example’s dataflow network and dividing by its cycles to completion.

These examples demand memory bandwidth. Across all 20 inputs, before our transformation, the network for the Treesort example averaged 0.37 memory accesses per cycle; RBsort averaged 0.37; RBmap, 0.33; Kdfilter, 0.27; and Mergesort averaged 0.37.

Figure 6.3a shows our technique increases MLP as measured by the access rate. For each example, this plot shows the distribution of the ratio between the access rate for a particular input fed to the original network and the transformed network. A ratio of 2 means our code transformation doubled the access rate, suggesting the generated circuit could benefit from multiple, parallel memories; a ratio of 1 would suggest parallel memories would rarely be used simultaneously, so a single large memory would perform equally well.

The varying sizes of the bars in Figure 6.3a indicate that the benefit of our transformation for potential MLP depends on the algorithm and sometimes the input data. Our code transformation technique works best when the first divide leads to equal amounts of work to conquer. Treesort, RBsort, and RBmap each walk a binary tree, so the balance and hence available parallelism depends on the structure of the input tree. Treesort constructs a tree whose structure can range from completely unbalanced to perfectly balanced depending on input, leading to the extreme variation shown in Figure 6.3a. RBsort also builds and walks a tree to sort its input, but exhibits less variation because it uses a red-black tree, which remains roughly height-balanced regardless of input. RBmap builds a similar red-black tree, but performs just a constant amount of work per node and is thus less affected by imbalance than RBsort’s *append* operation, where the amount of work per node depends on its number of children.

Kdfilter and Mergesort vary little across inputs because they split their inputs nearly perfectly in half, avoiding performance-robbing load imbalance. Mergesort performs a perfect split followed by identical work for both halves of the list, allowing it to produce a consistent 1.8× improvement in MLP. Kdfilter includes many calls to Mergesort, but also performs other operations that do not partition so perfectly, giving a slightly lower improvement of 1.7×.

6.3.2 Exploiting Memory-Level Parallelism

I analyzed how well our circuits exploited the MLP exposed by our code transformation by measuring their performance under a more realistic memory model. Specifically, to tease apart the effects of code transformation from cache partitioning, I ran the circuits from the previous exper-

Table 6.2: Heap and Stack Partition Assignments for the Transformed & Partitioned Examples

Test	**512K**	**256K**	**128K**	**128K**
Mergesort	Heap_C	Heap	Stack	Stack_C
Treesort	Heap	Stack_C	Heap_C	Stack
RBsort	Heap	Heap_C	Stack	Stack_C
RBmap	Heap	Heap_C	Stack	Stack_C
Kdfilter	Heap	Heap_C	Stack	Stack_C

iments (Section 6.3.1) with both a single monolithic cache and the multiple caches generated by our partitioning scheme. The sizes and roles of each cache assigned to our transformed circuits are listed in Table 6.2; the cache assignments for the untransformed circuits are less meaningful given the lack of duplicated types (i.e., dividing heaps from stacks is the most important aspect).

I again simulated the four configurations on 20 random inputs (expect Kdfilter: I simulated it on four random inputs of 8192 points due to excessive simulation time). I limited total cache capacity to 1 MB and backed them by a single, large, slow off-chip DRAM (our "realistic" memory model; see Section 6.3).

Figure 6.3b shows the distribution of speedups (in cycles, relative to the untransformed, monolithic cache case) for each example after partitioning the cache, transforming the code, and both. Partitioning the cache without transforming the code to increase MLP (left bars) produces only modest benefits, likely due to functions that occasionally access stack and heap caches in parallel. In these cases, the varying test inputs have little effect on the circuit's performance since the total amount of work is roughly the same across all inputs for a given unoptimized test.

Applying our code transformation without cache partitioning (middle bars) can take advantage of overlapping memory accesses from parallel tasks, but degraded performance for some inputs to Treesort and RBsort. An unlucky input to these examples can direct a large load to the "copied" task; the copying overhead introduced by the *toC* and *fromC* conversion functions dominates any speedup from parallel tasks. This occurs less in RBmap due to my optimization: I convert the structure before performing a much lengthier computation (traversing the whole tree ten times), so the overhead is proportionally smaller.

Overall, our code transformation performs best when it can exploit the increased memory bandwidth of a partitioned cache (right bars in Figure 6.3b), often achieving the performance gains predicted by the increased MLP observed in my first experiments (Section 6.3.1). RBmap, Mergesort, and Kdfilter exhibit the highest increases in MLP and the biggest speedups when given partitioned caches. Kdfilter exhibits a fairly consistent 1.7× speedup; Mergesort a similarly consistent 1.9×, and RBmap falls somewhere between the two. RBmap's performance varies more, however, even exceeding 2× on certain inputs. Such an extreme speedup is due to a significant reduction in aggregate miss rate across the partitioned caches. To test this hypothesis, I re-ran RBmap with a different memory allocation policy (I doled out address 0,3,6,... instead of 0,1,2,...) and found the cache miss rate increased (but still entailed a speedup of 1.7×).

I feared nondeterministic assignment of a program's "extra" types to caches could affect these experimental results, but found experimentally it did not matter. Our cache partitioning policy assigns the stacks and heaps of each DAC function (and those reachable from it) to distinct caches, but assigns all remaining heap (stack) types to one of the two heap (stack) caches at random, i.e., each additional heap type may be assigned to either of the heap caches; a similar assignment is done for excess stack types. There were usually too many assignments to test exhaustively, so I randomly sampled enough to give us a 90% confidence level. I found only a 2% variance in completion time with a 5% margin of error, so the assignment of "extra" types did not affect these experiments.

Compared to their behavior under an idealized memory model, unbalanced workloads (e.g., in Treesort and RBsort) perform worse under a realistic memory model, again because the conversion overhead (i.e., inter-cache copying) overshadows the meager benefit of task-level parallelism with unbalanced workloads. In extreme cases (e.g., certain input patterns to Treesort), the increase in memory bandwidth from partitioning the cache goes unused because of the load imbalance, and conversion overhead exceeds any benefit of parallelism, leaving the performance worse than the baseline.

I conclude that our technique works best when the divide-and-conquer operation produces nearly balanced workloads that can take advantage of a partitioned cache's additional bandwidth.

Chapter 7

Packing Recursive Data Types

The previous chapter discussed the first major optimizing pass in our compiler, which enables more parallelism in circuits implementing irregular divide-and-conquer algorithms. This chapter covers the second: an algorithm that packs recursive data types to increase data-level parallelism and reduce trips to off-chip memory. For example, our algorithm can transform a program that operates on linked lists with a single data element per cell into one that performs the same operations on lists with two or more elements per cell. Our algorithm also works on trees and similar recursive data types. While presented as operating on Core, this algorithm may be applied to any pure functional program containing recursive data types.

Running this optimization as part of the compilation process produces equivalent circuits that make fewer memory accesses and complete their work in fewer clock cycles: across eleven benchmark programs, our algorithm reduced memory operations by 1.5× to 4×; nine of the benchmarks also experienced speedups of 1.2× to 2.5×.

In effect, our algorithm improves the spatial locality of data structures without relying on cache blocks. Since our technique stores the same data in fewer cells, it reduces the number of distinct memory accesses (i.e., the number of cells that need to be read) and may also reduce the overall memory traffic since fewer inter-cell pointers are needed to represent the same structure. Of course, a programmer could manually perform such a transformation, but doing so is a detailed, error-prone process that requires the programmer to consider many edge cases—a perfect task for a compiler.

To my knowledge, this algorithm is the first to automatically pack arbitrary recursive data types (i.e., not just lists) and transform functions to operate on them; we are the first to apply such an algorithm in a hardware synthesis setting. Furthermore, most high-level synthesis tools balk at pointer operations and recursive functions; our algorithm not only works in such a setting, it improves the quality of results.

After outlining how it operates on a simple list example (Section 7.1), I present a detailed description of our algorithm (Section 7.2). I finish by presenting experimental results that show our algorithm consistently reduces memory operations and can often improve the memory footprint and overall running time of our compiler-generated dataflow networks (Section 7.3).

7.1 An Example: Appending Lists

In this section, I illustrate how our algorithm packs the lists in the *append* function, which concatenates two lists. Our algorithm operates on Core programs after polymorphism, duplicate names, and local and anonymous functions have been removed, but before recursive functions and data types have been replaced with tail-recursion and pointers.

This example involves the *List* type first presented in Chapter 3, which has two alternatives: *Nil*, the empty list, and *Cons*, a cell holding an integer and a reference to the rest of the list:

```
data List = Nil
          | Cons Int List
```

We code *append* with pattern matching and recursion: when the first list is empty, *append* returns the second list, otherwise, it splits the first list into a head element (x) and a tail (xs), calls itself recursively on the tail, and calls *Cons* on the result and the head to construct a new cell:

```
append :: List → List → List
append z y = case z of
  Nil       → y
  Cons x xs → Cons x (append xs y)
```

Our algorithm transforms *append* into an equivalent function that operates on groups of list elements. The user sets a parameter called the *packing factor* to specify how many times a recursive type should be inlined to bring more data into each cell. For this example, we use a packing factor of 1, so our algorithm begins by inlining the recursive *List* type once, resulting in a packed list type that can be an empty cell, an unpacked cell holding a single integer (needed, e.g., for lists with an odd number of elements), or a packed cell holding two integers:

```
data PList = PNil                  -- Empty list
           | UCons Int     PList   -- Unpacked cell
           | PCons Int Int PList   -- Packed cell
```

Our algorithm then generates functions *pack* and *unpack* that convert between ordinary and packed lists:

```
pack :: List → PList   -- Pack a list
pack arg = case arg of Nil         → PNil
                       Cons a as  → case as of
                                      Nil       → UCons a   (pack as)
                                      Cons b bs → PCons a b (pack bs)

unpack :: PList → List   -- Unpack a list
unpack arg = case arg of PNil           → Nil
                         UCons c    cs  → Cons c (unpack cs)
                         PCons d e  es  → Cons d (Cons e (unpack es))
```

By construction, these function are inverses: *unpack* ∘ *pack* and *pack* ∘ *unpack* are each the identity function.

