SHard: a Scheme to Hardware Compiler

Xavier Saint-Mleux
Université de Montréal
saintmlx@iro.umontreal.ca

Marc Feeley
Université de Montréal
feeley@iro.umontreal.ca

Jean-Pierre David
École Polytechnique de Montréal
jpdavid@polymtl.ca

Abstract

Implementing computations in hardware can offer better performance and power consumption than a software implementation, typically at a higher development cost. Current hardware/software co-design methodologies usually start from a pure software model that is incrementally transformed into hardware until the required performance is achieved. This is often a manual process which is tedious and which makes component transformation and reuse difficult. We describe a prototype compiler that compiles a functional subset of the Scheme language into synthesizable descriptions of dataflow parallel hardware. The compiler supports tail and non-tail function calls and higher-order functions. Our approach makes it possible for software developers to use a single programming language to implement algorithms as hardware components using standardized interfaces that reduce the need for expertise in digital circuits. Performance results of our system on a few test programs are given for FPGA hardware.

1. Introduction

Embedded systems combine software and hardware components. Hardware is used for interfacing with the real world and for accelerating the lower-level processing tasks. Software has traditionally been used for implementing the higher-level and more complex processing logic of the system and when future changes in functionality are expected. The partitioning of a system into its hardware and software components is a delicate task which must take into account conflicting goals including development cost, system cost, time-to-market, production volume, processing speed, power consumption, reliability and level of field upgradability.

Recent trends in technology are changing the trade-offs and making hardware components cheaper to create. Reconfigurable hardware is relatively recent and evolves rapidly, and can allow the use of custom circuits when field upgradability is desired or when production volume is expected to be low. Modern ASICs and FPGAs now contain enough gates to host complex embedded systems on a single chip, which may include tens of processors and dedicated hardware circuits. Power consumption becomes a major concern for portable devices. Specialized circuits, and in particular asynchronous and mixed synchronous/asynchronous circuits, offer better power usage than their equivalent software version running on a general purpose CPU or in synchronous logic [9][21][3].

The field of hardware/software co-design [11] has produced several tools and methodologies to assist the developer design and partition systems into hardware and software. Many tools present two languages to the developer, one for describing the hardware and the other for programming the software. This widespread approach has several problems. It requires that the developer learn two languages and processing models. The hardware/software interfaces may be complex, artificial and time consuming to develop. Any change to the partitioning involves re-writing substantial parts of the system. It is difficult to automate the partitioning process in this methodology.

Our position, which is shared by other projects such as SPARK-C [12], SpecC [8] and Handel-C [6], is that it is advantageous to employ a single language for designing the whole system (except perhaps the very lowest-level tasks). We believe that the partitioning of a system into hardware and software should be done by the compiler with the least amount of auxiliary partitioning information provided by the developer (e.g. command line options, pragmas, etc). This partitioning information allows the developer to optimize for speed, for space, for power usage, or other criteria. Moreover this information should be decoupled from the processing logic so that components can be reused in other contexts with different design constraints.

As a first step towards this long-term goal, we have implemented a compiler for a simple but complete parallel functional programming language which is fully synthesizable into hardware. Although our prototype compiler does not address the issue of automatic partitioning, it shows that it is possible to compile a general purpose programming language, and in particular function calls and higher-order functions, into parallel hardware.

We chose a subset of Scheme [17] as the source language for several reasons. Scheme's small core language allowed us to focus the development efforts on essential programming language features. This facilitated experimentation with various hardware specific extensions and allowed us to reuse some of the program transformations that were developed in the context of other Scheme compilers, in particular CPS conversion and 0-CFA analysis.

Our compiler, SHard, translates the source program into a graph of asynchronously connected instantiations of generic "black box" devices. Although the model could be fully synthesized with asynchronous components (provided a library with adequate models) we have validated our approach with an FPGA-based synchronous implementation where each asynchronous element is replaced by a synchronous Finite State Machine (FSM). Preliminary tests have been successfully performed using clockless implementations for some of the generic components, combined with synchronous FSMs for the others. Off-the-shelf synthesis tools are used to produce the circuit from the VHDL file generated by the compiler.

Throughout this project, emphasis has been put on the compilation process. To that effect, minimal effort has been put on opti-

Proceedings of the 2006 Scheme and Functional Programming Workshop
University of Chicago Technical Report TR-2006-06

mization and on creating efficient implementations of the generic hardware components that the back-end instantiates. Demonstrating the feasibility of our compilation approach was the main concern. While SHard is a good proof of concept its absolute performance remains a future goal.

In Section 2 we give a general overview of our approach. In Section 3 the source language is described. The hardware building blocks of the dataflow architecture are described in Section 4 and Section 5 explains the compilation process. Section 6 explores the current memory management system and possible alternatives. Section 7 illustrates how behavioral simulations are performed and Section 8 outlines the current RTL implementation. Experimental results are given in Section 9. We conclude with related and future work in Section 10.

2. Overview

The implementation of functions is one of the main difficulties when compiling a general programming language to hardware. In typical software compilers a stack is used to implement function call linkage, but in hardware the stack memory can hinder parallel execution if it is centralized. The work on Actors [14] and the Rabbit Scheme compiler [13] have shown that a tail function call is equivalent to message passing. Tail function calls have a fairly direct translation into hardware. Non-tail function calls can be translated to tail function calls which pass an additional continuation parameter in the message. For this reason our compiler uses the Continuation Passing Style (CPS) conversion [2] to eliminate non-tail function calls. Each message packages the essential parts of a computational process and can be viewed as a process token moving through a dataflow architecture. This model is inherently parallel because more than one token can be flowing in the circuit. It is also energy efficient because a component consumes power only if it is processing a token. The main issues remaining are the representation of function closures in hardware and the implementation of message passing and tail calls in hardware.

Many early systems based on dataflow machines suffer from a memory bottleneck [10]. To reduce this problem our approach is to distribute throughout the circuit the memories which store the function closures. A small memory is associated with each function allocation site (`lambda`-expression) with free variables. The allocation of a cell in a closure memory is performed whenever the corresponding `lambda`-expression is evaluated. To avoid the need for a full garbage collector we deallocate a closure when it is called. Improvements on this simple but effective memory management model are proposed in Section 6.

By using a data flow analysis the compiler can tell which function call sites may refer to the closures contained in a specific closure memory. This is useful to minimize the number of busses and control signals between call sites and closure memories.

To give a feel for our approach we will briefly explain a small program. Figure 1 gives a program which sorts integers using the mergesort algorithm. The program declares an input channel (`cin`) on which groups of integers are received sequentially (as $\langle n, x_1, x_2, \ldots, x_n \rangle$), and an output channel (`cout`) on which the sorted integers are output. The program also declares functions to create pairs and lists as closures (`nil` and `cons`), the mergesort algorithm itself (functions `sort`, `split`, `merge`, and `revapp`), functions to read and write lists on the I/O channels (`get-list` and `put-list`) and a "main" function (`doio`) which reads a group, sorts it, outputs the result and starts over again. Note also that the predefined procedure `eq?` can test if two closures are the same (i.e. have the same address).

