A Scheme compiler for Hardware Dataflow Machines

Xavier Saint-Mleux[*,1], Marc Feeley[*,1], Jean-Pierre David[†,2]

[*] *Université de Montréal*

[†] *École polytechnique de Montréal*

ABSTRACT

We describe a prototype compiler that compiles general functional programs into dataflow parallel hardware. The compiler supports non-tail function calls and higher-order functions. Our approach makes it possible for software developers to design hardware embedded systems with little prior knowledge of electronic circuits.

1 Introduction

This paper presents a compiler for a subset of the Scheme functional language that generates descriptions of hardware dataflow machines as output, without any human intervention in the process. The dataflow machine will be fully implemented in hardware, for example in an FPGA. It is derived from an existing software compiler, with a back-end that instantiates generic components written in VHDL. The output specification can then be fed to synthesis tools to get an actual physical implementation of the circuit.

The compiler as described below is a work in progress and is no more than a proof of concept but it can already be used to easily and safely implement any algorithm in hardware. Several successful tests have been performed using Synplicity and Altera synthesis software and an Altera Stratix FPGA.

2 Approach

The basic idea is to use distributed local memories instead of a global one in order to maximize the potential for parallel circuits. Many systems based on dataflow machines suffer from a bottleneck for memory accesses[GC94]; this is what we aim to avoid. In our prototype, the only kind of data allocation that may occur is the creation of a function closure. Therefore, each procedure that actually needs to remember about a local environment is associated with its own block of memory in hardware. Data is allocated each time a closure is created and it is automatically deallocated when the procedure is called.

[1]E-mail: {saintmlx,feeley}@iro.umontreal.ca

[2]E-mail: jpdavid@polymtl.ca

3 Source Language

The source language is a functional language based on S-expressions and using a parenthesized syntax. It supports the following types of primitive expressions:
- Integer literals
- Variable references
- Procedures: `(lambda (`*params*`)` *body*`)`
- Procedure calls
- Conditionals: `(if` *cond true-exp false-exp*`)`
- Binding constructs: `let`, `letrec` and `par`
- Expression sequences
- I/O channel procedures: `(input-chan` *name*`)` and `(output-chan` *name*`)`

Primitives are supplied for arithmetic operations and comparisons on integers. As an example the recursive factorial function can be implemented and called as follows:

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

Currently, the only types of data supported are integers and closures. 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 a b))))
      (car (lambda (h t) h)))
  (let ((pair (cons 3 4)))
    (pair car)))
```

The procedure call `(cons 3 4)` allocates memory for the two integers, but this memory is reclaimed as soon as `pair` is called; `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.

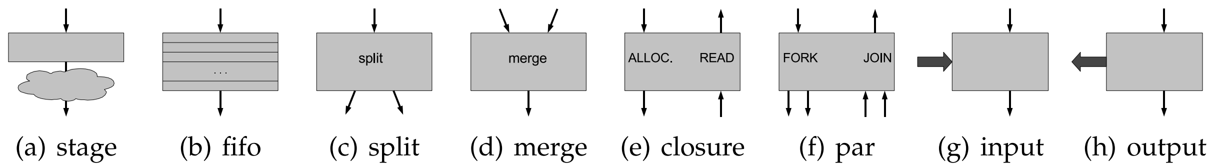
The `par` binding construct is syntactically and semantically similar to the `let` construct but it indicates that the binding expressions should be evaluated in parallel. They can be seen as a calculation with a continuation that takes several "return" values.

The I/O channel procedures create named input or output channels that are used like procedures. Input channels are thunks (functions with no arguments) that return the value read and 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 circuit. 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 again:

