From Functional Programs to Pipelined Dataflow Circuits

Richard Townsend
Columbia University (USA)
rtownsend@cs.columbia.edu

Martha A. Kim
Columbia University (USA)
martha@cs.columbia.edu

Stephen A. Edwards
Columbia University (USA)
sedwards@cs.columbia.edu

Abstract

We present a translation from programs expressed in a functional IR into dataflow networks as an intermediate step within a Haskell-to-Hardware compiler. Our networks exploit pipeline parallelism, particularly across multiple tail-recursive calls, via non-strict function evaluation. To handle the long-latency memory operations common to our target applications, we employ a latency-insensitive methodology that ensures arbitrary delays do not change the functionality of the circuit. We present empirical results comparing our networks against their strict counterparts, showing that non-strictness can mitigate small increases in memory latency and improve overall performance by up to 2×.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Compilers; B.6.3 [*Logic Design*]: Design Aids

Keywords Functional Language, Dataflow Networks, Non-strict Evaluation

1. Introduction

A growing fraction of the area in modern chips is dedicated to application-specific accelerators. These specialized cores consume less energy to perform a task than a general-purpose processor, and energy consumption is of critical, growing concern.

The work we present here is part of an ongoing project to simplify custom accelerator design. Because most accelerators are synthesized from register-transfer level (RTL) descriptions, their production is a tedious, error-prone process that impedes exploring design trade-offs (e.g., between computing resources and memory).

We present a compiler that synthesizes dataflow networks from algorithms expressed in a functional intermediate representation (IR) dubbed "Floh." We target dataflow networks because they are modular, inherently parallel, naturally "patient" about long, varying latencies, and they can yield high-speed hardware implementations [7]. 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 complex parallel algorithms [17]. These abstractions also present optimization opportunities in our compiler that may otherwise be infeasible due to side effects or direct control over pointers.

Many high-level synthesis techniques deal well with "scientific" algorithms expressed as loop nests with affine array indices and limited conditionals [20]. Instead, we target algorithms with complicated control and irregular memory access patterns that operate on data structures such as lists, trees, and graphs. These kinds of problems motivate the source and target of our compilation: functional languages present elegant solutions to such algorithms (especially those with recursion), and patient dataflow networks can mitigate the long latencies associated with today's memories. Since irregular algorithms appear in many settings, we and others are working to address the synthesis challenge they present [16, 29].

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. [5] 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. Our 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.

Motivation for this work was partly driven by shortcomings observed in previous research. The SHard compiler [23] translates functional programs into dataflow networks before generating hardware, but their strict function evaluation hinders their ability to exploit parallelism. Conversely, the MIT tagged-token dataflow architecture [1] dynamically generates dataflow graphs with fully non-strict functions, but their stored-program implementation relies on unbounded buffering, an impossible restriction in hardware.

We make two main contributions to attack these limitations: a largely syntax-directed translation of a functional IR into an abstract dataflow model that exhibits pipeline and other forms of parallelism, and a technique for implementing such abstract networks in hardware with limited, bounded buffering. Our translation handles algebraic data types, non-recursive function calls, and groups of mutually tail-recursive functions; an earlier pass in our compilation flow dismantles programs with arbitrary recursion into this form using a technique presented elsewhere [30].

Our paper is structured as follows. Section 2 describes our starting point, the "Floh" IR; Section 3 introduces our target, an abstract dataflow model with unbounded buffers between nodes. Our translation operates in two steps: Section 4 presents the translation from Floh to dataflow with unbounded buffers; Section 5 shows how to practically implement such networks in hardware. Section 6 presents our experimental results, which show our compiler creates systems that exploit pipeline parallelism and can cope with memory latency. We defer our discussion of related work to Section 7.

2. Our Intermediate Representation "Floh"

Our synthesis process begins from the Floh ("Functional Language On Hardware") intermediate language, whose syntax is shown in Figure 1. We based this syntax on the Core language of the Glasgow

CC'17, February 5–6, 2017, Austin, TX, USA
© 2017 ACM. 978-1-4503-5233-8/17/02...$15.00
http://dx.doi.org/10.1145/3033019.3033027

program	::=	*type-def** *func-def*$^+$	
type-def	::=	**data** *Tcon* = (*Dcon type*$^+$)$^+$	
func-def	::=	*fid vid*$^+$ = *expr*	
expr	::=	*vid*	Variable
		fid vid$^+$	Function call
		Dcon vid$^+$	Data constructor
		let (*vid* = *expr*)$^+$ **in** *expr*	Variable binding
		case *vid* **of** (*pattern* → *expr*)$^+$	Conditional
pattern	::=	*Dcon* (*vid* | _)$^+$	
type	::=	**Int**	Finite Integer
		Go	Trigger (see Section 2.2)
		Tcon	Algebraic type constructor
vid	::=	Variable identifier	
fid	::=	Function identifier or integer literal (see Section 2.2)	
Dcon	::=	Data constructor identifier (capitalized)	
Tcon	::=	User-defined type name (capitalized)	

Figure 1. The syntax of Floh, our compiler's functional IR

Haskell Compiler [21] to attain a simple but rich IR with inherent parallelism (described in Section 2.1). By design, we limit Floh's syntax to simplify its translation into dataflow; our prototype compiler, built on GHC, accepts a Haskell subset providing first-class functions and general recursion, and transforms it into Floh.

A Floh program consists of type and function definitions. An algebraic data type definition (*type-def*) defines a new type named *Tcon* consisting of one or more "variants". Each variant consists of a globally unique name called a data constructor (*Dcon*) and one or more type fields. A function definition binds an expression (the body of the function) to a function name. Each function has one or more named arguments; we prohibit partial function application.

We allow only tail-recursion in functions; our compiler uses the technique of Zhai et al. [30] to transform arbitrary recursion into tail-recursion with an explicit stack. Although these stacks are currently in the heap, we will eventually use custom stacks.