Figure 2 sketches the hardware components which are generated by the compiler to implement the `sort` function at line 33. The `sort` function can be called from three different places (at lines 57, 40 and 41; "callers" 1, 2 and 3 respectively) and "merge" components are used to route all function call requests to the function's body (A). The body starts with a fifo buffer (B) followed by the implementation of the test at line 34 ("stage" and "split" components, C). If the list is not empty, function `split` is called (line 36) after allocating a continuation closure for returning the result (D). This continuation, when called, allocates another continuation (the function at line 37) and calls `split`'s result (which is a function closure representing a pair) with it (E). Next, the test at line 38 is implemented like the previous one (F). If needed, two processes are forked with recursive calls to `sort` and their results are merged after both complete (G) (this is achieved with the `par` construct at line 40, which is syntactically like a `let` but evaluates all its binding expressions in parallel).

```
 1. (letrec
 2.     ((cin (input-chan cin))
 3.      (cout (output-chan cout))
 4.      (nil (lambda (_) 0))
 5.      (cons (lambda (h t) (lambda (f) (f h t))))
 6.      (revapp (lambda (L1 L2)
 7.                (if (eq? nil L1)
 8.                    L2
 9.                    (L1 (lambda (h t)
10.                          (revapp t (cons h L2)))))))
11.      (split (lambda (L L1 L2)
12.               (if (eq? nil L)
13.                   (cons L1 L2)
14.                   (L (lambda (h t)
15.                        (split t (cons h L2) L1))))))
16.      (merge (lambda (L1 L2 L)
17.               (if (eq? nil L1)
18.                   (revapp L L2)
19.                   (if (eq? nil L2)
20.                       (revapp L L1)
21.                       (L1
22.                        (lambda (h1 t1)
23.                          (L2
24.                           (lambda (h2 t2)
25.                             (if (< h1 h2)
26.                                 (merge t1
27.                                        (cons h2 t2)
28.                                        (cons h1 L))
29.                                 (merge
30.                                  (cons h1 t1)
31.                                  t2
32.                                  (cons h2 L)))))))))))
33.      (sort (lambda (L)
34.              (if (eq? nil L)
35.                  nil
36.                  ((split L nil nil)
37.                   (lambda (L1 L2)
38.                     (if (eq? nil L2)
39.                         L1
40.                         (par ((s1 (sort L1))
41.                               (s2 (sort L2)))
42.                           (merge s1 s2 nil)))))))))
43.      (get-list (lambda (n)
44.                  (if (= 0 n)
45.                      nil
46.                      (cons (cin)
47.                            (get-list (- n 1))))))
48.      (put-list (lambda (L)
49.                  (if (eq? nil L)
50.                      (cout nil)
51.                      (L (lambda (h t)
52.                           (cout h)
53.                           (put-list t))))))
54.      (doio (lambda ()
55.              (let ((n (cin)))
56.                (let ((L (get-list n)))
57.                  (put-list (sort L))
58.                  (doio))))))
59.   (doio))
```

Figure 1. Mergesort program

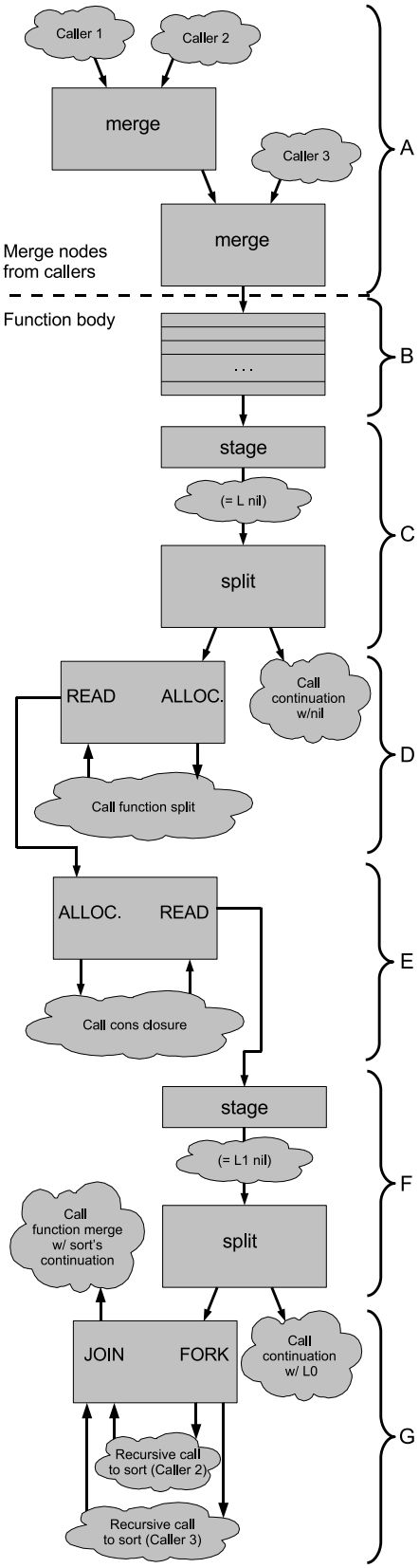

Figure 2. mergesort's `sort` function

3. Source Language

The source language is a lexically-scoped mostly functional language similar to Scheme [17]. It supports the following categories of expressions:

- Integer literal
- Variable reference
- Function creation: `(lambda (`*params*`)` *body*`)`
- Function call
- Conditional: `(if` *cond true-exp false-exp*`)`
- Binding: `let`, `letrec` and `par`
- Sequencing: `begin`
- I/O channel creation: `(input-chan` *name*`)` and `(output-chan` *name*`)`
- Global vectors: `make-vector`, `vector-set!` and `vector-ref`.

Primitive functions are also provided for integer arithmetic operations, comparisons on integers, bitwise operations and, for performance evaluation, timing. For example, the recursive factorial function can be defined and called as follows:

```
(letrec ((fact (lambda (n)
                 (if (< n 2)
                     1
                     (* n (fact (- n 1)))))))
  (fact 8))
```

In our prototype, the only types of data supported are fixed width integers, global vectors and function closures. Booleans are represented as integers, with zero meaning false and all other values meaning true. Closures can be used in a limited fashion to create data structures, as in the following example:

```
(let ((cons (lambda (h t)
              (lambda (f) (f h t))))
      (car (lambda (p)
             (p (lambda (h t) h)))))
  (let ((pair (cons 3 4)))
    (car pair)))
```

While this is a simple and effective way of supporting data structures, the programmer has to adapt to this model. The function call `(cons 3 4)` allocates memory for the closure containing the two integers, but this memory is reclaimed as soon as `pair` is called inside the `car` function; `pair` cannot be called again and the value `4` is lost. The only way to fetch the content of a closure is to call it and then recreate a similar copy using the data retrieved. Possible improvements are discussed in Section 6.