Next, our algorithm transforms *append* to consume and produce packed lists by wrapping *pack* around the body of *append* and *unpack* around the recursive call, replacing appearances of its arguments *z* and *y* with *unpack z* and *unpack y*, and applying *pack* to the arguments of *append*'s recursive call. This produces a correct, but inefficient program:

```
append :: PList → PList → PList
append z y = pack (case unpack z of
   Nil        → unpack y
   Cons x xs  → Cons x (unpack (append (pack xs) (pack (unpack y )))))
```

Because, *pack* and *unpack* are inverse functions, "pack (unpack y)" is just "y." Our algorithm performs this simplification, giving

```
append z y = pack (case unpack z of
   Nil        → unpack y
   Cons x xs  → Cons x (unpack (append (pack xs) y )))
```

To eliminate *pack* and *unpack* calls, our algorithm first inlines the *unpack* function in the *case* scrutinee *unpack z*:

```
append z y = pack (case (case z of -- unpack function inlined
                           PNil          → Nil
                           UCons c   cs  → Cons c (unpack cs)
                           PCons d e es  → Cons d (Cons e (unpack es))) of
  Nil        → unpack y
  Cons x xs  → Cons x (unpack (append (pack xs) y)))
```

This enables Jones and Santos's [101] "case-of-case" and "case-of-known-constructor" simplifications, giving

```
append z y = pack (case z of
  PNil          → unpack y
  UCons c   cs  → Cons c (unpack (append (pack (unpack cs))          y))
  PCons d e es  → Cons d (unpack (append (pack (Cons e (unpack es))) y)))
```

Next, we perform our central trick: we inline all recursive calls of the *append* function to produce a structure mimicking that of the now-packed *PList* type (e.g., both manipulate pairs of values). Such inlining exposes expressions of the form *unpack* (*pack* ...) that we can eliminate. The packing factor controls how many times we repeat this step (once in this example).

```
append z y = pack (case z of
  PNil          → unpack y
  UCons c   cs  → Cons c (unpack (pack (case cs of
    PNil          → unpack y
    UCons f   fs  → Cons f (unpack (append fs                            y))
    PCons g h hs  → Cons g (unpack (append (pack (Cons h (unpack hs))) y )))))
  PCons d e es  → Cons d (unpack (pack (case pack (Cons e (unpack es)) of
    PNil          → unpack y
    UCons f   fs  → Cons f (unpack (append fs                            y))
    PCons g h hs  → Cons g (unpack (append (pack (Cons h (unpack hs))) y ))))))
```

Now, we push all functions applied to each *case* expression through to its alternatives and eliminate adjacent *pack* and *unpack* calls:

```
append z y = case z of PNil          → y
                       UCons c  cs   → case cs of
 PNil         → pack (Cons c (unpack y))
 UCons f   fs → pack (Cons c (Cons f (unpack (append fs                            y ))))
 PCons g h hs → pack (Cons c (Cons g (unpack (append (pack (Cons h (unpack hs))) y ))))
                       PCons d e es  → case pack (Cons e (unpack es)) of
 PNil         → pack (Cons d (unpack y))
 UCons f   fs → pack (Cons d (Cons f (unpack (append fs                            y ))))
 PCons g h hs → pack (Cons d (Cons g (unpack (append (pack (Cons h (unpack hs))) y ))))
```

Finally, we inline the *pack* calls and, as before, apply the case-related simplifications and remove adjacent *pack* and *unpack* calls to give the final version of *append*, now free of calls to *pack* and *unpack*:

```
append z y = case z of
  PNil          → y
  UCons c   cs  → case cs of
                    PNil         → UCons c y
                    UCons f fs   → PCons c f (append fs            y)
                    PCons g h hs → PCons c g (append (UCons h hs) y)
  PCons d e es  → PCons d e (append es y)
```

When realized in hardware, this new function runs faster because it makes fewer memory accesses (assuming the input list is comprised of *PCons* cells). But this performance gain is not without cost: the code of the new function is bigger, primarily due to recursive inlining, and thus requires more hardware resources. More selective recursive inlining would limit code growth, but then certain packable structures produced by recursive functions might remain unpacked, nullifying the advantages of our algorithm. I discuss some heuristics that explore these tradeoffs in Section 7.2.5.

7.2 Packing Algorithm

While a programmer could probably manually devise the packed version of *append* shown above, our algorithm—a series of semantics-preserving rewrites that rely on *pack* and *unpack* being inverse functions—derives this mechanically and works on arbitrary recursive functions. Here, I describe it in detail.

Throughout this section, I make the lambda expressions implementing functions explicit, i.e., a function's arguments are shown on the right-hand side of its definition between the "λ" and "$\rightarrow$" symbols. This is done to convey how function inlining actually works in the algorithm; although inlining steps introduce anonymous functions, they are always eliminated by the end of the algorithm, maintaining the invariant that Core programs only contain top-level, named functions at this point in the compiler.

7.2.1 Packed Data Types

The first step of our algorithm identifies each *packable* data type: an algebraic type with exactly one recursive variant. As explained later, this restriction simplifies the algorithm, reduces the code growth incurred by function inlining, and captures the ubiquitous list and tree types that implement numerous data structures, e.g., stacks, dictionaries, and K-d trees [77]. Thus, we pack recursive types of the form

```
data T = B t
       | R s T₁ ··· Tₖ
```

Here, t and s are non-recursive fields (although they could be some other recursive type $S \neq T$) and R has one or more recursive fields $T_1 \ldots T_k$. We call B the *base* variant and R the *recursive* variant of the packable type T. We use the form above for clarity; in general, T may have more than one base variant, all variants may have zero or more non-recursive fields, and R's non-recursive and recursive fields may be interspersed in its definition.

The corresponding *singly packed* type, P, is of the form

```
data P = B′ t
       | U s P₁ ··· Pₖ
       | P s s₁ P₁₁ ··· P₁ₖ ··· sₖ Pₖ₁ ··· Pₖₖ
```

where the B' and U variants are analogous to the B and R variants in the unpacked type T. P is the *packed* variant, consisting of $k+1$ copies of the "payload" s and $k \times k$ recursive instances. The P variant of a singly packed type can be expressed more succinctly as $P\ s\ (s\ P^k)^k$.

Both our generation of packed variants and the definition of a packable type (a type with exactly one recursive variant) seem restrictive at first glance, but are motivated with the goal of reducing exponential code growth. We obtained the packed variant by inlining each recursive field in the recursive variant U, but we could have generated other "less-packed" variants by selectively inlining sets of fields. However, for a variant with a set of k recursive fields, there are $2^k - 1$ non-empty subsets of fields (each corresponding to an inlining choice), so the most general packed type would have $2^k - 1$ packed variants. As seen in Section 7.1, our algorithm introduces *case* expressions that pattern match on each variant of the packed type, so such a general scheme would entail code growth exponential in k. A similar case would occur if we defined packable types to have $j \geq 1$ recursive variants; a variant with k recursive fields would entail j^k packed variants (since there are j inlining choices for each field). By limiting our packable types and the form of a packed variant, we ensure that our *case* expressions will always match on a constant number of variants.

However, exponential growth is still possible. P's definition depends on a user-defined integer n called the *packing factor*. This value determines how many times we inline the recursive fields in U's definition to obtain P. Here we assumed a packing factor of $n = 1$: each of U's recursive fields P_i was inlined once to yield $s_i\ \mathrm{P}_{i1}\ \ldots\ \mathrm{P}_{ik}$. Increasing the packing factor gives exponentially larger cells when $k > 1$:

Packing Factor	P variant
(unpacked) 0	$\mathrm{P}\ s\ \mathrm{P}^k$
1	$\mathrm{P}\ s\ (s\ \mathrm{P}^k)^k$
2	$\mathrm{P}\ s\ (s\ (s\ \mathrm{P}^k)^k)^k$
3	$\mathrm{P}\ s\ (s\ (s\ (s\ \mathrm{P}^k)^k)^k)^k$

To generate completely packed cells, the algorithm inlines all recursive function calls n times. If the function has k recursive calls (typical when traversing a recursive structure), this can cause exponential code growth, leading to exponential resource usage in hardware; in Section 7.2.5 I discuss heuristics that help retard this growth.

7.2.2 Pack and Unpack Functions

Our algorithm next defines conversion functions *pack* and *unpack* that convert between types T and P. For a packing factor of 1, we construct the bodies of these functions from the following templates.

```
pack :: T → P
pack = λw → case w of
  B x              → B′ x
  R x (R y z^k)^k  → P x (y (pack z)^k)^k
  R x z^k          → U x (pack z)^k
```

```
unpack :: P → T
unpack = λw → case w of
  B′ x           → B x
  P x (y z^k)^k  → R x (R y (unpack z)^k)^k
  U x z^k        → R x (unpack z)^k
```

The recursive *pack* function takes a T and produces an equivalent P. There are three cases. A base variant B is simply renamed to a B′; any payload is copied. The second case generates a packed variant P from a "fully populated" R variant, that is, one whose recursive references are all to R variants. The third case handles all the other cases (e.g., a binary tree node with only one branch) by generating an unpacked variant U. As we mentioned above, for something like a binary tree we could consider additional variants (e.g., left branch only and right branch only), but we handle all of these variants with a single catch-all variant to minimize code complexity.

By design, the recursive *unpack* function is the inverse of *pack* through symmetry: the patterns and expressions swap roles to make *unpack* exactly reverse the work of *pack*. These two functions satisfy the *packing identity*: *pack* (*unpack p*) = *p* and *unpack* (*pack u*) = *u*.

Simple structural induction shows the packing identity holds; for example, here is the proof that applying *unpack* after *pack* leaves the original structure unchanged:

Theorem 1. *For all u of packable type T, unpack (pack u) = u.*

Proof. By induction over the packable data type u. All equalities shown are by definition unless stated otherwise.

Base Case: u = B x. Then we have

```
  unpack (pack u)
= unpack (pack (B x))
= unpack (B′ x)
= B x
= u
```

Inductive Case 1: u = R x z^k, where at least one z is not a recursive variant. Assume that our theorem holds for each z. Then we have

```
  unpack (pack u)
= unpack (pack (R x z^k))
= unpack (U x (pack z)^k)
= R x (unpack (pack z))^k
= R x z^k
= u
```

where the penultimate equality is due to the inductive hypothesis.

Inductive Case 2: u = R x q^k, where each q is itself a recursive variant R y z^k. Assume that our theorem holds for each z. Then we have

```
  unpack (pack u)
= unpack (pack (R x (R y z^k)^k))
= unpack (P x (y (pack z)^k)^k)
= R x (R y (unpack (pack z))^k)^k
= R x (R y z^k)^k
= u
```

where the penultimate equality is due to the inductive hypothesis. □

For higher packing factors, the second *case* alternative of each function grows more complicated because it needs to work with multiple fully-populated tree levels. In general, these alternatives are

```
R x (R y1 (··· (R yn z^k)^k···)^k)^k →
                    P x (y1 (··· (yn (pack z)^k)^k···)^k)^k

P x (y1 (··· (yn z^k)^k···)^k)^k →
                    R x (R y1 (··· (R yn (unpack z)^k)^k···)^k)^k
```

7.2.3 Injection and Hoisting

After creating packed data types, our algorithm transforms functions by first injecting seemingly redundant (but semantics-preserving) calls to *pack* and *unpack* throughout the program, then "hoisting" certain calls so functions take and return the packed types. After this step, the bodies of the functions continue to operate on unpacked data; later (Section 7.2.4) we will also transform the function bodies to operate directly on packed data by inlining and simplifying.

To inject the packed type in the program, we first surround every expression of packable type T with calls to *pack* and *unpack*, i.e.,

$$\text{if } e :: \mathrm{T}, \quad e \quad \text{becomes} \quad (\mathit{unpack}\ (\mathit{pack}\ e))$$

where *pack* :: T → P and *unpack* :: P → T are the complementary functions described above. Since *unpack* ∘ *pack* :: T → T is the identity function (by Theorem 1) and we apply it to expressions of type T, it follows that injection leaves a well-typed program's meaning unchanged.