2.1 Expressions

The most primitive expression is a variable: a function parameter or a local name bound by a *let* construct or by a pattern in a *case* construct. In keeping with tradition, we use the term "variable" even though it is a misnomer: our "variables" are immutable.

Function calls require one or more arguments, which must all be variables; we can pass an expression's value to a function by first binding the expression to a variable with a *let*. The number and type of arguments to each function must be consistent with the function's type, which is inferred from its definition. Data constructors, which always have one or more arguments, behave like functions that create objects: if type T has a variant defined as D $t_1 \dots t_k$, the expression D $v_1 \dots v_k$, where variable v_i is of type t_i, creates an object of type T.

Unlike Haskell, Floh uses different strictness policies for data constructors and function calls to balance simplicity with performance. Data constructors are *strict*: they evaluate all their arguments before producing a result, simplifying the memory system semantics and eliminating the bookkeeping overhead of Haskell's lazy evaluation scheme. Floh functions are only *strict in their first argument*: a function may begin evaluation after the first argument is available, but requires the arrival of all other arguments before returning a value, even if some are unused. 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.

The *let* construct binds one or more local variables to expressions. These 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.

The *case* construct is a multi-way conditional that selects an expression to evaluate according to a matching pattern. A *case's* argument (a *vid*) must be bound to a data constructor for matching; the *case* has exactly one pattern for each variant of *vid's* data type, and each pattern must specify every argument for its data constructor. After matching its argument against one of the patterns, the *case* extracts the fields of the data constructor, binding each to the new local variables named in the pattern or ignoring them if the corresponding argument is the wildcard character (underline). This *case* construct is more restrictive than the full pattern-matching mechanism in many modern functional languages, but richer patterns can be decomposed to this simpler form.

2.2 Numeric Literals and the *Go* Type

Numeric literals are deliberately missing from Floh to simplify its translation to dataflow. To avoid the need for "source" nodes that generate data without being prompted (which lead to scheduling headaches), we insist that the evaluation of each expression is triggered by the evaluation of at least one variable. Traditional numeric literals would violate this invariant.

Each numeric literal in Floh is generated by a primitive function that takes a single-valued type called "*Go*" 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. For example:

```
add42 :: Go → Int → Int
add42 g x = let fortyTwo = 42 g in
            add x fortyTwo
```

Here, *42* is a function that generates the value 42 when given a *Go*-valued argument *g*. This value is then bound to *fortyTwo* and added to *x*. The type signature here is for illustration only; all signatures are inferred in Floh and are not part of the syntax.

A program cannot produce a *Go* object; the environment supplies a single *Go* object for the program, and functions pass it around as an argument (like the *g* above). 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.

We also use the *Go* type for constant data constructors. The *Bool* type below illustrates this: both the *True* and *False* data constructors take a single *Go* argument, and functions operating on Booleans (such as the logical *not* function) take an additional *g* argument to pass the *Go* type around. *Case* statements take these *Go* fields into consideration, but ignore them with wildcard characters.

```
data Bool = True  Go
            False Go

not :: Go → Bool → Bool
not g boolVal = case boolVal of
                  True  _ → False g
                  False _ → True  g
```

While our *Go* machinery clutters Floh programs, it simplifies our 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. By contrast, Arvind and Nikhil [1] use two different policies for literals: many are subsumed into the operator using them; others use a "trigger" signal similar to our *Go*. We believe that our single-policy approach is simpler. In practice, our compiler takes Haskell programs with traditional constants and inserts the needed *Go* machinery as part of its translation into Floh.

2.3 Types and Memory

Values in our language are either finite integers, the single-variant *Go* type, or non-recursive algebraic data types. We omit recursive types due to their unboundedness: statically-sized (i.e., bounded) bit vectors encode our data types in hardware, and a recursive type cannot be bounded at compile time, in general.

We model recursive data types with type-specific pointers and a heap. For example, a binary tree of integers uses types

```
data Btree = Branch Bptr  Int  Bptr
             Leaf  Go
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 (note the use of *Go* for this constant data constructor).

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 Floh 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.

2.4 An Example: Map

Figure 2 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
              []  → []
              (x:xs)  → 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 uses the techniques described in Zhai et al. [30] to translate the recursive Haskell function into the tail-recursive Floh program in Figure 2. It transforms recursive functions into continuation-passing style, performs lambda lifting to name each continuation as a global function, creates a continuation type to encode these new functions, and finally builds a pair of functions that operate on the new type to handle the calls (*call*) and continuations (*cont*) of the *map* function. The continuations in our language behave like activation records and can be managed on a stack.

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

data ContPtr = ContPtr  Int
data Continuation  = C1 Int  ContPtr
                     C0 Go

map lp  g =  let  c0 = C0 g              in
             let  sp  = stackWrite  c0  in
             call  lp  g  sp

call   lp  g  sp =  let   le  =  listRead   lp  in
                    case  le  of
                    Cons x  xs  →  let   nc  = C1 x  sp          in
                                   let   nsp  =  stackWrite  nc  in
                                   call  xs  g  nsp
                    Nil  _      →  let   nil  = Nil  g          in
                                   let  lpn  =  listWrite   nil  in
                                   cont  sp  lpn

cont  sp  lp  =  let   se  = stackRead sp  in
                 case  se  of
                 C1 x nsp  →  let  fx  = f  x              in
                              let  nle  = Cons fx  lp      in
                              let  nlp  =  listWrite  nle  in
                              cont  nsp  nlp
                 C0 _      → lp
```

Figure 2. 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 *cont* function pops each element *x* from the stack, applies *f* to it, and prepends the result to the returned list.

In Figure 2, the *map* function receives a list pointer and a *Go* object as arguments, pushes an initial terminal continuation (*C0*) on the stack, and then starts *call*. The *call* function reads a cell of the input list and either pushes its contents in a *C1* continuation on the stack before tail-recursing or writes an empty list to the heap and invokes *cont*. The *cont* 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, *cont*'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.

3. Dataflow Networks

We translate a Floh program into an idealized dataflow network with unbounded buffers, which we ultimately convert into hardware with finite buffers; see Section 5. 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.