The `par` binding construct is syntactically and semantically similar to the `let` construct but it indicates that the binding expressions can be evaluated in parallel and that their evaluation must be finished before the body is evaluated. They can be seen as a calculation with a continuation that takes several return values. They can be used for manual parallelization when automatic parallelization (Section 5.1) is turned off or when expressions with side-effects may run concurrently.

The I/O channel creation forms create a functional representation of named input and output channels. Input channels are functions with no arguments that return the value read. Output channels take the value to be written as an argument and always return 0. The name given as an argument to `input-chan` and `output-chan` will be used as a signal name in the top-level VHDL circuit description. For example, the following specification creates a circuit that adds the values read from two different input channels, writes the sum on an output channel and starts over again:

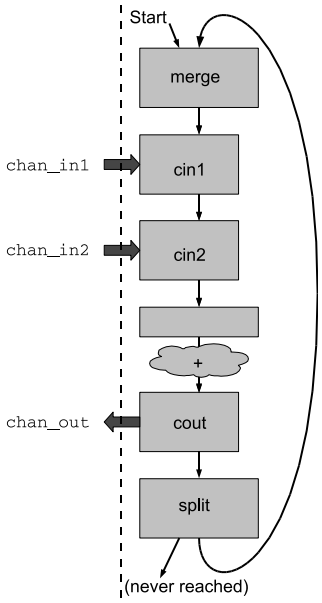

Figure 3. I/O example

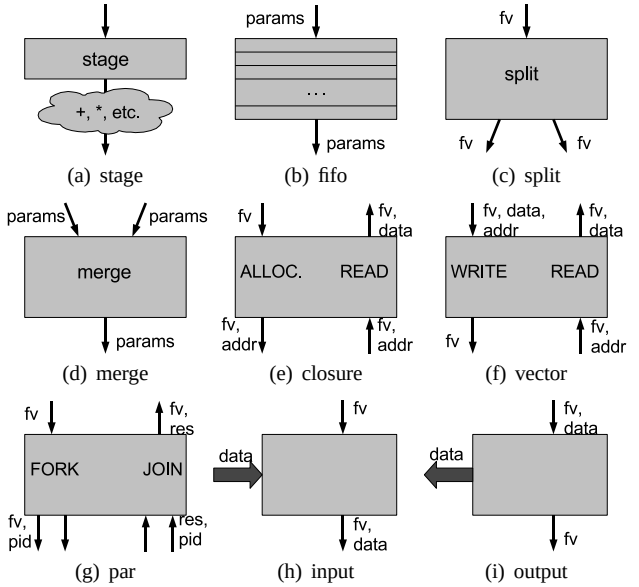

Figure 4. Generic Hardware Components

```
(let ((cin1 (input-chan chan_in1))
      (cin2 (input-chan chan_in2))
      (cout (output-chan chan_out)))
  (letrec ((doio (lambda ()
                   (cout (+ (cin1) (cin2)))
                   (doio))))
    (doio)))
```

This example is illustrated in Figure 3 using the components described in Section 4.

Like any other function with no free variables, channel procedures can be called any number of times since no closure allocation or deallocation is performed.

Mutable vectors are created with the `make-vector` primitive, which is syntactically similar to its Scheme equivalent. Currently, only statically allocated vectors are supported (i.e. vectors must be created at the top-level of the program). Our memory management model would have to be extended to support true Scheme vectors.

To simplify our explanations, we define two classes of expressions. *Trivial* expressions are literals, lambda expressions and references to variables. *Simple* expressions are either *trivial* expressions or calls of primitive functions whose arguments are *trivial* expressions.

4. Generic Hardware Components

The dataflow circuit generated by the compiler is a directed graph made of instantiations of the 9 generic components shown in Figure 4. The components are linked using unidirectional data busses which are the small arrows entering and leaving the components in the figure. Channels carry up to one message, or *token*, from the source component to the target component. Each channel contains a data bus and two wires for the synchronization protocol. The *request* wire, which carries a signal from the source to target, indicates the presence of a token on the bus. The *acknowledge* wire, which carries a signal from the target to the source, indicates that the token has been received at the target. The two signals implement a four-phase handshake protocol (i.e. ↑ Req, ↑ Ack, ↓ Req, ↓ Ack).

The following generic components are used in the system:

- **Stage** (Fig. 4(a)): Stages are used to bind new variables from *simple* expressions. Every `let` expression in which all bindings are from *simple* expressions are translated into stages in hardware; this is the case for all `let` expressions at the end of compilation. The stage component has an input channel that carries a token with all live variables in the expression that encloses the `let`. It has one output channel that sends a token with the same information, which then goes through a combinatorial circuit that implements all the *simple* expressions; the stage component is responsible for all synchronization so it must take into account the delay of the combinatorial part. The final token is sent to the component that implements the `let`'s body with all live variables at that point.
- **Fifo** (Fig. 4(b)): Fifos are used as buffers to accumulate tokens at the beginning of functions that might be called concurrently by several processes. Fifos are necessary in some situations to avoid deadlocks. They are conceptually like a series of n back-to-back stages but are implemented using RAM blocks in order to reduce the latency from input to output and the size of the circuit.
- **Split** (Fig. 4(c)): Split components are used to implement conditional (`if`) expressions. They have an input channel that receives a token with all live variables in the expression that encloses the conditional. The test expression itself is a reference to a boolean variable at the end of compilation so it is received directly as a wire carrying a 0 or a 1. Every token received is routed to the appropriate component through either one of two output channels, representing the *then* and *else* branches of the conditional expression. The appropriate branch will get a token carrying all live variables in the corresponding expression.
- **Merge** (Fig. 4(d)): Merge components are used to route tokens from a call site to the called function whenever there is more than one possible call site for a given function. Tokens received at the two input channels contain all the parameters of the function. In the case of a closure call, a pointer to the corresponding closure environment is also contained in the token. An arbiter ensures that every token received is sent to the component that implements the function's body, one at a time. Merge compo-

nents are connected together in a tree whenever there is more than two call sites for a function.