Next, we apply three "hoisting" rules that move around the injected calls to *pack* and *unpack* to make the program's functions take and return packed types.

Top-Level Definitions that Return Type T There are two kinds of top-level definitions: functions and variables.

After injection, every definition of a function f that returns type T and every call to that function will have the form

```
f  ::  ···  → T
f  = λ ···  → unpack (pack e)     -- Definition of f

··· unpack (pack (f ··· )) ···    -- Call of f
```

where e is an arbitrary expression of type T. Referential transparency (a property of our pure language) dictates that a call to some function f may be replaced with its body (after substituting in its arguments); thus, applying another function g to the result of every f call is equivalent to simply applying g once to f's body instead. Our hoisting rules leverage this property; the first hoists the call to *pack* from each of f's call sites to its definition, which changes f's return type to P:

```
f  ::  ···  → P
f  = λ ···  → pack (unpack (pack e))  -- Definition of f

··· unpack (f ··· ) ···               -- Call of f
```

Now we remove the redundant *unpack*/*pack* pair:

```
f  ::  ···  → P
f  = λ  ···  → pack e              -- Definition of f

···  unpack (f  ···  )  ···        -- Call of f
```

We apply a similar procedure to (top-level) variables of type T to transform them to variables of type P. These go from

```
v  ::  P
v  = unpack (pack e)          -- Definition of v

···  unpack (pack v)  ···     -- Use of v
```

to

```
v  ::  P
v  = pack e             -- Definition of v

···  unpack v ···       -- Use of v
```

Function Argument of Type T When a function f has an argument x of type T, injection wraps references to x in f's definition with *unpack*/*pack* pairs. Similarly, wherever f is called, the expression passed as x is also wrapped, i.e., if x is f's first argument, this looks like

```
f  ::  T →  ···
f  = λx  ···  →  ··· (unpack (pack x)) ···   -- Argument x of type T

···  f  (unpack (pack e ))  ···   -- Passing argument e of type T
```

We hoist the *unpack* call enclosing the argument into the body of the lambda term to change the type of the argument:

```
f  ::  P →  ···
f  = λx  ···  →  ···  (unpack (pack (unpack x )))  ···

···  f  (pack e)  ···
```

and simplify

```
f :: P → ···
f = λx ··· → ··· (unpack x) ··· -- Have argument of type P

··· f (pack e) ··· -- Passing argument of type P
```

Unpacked Types in Other Types The hoisting rules remove the unpacked type from the program (the definitions of *pack* and *unpack* are the exception; they still use the unpacked type by definition). The first two rules transform functions that operate on unpacked types; the third transforms other types that include unpacked types.

When a type S ≠ T has a variant defined as $C \ldots$ T $\ldots$, calls to the C constructor will have a corresponding argument of type T. Pattern matching on an object of type S will thus yield an alternative for C that binds an argument v of type T. After injection, such terms will be of this form:

```
data S = C ··· T ··· -- Another type that contains T

··· C ··· (unpack (pack e)) ··· -- Call of constructor

··· case x of -- Pattern v of type T
      C ··· v ··· → ··· (unpack (pack v)) ···
```

This hoisting transformation treats the pattern match like the body of a function: the type field and its use are modified:

```
data S = C ··· P ··· -- P replaced with packed type T

··· C ··· (pack e) ··· -- Argument is packed

··· case x of -- Pattern v now of type P
      C ··· v ··· → ··· (unpack v) ···   -- Unpack field
```

7.2.4 Simplification

After injection and hoisting, many *pack* and *unpack* calls remain scattered throughout the program. This version would run more slowly than the original since these calls perform redundant computation. Following work on deforestation [59, 129], we perform semantics-preserving transformations to remove these calls and produce a program that operates directly on packed types.

These transformations are largely standard in the functional language community [101]. We follow Jones and Launchbury's descriptions [98].

Variable Inlining If a variable or function v names an expression e, any reference to v may be replaced with e provided variable names are changed to avoid collisions. If v is a local variable, inlining can only occur within its local scope.

Beta Reduction After inlining a function, we have a lambda $(\lambda v_1 \ldots v_n \rightarrow e)$ applied to arguments $a_1 \ldots a_n$; beta reduction replaces every free instance of v_i in e with a_i. We use the notation $e[y/x]$ to indicate y replaces all occurrences of x in e.

$$(\lambda v_1 ... v_n \rightarrow e)\ a_1 ... a_n = e[a_1/v_1, ..., a_n/v_n]$$

Case-of-Case A *case* expression that scrutinizes another *case* can be pushed to each of the inner *case*'s alternatives.

$$\begin{aligned} &\textbf{case}\ (\textbf{case}\ e\ \textbf{of}\ (p_1 \rightarrow e_1) ... (p_k \rightarrow e_k))\ \textbf{of}\ alts \\ &= \textbf{case}\ e\ \textbf{of}\ (p_1 \rightarrow \textbf{case}\ e_1\ \textbf{of}\ alts) ... (p_k \rightarrow \textbf{case}\ e_k\ \textbf{of}\ alts) \end{aligned}$$

Case-of-Pattern If a *case* expression scrutinizes a constructor expression, we replace the *case* with the appropriate alternative expression and perform a pattern substitution.

$$\begin{aligned} &\textbf{case}\ c\ v_1 ... v_k\ \textbf{of}\ ...\ (c\ v_1' ... v_k' \rightarrow e_i)\ ... \\ &= e_i[v_1/v_1', ..., v_k/v_k'] \end{aligned}$$

Our simplification procedure has six steps, described below. After each step of the simplification process, we clean up the program by repetitively applying the Case-of-Case and Case-of-Pattern transformations and the packing identity until reaching a fixed point.

For illustration, we use the following function, which traverses a binary tree of integers, incrementing each element by 1. We assume that injection and hoisting have occurred.

```
data T = B Int | R Int T T
f = λarg → pack (case (unpack arg) of
  B x → B (x+1)
  R x z1 z2 → R (x+1) (unpack (f (pack z1 )))
                      (unpack (f (pack z2 ))))
```

Step 1 Inline any *unpack* call scrutinized by a *case* expression, renaming all variables and patterns within the inlined expression to avoid conflicts. In our example, **case** (*unpack arg*) **of** fulfills this condition, so we rewrite it to

```
case ((λp → case p of
  B′y → B y
  P y1  y2 k1 k2  y3 k3 k4 → R y1 (R y2 (unpack k1) (unpack k2))
                                  (R y3 (unpack k3) (unpack k4))
  U y k1 k2 → R y (unpack k1) (unpack k2)) arg) of ···
```

Beta reduction applies the inlined function to its argument (*arg*), and the "cleanup" phase performs a Case-of-Case and three Case-of-Pattern transformations, giving a function that pattern matches on packed binary trees:

```
f = λarg → pack (case arg of
  B′y → B (y+1)
  P y1  y2 k1 k2  y3 k3 k4 → R (y1+1)
                              (unpack (f (pack (R y2 (unpack k1) (unpack k2 )))))
                              (unpack (f (pack (R y3 (unpack k3) (unpack k4 )))))
  U y k1 k2 → R (y+1) (unpack (f k1)) (unpack (f k2 )))
```

When a function pattern matches on both an argument *v* of a packable type and on any of *v*'s pointer patterns, we have to reapply this step multiple times. Consider a function *g* that pattern matches on a list and on the list's tail. After injection, hoisting, and the application of our packing identity we could have, e.g.,

```
g = λv → ···  case unpack v of
                Nil       → ···
                Cons x xs → ···  case xs of ···
```

After performing Step 1 once, we would then have

```
g = λv → ··· case v of
              PNil          → ···
              PCons x1 x2 xs → ··· case unpack xs of ···
              UCons x xs     → ··· case unpack xs of ···
```

We thus apply this step until no *unpack* calls are scrutinized; if the original function had n nested *cases*, each scrutinizing a pointer pattern from a previous *case*, we will have to apply this step n times.

Step 2 Inline all recursive function calls (again renaming variables and patterns), beta reducing each time. We perform this step multiple times according to the packing factor before applying the cleanup phase. Applying a packing factor of 1 to our f example (i.e., inline every call of f once), we get

```
f = λarg → pack (case arg of
  B′y → B (y+1)
  P y1  y2 k1 k2  y3 k3 k4 → R (y1+1) (case pack (R y2 (unpack k1) (unpack k2)) of
                                        B′··· ; P··· ; U···)
                                       (case pack (R y3 (unpack k3) (unpack k4)) of
                                        B′··· ; P··· ; U···)
  U y k1 k2 → R (y+1) (case k1 of B′··· ; P··· ; U···)
                      (case k2 of B′··· ; P··· ; U···))
```

where B′··· ; P··· ; U··· are the patterns and expressions in the body of f from step 1, i.e., B′y → B (y+1) ; P y1 y2 k1 k2 y3 k3 k4 → ··· .

Step 3 Push functions and data constructors being passed a *case* argument into the *case*. We repeat this until we reach a fixed-point, and also apply it to arguments that are *let* expressions. E.g.,

```
f e (case s of C → a
               D → b) g
```

to

```
case s of C → f e a g
          D → f e b g
```

In our example, expressions of the form R e (**case**···) (**case**···) reside in the P and U alternatives of the top *case*. This step will push the R, e, and second *case* down to each alternative of the first, then the R and its first two arguments will be further pushed into each of the second case's alternatives. Finally, the outer *pack* call will be pushed down to every alternative. We only present the full alternatives that will produce fully packed cells after the last step of the algorithm:

```
f = λarg → case arg of
  B′y → pack (B (y+1))
  P y1  y2 k1 k2  y3 k3 k4 → case pack (R y2 (unpack k1) (unpack k2)) of
    B′··· ; P··· ; U w m1 m2 → case pack (R y3 (unpack k3) (unpack k4)) of
      B′··· ; P··· ; U z n1 n2 → pack (R (y1+1)
                                          (R (w+1) (unpack (f m1)) (unpack (f m2)))
                                          (R (z+1) (unpack (f n1)) (unpack (f n2 ))))
  U y k1 k2 → case k1 of
    B′··· ; P··· ; U w m1 m2 → case k2 of
      B′··· ; P··· ; U z n1 n2 → pack (R (y+1)
                                          (R (w+1) (unpack (f m1)) (unpack (f m2)))
                                          (R (z+1) (unpack (f n1)) (unpack (f n2 ))))
```

Step 4 If we have a variable *v* that defines a packed constructor expression at the top-level *v* = *pack* (*c* . . .), **inline *v*** wherever it is used. This step has the same goal as step 3: repositioning expressions to maximize the number of packed variants generated when we finally inline our *pack* calls.