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 [30, 11] does not accommodate delays as readily.

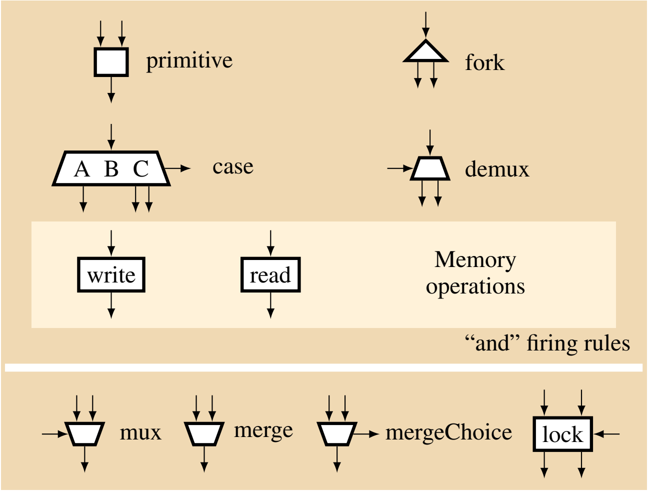


Figure 3. Our menagerie of dataflow nodes. Those above the line require data on every input channel to fire.

A dataflow network consists of a collection of *nodes* connected via unbounded point-to-point FIFO *channels* that convey typed, data-carrying *tokens*. All tokens on a particular channel have the same Floh type. When a node *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* node has sufficient input tokens to fire.

At this level, our networks resemble Kahn Process Networks (KPNs) [15]: a KPN comprises a set of deterministic nodes that communicate via tokens passed along unbounded FIFOs. Since we use a nondeterministic merge (arbiter) node in our networks, we do not exactly follow the KPN model and thus cannot rely on Kahn's proof of determinism. However, our networks are 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.

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 nodes (i.e., those with sufficient tokens on their input channels). The choice of which nodes actually fire is nondeterministic.

Figure 3 lists the types of nodes in our networks; each has its own firing policy. Because each channel is an unbounded FIFO that can always accept another token, a node's ability to fire solely depends on the presence of tokens on its input channels.

Each node on the top part of Figure 3 requires exactly one token per input to fire. A *primitive* node models a constant, simple arithmetic or Boolean function and produces a single output token when it fires. A *fork* node consumes a single input token and copies it to each of its output channels.

A *case* node models the core of the Floh *case* construct. Each case node has a single input that accepts tokens of a specific algebraic data type, an output (drawn on the right) that reports the variant of the input token, and a potentially empty set of output channels for each variant of the algebraic type. We only provide a channel for a field if the corresponding case alternative uses it.

When a case fires, it consumes its input token, emits a "choice" token indicating the input's variant, splits the input token into its constituent fields, and generates tokens on the existing output channels for that variant. Figure 4 illustrates the three ways a particular three-variant case node may fire. The omission of a channel for *B's* field means that the field is not used in *B's* alternative expression.

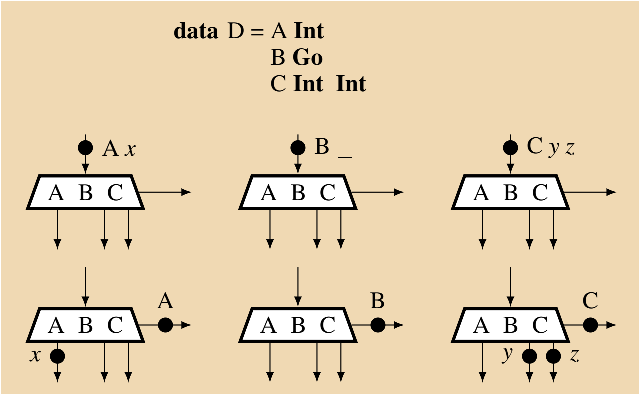


Figure 4. The three ways a particular 3-variant case node can fire. The top and bottom rows indicate the state of the node's channels before and after firing, respectively. Labeled black dots represent the location of tokens and their values.

A *demux* node routes an input token (from the top) to one of its output channels depending on the value of a "choice" token (from the side). A case node is one source of such "choice" tokens.

Memory *read* and *write* nodes behave like primitive functions 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 node takes a data token as input and generates an address token as output; a read node does the opposite. Together, these nodes maintain the deterministic memory operation invariant discussed in 2.3.

Our translation treats *read* and *write* nodes abstractly: as independent and non-interfering, but their eventual implementation will be more subtle. For example, the memory system much 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. We also intend to divide up memory into multiple regions, e.g., based on their type. This means we must both size these regions wisely and, to avoid local out-of-memory situations (e.g., stack overflows), back local memories with a larger off-chip memory. The abstract memory operations suffice for our purposes here; we will demonstrate a realistic, garbage-collected memory system in future work.

The nodes on the bottom half of Figure 3 only require tokens on a subset of their inputs to fire. A *mux* node is the opposite of the demux node: 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* node 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* nodes 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 merge node, manages access to a shared resource.

A *lock* node limits calls to a cluster of mutually recursive functions; after receiving arguments for a given call, it blocks additional inputs until an "unlock" token arrives. A lock node for a function of *n* arguments has *n* pairs of inputs and outputs and a single "unlock" input on the side. It operates in two phases. In the first phase, the node passes a single argument token from each of its inputs to the corresponding output as soon as the input is available, blocking any further arguments that may arrive. Once a single token has been consumed from every top input, the lock enters the second phase, where it waits to consume a single "unlock" token on its side input before returning to the first phase.

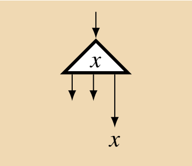


Figure 5. Translating a reference to a variable x: a connection is made to the fork node that distributes its value.

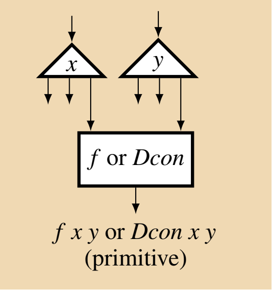


Figure 6. Translating a primitive function call, constant, or data constructor: each argument (one or more variables) is taken from a new connection to that variable's fork node and the result is the output of the primitive node. Constants are treated as single-argument functions.