- **Closure** (Fig. 4(e)): Closure components are used to allocate and read the environments associated with function closures. They have two pairs of input and output channels, one for allocating and the other for reading. When a closure needs to be allocated, a token containing all live variables in the expression that encloses the closure declaration is received. All free variables in the function are saved at an unused address in a local RAM and a token containing that address, a tag identifying the function and all variables that are live in the declaration's continuation is sent to the component that implements that continuation. On a closure call, a token containing both the address of the closure's environment and the actual parameters is received, the free variables are fetched from the local RAM and a token containing the free variables and the actual parameters is sent to the component that implements the function's body. The closure component's read channel is connected to the output of the merge node(s) and the channel for the value read goes to the component that implements the function body. Since each closure has its dedicated block of RAM, the data width is exactly what is needed to save all free variables and no memory is wasted. Closure environments can be written or read in a single operation.
- **Vector** (Fig. 4(f)): Vector components are used to implement global vectors introduced through the `make-vector` primitive. They have two pairs of input and output channels, one for writing and another for reading. When the `vector-set!` primitive is called, a token is received with all live variables, an address and data to be written at that address. The data is written and a token with the live variables is sent as output. When the `vector-ref` primitive is called, a token containing all live variables and an address is received. Data is read from that address and sent in an output token along with the live variables. A block of RAM is associated with each vector and is sized accordingly.
- **Par** (Fig. 4(g)): Par components are used to implement `par` binding expressions. Like the closure component, the par component has an allocation and a reading part, respectively called *fork* and *join*. When a token is received for *fork*, it contains all the live variables of the `par` expression. All variables that are free in the `par`'s body are saved in a local RAM, much like for a closure environment; the corresponding address is an identifier for the `par` binding expressions' continuation. Then, tokens are sent simultaneously (forked) to the components that implement the binding expressions. Each of these parallel tokens contains the binding expression's continuation pointer and free variables. When a token is received for *join*, the binding expression's return value is saved in the local RAM along with the continuation's free variables. When the last token for a given identifier is received for *join* the return value is sent to the `par`'s body along with the other branches' return values and the free variables saved in the local RAM for that identifier, and the memory for that identifier is deallocated. Currently only two binding expressions are supported.
- **Input** (Fig. 4(h)): Input components implement all declared input channels in the circuit. It can be viewed like a simplified *join* part of a par component: it waits until it has received tokens from both inputs before sending one as output. One of the input tokens represents the control flow and contains all live variables in the call to the input function. The other input token contains the data present on the corresponding top-level signal of the circuit. The output token contains data from both input tokens.
- **Output** (Fig. 4(i)): Output components implement output channels. They act like a simplified *fork* part of a par component: whenever a token is received as input, two output tokens are sent simultaneously as output: one to the corresponding top-level signal of the circuit and one to the component that implements the continuation to the call to the output function.

The system has been designed so that all components can be implemented either as synchronous (clocked) or asynchronous components. For easy integration with the other synthesis and simulation tools available to us, our prototype currently uses clocked components reacting to the rising edge of the clock.

Input and output components can be implemented to support different kinds of synchronization. All experiments so far have been done using a four-phase handshake with passive inputs and active outputs: input components wait until they receive a request from the outside world and have to acknowledge it while output components send a request to the outside world and expect an acknowledgment. This allows linking of separately compiled circuits by simply connecting their IO channels together.

5. Compilation Process

The core of the compilation process is a pipeline of the phases described in this section. The 0-CFA is performed multiple times, as sub-phases of parallelization and inlining, and as a main phase by itself.

5.1 Parallelization

The compiler can be configured to automatically parallelize the computation. When this option is used, the compiler looks for sets of expressions which can be safely evaluated concurrently (side-effect free) and binds them to variables using a `par` construct. This is done only when at least two of the expressions are non-*simple*, since *simple* expressions are evaluated concurrently anyways (they are implemented as combinatorial circuits in a single stage) and the `par` construct produces a hardware component that implements the fork-join mechanism, which would be useless overhead in this case.

This stage is implemented as four different sub-stages. First a control flow analysis is performed on the program (see Section 5.5) in order to determine which expressions may actually have side-effects and which functions are recursive. Then, for all calls with arguments that are non-*simple*, those arguments are replaced with fresh variable references and the modified calls form the body of a `let` that binds those variables to the original arguments. For example,

```
(f (fact x) (- (fib y) 5) 3)
```

becomes

```
(let ((v_0 (fact x))
      (v_1 (- (fib y) 5)))
  (f v_0 v_1 3))
```

Next, all `let`s are analyzed and those for which all binding expressions have no side-effects and are non-*simple* are replaced by `par`s. Finally, the transformed program is analyzed to find all `par`s that may introduce an arbitrary number of tokens into the same part of the pipeline. These are the `par`s for which at least two binding expressions loop back to the `par` itself (e.g. a recursive function that calls itself twice). Any recursive function that can be called from the binding expressions is then tagged as "dangerous". The reason for this last step is that recursive functions are implemented as pipelines that feed themselves and each of these can only hold a given number of tokens at a given time before a deadlock occurs.

This tagging is used later in the compilation process to insert fifo buffers to reduce the possibility of deadlock.

5.2 CPS-Conversion

The compiler uses the CPS-Conversion to make the function call linkage of the program explicit by transforming all function calls into tail function calls. Functions no longer return a result; they simply pass the result along to another function using a tail call. Functions receive an extra parameter, the continuation, which is a function that represents the computation to be performed with the result. Where the original function would normally return a result to the caller, it now calls its continuation with this result as a parameter.

Since all functions now have an extra parameter, all calls must also be updated so that they pass a continuation as an argument. This continuation is made of the "context" of the call site embedded in a new lambda abstraction with a single parameter. The body of the continuation is the enclosing expression of the call where the call itself is replaced by a reference to the continuation's parameter. Syntactically the call now encloses in its new argument the expression that used to enclose it. For example,

```
(letrec ((fact (lambda (x)
                 (if (= 0 x)
                     1
                     (* x (fact (- x 1)))))))
  (+ (fact 3) 25))
```

becomes

```
(letrec ((fact (lambda (k x)
                 (if (= 0 x)
                     (k 1)
                     (fact (lambda (r) (k (* x r)))
                           (- x 1))))))
  (fact (lambda (r) (+ r 25)) 3))
```

There are two special cases. The program itself is an expression that returns a value so it should instead call a continuation with this result, but the normal conversion cannot be used in this case since there is no enclosing expression. The solution is to use a primitive called `halt` that represents program termination.

Because of the parallel fork-join mechanism, we also need to supply the parallel expressions with continuations. This is done in a similar way using a `pjoin` primitive which includes information about the `par` form that forked this process. This represents the fact that parallel sub-processes forked from a process must be matched with each other once they complete. For example,

```
(par ((x (f 3))
      (y (f 5)))
  ...)
```

becomes

```
(par pid_123
    ((x (f (lambda (r) (pjoin pid_123 r 0)) 3))
     (y (f (lambda (r) (pjoin pid_123 r 1)) 5)))
  ...)
```

`pid_123` is bound by the `par` at fork time and corresponds to the newly allocated address in the local RAM. `pjoin`'s last parameter (`0` or `1`) distinguishes the two sub-processes.