For example, say we have two variables defined as

```
v1 = pack (R y (unpack z1) (unpack z2))
v2 = pack (R x (unpack v1) (unpack v1))
```

Inlining *v2*'s *pack* call would generate a U variant instead of P. However, if we first inline *v1* and apply our packing identity via the cleanup pass, *v2* becomes

```
v2 = pack (R x (R y (unpack z1) (unpack z2))
               (R y (unpack z1) (unpack z2 )))
```

Step 5 If we have a let-binding **let** *v* = *unpack e* **in** *e'* and all uses of *v* in *e*′ are of the form (*pack v*), **apply our packing identity** by removing these conversion calls. The binding becomes **let** *v* = *e* **in** *e'* and each (*pack v*) in *e*′ becomes *v*.

Step 6 **Inline pack calls**, apply beta reduction, and perform the cleanup pass. We repeat this step until no pack calls remain.

```
f = λarg → case arg of
  B′y → B′ (y+1)
  P y1  y2 k1 k2  y3 k3 k4 →
                  P (y1+1) (y2+1) (f k1) (f k2)
                           (y3+1) (f k3) (f k4)
  U y k1 k2 → case k1 of B′··· ; P··· ;
    U w m1 m2 → case k2 of B′··· ; P··· ;
      U z n1 n2 → P (y+1) (w+1) (f m1) (f m2)
                          (z+1) (f n1) (f n2)
```

Here, the unpacked alternative still contains the nested *case* expressions introduced by the pushing of step 3. While contributing to the overall code growth issue pervasive in any inlining-based optimization, these extra *cases* can help produce packed structures, as seen above when an unpacked cell has unpacked cells as its children. This additional code helps improve performance at the cost of increased area in the final circuitry (detailed in Section 7.3).

7.2.5 Heuristics to Limit Code Growth

Our algorithm's recursive inlining (step 2 of the simplification phase) is key for functions to produce packed data types directly: it introduces additional data constructor operations (e.g., *Cons*) that are consolidated into packed variants. Unfortunately, this inlining leads to a potentially exponential increase in code size (and hence hardware resources) for tree-like types and functions. To retard this increase, we employ heuristics to selectively inline functions whose form leads to packed results and small functions that are cheap to inline.

First, we select functions that generate packable data: those that return a data constructor with an argument that is a recursive call. The *append* function (Section 7.1) has this form:

```
append z y = ··· Cons x xs → Cons x (append xs y)
```

To minimize code growth from inlining such functions, we also insist that either the recursive call does not take any packable arguments (so no *unpack* calls are scrutinized and thus inlined by Step 1 of Section 7.2.4) or at least one of its arguments is a pointer pattern, e.g., the *xs* pattern above. The second requirement is designed to select smaller functions for inlining: if recursive function *f* takes packable arguments, but none of the arguments passed to its recursive call are pointer patterns, then the call's arguments of packable type must be generated by additional calls to other functions. These additional calls clutter *f*'s body; inlining *f* would thus lead to copies of these calls, contributing to its overall growth.

In addition to examples such as *append* (Section 7.1) and the simple tree map (Section 7.2.4), we find this heuristic identifies more subtle functions that make good candidates, such as this *split* function used by a *mergesort* implementation to split a list into a pair of lists of even and odd-numbered elements:

```
split  w = case w of          -- Match on input
 Nil         → (Nil, Nil)
 Cons x xs  → case xs of -- Match on pointer pattern
   Nil          → (w, Nil)
   Cons y ys  → case  split  ys  of -- Recurse on pointer pattern
     (a, b)  → (Cons x a, Cons y b)  -- Return data constructor
```

Our first heuristic does not consider functions that merely traverse packed types (e.g., list length) or tail-recursive functions that accumulate a packable type. To capture these, we also inline functions below a certain "size" according to a variant of the metric employed by the Glasgow Haskell Compiler [99]. We compute a function's size by traversing all of its body's subexpressions, adding 1 to a running total for each of the following constructs encountered: variable, literal, or data constructor expressions (except for variable expressions scrutinized by a *case*); local variable definitions; and data constructor patterns.

7.3 Experimental Evaluation

I tested our algorithm on eleven programs (Table 7.1) and evaluated the speed and size of the circuits that were eventually produced from its output. As was our intent, our algorithm consistently reduced the number of memory accesses (by up to a factor of two at a packing factor of 1). This usually reduced execution times (around 25% for a packing factor of 1) and total memory traffic (bits transferred) at the cost of an increase in circuit area, which follows the size of the generated code.

For list-like data structures, our algorithm is practical at higher packing factors that generate list cells with three or more elements. For tree-like data structures, however, packing factors of 2 or more lead to impractically large circuits (e.g., ten times larger than the baseline) and in some cases overwhelmed the downstream tools used in our experimental framework. This result is understandable since data and code size increases exponentially for tree-like types but only linearly for lists.

7.3.1 Testing Scheme

I implemented our algorithm as a pass in our compiler and set the “size” threshold for our second code growth heuristic (Section 7.2.5) to 25. I used Verilator to generate a C++ simulator from the SystemVerilog output by the compiler and linked it to my memory simulator to gather performance measurements such as the number of simulated cycles and memory accesses. I ran my memory simulator in its “realistic” mode for all experiments (see Section 6.3 for the cache and DRAM parameters simulated).

I ran Intel’s Quartus 15.0 on the generated SystemVerilog for area estimates (Table 7.1). As a result, the area estimates do not consider the area of any memory system, just the core datapath and controller. The area units are ALMs for a midrange Cyclone V 5CSXFC6D6F31C8ES FPGA.

Recall that the compiler implements each recursive type as a bit vector including tag bits to encode the variant and type-specific pointers for recursive references. This implementation covers both our packable types and the compiler-generated types for recursive functions’ continuations. Larger functions tend to need larger continuations to store more free variables; our results here reflect this trend.

Table 7.1 lists the benchmark applications. Append, Length, Filter, and Foldl each traverse a list of integers and, respectively, concatenate it to another list, count the elements, remove even elements, or sum elements. Transpose performs matrix transpose on a list of lists. Life executes 100 steps of the “gridless” version of the Game of Life (from RosettaCode.com). Mergesort sorts a list of integers by splitting, recursing, and merging; Treesort does so by building a binary search tree then building a sorted result from an in-order traversal. DFS searches a binary tree for a value; Treeflip swaps the branches of each node of a tree. Kdfilter is Kanungo et al.’s K-means clustering algorithm [77], used in machine learning and image processing.

Each test generates its own inputs with a recursive function, which our algorithm transforms to produce packed structures. Transpose builds a 16×128 matrix; Life takes a list of points encoding a “glider”; DFS, Treesort, and Treeflip each build a complete binary search tree of 16384 integers; Kdfilter builds a K-d tree from a list of 8192 random 2D points; the remaining tests operate on lists of 16384 random integers.

Test	Size	Runtime	Traffic	Area	Area Increase		
	(LoC)	(cycles)	(KB)	(ALMs)	1	2	3
Append	25	1200k	1400	2500	2.3×	3.2×	4.3×
Length	29	710	790	1700	1.5	2.0	2.8
Foldl	28	690	790	1700	1.5	2.1	3.0
Filter	25	960	1100	2200	2.7	6.9	18
Mergesort	36	19000	27000	6000	5.6		
Transpose	43	240	320	5100	4.7	21	
Life	117	3700	5100	21000	4.6		
DFS	42	1200	1300	2000	3.4	20	
Treesort	39	6100	7500	3500	2.8	13	
Treeflip	44	2300	2900	2600	6.8		
Kdfilter [77]	377	100000	150000	37000	5.2		

Table 7.1: Baseline measurements for my benchmarks and area increases with packing factor. *Size* is lines of code in Haskell source; *Runtime* is simulated execution time of the circuit (in thousands of cycles); *Traffic* is the total amount of memory read and written (in kilobytes); *Area* is the area (in adaptive logic modules for a Cyclone V FPGA). *Area Increase* is the fractional increase under packing factors 1, 2, and 3 (2× is doubling).

7.3.2 Experimental Results

I evaluated the impact of our algorithm on our compiled circuits by constructing a packed version of each benchmark and comparing that to the unpacked (original) baseline. I swept the packing factor from 1 to 3, but the compiler was unable to produce circuits for the larger examples at high packing factors because our algorithm generated functions with more than 64 recursive calls, surpassing a limit imposed by the compiler. Table 7.1 lists baseline numbers (the area measurements were computed with Paolo Mantovani's help); Figure 7.1 shows performance improvements.

I measured the total number of each circuit's cycles to completion, memory accesses, and bits accessed. Memory accesses represent the number of memory requests from the circuit to the memory system; memory bits counts the total number of bits transferred (request sizes vary).

Figure 7.1 depicts these results; smaller bars are better. Each cluster of bars represents a particular metric for a given benchmark; results are normalized to the original, unpacked version of each benchmark. For memory and bit accesses, each bar is partitioned into reads (solid) and writes (open); higher packing factors use darker colors.

The results are promising: packing consistently reduces memory accesses, with reductions from 1.5× to 2× under a packing factor of 1. Increasing the packing factor leads to reductions as high as 4×, but only the simplest list tests achieve this reduction with comparable increases

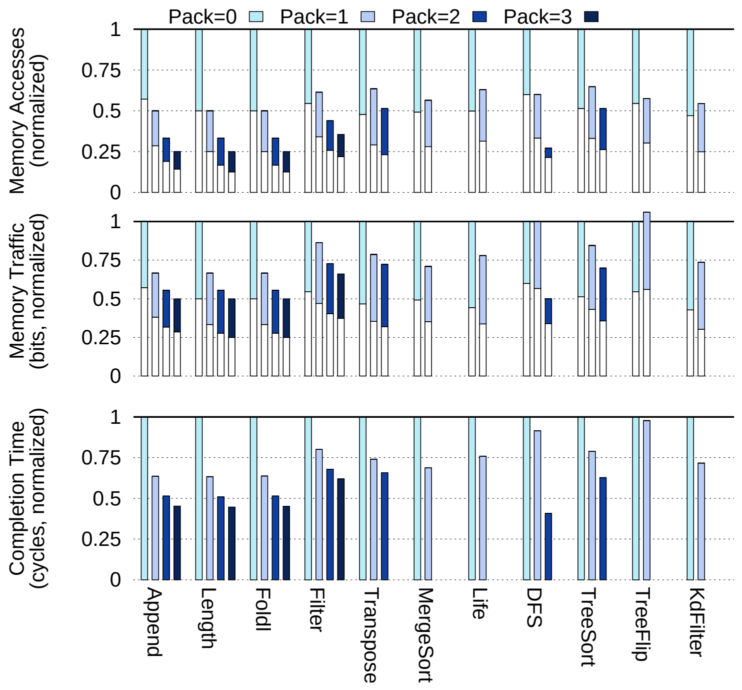


Figure 7.1: Performance under various degrees of packing (shorter is better): total number of memory accesses, total memory traffic in bits, and completion time in cycles. The numbers for each benchmark are normalized to its unpacked case (Table 7.1 lists baselines). For accesses and traffic, solid bars denote reads; open bars are writes.

in area; a packing factor greater than 1 only makes sense if there are few recursive calls and the types being packed are list-like. Regardless, packing also generally reduces the number of bits transferred and cycles to completion.