4. Translation from Floh to Dataflow

In this section, we describe our translation procedure that transforms a Floh program into a dataflow network. Overall, each function is transformed into a subgraph of the network with at least one input channel per argument and one output channel (except for tail-recursive calls). 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, but may take other values from the external environment.

4.1 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 (including, perhaps, a channel for *Go* tokens that trigger constants), 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 node; a reference to a variable in an expression adds an output port and channel from that variable's fork, as shown in Figure 5 (this may produce unary fork nodes, which we optimize away). This implicitly assumes every reference to a variable in an expression will be consumed, which our translation rules enforce.

A call to a built-in function (arithmetic, Boolean logic, and memory operations) or data constructor is converted into a single node. As shown in Figure 6, we add a new connection from each argument's fork (each argument is necessarily a variable) to the appropriate input port. Constants are handled similarly: they take a single *Go*-valued argument passed in from the appropriate fork. The result channel from such an expression is the output channel from the function/constructor/constant node.

Translating a *let* construct, depicted in Figure 7, 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 nodes and calls to user-defined functions, especially tail-recursive calls, are context-dependent; we describe them in the next sections.

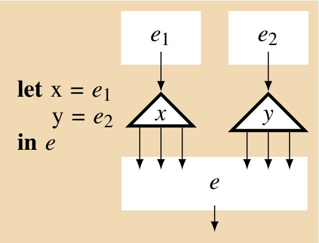


Figure 7. Translating a *let* construct: Each of the newly-bound variables is evaluated and connected to fork nodes that make their values available to the body.

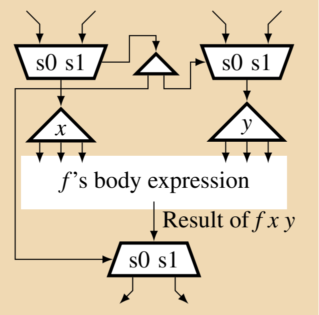


Figure 8. Translating a simple two-argument function f with two external call sites, *s0* and *s1*. A mergeChoice and mux node synchronize calls from the two sites; a demux routes the result to the caller.

4.2 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 calls to clusters is difficult because the subgraph for a cluster may return results from multiple calls out of order. While pipelining clusters would be possible by tagging tokens and adding reorder buffers, we have not yet attempted to do so. We plan to consider this in future work.

Figure 8 shows how each simple function becomes a collection of nodes surrounding the translation of the function's body. Arguments are channeled from call sites to the function via merge and mux nodes; 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. The choice token is also sent to a demux that routes the result of the body expression back to the appropriate caller. Each argument—the output of either the merge or one of the mux nodes—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 input to each of the merge and mux nodes for the arguments and another output to the demux.

Here, we make an important choice that separates us from similar dataflow translations: function calls are not strict. In particular, the nodes comprising a function's body may start firing before every function argument is available; once the first argument from a given call site passes through the merge node, 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 help 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 call sites. We avoid this problem with a single merge node that dictates which call site to service.

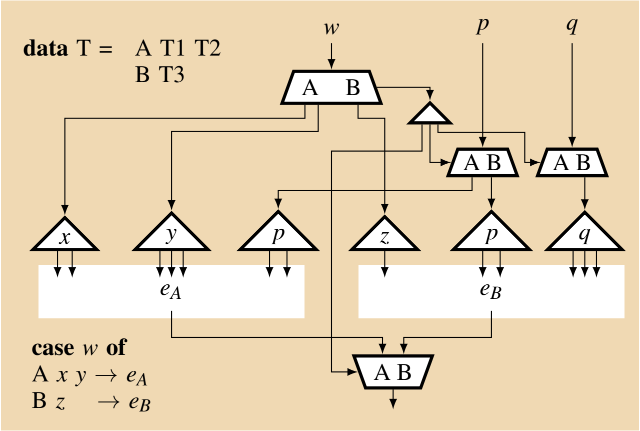


Figure 9. Translating a *case* construct: a case node checks the data constructor of input w and splits the token into fields: x and y for A or z for B. The case also produces a choice token that indicates which branch was selected (A or B); this token drives both the mux that selects the case's result 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.

Although nondeterministic merge nodes 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.

Figure 9 illustrates how a *case* construct is translated in general (some *cases* within clustered functions require special treatment). The network is built around a case node that takes in the argument, identifies which data constructor it represents, and sends that constructor's fields out (if they are referenced) to its alternative expression as newly bound variables.

The *case* node also generates a token that encodes which alternative was selected, which we use to steer local variables to different alternatives. A 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 9 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 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 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 *cases* 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 10 illustrates our solution to this problem; two of the *case's* alternatives return results while a third yields a tail-recursive call. A standard case node still examines and dismantles the algebraic data type into fields, and a choice token still steers live variables (not shown in Figure 10), 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 node because tail-recursive calls make it difficult to determine where a result will ultimately come from.

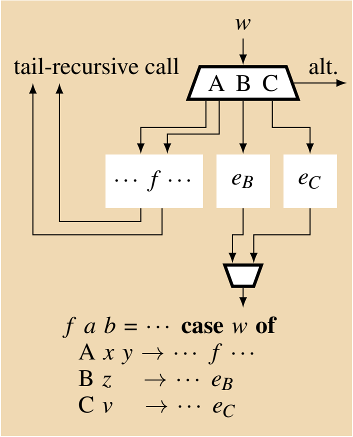


Figure 10. Translating a *case* construct containing tail-recursive calls. Values produced by the alternatives are collected at a merge node; arguments for intra-cluster tail calls are fed to each function's internal call site machinery. Not shown are the demuxes for live variables, which are treated the same as in Figure 9.