5.3 Lambda Lifting

Lambda lifting [16] is a transformation that makes the free variables of a function become explicit parameters of this function. Using this transformation, local functions can be lifted to the top-level of the program. Such functions have no free-variables and are called combinators. For example,

```
(let ((x 25))
  (let ((f (lambda (y) (+ x y))))
    (f 12)))
```

becomes

```
(let ((x 25))
  (let ((f (lambda (x2 y) (+ x2 y))))
    (f x 12)))
```

which is equivalent to

```
(let ((f (lambda (x2 y) (+ x2 y))))
  (let ((x 25))
    (f x 12)))
```

Since a combinator has no free-variables, it doesn't need to be aware of the environment in which it is called: all the values that it uses are explicitly passed as parameters. Combinators are closures that hold no data and therefore, in our system, we do not assign a closure memory to them. For example, if function `f` is not lambda lifted in the above example, it needs to remember the value of `x` between the function declaration and the function call; this would normally translate to a memory allocation at the declaration and a read at the call (see Section 5.6). After lambda lifting, `x` does not need to be known when `f` is declared since it will be explicitly passed as parameter `x2` on each call to `f`. The use of lambda lifting in our compiler helps to reduce the amount of memory used in the output circuit and to reduce latency.

Lambda lifting is not possible in all situations. For example:

```
(letrec ((fact (lambda (k x)
                 (if (= 0 x)
                     (k 1)
                     (fact (lambda (r) (k (* x r)))
                           (- x 1))))))
  (fact (lambda (r) r) 5))
```

This is a CPS-converted version of the classic factorial function. In this case, function `fact` needs to pass its result to continuation `k`, which can be the original continuation `(lambda (r) r)` or the continuation to a recursive call `(lambda (r) (k (* r x)))`. The continuation to a recursive call needs to remember about the parameters to the previous call to `fact` (`k` and `x`). We could add those free variables as parameters, like `(lambda (r k x) (k (* r x)))`, but then `fact` would need to know about the parameters to its previous call in order to be able to call its continuation, thus adding parameters to `fact` as well. Since `fact` is a recursive function and each recursive call needs to remember the parameters of the previous call, we would end up with a function that needs a different number of arguments depending on the context in which it is called, and this number could be arbitrarily large. Such cases are handled by closure conversion (Section 5.6) which identifies which closures actually contain data that needs to be allocated. In the `fact` example, the allocation of the free variables of the continuation to a recursive call (`k` and `x`) corresponds to the allocation of a stack frame in a software program.

5.4 Inlining

Inlining is a transformation which puts a copy of a function's body at the function's call site. In the circuit this corresponds to a duplication of hardware components. Although the resulting circuit is larger than could be, the circuit's parallelism is increased, which can yield faster computation. For this reason the compilation process includes an optional inlining phase.

The only information given to this phase by the developer is the maximum factor by which the code size should grow. Code size and circuit size is roughly approximated by the number of nodes in the corresponding AST. Since parallelism can only occur within

`par` expressions, inlining is done only in `par` binding expressions that have been tagged as "dangerous" by the parallelization phase (see Section 5.1). No inlining will occur in a program that does not exploit parallelism.

The inlining process is iterative and starts by inlining functions smaller than a given "inlining" size in all the identified expressions. If no function can be inlined and the desired growth factor has not been reached, this inlining size is increased and the process is iterated. Since inlining a function introduces new code in a `par`'s binding expressions, this can offer new candidate functions for inlining which may be much smaller than the current inlining size. The inlining size is therefore reset to its initial value after every iteration in which inlining actually occurred.

At each iteration, the number of callers for each inlinable function is calculated, functions are sorted from the most called to the least called and then treated in that order. The goal is to try to first duplicate components that have more chances of being a sequential bottleneck. 0-CFA is also done at each iteration in order to be aware of new call sites and the call sites that have vanished.

This method does not consider the fact that the size of the resulting circuit is not directly related to the size of the AST. In particular, the networking needed to connect calls to function bodies may grow quadratically as functions are duplicated. This is due to the fact that calls must be connected to every function possibly called, and the number of call sites also grows when code grows. An example of this is given in Section 9.

5.5 0-CFA

The 0-CFA (Control Flow Analysis [18]) performs a combined control flow and data flow analysis. Using the result of the 0-CFA, the compiler builds the control flow graph of the program. This is a graph that indicates which functions may be called at each function call site. This graph indicates how the circuit's components are interconnected, with each edge corresponding to a communication channel from the caller to the callee. When several edges point to the same node, we know that this function needs to be preceded by a tree of merge components (e.g. part A of Figure 2).

This analysis is also used for automatic parallelization and inlining as explained in Sections 5.1 and 5.4, and to assign locally unique identifiers to functions (see Section 5.7).

Abstract interpretation is used to gather the control flow information and that information is returned as an abstract value for each node in the AST. An abstract value is an upper bound of the set of all possible values that a given expression can evaluate to. In our case, all values other than functions are ignored and the abstract value is just a list of functions which represents the set containing those functions along with all non-function values.

5.6 Closure Conversion

Closure conversion is used to make explicit the fact that some functions are actually closures that contain data (free-variables); those are the functions that could not be made combinators by lambda lifting (Section 5.3). This conversion introduces two new primitives to the internal representation: `%closure` and `%clo-ref`. The `%closure` primitive is used to indicate that a function actually is a closure for which some data allocation must be made; its first parameter is the function itself and the rest are values to be saved in the closure memory. The `%clo-ref` is used within closures to indicate references to variables saved in the closure memory; it has two parameters: the first is a "self" parameter that indicates the address at which the data is saved and the second one is an offset within the data saved at that address (field number within a record), with `0` representing the function itself (not actually saved in memory). For example,

```
(letrec ((fact (lambda (k x)
                 (if (= 0 x)
                     (k 1)
                     (fact (lambda (r) (k (* x r)))
                           (- x 1))))))
  (fact (lambda (r) r) 5))
```

becomes

```
(letrec ((fact (lambda (k x)
                 (if (= 0 x)
                     ((%clo-ref k 0) k 1)
                     (fact
                       (%closure
                         (lambda (self r)
                           ((%clo-ref
                              (%clo-ref self 2)
                              0)
                            (%clo-ref self 2)
                            (* r (%clo-ref self 1))))
                         x
                         k)
                       (- x 1)))))))
  (fact (%closure (lambda (self r) r)) 5))
```

5.7 Finalization

The finalization stage consists of three sub-stages: a trivial optimization, a "cosmetic" transformation to ease the job of the backend, and information gathering.

The first sub-stage merges sequences of embedded `let` expressions into a single `let`, when possible. It checks for a `let` in the body of another one and extracts the bindings in the embedded `let` that do not depend on variables declared by the embedding `let`. Those extracted bindings are moved up from the embedded `let` to the embedding one. If the embedded `let` ends up with an empty binding list, it is replaced by its own body as the body of the embedding `let`. For example,

```
(let ((a (+ x 7)))
  (let ((b (* y 6)))
    (let ((c (- a 3)))
      ...)))
```

becomes

```
(let ((a (+ x 7))
      (b (* y 6)))
  (let ((c (- a 3)))
    ...)))
```

This is done because each `let` translates directly to a pipeline stage in hardware; instead of operations being done in sequence in several stages, they are done concurrently in a single stage thus reducing latency and circuit size.