For Append, Length, Foldl, and Filter, a packing factor of n decreases reads by a factor of $n+1$. This is the maximum expected: the number of list elements the algorithm must consider remains unchanged, yet each packed list cell contains $n+1$ elements and the input list is completely packed.

Filter performs almost as well, but writes some unpacked cells after traversing its input list, reducing the improvement. Increasing the packing factor reduces the likelihood that Filter can generate a fully packed cell, reducing the potential gain.

Packing also decreases the overall number of bits transferred and completion time of these tests, but by a smaller factor than memory traffic. Since packing does not affect the total amount of data (integers) the benchmark must process and store, any reductions in the total number of bits transferred due to packing arises from the elimination of certain pointers in the packed data structures. Because Filter is unable to completely pack its results, packing Filter reduces the total number of bits transferred less than, say, packing Append.

The other, more complex list tests produce more unpacked cells and thus experience smaller performance gains. Mergesort splits its input list recursively, eventually reaching single-element lists which must be stored in unpacked cells, but the subsequent merge procedure builds packed cells (using the nested *case* expressions in the function's unpacked alternative, discussed at the end of Section 7.2.4). Unpacked cells reduce Mergesort's gains by 10% compared to the simplest list tests.

Given a list of lists representing a matrix, Transpose uses one function to build a new row comprising the head of each nested list, another to build a new matrix from the tail of each nested list, and a third to prepend the new row to the new matrix. Although the head and tail collection functions are fully packed, our heuristic from Section 7.2.5 does not select the new row construction function for inlining, leading to more unpacked cells. A similar issue occurs in Life due its use of nested lists; both tests deal with more unpacked cells than Mergesort, leading to more bits accessed and higher completion times. However, both tests still achieve speedups of 1.3×, and experience less growth than Mergesort, showing that our heuristic can trade performance gains for area savings.

The Kdfilter test is our most complex, yet it still achieves memory reductions similar to our list tests even though only some of its data structures are ultimately packed. It consists of two main recursive functions: one constructs a K-d tree from an input list of points, using a *mergesort* function to balance the points across the tree; the other performs Kanungo et al.'s "filtering" variant of the K-means clustering algorithm on the tree. While our algorithm successfully packs the lists passed to the *mergesort* function, our heuristic rejects for inlining both the tree construction function (because it is the wrong form) and the K-d tree filtering algorithm (because it is too large). Nevertheless, our algorithm still achieves nearly a 2× reduction in memory accesses and a 1.3× reduction in memory traffic and completion for this benchmark.

The tree benchmarks vary the most because, ironically, higher packing factors can lead to fewer fully packed tree nodes. The capacity of packed tree nodes grows exponentially with the packing factor: if each node holds a single data element unpacked, they hold three elements at a packing factor of 1, seven at 2, and fifteen at 3. As such, there may be many exceptional cases near the leaves.

Packing trees does reduce the number of cells traversed (and hence the number of accesses), but DFS and Treeflip access more total bits when packed once because the nodes (and continuations implementing the functions' recursion) are much larger and are always read completely. DFS does see a reduction in total bits and execution cycles at a packing factor of 2 (the element being searched for in the tree is found with significantly fewer accesses), but this comes at a massive increase in circuit area.

Treeflip is not so lucky. Under any packing factor, every packed tree cell is read and written by the main recursive flipping function, whose continuations are large and numerous enough that more bits are accessed under packing.

Treesort reads every tree node as it builds a (packed) result list. The list cells and continuations are small enough to yield overall reductions in cycles and bits accessed.

Although these results confirm that our algorithm can improve performance, both memory and absolute time, they are only truly meaningful if the circuit generated is of reasonable size. Table 7.1 shows that all tests comprise a reasonable amount of area under a packing factor of 1, but any higher packing factor only makes sense for the simplest list tests. The exponential growth caused by inlining make tests like Transpose and DFS infeasible at higher packing factors; inlining always has this potential affect when used as an optimization.

Chapter 8

Conclusions and Further Work

8.1 Conclusions

As stated in Chapter 1, my thesis is that pure functional programs exhibiting irregular memory access patterns can be compiled into specialized hardware and optimized for parallelism. This thesis has been supported with the following contributions:

- Program transformations that (1) remove language constructs precluding direct hardware translation and (2) bring pure functional programs closer to a hardware representation.
- A (mostly) syntax-directed translation from a functional IR into patient dataflow networks.
- Compositional dataflow circuits that correctly implement our abstract networks.
- A nondeterministic merge actor enabling pipeline parallelism across recursive calls.
- Buffering heuristics to prevent deadlock in our compiler-generated networks.
- A type-based partitioning scheme for on-chip memory.
- Optimizations for irregular functional programs realized as hardware dataflow networks.

These contributions have been implemented in a working compiler, which translates Haskell programs into latency-insensitive dataflow circuits. The compiler demonstrates that irregular functional programs can be synthesized into hardware circuits; the two optimizations from Chapter 6 and Chapter 7 show how we can enable and exploit more parallelism in these circuits.

This work has thus extended the state-of-the art in high-level synthesis, showing how to synthesize hardware in the face of recursion, dynamic data structures, and irregular memory access patterns.

8.2 Further Work

The rest of this chapter considers how our work could be strengthened and extended to further support my thesis.

8.2.1 Formalism

Most of our contributions concern program transformations or translations between different models of computation. Although we have empirical evidence of their correctness, formal proofs are required to rigorously claim that they do not affect the functionality of the program being modified. We could use the traditional compiler researcher's proof scheme for the compiler passes of Section 3.3 and code transformations in Chapter 6 and Chapter 7: present the operational or denotational semantics for the input language, cast the transformation as rules within the semantic formalism, and show that applying these rules leaves the output of a program unchanged for a given input.

The correctness of our translation from software to dataflow (Section 4.3) is harder to prove, since it connects one model of computation (pure functions) to another (abstract dataflow networks). While formal systems exist to define both ends of the translation (denotational/operational semantics for the source; Kahn Process Networks for the target), it is not obvious how to connect the two mathematically without a new formalism capturing both models.

My intuition suggests that we can prove that our compiler-generated networks are deadlock-free and deterministic. Proving deadlock-freedom for arbitrary dataflow networks is undecidable [17], but we generate specific parameterized subnetworks for each language construct found in a Floh program. It seems likely that our networks are deadlock-free when every channel has at least one buffer on it.

As explained in Chapter 5, our use of a nondeterministic merge node prevents the use of Kahn's formalism to claim determinism for our overall network behavior. Empirically, we have not witnessed any functional nondeterminism in a network's behavior for a given input and buffering scheme, i.e., feeding the same input into a well-buffered network (enough buffers to prevent deadlock) always produces the same output. To prove that a such a network is functionally deterministic, we either require a different, non-Kahn formalism to describe network behavior (perhaps using the Colored Petri Net model [73]) or a way to circumvent the need for a nondeterministic merge node in our networks.

8.2.2 Extending the Compiler

Our compiler currently operates on a subset of modern Haskell; future work could extend it to handle more language features. The clearest extension is to admit higher-order functions, one of the key constructs in Haskell that enables higher programmer productivity. Defunctionalization [37] translates these functions into first-order form, which our compiler already handles. A preliminary defunctionalization pass exists in our compiler (implemented by Lizzie Paquette) and can successfully transform some simple higher-order functions like *map* and *foldl*, but additional work is necessary to handle arbitrary higher-order functions and higher-order (or partially applied) data constructors.

Compiling some Haskell features like exceptions and I/O introduces built-in primitive functions with no clear hardware analogue. Since most of the benchmarks in Haskell's standard "nofib" testsuite are driven with I/O functions, we are unable to compile any of them directly, severely limiting our ability to test our compiler on complex, peer-approved programs. Our compiler should be extended to recognize these constructs and either replace them with hardware-facing components (e.g., change a function that waits for user input to a dataflow actor waiting for an input token from the environment) or ask the user to provide a set of constant values to feed into the circuit whenever it expects input from the environment.

8.2.3 Synthesizing a Realistic Memory System

The compiler has two memory systems that it can target, but neither is wholly preferable. The first, presented in Section 5.6.5, associates each recursive type with an on-chip bit vector array memory; each memory has the same number of slots (each tailored to the associated type's bit-width), a private address space, and operates without a backing store or the ability to free memory. This system is synthesizable, but severely limits the size of the data structures generated by a network (due to the lack of off-chip backing storage). The second system is our default cache-based hierarchy (Section 6.2), which admits much larger data structures but uses a uniform memory representation for heap objects and only operates in simulation.

We envision a pairing of these two systems with further augmentations to take advantage of an FPGAs customizability and high memory bandwidth. This ideal memory system would provide type-based on-chip memories with object-specific bit-widths (e.g., a 66-bit list object would be stored in a 66-bit memory slot, instead of a 96-bit slot aligned to uniform 32-bit values) and avoid the energy-intensive logic required by traditional cache implementations. Instead, static analysis paired with profiling would guide the transfer of data between on- and off-chip memory to minimize high-latency trips off-chip (Dominguez et al. [42] propose a similar technique), and we would marshal the irregularly sized data in each on-chip memory into a uniform representation for off-chip DRAM.

Throughout this work, we assume the presence of a garbage collector (GC) in hardware for our memory system. Others in our research group are currently developing a hardware implementation of a stop-the-world mark-and-sweep collection [132]: it uses special dataflow buffers to pass any pointers floating in a network off to the collector when GC is triggered, follows these pointers to determine which memory objects are reachable (marking any reached), sweeps the system to free any memory storing unmarked data, and sends a signal to the special buffers that allows them to continue firing as before.

8.2.4 Improved and Additional Optimizations

The two optimizations presented in this thesis vary in their effectiveness across different kinds of programs; they both have potential for improvement.

The results in Section 6.3 suggest that our divide-and-conquer optimization works best on algorithms that evenly divide their workloads (e.g., splitting a list in half or traversing the branches of a balanced binary tree). If the workload is unbalanced, we waste the extra memory bandwidth provided by our partitioned memory system and introduce excessive overhead with our type conversion functions. A new architecture based on task-level abstraction (e.g., similar to that of the ParallelXL [22] or TAPAS [89] systems discussed in Chapter 2) could remove these inefficiencies by dynamically load balancing the data structure across independent dataflow networks, removing any need for the type conversion functions. This would require a significant change to our translation algorithm, though, which would no longer be syntax-directed (additional machinery would be added to pass data between independent networks).

Our packing algorithm has the potential to increase a tree-based program's memory traffic and exponentially increase its area requirements at higher packing factors. If the original program did not traverse every tree node (e.g., as in depth-first search), the packed version will access more data (usually unnecessarily) than the original; a preprocessing step could be added to the algorithm to detect these kinds of functions statically (e.g., by determining whether all branches are passed to recursive calls), and avoid selecting them for packing. When the full structure is traversed, the packed heap accesses should not increase memory traffic (same amount of data, fewer pointers), but the continuations implementing function recursion may get excessively large. These continuations store the free variables found in a function, even if they refer to the same value. Sometimes they also hold multiple values which are eventually combined with an arithmetic or boolean operation. These issues could be solved by modifying the recursion removal pass (Section 3.3.4) to find common values bound to different variables and remove them, or apply any combining operations to free variables earlier to decrease the size of the continuations and lower the overall memory footprint.