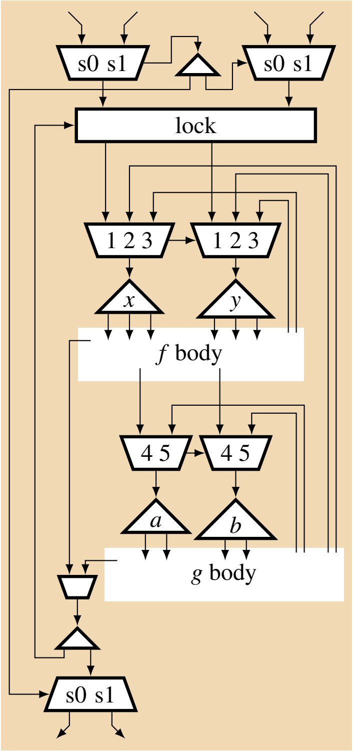


Figure 11. Translating function clusters. Functions f and g comprise the cluster since they call one another recursively. Any values produced by members of a cluster are merged together to form the cluster's output channel; using a mux instead could lead to deadlock. We omit local demuxes for clustered functions for the same reason. A lock node prevents multiple external calls from overlapping within a cluster; the presence of a token on the cluster's output channel triggers the unlocking of the lock node, allowing another external call to access the cluster.

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 merge and muxes for the arguments and a demux for the result as in the simple function case. Functions within the cluster are translated differently, however.

Figure 11 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 still has a merge and group of muxes that manage intra-cluster calls to it, but the results of each function are passed to a single merge node for the cluster (i.e., rather than the per-function demuxes used in our translation of simple functions). Again, our use of a merge node is motivated by the presence of tail-recursion.

The other substantial difference in translating a cluster is a lock node that blocks multiple external calls from accessing the cluster. 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 lock node passes exactly one external function call at a time into the internals of the cluster. The lock accepts exactly one token on each (top) input channel and blocks 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 11 and passing it as an "unlock" token to the lock.

4.4 Putting It All Together: Translating the Map Example

Figure 12 shows the dataflow network our procedure generates for the *map* example introduced in Figure 2. As described in Section 2.4, 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 *cont* contain tail-recursive calls but are not mutually recursive, so each is treated as a cluster. Thus, we place a *lock* node on both of their inputs to ensure they will not accept another outside call until they have generated an output.

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 1, which reads the input list, can race ahead, producing data tokens on channel 4. These tokens are eventually consumed by loop 3, which places them on the stack as a series of "C1" objects. 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 *cont* 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 *cont* is non-strict so loop 5 can race ahead, perhaps even filling *f*'s pipeline to greatly improve parallelism.

In Section 6, we quantify how well our technique exposes parallelism in this example and others.

5. Dataflow Networks in Hardware

We take a structural, distributed approach to implementing dataflow networks in hardware: each node becomes a small block of combinational logic, 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 [1] found. Our approach thus maintains the parallelism of the dataflow network while enabling high-speed hardware, since far-flung parts of the circuit do not need to communicate with a global memory or each other.

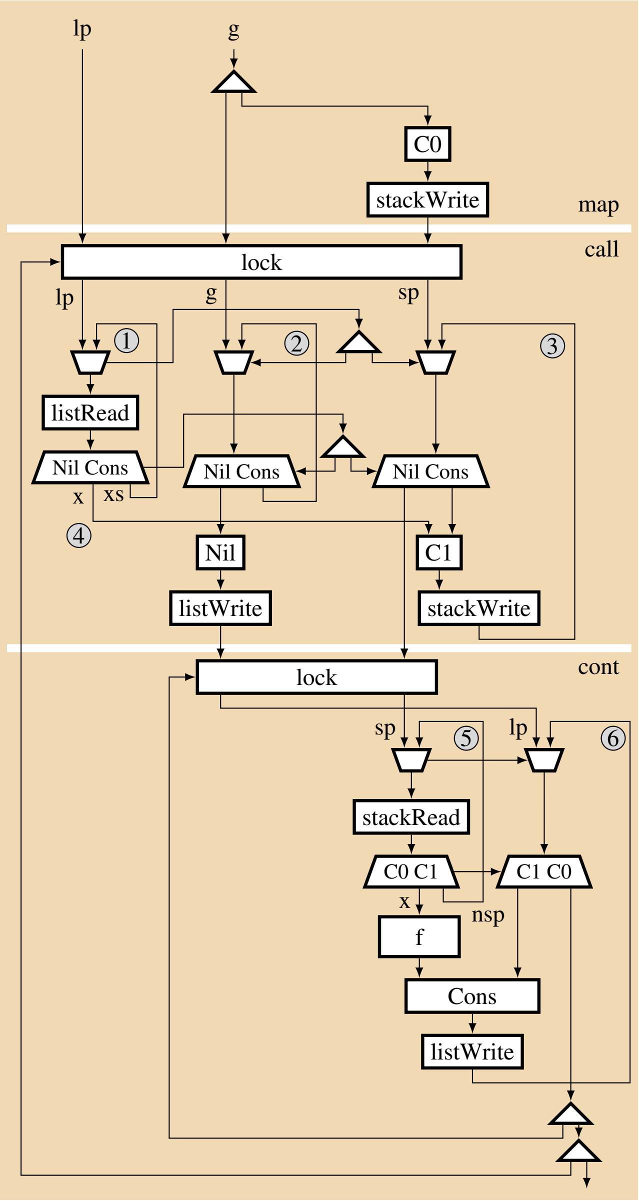
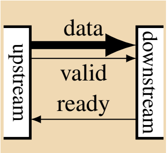


Figure 12. A dataflow graph for the *map* function from Figure 2. This initializes the stack (*map*); walks the input list, pushing each element on the stack (*call*); then pops each element off the stack, applies *f*, and places the result at the head of the new list (*cont*). Tail calls to *call* and *cont* are not strict, decoupling loops 1 and 3, and more importantly, loops 5 and 6, to enable pipelining.