The next sub-stage is used to make explicit the fact that closures must be allocated before they are used. At this point in the compiler, arguments to calls are all values (literals or closures) or references to variables, so that a function call would be a simple connection between the caller and the callee. The only exception to this is that some closures contain data that must be allocated and these are represented by both a lambda identifier and an address that refers to the closure memory. To make everything uniform, closures that contain data are lifted in a newly created `let` that embeds the call. This way, we now have a special case of `let` that means "closure allocation" and the function call becomes a single stage where all arguments can be passed the same way. For example,

```
(foo 123 x (%closure (lambda (y) ...) a b))
```

becomes

```
(let ((clo_25 (%closure (lambda (y) ...) a b)))
  (foo 123 x clo_25))
```

so that it is clear that `a` and `b` are allocated in the closure memory in a stage prior to the function call.

The last sub-stage of finalization is used to assign locally unique identifiers to lambda abstractions to be used in the circuit instead of globally unique identifiers. The reason for this is that IDs take $\lceil \log_2 n \rceil$ bits to encode, where n can be the number of lambdas in the whole program (global IDs), or the number of lambdas in a specific subset (local IDs); our aim is to reduce the width of busses carrying those IDs. Since we have previously performed a 0-CFA, it is possible to know which lambdas may be called from a given call site. We first make the set of all those sets of functions and then merge the sets of functions that have elements in common until all are disjoint. This ensures that each lambda has an ID that, while not being necessarily unique in the program, gives enough information to distinguish this lambda from others in all call sites where it might be used.

5.8 Back-End

The back-end of the compiler translates the finalized Abstract Syntax Tree (AST) into a description of the circuit. The description is first output in an intermediate representation that describes the instantiation of several simple generic components and the data busses used as connections between them. This intermediate representation can be used for simulation and it is translated to VHDL through a secondary, almost trivial back-end (see Section 8).

Data busses are of two different types: `bus` and `join`:

- `(bus` *`width val`*`)`: describes a bus *width* bits wide with an initial value of *val*.
- `(join` *`subbus`*$_1$ `...` *`subbus`*$_n$`)`: describes a bus which is made of the concatenation of one or more other busses.

Integer literals and variable references are translated to busses; literals are constant busses while references are busses that get their value from the component that binds the variable (e.g. stage). All components that implement expressions through which some variable is live will have distinct input and output busses to carry its value, like if a new variable was bound at every pipeline stage.

As explained in Section 4, each `let` expression is translated to a stage component followed by a combinatorial circuit that implements the binding expressions and the component that implements the `let`'s body. The binding of a variable that represents a closure is translated to the *alloc* part of a closure component. Parallel bindings are translated to par components with the *fork* outputs and the *join* inputs connected to the binding expressions and the *join* output connected to the component that implements the `par`'s body. At this point in compilation, `letrec` bindings contain nothing else than function definitions so they are translated to the implementation of their body and all functions defined.

Function calls are translated to stage components where the combinatorial circuit is used to test the tag that identifies the function called and route the token accordingly. The result of the 0-CFA is used to determine which expressions call which functions. No stage is present if only one function can be called from a given point. Since all actual parameters are *trivial* expressions at this point in compilation, a connection from caller to callee is all that is needed.

Conditionals are translated to split components with each output connected to the component that implements the corresponding expression (*`true-exp`* or *`false-exp`*). As explained in Section 4, the condition is a *trivial* expression and its value is used directly to control a multiplexer.

Lambda abstraction declarations are translated to the bus that carries the function's or closure's identifying tag and address. Definitions are translated to the component that implements the function's body, possibly preceded by a tree of merge nodes and/or the *read* part of a closure component. The result of the 0-CFA is used to build the tree of merge nodes. Input and output channels are translated just like functions with an input or output component as a body.

Primitives may be translated in different ways: arithmetic primitives are translated to the equivalent combinatorial circuits while calls to the `timer` primitive – which returns the number of clock cycles since the last circuit reset – are similar to function calls to a global timer component. Other primitives are used internally by the compiler and each is translated in its own way. For example, the `halt` primitive terminates a process and the similar `pjoin` primitive indicates that a sub-process forked by a `par` has completed.

The compiler also adds an input and an output channel to the top-level circuit. The input channel carries tokens containing all free variables in the program and is used to start a new process; in most cases, this channel carries no data and is used only once since concurrent processes can be created using `par` expressions. The output channel carries tokens that contain the return value of the process; a token is output whenever the control reaches the `halt` primitive and indicates that a process has completed. Thus, the whole circuit can be seen as a function itself.

6. Memory Management

Memory management has been made as simple as possible since this is not our main research interest. As mentionned before, the memory associated to a closure is freed as soon as the closure is called. While this is enough to make our language Turing-complete, it imposes a very particular programming style, is error prone and makes inefficient tricks become indispensable in some situations.

The most desirable solution would, of course, be a full garbage collector as it normally exists in Scheme. Since closures can be stored in other closures (and this is always the case for continuations), a garbage collecting hardware component would need to be connected to all closure memories in a given system. The garbage collector would also need to be aware of all closures contained in tokens flowing in the system. Such a centralized component would hinder parallel execution and is in no way trivial to implement.

A reasonable alternative to a full garbage collector is to augment our memory management mechanism with a manual memory deallocation mechanism. This could be done as in many languages by using a "free" primitive. Closure memory components would need a new pair of input/output channels to support this, which would be connected to "free" call sites much like functions are connected to their call sites. It would also be possible to let the programmer indicate which closures are to be manually reclaimed and let the others be reclaimed automatically as is currently the case, thus reducing the size and latency of the resulting circuit.

Another issue is the amount of memory that is reserved for closures. Since each closure has its own block of RAM, this block has to be large enough to hold the largest number of closures that can exist concurrently, lest a deadlock might occur. Our prototype currently sets all closure memories to the same depth, which results in far more RAM being generated than necessary. One solution would be to use smaller closure memories and allow them to spill to a global memory; they would thus become local, distributed caches. Finding the optimal size for each local cache would be the main goal in order to minimize concurrent requests to the main memory. A non-trivial, multi-ported implementation of the global memory might be necessary in order to achieve good levels of parallelism.

Finally, the current implementation of vectors creates a bottleneck in parallel circuits since each vector is a single component and it cannot be duplicated like a function. A solution would be to split each vector into several independently accessible sub-vectors

controlled by a multiplexer which would route the request to the appropriate sub-vector.

7. Behavioral Simulation

The intermediate representation generated by the primary back-end is itself a Scheme program: it can be printed out in S-expression syntax and then executed to perform a simulation. This is done by using a simulator and behavioral descriptions of the components, both written in Scheme and included as external modules to the intermediate representation.

The simulator provides functions to manipulate wires and busses and to supply call-backs for some events: a signal transition, an absolute time from the beginning of simulation or a delay relative to the actual simulation time. In the behavioral simulator, it is therefore possible to write "when the input signal becomes 1, wait for 10ns and set the value of the output signal to 0" as:

```
(on input (lambda ()
            (if (= 1 (wire-value input))
                (lambda ()
                  (in 10 ;;time is in ns.
                      (lambda ()
                        (wire-update! output 1))))
                void)))
```

The program generated by the back-end also includes a test function which can be modified by hand to specify simulation parameters (input values, duration, etc). When the program is run, it produces output in VCD format (Value Change Dump, described in [1]). This output indicates the initial values of all signals in the circuit and all transitions that occurred during the simulation and can be sent to a standard waveform viewer (e.g. GTKWave).