The exponential area increase caused by higher packing factors is a more difficult problem to solve. Any optimization involving inlining is guaranteed to yield code growth, which corresponds to larger circuits. A more selective packing scheme could alleviate this issue, e.g., when packing a nested structure like a list of lists, only pack one level of the structure. For structures with multiple recursive references like trees, we could employ a "linear" packing scheme where only one branch is packed; this would lead to only one recursive call being inlined and should entail linear area increases as the packing factor grows. Unfortunately, such a scheme would involve load-balancing issues similar to those present in our divide-and-conquer algorithm; further research is needed to determine how to tackle these issues.

Even with the mentioned issues, the two optimizations presented in this dissertation generally improve our networks' performance; further gains could be achieved with additional optimizations. Our buffering heuristics only target network correctness; a new heuristic could additionally improve performance by exploiting more pipeline parallelism. This heuristic could leverage a formalism to estimate the maximum throughput potential of a network, and assign buffers to specific channels to realize that potential. Collins and Carloni [26] used *marked graphs* to achieve this goal for their latency-insensitive systems, but we cannot use their technique due to our use of data-dependent and nondeterministic actors found in our networks. Regardless, their method could be a starting point for future work on more performance-focused buffering.

Our compiler-generated dataflow networks can only service a single call to a recursive function at a time. We could enable more pipeline parallelism by servicing multiple calls simultaneously, but the network would need to distinguish tokens from different calls to avoid erroneous computation. One possible solution could leverage the work of Dennis [39] and Zebelein et al. [140]: colored tokens allow multiple simultaneous invocations of a function and allow tokens of different colors to overtake each other. Since two calls may finish out-of-order, we would need reorder buffers to ensure the functional correctness of the network. This solution would increase the area overhead of our circuits, but could entail significant performance increases.

Bibliography

[1] Mythri Alle, Antoine Morvan, and Steven Derrien. Runtime dependency analysis for loop pipelining in high-level synthesis. In *Proceedings of the 50th Design Automation Conference*. ACM, 2013.

[2] ARM. *AMBA 4 AXI4-Stream Protocol Specification Version 1.0*, March 2010. ARM IHI 0051A (ID030510).

[3] Arvind and R.S. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Transactions on Computers*, 39(3):300–318, March 1990.

[4] Arvind, R.S. Nikhil, D.L. Rosenband, and N. Dave. High-level synthesis: an essential ingredient for designing complex ASICs. In *Proceedings of the International Conference on Computer Aided Design (ICCAD)*, pages 775–782, San Jose, California, November 2004.

[5] Christiaan Baaij. Clash.tutorial, 2017. [Online; accessed 08-April-2019].

[6] Christiaan Baaij, Matthijs Kooijman, Jan Kuper, Arjan Boeijink, and Marco Gerards. Cλash: Structural descriptions of synchronous hardware using Haskell. In *Proceedings of the Euromicro Conference on Digital System Design (DSD)*, pages 714–721, Lille, France, September 2010.

[7] Christiaan Baaij and Jan Kuper. Using rewriting to synthesize functional languages to digital circuits. In *Proceedings of Trends in Functional Programming (TFP)*, volume 8322 of *Lecture Notes in Computer Science*, pages 17–33, Provo, Utah, 2014. Springer.

[8] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: Constructing hardware in a scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 1216–1225, New York, NY, USA, 2012. ACM.

[9] David F. Bacon, Perry Cheng, and Sunil Shukla. Parallel real-time garbage collection of multiple heaps in reconfigurable hardware. In *Proceedings of the International Symposium on Memory Management (ISMM)*, pages 117–127, Edinburgh, United Kingdom, 2014. ACM.

[10] Twan Basten and Jan Hoogerbrugge. Efficient execution of process networks. In Alan Chalmers, Majid Mirmehdi, and Henk Muller, editors, *Communicating Process Architectures (CPA)*, pages 1–14, Bristol, UK, September 2001. IOS Press.

[11] Yosi Ben-Asher and Nadav Rotem. Automatic memory partitioning: Increasing memory parallelism via data structure partitioning. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 155–162, Scottsdale, Arizona, 2010. ACM.

[12] Endri Bezati, Marco Mattavelli, and Jörn W. Janneck. High-level synthesis of dataflow programs for signal processing systems. In *Proceedings of the International Symposium on Image and Signal Processing and Analysis (ISPA)*, pages 750–755, Trieste, Italy, September 2013. The Institute of Electrical and Electronics Engineers (IEEE).

[13] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware design in Haskell. In *Proceedings of the International Conference on Functional Programming (ICFP)*, pages 174–184, Baltimore, Maryland, 1998.

[14] Shekhar Borkar and Andrew A. Chien. The future of microprocessors. *Commun. ACM*, 54(5):67–77, May 2011.

[15] Anastasia Braginsky and Erez Petrank. Locality-conscious lock-free linked lists. In Marcos K. Aguilera, Haifeng Yu, Nitin H. Vaidya, Vikram Srinivasan, and Romit R. Choudhury, editors, *Distributed Computing and Networking*, volume 6522 of *Lecture Notes in Computer Science*, pages 107–118. Springer Berlin Heidelberg, 2011.

[16] Manfred Broy. Nondeterministic data flow programs: How to avoid the merge anomaly. *Science of Computer Programming*, 10(1):65–85, February 1988.

[17] Joseph Tobin Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory using the Token Flow Model.* PhD thesis, University of California, Berkeley, 1993. Available as UCB/ERL M93/69.

[18] Bingyi Cao, Kenneth A. Ross, Martha A. Kim, and Stephen A. Edwards. Implementing latency-insensitive dataflow blocks. In *Proceedings of the International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pages 179–187, Austin, Texas, September 2015. The Institute of Electrical and Electronics Engineers (IEEE).

[19] Luca P. Carloni. The role of back-pressure in implementing latency-insensitive systems. *Electronic Notes in Theoretical Computer Science*, 146(2):61–80, 2006.

[20] Luca P. Carloni, Kenneth L. McMillan, and Alberto L. Sangiovanni-Vincentelli. Theory of latency-insensitive design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(9):1059–1076, September 2001.

[21] Josep Carmona, Jordi Cortadella, Mike Kishinevsky, and Alexander Taubin. Elastic circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(10):1437–1455, Oct 2009.

[22] Tao Chen, Shreesha Srinath, Christopher Batten, and G Edward Suh. An architectural framework for accelerating dynamic parallel algorithms on reconfigurable hardware. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 55–67. IEEE, 2018.

[23] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 269–284, New York, NY, USA, 2014. ACM.

[24] Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33(2):346–366, April 1932.

[25] Alessandro Cilardo and Luca Gallo. Improving multibank memory access parallelism with lattice-based partitioning. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(4):45, 2015.

[26] R. L. Collins and L. P. Carloni. Topology-based optimization of maximal sustainable throughput in a latency-insensitive system. Technical Report CUCS–008–07, Columbia University, Department of Computer Science, New York, New York, USA, February 2007.

[27] Rebecca L. Collins and Luca P. Carloni. Topology-based performance analysis and optimization of latency-insensitive systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(12):2277–2290, December 2008.

[28] Rebecca L. Collins, Bharadwaj Vellore, and Luca P. Carloni. Recursion-driven parallel code generation for multi-core platforms. In *Proceedings of Design, Automation, and Test in Europe (DATE)*, pages 190–195, Dresden, Germany, March 2010. EDAA.

[29] Jason Cong, Wei Jiang, Bin Liu, and Yi Zou. Automatic memory partitioning and scheduling for throughput and power optimization. In *Proceedings of the International Conference on Computer Aided Design (ICCAD)*, pages 697–704, San Jose, California, November 2009.

[30] Jason Cong, Wei Jiang, Bin Liu, and Yi Zou. Automatic memory partitioning and scheduling for throughput and power optimization. *ACM Transactions on Design Automation of Electronic Systems*, 16(2), March 2011. Article No. 15.

[31] Jason Cong, Peng Li, Bingjun Xiao, and Peng Zhang. An optimal microarchitecture for stencil computation acceleration based on non-uniform partitioning of data reuse buffers. In *Proceedings of the 51st Design Automation Conference*, pages 1–6, San Francisco, California, 2014. ACM.

[32] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. High-level synthesis for FPGAs: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491, April 2011.

[33] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms.* McGraw-Hill, New York, 2001.

[34] Jordi Cortadella, Marc Galceran-Oms, and Mike Kishinevsky. Elastic systems. In *Proceedings of the International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pages 149–158, Grenoble, France, July 2010. The Institute of Electrical and Electronics Engineers (IEEE).

[35] Jordi Cortadella, Mike Kishinevsky, and Bill Grundmann. Self: Specification and design of synchronous elastic circuits. In *ACM International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, San Jose, California, February 2006. Association for Computing Machinery.

[36] Steve Dai, Ritchie Zhao, Gai Liu, Shreesha Srinath, Udit Gupta, Christopher Batten, and Zhiru Zhang. Dynamic hazard resolution for pipelining irregular loops in high-level synthesis. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 189–194. ACM, February 2017.

[37] Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. In *Proceedings of Principles and Practice of Declarative Programming (PPDP)*, pages 162–174, New York, NY, USA, 2001. ACM.

[38] Robert H. Dennard, Fritz H. Gaensslen, Hwa-Nien Yu, V. Leo Rideout, Ernest Bassous, and Andre R. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, October 1974.

[39] Jack B. Dennis. First version of a data flow procedure language. In *Programming Symposium*, volume 19 of *Lecture Notes in Computer Science*, pages 362–376, Paris, France, April 1974. Springer.

[40] Giorgos Dimitrakopoulos, Anastasios Psarras, and Ioannis Seitanidis. *Microarchitecture of Network-on-Chip Routers: A Designer's Perspective.* Springer, Heidelberg, Germany, 2015.

[41] Julian Dolby. Automatic inline allocation of objects. In *Proceedings of Program Language Design and Implementation (PLDI)*, pages 7–17, New York, New York, May 1997. Association for Computing Machinery.

[42] Angel Dominguez, Sumesh Udayakumaran, and Rajeev Barua. Heap data allocation to scratch-pad memory in embedded systems. *Journal of Embedded Computing*, 1(4):521–540, 2005.

[43] Stephen A. Edwards, Richard Townsend, Martha Barker, and Martha A. Kim. Compositional dataflow circuits. *ACM Transactions on Embedded Computing Systems*, 18(1):5, February 2019.

[44] Stephen A. Edwards, Richard Townsend, and Martha A. Kim. Compositional dataflow circuits. In *Proceedings of the International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pages 175–184, Vienna, Austria, September 2017. Association for Computing Machinery.