Bounded buffers complicate node 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



valid	ready	Meaning
0	–	No token
1	1	Token Transfer
1	0	Token held upstream

Figure 13. The flow-control protocol, after Cao et al. [7]. Data and valid bits flow downstream; ready bits flow upstream.

et al. [7] present a solution to these issues by implementing each channel as a two-token buffer. This technique breaks any potential cycle or long combinational path with a flip-flop but hinders throughput, as tokens can only cross up to one node per cycle. Since some nodes do more work than others, grouping simple nodes 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 nodes together. Our circuits use the flow-control protocol shown in Figure 13, which presents a danger of a combinational cycle (and hence deadlock) if the *valid* signal depends on the *ready* signal and vice versa. Below, we discuss our implementation technique for avoiding these cycles.

5.1 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. [7]: 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 a node may never depend on its *ready* inputs in the same cycle, which turns out to be a delicate property to guarantee.

5.2 Stateless Nodes

Under our *valid*-then-*ready* evaluation order, nodes that produce a single token when fired have fairly straightforward flow control logic. Primitives nodes 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.

5.3 Stateful Nodes

Nodes 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 is to add a few bits of state to nodes that can generate multiple tokens: fork, case, lock, and mergeChoice. Specifically, each output is given a state bit that indicates whether a token has been generated from the 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. Once tokens have been generated on all the relevant outputs (e.g., every one for a fork; the appropriate group for a case) in a given round, all the bits are reset and the next round can begin in the next cycle. Locks are reset differently: they require the arrival of an unlock token before another round can begin.

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

5.4 Inserting Buffers

As stated above, we implement certain channels in our dataflow graph with the two-token buffers described in Cao et al. [7] 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.

Figure 14 shows part of a network after buffer insertion. Unbuffered cycles result in deadlock, so we first ensure cycles have been broken with buffers using the following heuristic: find the shortest unbuffered cycle in the graph (we modify Dijkstra's shortest-paths algorithm to solve this); find a node 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 nodes with multiple outputs to prevent throughput degradation on the other outputs that are not part of the cycle. In Figure 14, buffer 2 was inserted to break the cycle.

A far more subtle form of deadlock can arise from mismatched reconvergent paths. Multiple paths in a network are *reconvergent* if they share the same source and destination nodes but no others. These paths necessarily originate at multi-output nodes and terminate at multi-input nodes. Two such paths are *mismatched* if exactly one is buffered.

As an example of reconvergent deadlock, consider the dataflow graph shown in Figure 14, 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.

To prevent this situation, we find and buffer mismatched reconvergent paths after breaking cycles (since the latter may buffer some mismatched paths implicitly). Although finding all such paths is tractable in DAGs [22], it is NP-hard in general, requiring a heuristic solution.

Our heuristic leverages an approximation algorithm for counting reconvergent paths [28] 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 $\min(\mathrm{out}_i, \mathrm{in}_j)$ 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 $O(\min(\mathrm{out}_i, \mathrm{in}_j)(m + n\log n))$ time.

Our heuristic applies this algorithm to each pair (i, j) of multi-output (i) and multi-input (j) nodes, keeps any mismatched sets of

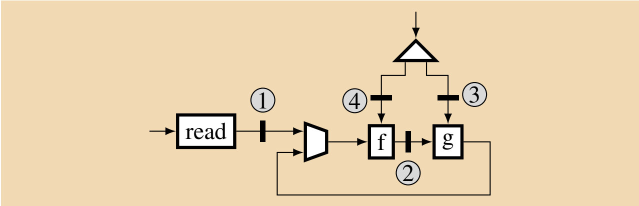


Figure 14. An example of our buffer allocation scheme. We insert buffers to break cycles in the network (e.g., 2) and prevent reconvergent deadlock (e.g., 3).

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 $d = \min(\text{out}_{max}, \text{in}_{max})$, 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 14, 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. In some cases, these excessive buffers improve performance by providing implicit pipelining. Other topologies are hurt by these buffers, as they can increase overall completion time without increasing throughput.

Our heuristic algorithm adds some non-determinism 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. Inserting minimal, efficient buffering is a difficult problem which we will address in future work.

6. Performance Evaluation

To evaluate the quality of the dataflow networks produced by this compilation pass, we simulate several examples. We analyze the impact of non-strict evaluation, the importance of argument order, and sensitivity to memory latency.

6.1 Methodology

Simulator To evaluate the performance of our generated dataflow networks, we wrote a simulator that executes a network on a given 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, our simulator runs a cycle-accurate model of a hardware implementation that employs the finite buffers of Cao et al. [7]. In particular, it models their single cycle latency.

In the other mode, our simulator calculates a lower bound on the number of cycles hardware would take by assuming ideal, fast buffering. Here, buffers are modeled as unbounded, but in each cycle, each node is limited to firing at most once. While such behavior is unrealizable, this value provides a lower bound on the cycles a particular network requires to process the input.

Test Programs We compiled six recursive Haskell programs into dataflow graphs 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. We 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.

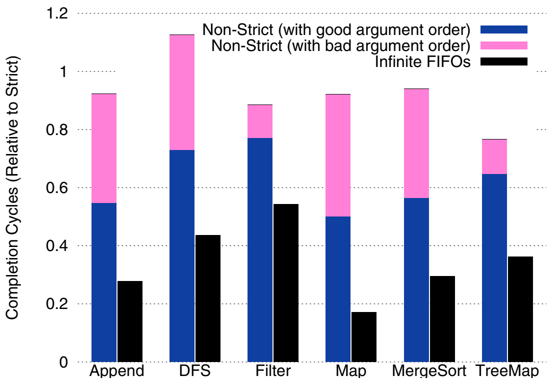


Figure 15. Non-strict evaluation is generally superior to strict. When combined with a good function argument ordering, the finite, non-strict implementations yield a 1.3–2× speedup over strict.

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 we recurse on the left and right subtrees, append the results, and prepend 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 *cont* 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 program is fed a *Go* object 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 main program.