8. Implementation

Hardware implementations are described using the VHDL language. All components listed in Section 4 are implemented as VHDL entities and architectures using generic parameters for bus widths, memory and fifo depths, etc. Most components have input signals for the global clock and reset signals.

For example, the stage VHDL component has an input channel and an output channel, and a `bus_width` generic parameter to specify the width of those channels. An internal register saves the input data at the rising edge of the clock on a successful handshake, and is cleared when the reset signal is asserted. Each channel is associated with a pair of wires that carry the request and acknowledge signals for synchronization; request signals go in the same direction as the data and acknowledge signals go the opposite way.

The top-level of the circuit is also translated from the Scheme program described above into a VHDL entity and architecture which instantiates all the necessary components. In addition to components described in Section 4, trivial combinatorial components like adders and equality testers are also used in the top-level.

The most difficult aspect of generating a VHDL circuit description is to handle `join` busses properly. There is no standard VHDL construct to express that some bus is in fact just an alias for the concatenation of other busses; these have to be translated to one-way assignments, either assigning the concatenation of several busses to a `join` bus or assigning a slice of a `join` to another bus. The rest is a straightforward translation of busses and components from the intermediate representation to VHDL, including bus renaming.

9. Results

We have tested our prototype on a number of programs to produce dataflow machines on an FPGA. The compiler's VHDL output is fed to Altera's Quartus-II development environment. The only human intervention necessary at this point is the assignment of the circuit's external signals to FPGA pins; other constraints can also be given to the synthesis tool, for example to force it to try to produce a circuit that runs at a specific clock speed.

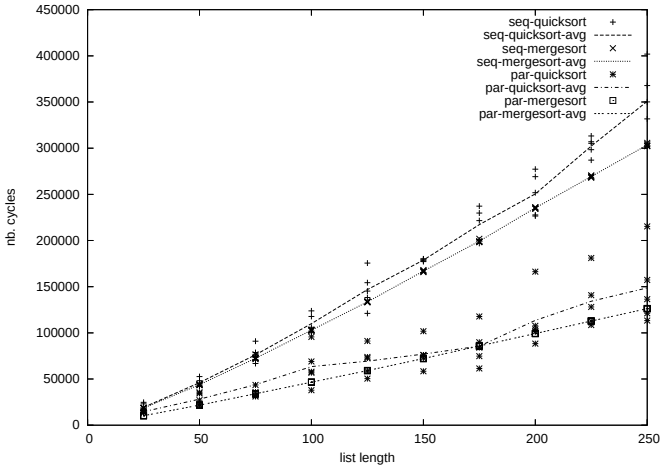

Figure 5. Sequential and parallel mergesort vs. quicksort, number of cycles as a function of list length

As an example, the quicksort and mergesort algorithms have been implemented in an Altera Stratix EP1S80 FPGA with a speed grade of -6. This FPGA contains 80,000 configurable cells. The list of elements is represented as a vector for quicksort and as a chain of closures for mergesort. The resulting circuits use about 11% and 14% of the reconfigurable logic and about 5% and 8% of the memory available in the FPGA, respectively, for lists of up to 256 16-bit integers and can run at clock rates above 80MHz. Also, mergesort is an algorithm for which the automatic parallelization stage of the compiler is useful.

Figure 5 shows the number of clock cycles required to sort lists of different lengths using mergesort and quicksort, for sequential and parallel versions of the algorithms. The parallel mergesort was automatically obtained by the compiler from a program without `par` expressions. Because of the vector mutations in the quicksort algorithm, the compiler could not obtain a parallel version automatically; it was necessary to manually insert a `par` expression for the recursive calls.

Figure 6 shows average clock cycles per element and compares sequential and parallel versions of both programs. It shows that a simple, sequential algorithm can gain a lot in terms of performance by using the parallelization stage of the compiler, or through simple modifications (changing `lets` to `pars`); performance is then limited by the amount of hardware used (e.g. components can be duplicated to gain more parallelism).

The fact that the quicksort algorithm is slower than the mergesort algorithm in our tests comes mainly from an inefficient implementation of vectors. Quicksort implemented using a chain of closures is, on average, faster than mergesort for sequential execution and about as fast for parallel execution.

Table 1 illustrates the effect of inlining (Section 5.4) on performance and circuit size. The program used for this test is the mergesort algorithm shown in Figure 1. In this program, the function which is inlined most often is `cons`, which has the effect of distributing the memory used to store the list in several independent memory blocks; with an inlining factor of 1.10, it is the only function that gets inlined and it is inlined five times out of a total

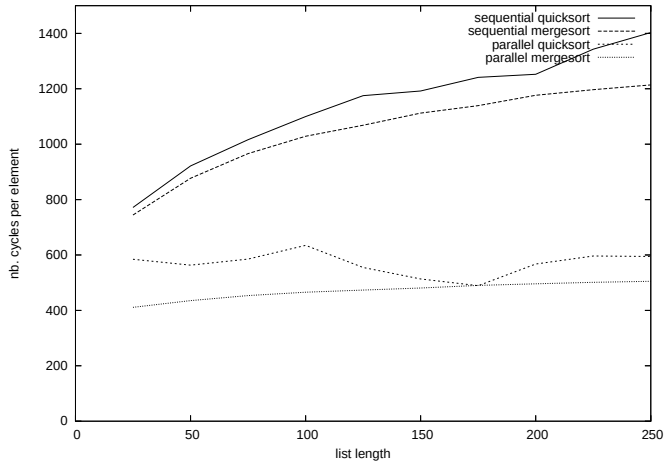

Figure 6. Parallel vs. Sequential mergesort and quicksort, average number of cycles per element as a function of list length

Inlining factor	% of logic	merge components	cycles to sort 250 elts.	% of baseline's (1.00) cycles
1.00	14	57	126,226	100.0
1.10	21	107	110,922	87.9
1.25	22	110	95,486	75.6
1.50	32	204	91,684	72.6
2.50	74	709	96,006	76.1

Table 1. Effect of inlining on mergesort

of seven call sites within `par`s. The proportion of logic is given for the Stratix EP1S80.

As mentioned in Section 5.4, the circuit size is not proportional to the AST size. To illustrate this, the number of merge components is given for each inlining factor. This outlines the fact that, by duplicating code, each function is potentially called from several more places. Area usage quickly becomes prohibitive as the inlining factor is increased. Also, more inlining does not always translate to a better performance: as the tree of merge components at each function entry gets bigger, the pipeline gets deeper and the latency increases; there is no need to have a lot more components than the maximum number of simultaneous tokens in the circuit.