[45] Johan Eker and Jörn W. Janneck. CAL language report: Specification of the CAL actor language. Technical Report UCB/ERL M03/48, EECS Department, University of California, Berkeley, December 2003.

[46] Hadi Esmaeilzadeh, Emily Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 365–376, San Jose, California, June 2011.

[47] Joachim Falk, Christian Haubelt, and Jürgen Teich. Efficient representation and simulation of model-based designs in systemc. In *Forum on Specification and Design Languages (FDL)*, volume 6, pages 129–134, Darmstadt, Germany, September 2006. ECSI.

[48] Leonidas Fegaras and Andrew Tolmach. Using compact data representations for languages based on catamorphisms. Technical Report 95-025, Oregon Graduate Institute, 1995.

[49] Matthew Fluet. Monomorphise, January 2015. http://mlton.org/Monomorphise [Online; accessed 23-January-2015].

[50] Simon Frankau. Hardware synthesis from a stream-processing functional language. Technical Report UCAM–CL–TR–824, Cambridge University, 2012.

[51] Simon Frankau and Alan Mycroft. Stream processing hardware from functional language specifications. In *Proceedings of the Hawaii International Conference on System Sciences*. The Institute of Electrical and Electronics Engineers (IEEE), 2003. 10 pp.

[52] Daniel P. Friedman and Mitchell Wand. *Essentials of Programming Languages*. MIT Press, third edition, 2008.

[53] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of Program Language Design and Implementation (PLDI)*, pages 212–233, FIXME, June 1998.

[54] Peter Gammie. Synchronous digital circuits as functional programs. *ACM Computing Surveys*, 46(2):article 21, November 2013.

[55] G. R. Gao, R. Govindarajan, and Prakash Panangaden. Well-behaved dataflow programs for DSP computation. In *Proceedings of the International Conference on Acoustics, Speech, & Signal Processing (ICASSP)*, volume 5, pages 561–564, San Francisco, California, March 1992. The Institute of Electrical and Electronics Engineers (IEEE).

[56] Marc Geilen and Twan Basten. Requirements on the execution of Kahn process networks. In *Proceedings of the European Symposium on Programming (ESOP)*, volume 2618 of *Lecture Notes in Computer Science*, pages 319–334, Warsaw, Poland, April 2003. Springer.

[57] Marc Geilen, Twan Basten, and Sander Stuijk. Minimising buffer requirements of synchronous dataflow graphs with model checking. In *Proceedings of the 42nd Design Automation Conference*, pages 819–824, Anaheim, California, 2005. ACM.

[58] Marc Geilen and Sander Stuijk. Worst-case performance analysis of synchronous dataflow scenarios. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 125–134, Scottsdale, Arizona, October 2010. ACM.

[59] Andrew Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. In *Proceedings of Functional Programming Languages and Computer Architecture (FPCA)*, pages 223–232, Copenhagen, Denmark, June 1993.

[60] Andy Gill, T. Bull, Garrin Kimmell, Erik Perrins, Ed Komp, and B. Werling. Introducing Kansas Lava. In *Proceedings of the International Symposium on Implementation and Application of Functional Languages*, volume 6041 of *Lecture Notes in Computer Science*, November 2009.

[61] Andy Gill, Tristan Bull, Andrew Farmer, Garrin Kimmell, and Ed Komp. Types and associated type families for hardware simulation and synthesis: The internals and externals of Kansas Lava. *Higher-Order and Symbolic Computation*, pages 1–20, 2013.

[62] Manish Gupta, Sayak Mukhopadhyay, and Navin Sinha. Automatic parallelization of recursive procedures. *International Journal of Parallel Programming*, 28(6):537–562, 2000.

[63] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[64] Cordelia V. Hall. Using Hidley-Milner type inference to optimise list representation. In *Proceedings of the ACM Symposium on LISP and Functional Programming (LFP)*, pages 162–172, Orlando, Florida, June 1994.

[65] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, Hiroaki Takada, and Katuya Ishii. CHStone: A benchmark program suite for practical C-based high-level synthesis. In *iscas*, pages 1192–1195, Seattle, Washington, May 2008.

[66] Pieter H. Hartel, Theo C. Ruys, and Marc C. W. Geilen. Scheduling optimisations for SPIN to minimise buffer requirements in synchronous data flow. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 161–170, Portland, Oregon, October 2008. The Institute of Electrical and Electronics Engineers (IEEE).

[67] Christian Haubelt, Joachim Falk, Joachim Keinert, Thomas Schlichter, Martin Streubühr, Andreas Deyhle, Andreas Hadert, and Jürgen Teich. A SystemC-based design methodology for digital signal processing systems. *EURASIP Journal on Embedded Systems*, 2007(1), 2007. Article ID 47580.

[68] Haruo Hosoya and Benjamin Pierce. Regular expression pattern matching for XML. *ACM SIGPLAN Notices*, 36(3):67–80, 2001.

[69] Haruo Hosoya and Benjamin C. Pierce. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology (TOIT)*, 3(2):117–148, May 2003.

[70] Intel Corporation, Santa Clara, California. *8008 8-Bit Parallel Central Processor Unit Users Manual*, November 1972.

[71] Jörn W. Janneck, Ian D. Miller, David B. Parlour, Ghislain Roquier, and Matthieu Wipliez Mickaël Raulet. Synthesizing hardware from dataflow programs: An MPEG-4 simple profile decoder case study. *Journal of Signal Processing Systems*, 63(2):241–249, July 2009.

[72] Jörn W. Janneck, Ian D. Miller, David B. Parlour, Ghislain Roquier, Matthieu Wipliez, and Mickaël Raulet. Synthesizing hardware from dataflow programs. *Journal of Signal Processing Systems*, 63(2):241–249, May 2011.

[73] Kurt Jensen. Coloured petri nets. In *Petri nets: central models and their properties*, pages 248–299. Springer, 1987.

[74] Thomas Johnsson. Lambda lifting: Transforming programs to recursive equations. In *Proceedings of Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 190–203, Nancy, France, 1985. Springer.

[75] Lana Josipović, Radhika Ghosal, and Paolo Ienne. Dynamically scheduled high-level synthesis. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 127–136. ACM, 2018.

[76] Gilles Kahn. The semantics of a simple language for parallel programming. In *Information Processing 74: Proceedings of IFIP Congress 74*, pages 471–475, Stockholm, Sweden, August 1974. North-Holland.

[77] Tapas Kanungo, David M Mount, Nathan S Netanyahu, Christine D Piatko, Ruth Silverman, and Angela Y Wu. An efficient k-means clustering algorithm: Analysis and implementation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(7):881–892, July 2002.

[78] Joachim Keinert, Martin Streubühr, Thomas Schlichter, Joachim Falk, Jens Gladigau, Christian Haubelt, Jürgen Teich, and Michael Meredith. Systemcodesigner—an automatic esl synthesis approach by design space exploration and behavioral synthesis for streaming applications. *ACM Transactions on Design Automation of Electronic Systems*, 14(1):1:1–1:23, January 2009.

[79] Morihiro Kuga, Kansuke Fukuda, Motoki Amagasaki, Masahiro Iida, and Toshinori Sueyoshi. High-level synthesis based on parallel design patterns using a functional language. In *Proceedings of the 8th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies*, HEART2017, pages 23:1–23:6, New York, NY, USA, 2017. ACM.

[80] Milind Kulkarni, Martin Burstcher, Călin Caşcaval, and Keshav Pingali. Lonestar: A suite of parallel irregular programs. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 65–76, Boston, Massachusetts, April 2009.

[81] Edward A. Lee and Eleftherios Matsikoudis. The semantics of dataflow with firing. In *From Semantics to Computer Science: Essays in memory of Gilles Kahn*, chapter 4, pages 71–94. Cambridge University Press, Cambridge, UK, 2008.

[82] Edward A. Lee and Thomas M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, May 1995.

[83] Cheng-Hong Li, Rebecca Collins, Sampada Sonalkar, and Luca P. Carloni. Design, implementation, and validation of a new class of interface circuits for latency-insensitive design. In *Proceedings of the International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pages 13–22, Nice, France, May 2007. The Institute of Electrical and Electronics Engineers (IEEE).

[84] Feng Liu, Soumyadeep Ghosh, Nick P Johnson, and David I August. CGPA: Coarse-grained pipelined accelerators. In *Proceedings of the 51st Design Automation Conference*, pages 1–6, San Francisco, California, 2014. ACM.

[85] H-W Loidl et al. Comparing parallel functional languages: Programming and performance. *Higher-Order and Symbolic Computation*, 16(3):203–251, 2003.

[86] Zhonghai Lu, Ingo Sander, and Axel Jantsch. A case study of hardware and software synthesis in ForSyDe. In *Proceedings of the International Symposium on System Synthesis (ISSS)*, pages 86–91, Kyoto, Japan, 2002. ACM.

[87] Wayne Luk, Teddy Wu, and Ian Page. Hardware-software codesign of multidimensional programs. In *Proceedings of the Symposium on FPGAs for Custom Computing Machines (FCCM)*, pages 82–90, Napa Valley, California, April 1994. IEEE, IEEE.

[88] Paolo Mantovani, Giuseppe Di Guglielmo, and Luca P. Carloni. High-level synthesis of accelerators in embedded scalable platforms. In *Proceedings of the Asia South Pacific Design Automation Conference (ASP-DAC)*, pages 204–211. The Institute of Electrical and Electronics Engineers (IEEE), 2016.

[89] Steven Margerm, Amirali Sharifian, Apala Guha, Arrvindh Shriraman, and Gilles Pokam. TAPAS: Generating parallel accelerators from parallel programs. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 245–257. IEEE, 2018.

[90] Chenyue Meng, Shouyi Yin, Peng Ouyang, Leibo Liu, and Shaojun Wei. Efficient memory partitioning for parallel data access in multidimensional arrays. In *Proceedings of the 52nd Annual Design Automation Conference*, page 160. ACM, 2015.

[91] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.

[92] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, April 19 1965.

[93] Orlando Moreira, Twan Basten, Marc Geilen, and Sander Stuijk. Buffer sizing for rate-optimal single-rate dataflow scheduling revisited. *IEEE Transactions on Computers*, 59(2):188–201, 2010.

[94] Kazutaka Morita, Akimasa Morihata, Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. Automatic inversion generates divide-and-conquer parallel programs. *Proceedings of Program Language Design and Implementation (PLDI)*, 42(6):146–155, 2007.

[95] Alan Mycroft and Richard W. Sharp. Hardware synthesis using SAFL and application to processor design. In *Proceedings of Correct Hardware Design and Verification Methods (CHARME)*, volume 2144 of *Lecture Notes in Computer Science*, pages 13–39, Livingston, Scotland, September 2001.

[96] R. Nane, V. M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels. A survey and evaluation of FPGA high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(10):1591–1604, Oct 2016.

[97] Thomas M. Parks. *Bounded Scheduling of Process Networks*. PhD thesis, University of California, Berkeley, 1995. Available as UCB/ERL M95/105.