6.2 Strict vs. Non-strict Tail Recursive Calls

Our first experiment measures the performance impact of our non-strict function evaluation policy intended to enable pipelining. We generate each test's dataflow graph under three different policies: non-strict tail-recursive calls with infinite FIFOs on each channel, non-strict with finite buffering, and strict with finite buffering. We also vary 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 *cont* in Figure 12), which enables the function to start being 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 input program's syntax, i.e., neither our translation nor our buffering scheme affects this ordering.

Figure 15 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.

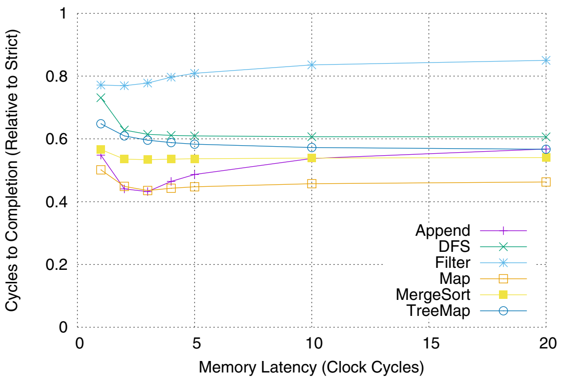


Figure 16. Mitigating increasing memory latency with non-strict function evaluation.

Figure 15 shows non-strictness with proper argument ordering yields faster completion times, which we 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.

6.3 Sensitivity to Memory Latency

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

Figure 16 shows how long it took each program to run under increasing memory latency. Again, we used strict functions as our baseline and calculated the improvement under a non-strict policy (under "good" argument ordering throughout).

The initially negative slopes in Figure 16 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, we attribute these results to unbalanced buffer capacities along reconvergent paths in our networks.

To explain, consider two reconvergent paths originating at some node 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 others have presented solutions for finding these reconvergent paths [22, 28], determining how to distribute buffers along these paths remains a difficult problem that requires further study.

6.4 Sensitivity to Function Latency

We 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.

We 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 16 (we omit the exact results for space): after a slight widening of the gap between non-strict and strict, non-strict completion cycles rise before plateauing off. This further supports our previous conclusions that our buffer allocation scheme is not mature enough to fill up long pipelines, be they be functional or memory-related.

7. Related Work

The high-level synthesis community has produced a wealth of tools for synthesizing hardware from software specifications [20] most often coded in C. These facilitate design space exploration [18] and can produce low-power, application-specific cores [27], but they have difficulty with concurrent accesses to shared resources, algorithms with irregular memory access, and complex control.

Bluespec [2] takes an alternate approach, drawing 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. Our flow control effectively acts as a distributed scheduler.

Our translation of a functional language to dataflow was inspired by that of Arvind and Nikhil [1], but differs in an important way: 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. Since our goal is low-power, high-performance custom hardware, our solution will produce superior results because it avoids general-purpose overhead.

Arvind and Nikhil's virtual approach allows them to support unbounded buffers. While this does eliminate the danger of insufficient buffering, it requires the introduction of additional nodes and channels to throttle loops. Our hardware uses only finite buffers and naturally throttles itself, although choosing appropriate buffer capacities is not a trivial problem.

Finally, our *Go* type leads to a simpler translation than Arvind and Nikhil, who rely on additional rules for producing constants.

Like ours, the SHard compiler of Saint-Mleux et al. [23] compiles a functional language (Scheme) into a dataflow representation to produce custom hardware. However, they only implement strict functions and do not exploit the pipelined parallelism we can. Their treatment of memory is unusual: they *only* directly support closures (a lambda lifting pass removes them in our compiler). In their system, data structures such as lists must be coded as closures.

The FLaSH compiler of Mycroft and Sharp [19, 25] also compiles a functional language into hardware. Their original language (SAFL) was simpler than our Floh IR; they later added parametric channels (SAFL+) [24] and a type-based approach for direct stream processing (SASL) [10]. Their technique for sharing resources (i.e., functions called from multiple places) inspired ours. They, too, place an arbiter at the entry to a shared function, remember which caller gained access, and finally route the result back to the caller. However, their functions cannot be pipelined; only a single call is handled at once.

Kuper's Cλash project [4, 3] has similar goals as ours and a Haskell front-end, but requires the programmer to specify what calculations are performed in each hardware clock cycle. Such a synchronous specification is more in the spirit of a hardware description language. Our model is at a higher level of abstraction. Furthermore, Cλash only supports recursively defined variables; our compiler can handle recursive functions and data types.

Like us, Ghica et al. [12] synthesizes recursive functions into dataflow-like hardware, but they start from a imperative language with side-effects, making it harder to analyze.

Gammie's survey [11] addresses how researchers have attempted to describe *arbitrary* digital circuits with functional programs. A *structural* approach is typical, in which functions use gate-level constructs to operate over streams of data. E.g., Sheeran's μFP language [26] champions higher-order combinators for composing circuit primitives; the various Lava variants [6, 13] implement an embedded hardware description language by harnessing Haskell's type classes. We, instead, take a *behavioral* approach, presenting designers with a higher level of abstraction that allows the compiler more flexibility to optimize the result.

Others, notably Janneck [14], have synthesized dataflow networks to hardware, but few technical details of this are public.

Carloni's Latency-Insensitive Design [8] and Elastic Circuits [9] inspired our synthesized dataflow hardware, but our approach is far richer because of its support of data-dependent behavior.

8. Conclusions

We have presented a largely syntax-directed translation from a functional IR to distributed, parallel dataflow networks to be realized in hardware. Our networks are designed to exploit pipeline parallelism via non-strict function evaluation, especially across multiple calls to tail-recursive functions. We realize these networks in hardware by introducing finite buffering and leveraging a latency-insensitive flow-control protocol.

We measured the efficacy of our translation by compiling some Haskell programs into dataflow networks and comparing their completion times against two other implementation policies: infinite buffering and finite buffering with strict evaluation. These experiments showed that our networks generally outperform their strict counterparts with speedups of 1.3–2×, indicating that our compiler was able to automatically infer pipelines, but that further gains could be achieved with a stronger buffer allocation heuristic. Furthermore, the improvement shown by a non-strict policy over a strict policy widens under low memory latencies, but remains constant as memory latency increases further, suggesting that there remains more parallelism to exploit with other techniques.