To test the implementation of vectors we wrote a program which interprets a machine language for a custom 16-bit processor. Vectors are used to implement the RAM and the program memory. The instruction set contains 21 simple 16-bit instructions, some of which use a single immediate integer value. With the RAM and program memory both at 4096 elements deep, the circuit uses only 10% of the logic and 3% of the memory in a Stratix EP1S80. Unfortunately the execution speed is poor, in part because our language's lack of a `case` construct forced us to use nested `if`s to decode the instructions. It is exciting to consider that with some extensions to our system it might be possible to generate a "Scheme machine" processor by compiling an `eval` suitably modified for our system. Moreover, a multithreaded processor could be obtained easily by adding to the instruction set operations to fork new threads.

Tests have also been performed on the SHA-1 hashing algorithm. Since this algorithm always uses a fixed amount of memory, it has been written so that it does not use memory allocated data structures. Instead, each function receives all the values it needs as separate parameters. Input data is received in a stream from an input channel and new values are read only when the circuit is ready to process them. This has the effect of reducing latency since fewer closures have to be allocated, but it also means that tokens, and therefore data busses, can be very large. Closure memories for continuations also need to store more variables and the circuit ends up taking 39% of the logic and 23% of the memory in a Stratix EP1S80 device. This program highlights several situations in which simple optimizations could be added to the compiler to reduce the size of the circuit.

10. Conclusions

We have presented a compiler that automatically transforms a high level functional program into a parallel dataflow hardware description. The compilation process, from a Scheme-like language to VHDL, requires no user intervention and the approach has been validated on non-trivial algorithms. Our system handles tail and non-tail function calls, recursive functions and higher-order functions. This is done using closure memories which are distributed throughout the circuit, eliminating bottlenecks that could hinder parallel execution. The dataflow architecture generated is such that it could be implemented with power-efficient asynchronous circuits.

10.1 Related Work

Other research projects have studied the possibility of automatic synthesis of hardware architectures using software programming languages. Lava [4] allows the structural description of low-level combinatorial circuits in Haskell by the use of higher-order functions. It does not translate functional programs into hardware. Handel-C [6] is a subset of the C language which can be compiled directly to hardware, but it lacks support for features which are common in C, like pointers. Moreover it only supports inlined functions ("macros" which cannot be recursive). Scheme has also been applied to hardware synthesis in the context of the Scheme Machine project at Indiana University [20][15][5]. That work also does not support non-tail function calls and higher-order functions.

10.2 Future Work

In this work, our focus was to show that it is feasible to compile a functional description of a computation into a parallel circuit. We think it would be good to implement our generic hardware components in asynchronous logic to target very low power circuits. Asynchronous FPGAs [19] are being designed and these chips would be the perfect targets for our approach. As mentioned previously, an exciting prospect is the application of our compilation technique to hardware/software co-design for reconfigurable chips containing embedded processors and to Globally Asynchronous Locally Synchronous (GALS) architectures [7] which allow very high speed and massively parallel execution by eliminating the need for a global clock.

Several optimizations normally applied to software programs can be added to our compiler to produce more efficient circuits. For example, constant propagation can be used to reduce the width of busses and the size of memories, and even eliminate some superfluous closures. The simple inlining technique described in Section 5.4 could be replaced by a more clever one or one that can take into account the amount of logic available to implement the circuit or the desired level of parallelism. Common subexpression elimination, which compacts the circuit and reduces parallelism, may also be interesting to explore for space constrained applications.

As explained in Section 6, several improvements could be made to the memory management.

Our language lacks some useful constructs, such as `case` expressions, dynamically allocatable vectors, and data types, which would greatly enhance its expressiveness. A type system would

also be useful to determine the width of busses and memories and to perform static type checking.

References

[1] *IEEE Std 1364-2001 Verilog® Hardware Description Language*. IEEE, 2001.

[2] A. W. Appel and T. Jim. Continuation-passing, closure-passing style. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 293–302. ACM Press, 1989.

[3] G. M. Birtwistle and A. Davis, editors. *Asynchronous Digital Circuit Design*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1995.

[4] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: hardware design in Haskell. In *ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 174–184, New York, NY, USA, 1998. ACM Press.

[5] R. G. Burger. The Scheme Machine. Technical Report Technical Report 413, Indiana University, Computer Science Department, August 1994.

[6] Celoxica. *Handel-C Language Reference Manual RM-1003-4.0*. http://www.celoxica.com, 2003.

[7] A. Chattopadhyay and Z. Zilic. GALDS: a complete framework for designing multiclock ASICs and SoCs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 13(6):641–654, June 2005.

[8] D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, and S. Zhao, editors. *SpecC: Specification Language and Methodology*. Springer, 2000.

[9] D. Geer. Is it time for clockless chip? *Computer*, pages 18–21, March 2005.

[10] C. Giraud-Carrier. A reconfigurable dataflow machine for implementing functional programming languages. *SIGPLAN Not.*, 29(9):22–28, 1994.

[11] R. Gupta and G. D. Micheli. Hardware/Software Co-Design. In *IEEE Proceedings*, volume 85, pages 349–365, March 1997.

[12] S. Gupta, N. Dutt, R. Gupta, and A. Nicola. SPARK : A High-Level Synthesis Framework For Applying Parallelizing Compiler Transformations. In *International Conference on VLSI Design*, New Delhi, India, January 2003.

[13] J. Guy L. Steele. Rabbit: A Compiler for Scheme. Technical report, Cambridge, MA, USA, 1978.

[14] C. Hewitt, P. Bishop, and R. Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proc. of the 3rd International Joint Conference on Artificial Intelligence*, pages 235–245, 1973.

[15] S. D. Johnson. Formal derivation of a scheme computer. Technical Report Technical Report 544, Indiana University Computer Science Department, September 2000.

[16] T. Johnsson. Lambda lifting: transforming programs to recursive equations. In *Functional programming languages and computer architecture. Proc. of a conference (Nancy, France, Sept. 1985)*, New York, NY, USA, 1985. Springer-Verlag Inc.

[17] R. Kelsey, W. Clinger, and J. Rees (eds.). Revised^5 Report on the Algorithmic Language Scheme. In *Higher-Order and Symbolic Computation*, volume 11, August 1998.

[18] O. G. Shivers. *Control-flow analysis of higher-order languages of taming lambda*. PhD thesis, Carnegie Mellon University, 1991.

[19] J. Teifel and R. Manohar. An Asynchronous Dataflow FPGA Architecture. *IEEE Transactions on Computers (special issue)*, November 2004.

[20] M. E. Tuna, S. D. Johnson, and R. G. Burger. Continuations in Hardware-Software Codesign. In *IEEE Proceedings of the International Conference on Computer Design*, pages 264–269, October 1994.

[21] C. Van Berkel, M. Josephs, and S. Nowick. Applications of asynchronous circuits. In *Proceedings of the IEEE*, volume 87, pages 223–233, Feb. 1999.