[98] Simon L. Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In J. Hughes, editor, *Proceedings of the Conference on Functional Programming and Computer Architecture*, pages 636–666, Cambridge, Massachussets, USA, 26–28 August 1991. Springer-Verlag LNCS523.

[99] Simon L. Peyton Jones and Simon Marlow. Secrets of the Glasgow Haskell Compiler inliner. *Journal of Functional Programming*, 12:393–434, September 2002.

[100] Simon L. Peyton Jones and André L.M. Santos. Compilation by transformation in the Glasgow Haskell Compiler. In Kevin Hammond, DavidN. Turner, and PatrickM. Sansom, editors, *Functional Programming, Glasgow 1994*, Workshops in Computing, pages 184–204. Springer London, 1995.

[101] Simon L. Peyton Jones and André L.M. Santos. A transformation-based optimiser for Haskell. In *Science of Computer Programming*, pages 3–47. Elsevier, 1998.

[102] Keshav Pingali and Arvind. Efficient demand-driven evaluation. part 1. *ACM Transactions on Programming Languages and Systems*, 7(2):311–333, 1985.

[103] Kenneth Platz, Neeraj Mittal, and Subbarayan Venkatesan. Practical concurrent unrolled linked lists using lazy synchronization. In MarcosK. Aguilera, Leonardo Querzoni, and Marc Shapiro, editors, *Principles of Distributed Systems*, volume 8878 of *Lecture Notes in Computer Science*, pages 388–403. Springer International Publishing, 2014.

[104] Rafael Trapani Possignolo, Elnaz Ebrahimi, Haven Skinner, and Jose Renau. Fluid pipelines: Elastic circuitry meets out-of-order execution. In *Proceedings of the IEEE International Conference on Computer Design (ICCD)*, pages 233–240. The Institute of Electrical and Electronics Engineers (IEEE), October 2016.

[105] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 13–24, Piscataway, NJ, USA, 2014. IEEE Press.

[106] Ingmar Raa. Recursive functional hardware descriptions using CλaSH. Master's thesis, University of Twente, The Netherlands, November 2015.

[107] Brandon Reagen, Robert Adolf, Yakun Sophia Shao, Gu-Yeon Wei, and David Brooks. Machsuite: Benchmarks for accelerator design and customized architectures. In *Proceedings of the International Symposium on Workload Characterization (IISWC)*, pages 110–119. The Institute of Electrical and Electronics Engineers (IEEE), 2014.

[108] M. W. Roberts and P. K. Lala. Algorithm to detect reconvergent fanouts in logic circuits. *IEE Proceedings E - Computers and Digital Techniques*, 134(2):105–111, March 1987.

[109] Radu Rugina and Martin Rinard. Automatic parallelization of divide and conquer algorithms. In *Proceedings of Principles and Practice of Parallel Programming (PPoPP)*, pages 72–83, Atlanta, Georgia, 1999. ACM, ACM.

[110] Radu Rugina and Martin Rinard. Recursion unrolling for divide and conquer programs. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing (LCPC)*, volume 2017 of *Lecture Notes in Computer Science*, pages 34–48, Yorktown Heights, New York, August 2000.

[111] Xavier Saint-Mleux, Marc Feeley, and Jean-Pierre David. SHard: a Scheme to hardware compiler. In *Proceedings of the Scheme and Functional Programming Workshop (SFPW)*, pages 39–49, Portland, Oregon, September 2006. University of Chicago technical report TR–2006–06.

[112] Ingo Sander. *System Modeling and Design Refinement in ForSyDe.* PhD thesis, Royal Institute of Technology, Stockholm, Sweden, April 2003.

[113] Ingo Sander and Axel Jantsch. System synthesis based on a formal computational model and skeletons. In *Proceedings of the IEEE Computer Society Workshop on VLSI*, pages 32–39, Orlando, Florida, April 1999. The Institute of Electrical and Electronics Engineers (IEEE).

[114] Tao B. Schardl, William S. Moses, and Charles E. Leiserson. Tapir: Embedding fork-join parallelism into LLVM's intermediate representation. In *Proceedings of Principles and Practice of Parallel Programming (PPoPP)*, PPoPP '17, pages 249–265, New York, NY, USA, 2017. ACM.

[115] Charles L. Seitz. System timing. In Carver Mead and Lynn Conway, editors, *Introduction to VLSI Systems*, chapter 7, pages 218–262. Addison-Wesley, Reading, Massachusetts, 1980.

[116] Zhong Shao, John H. Reppy, and Andrew W. Appel. Unrolling lists. In *Proceedings of the ACM Symposium on LISP and Functional Programming (LFP)*, pages 185–195, Orlando, Florida, June 1994.

[117] Richard W. Sharp and Alan Mycroft. The FLaSH compiler: Efficient circuits from functional specifications. Technical Report tr.2000.3, AT&T Laboratories Cambridge, 2000.

[118] Richard W. Sharp and Alan Mycroft. A higher-level language for hardware synthesis. In *Proceedings of Correct Hardware Design and Verification Methods (CHARME)*, pages 228–243. Springer, 2001.

[119] Mary Sheeran. μFP, an algebraic VLSI design language. In *Proceedings of the ACM Symposium on LISP and Functional Programming (LFP)*, pages 104–112, Austin, Texas, August 1984.

[120] Guy L. Steele. Rabbit: A compiler for Scheme. Technical Report AI-TR-474, MIT Press, 1978.

[121] Sander Stuijk, Marc C.W. Geilen, and Twan Basten. Throughput-buffering trade-off exploration for cyclo-static and synchronous dataflow graphs. *IEEE Transactions on Computers*, 57(10):1331–1345, 2008. Special section on Programming Models and Architectures for Embedded Systems.

[122] Mingxing Tan, Gai Liu, Ritchie Zhao, Steve Dai, and Zhiru Zhang. ElasticFlow: A complexity-effective approach for pipelining irregular loop nests. In *Proceedings of the International Conference on Computer Aided Design (ICCAD)*, pages 78–85, Austin, Texas, November 2015. IEEE.

[123] Richard Thavot, Romuald Mosqueron, Julien Dubois, and Marco Mattavelli. Hardware synthesis of complex standard interfaces using CAL dataflow descriptions. In *Proceedings of Design and Architectures for Signal and Image Processing (DASIP)*, pages 127–134, Sophia Antipolis, France, October 2009. ECSI.

[124] Andrew Tolmach, Tim Chevalier, and The GHC Team. An external representation for the GHC core language (for GHC 6.10), April 2010.

[125] Richard Townsend, Martha A. Kim, and Stephen A. Edwards. From functional programs to pipelined dataflow circuits. In *Proceedings of Compiler Construction (CC)*, pages 76–86, Austin, Texas, February 2017. ACM.

[126] Stavros Tripakis, Rhishikesh Limaye, Kaushik Ravindran, and Guoqiang Wang. On tokens and signals: Bridging the semantic gap between dataflow models and hardware implementations. In *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV)*, pages 51–58, Samos, Greece, July 2014. The Institute of Electrical and Electronics Engineers (IEEE).

[127] Rob V. van Nieuwpoort, Thilo Kielmann, and Henri E. Bal. Satin: Efficient parallel divide-and-conquer in java. In *European Conference on Parallel Processing (Euro-Par)*, volume 1900 of *Lecture Notes in Computer Science*, pages 690–699, Munich, Germany, August 2000. Springer.

[128] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. Conservation cores: Reducing the energy of mature computations. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 205–218, Pittsburgh, Pennsylvania, March 2010.

[129] Philip Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, June 1990.

[130] Yuxin Wang, Peng Li, Peng Zhang, Chen Zhang, and Jason Cong. Memory partitioning for multidimensional arrays in high-level synthesis. In *Proceedings of the 50th Design Automation Conference*, Austin, Texas, June 2013. ACM.

[131] Douglas R. White and Mark Newman. Fast approximation algorithms for finding node-independent paths in networks. Technical Report 01–07–035, Santa Fe Institute, New Mexico, 2001.

[132] Paul R. Wilson. Uniprocessor garbage collection techniques. In Yves Bekkers and Jacques Cohen, editors, *Memory Management*, volume 637 of *Lecture Notes in Computer Science*, pages 1–42. Springer Berlin Heidelberg, 1992.

[133] Felix Winterstein, Samuel Bayliss, and George A Constantinides. High-level synthesis of dynamic data structures: A case study using Vivado HLS. In *Proceedings of Field-Programmable Technology (FPT)*, pages 362–365. The Institute of Electrical and Electronics Engineers (IEEE), 2013.

[134] Felix Winterstein, Kermin E Fleming, Hsin-Jung Yang, and George A Constantinides. Custom multicache architectures for heap manipulating programs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(5):761–774, 2017.

[135] Felix Winterstein, Kermin E. Fleming, Hsin-Jung Yang, John Wickerson, and George A. Constantinides. Custom-sized caches in application-specific memory hierarchies. In *Proceedings of the International Conference on Field-Programmable Technology (FPT)*, pages 144–151, Queenstown, New Zealand, 2015. IEEE.

[136] Lisa Wu, Andrea Lottarini, Timothy K. Paine, Martha A. Kim, and Kenneth A. Ross. Q100: The architecture and design of a database processing unit. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 255–268, Salt Lake City, Utah, March 2014.

[137] Lisa Wu, Orestis Polychroniou, Raymond J. Barker, Martha A. Kim, and Kenneth A. Ross. Energy analysis of hardware and software range partitioning. *ACM Transactions on Computing Systems*, 32(3):8, August 2014. 24 pages.

[138] Xilinx Corporation. *Vivado Design Suite: User Guide*, 2018. UG902 (v2018.3).

[139] Hsin-Jung Yang, Kermin E. Fleming, Michael Adler, Felix Winterstein, and Joel Emer. Scavenger: Automating the construction of application-optimized memory hierarchies. In *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8. IEEE, 2015.

[140] Christian Zebelein, Christian Haubelt, Joachim Falk, Tobias Schwarzer, and Jürgen Teich. Model-based actor multiplexing with application to complex communication protocols. In *Proceedings of Design, Automation, and Test in Europe (DATE)*, pages 216–219, Dresden, Germany, March 2014. The Institute of Electrical and Electronics Engineers (IEEE).

[141] Kuangya Zhai, Richard Townsend, Lianne Lairmore, Martha A. Kim, and Stephen A. Edwards. Hardware synthesis from a recursive functional language. In *Proceedings of the International Conference on Hardware/-Software Codesign and System Synthesis (CODES+ISSS)*, pages 83–93, Amsterdam, The Netherlands, October 2015. IEEE.

[142] Ritchie Zhao, Gai Liu, Shreesha Srinath, Christopher Batten, and Zhiru Zhang. Improving high-level synthesis with decoupled data structure optimization. In *Proceedings of the 53rd Design Automation Conference*, Austin, Texas, June 2016. ACM.

[143] Yuan Zhou, Khalid Musa Al-Hawaj, and Zhiru Zhang. A new approach to automatic memory banking using trace-based address mining. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 179–188, Monterey, California, February 2017. ACM.