```
(let ((cin0 (input-chan chan_in0))
      (cin1 (input-chan chan_in1))
      (cout (output-chan chan_out)))
  (letrec ((doio (lambda () (cout (+ (cin1) (cin2)))
                   (doio))))
    (doio)))
```

4 Generic Hardware Components

The output circuit will be made of instantiations of the following generic components:
- Stage (Fig. 1(a)): A register followed by some combinatorial circuit that implements primitive calls such as addition and comparison.



(a) stage (b) fifo (c) split (d) merge (e) closure (f) par (g) input (h) output

Figure 1: Generic Components

- Fifo (Fig. 1(b)): A memory that can accumulate tokens when the rest of the circuit is not ready to process them.
- Split (Fig. 1(c)): Sends any token received to either output channel depending on a supplied boolean value.
- Merge (Fig. 1(d)): Sends tokens received on any input to the output, one at a time.
- Closure (Fig. 1(e)): Memory to allocate the local environment for a function closure. When a token is received for allocation (closure instantiation), part of its data is saved in the memory at an unused address and the rest is sent as output along with that address. When a token is received for reading (closure call), part of its data is used as an address to fetch the environment from memory and this environment is sent as output with the rest of the input data; that address is marked as free again.
- Par (Fig. 1(f)): Memory for the synchronization of expressions evaluated in parallel. When a token is received for forking, part of its data is saved in the local memory at an unused address and two tokens containing the rest of the input data and that address are sent simultaneously as outputs. When a token is received for a join, part of its data is used as an address in the memory. If it is the first token to arrive for that address, its data is saved along with what was already at that address; if it is the second one, saved values are fetched from the given address and sent as output along with the input data.
- Input (Fig. 1(g)): Reads data from an external synchronized input channel.
- Output (Fig. 1(h)): Writes data to an external synchronized output channel.

5 Back-End

The back-end creates pipelines by connecting instances of generic components one after the other. Tokens flowing through these pipelines will contain all live variables at a given point in the program. All values, integers or closures, use busses of the same fixed width in bits; closures are identified by a procedure ID and possibly an address to the local environment.

After the rest of the compilation process (CPS conversion[Appe89], 0-CFA[Shiv91], etc.), the results of all simple expressions (primitive applications to literals or variable references) are bound to variables in `let` expressions. Each let is translated into a stage followed by a combinatorial circuit that implements the required primitives, itself followed by the circuit that corresponds to the `let`'s body.

Procedure calls are implemented by sending a token with all parameters to the part of the circuit that implements the procedure. The procedure ID is compared simultaneously with all possible IDs for this calling point and the data is routed accordingly.

Procedures can be called from different points in the program so they are preceded by a tree of merge nodes that route tokens from any calling point to the circuit that implements the body of the procedure.

Closures are implemented using the component of the same name. Allocation is done when the closure is declared and the reading stage is placed at the beginning of the closure's body.

Conditionals are implemented as a split node where the value of the conditional expression is used to route the input token to either the "true" branch or the "false" one.

Parallel bindings are implemented as one or more `par` nodes in a tree. Each circuit that corresponds to a parallel expression is connected to a fork output and to a join input. The body of the `par` expression is connected to the join output.

Input and output channels are implemented like procedures with input or output components as a body.

6 Results

We have tested the prototype on classic functional algorithms to produce dataflow machines on an FPGA. The compiler's VHDL output is fed to Altera's Quartus-II development environment and synthetised using Synplify Pro. 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.

As an example, a quicksort algorithm has been implemented in an Altera Stratix EP1S80 FPGA with a speed grade of -6. As explained above, the data structures (lists) are implemented as closures. The resulting circuit uses about 12% of the reconfigurable logic and about 7% of the memory available in the FPGA for lists of up to 256 16-bit elements and can run at clock rates above 80MHz.

7 Conclusions and Future Work

Even though the compiler presented in this paper is a complete working system that can generate circuits for any algorithm, it lacks many of the primitives that would be needed to efficiently implement some basic concepts such as data structures and the procedures to manipulate them. Also, care has to be taken when writing circuits that use parallel expressions since those might introduce deadlocks.

Future work will concentrate among other things on data structures, automatic and manual reclaiming of unused storage, safety of parallel expressions, different modes for input/output channels and compile-time optimizations.

References

[Appe89] A. 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.

[GC94] C. GIRAUD-CARRIER. A reconfigurable dataflow machine for implementing functional programming languages. *SIGPLAN Not.*, 29(9):22–28, 1994.

[Shiv91] O. SHIVERS. *Control-flow analysis of higher-order languages of taming lambda*. PhD. Thesis, Carnegie Mellon University, 1991.