Acknowledgments

Andrea Lottarini suggested the use of stateful *fork* nodes.

This work was funded by the National Science Foundation under grant CCF-1162124.

References

[1] Arvind and R. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Trans. Comp.*, 39(3):300–318, Mar. 1990.

[2] Arvind et al. High-level synthesis: an essential ingredient for designing complex ASICs. In *Intl. Conf. Comp. Aided Design (ICCAD)*, pp. 775–782, San Jose, CA, Nov. 2004.

[3] C. Baaij et al. Cλash: Structural descriptions of synchronous hardware using Haskell. In *Euromicro Digital Sys. Design (DSD)*, pp. 714–721, Lille, France, Sept. 2010.

[4] C. P. R. Baaij and J. Kuper. Using rewriting to synthesize functional languages to digital circuits. In *Proc. Trends in Functional Programming (TFP)*, LNCS vol. 8322, pp. 17–33, Provo, Utah, 2014. Springer.

[5] D. F. Bacon et al. Parallel real-time garbage collection of multiple heaps in reconfigurable hardware. In *Proc. Intl. Symp. Memory Management (ISMM)*, pp. 117–127, Edinburgh, UK, 2014.

[6] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: Hardware design in Haskell. In *Proc. Intl. Conf. Functional Programming (ICFP)*, pp. 174–184, Baltimore, Maryland, 1998.

[7] B. Cao et al. Implementing latency-insensitive dataflow blocks. In *Form. Meth. Mod. Codesign (MEMOCODE)*, Austin, TX, Sep. 2015.

[8] L. P. Carloni. The role of back-pressure in implementing latency-insensitive systems. *Elect. Notes Theo. CS*, 146(2):61–80, 2006.

[9] J. Carmona et al. Elastic circuits. *IEEE Trans. CAD*, 28(10):1437–1455, Oct 2009.

[10] S. Frankau and A. Mycroft. Stream processing hardware from functional language specifications. In *Hawaii Intl. Conf. Sys. Sci.*, 2003.

[11] P. Gammie. Synchronous digital circuits as functional programs. *ACM Computing Surveys*, 46(2), article 21, Nov. 2013.

[12] D. R. Ghica. Function interface models for hardware compilation. In *Proc. of the Intl. Conf. on Formal Methods and Models for Codesign (MEMOCODE)*, pp. 131–142, Cambridge, UK, July 2011.

[13] A. Gill et al. Types and associated type families for hardware simulation and synthesis: The internals and externals of Kansas Lava. *Higher-Order and Symbolic Computation*, pp. 1–20, 2013.

[14] J. W. Janneck et al. Synthesizing hardware from dataflow programs. *Journal of Signal Processing Systems*, 63(2):241–249, May 2011.

[15] G. Kahn. The semantics of a simple language for parallel programming. In *Information Processing 74: Proc. of IFIP Congress 74*, pp. 471–475, Stockholm, Sweden, Aug. 1974. North-Holland.

[16] M. Kulkarni et al. Lonestar: A suite of parallel irregular programs. In *Proc. of the Intl. Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 65–76, Boston, Massachusetts, Apr. 2009.

[17] H.-W. Loidl et al. Comparing parallel functional languages. *Higher-Order and Symbolic Computation*, 16(3):203–251, 2003.

[18] P. Mantovani et al. High-level synthesis of accelerators in embedded scalable platforms. In *Proc. Asia South Pacific Design Automation Conf. (ASP-DAC)*, pp. 204–211. IEEE, 2016.

[19] A. Mycroft et al. Hardware synthesis using SAFL and application to processor design. In *Correct HW Design Verif. Methods (CHARME)*, LNCS vol. 2144, pp. 13–39, Livingston, Scot., Sept. 2001.

[20] R. Nane et al. A survey and evaluation of FPGA high-level synthesis tools. *IEEE Trans. CAD*, 35(10):1591–1604, Oct 2016.

[21] S. Peyton Jones and A. Santos. Compilation by transformation in the Glasgow Haskell Compiler. In *Functional Programming, Glasgow 1994*, pp. 184–204. Springer.

[22] M. W. Roberts and P. K. Lala. Algorithm to detect reconvergent fanouts in logic circuits. *IEE Proc. E—Comp. and Dig. Tech.*, 134(2):105–111, March 1987.

[23] X. Saint-Mleux et al. SHard: a Scheme to hardware compiler. In *Scheme and Func. Prog. (SFPW)*, pp. 39–49, Portland, OR, Sep. 2006.

[24] R. Sharp and A. Mycroft. A higher-level language for hardware synthesis. In *Correct Hard. Design and Verif. Methods (CHARME)*, pp. 228–243. Springer, 2001.

[25] R. W. Sharp and A. Mycroft. The FLaSH compiler: Efficient circuits from functional specs. tr.2000.3, AT&T Labs Cambridge, 2000.

[26] M. Sheeran. μFP, an algebraic VLSI design language. In *Symp. LISP and Funct. Prog. (LFP)*, pp. 104–112, Austin, Texas, Aug. 1984.

[27] G. Venkatesh et al. Conservation cores: Reducing the energy of mature computations. In *Arch. Support for Prog. Lang. and Operating Sys. (ASPLOS)*, pp. 205–218, Pittsburgh, Pennsylvania, Mar. 2010.

[28] D. R. White and M. Newman. Fast approximation algorithms for finding node-independent paths in networks. Technical Report 01–07–035, Santa Fe Institute, New Mexico, 2001.

[29] F. Winterstein et al. High-level synthesis of dynamic data structures: A case study using Vivado HLS. In *Proc. of Field-Programmable Technology (FPT)*, pages 362–365. IEEE, 2013.

[30] K. Zhai et al. Hardware synthesis from a recursive functional language. In *Proc. Conf. Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Amsterdam, The Netherlands, Oct. 2015.