Compiling Recursion to Reconfigurable Hardware using CLaSH

Ruud Harmsen
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
r.harmsen@student.utwente.nl

ABSTRACT
Recursion is an important problem solving technique and implementing it on hardware is not trivial. In this research several methods on how to do this are discussed and one is implemented. The goal is to find out if it is possible to automate the process of making recursion possible on hardware and add this automation to the CLaSH compiler. The CLaSH compiler is a compiler from Haskell to VHDL, Haskell has a much higher level of abstraction than VHDL does, so should give an easier way to implement hardware designs. This compiler is still a work in progress, one of the missing things is support for recursion. This paper shows how to compile simple recursion to a stack-based framework which can be used in CLaSH.

Keywords
CLaSH, hardware recursion, FPGA

1. INTRODUCTION
Recursion is an important and powerful problem solving technique, which is used in important algorithms e.g. search and routing algorithms. Recursion is a method where the solution to a problem depends on solutions to smaller instances of the same problem. The power of recursion evidently lies in the possibility of defining an infinite set of objects by a finite statement. Implementing recursion on reconfigurable hardware like a Field programmable gate array (FPGA) is not trivial. Some examples for implementing recursion on hardware (FPGA) is done by Mihhailov et al.[6], the study implemented a parallel sorting algorithm on an FPGA. Maruyama et al.[5] show 2 algorithms which shows that recursion can be faster on reconfigurable hardware, the speed of some simple combinatorial algorithms are 4.1 to 6.7 times faster on a relatively slow FPGA (compared to a processor). The first attempt to do this was made by Tsutomu Maruyama et al.[5], where recursion was implemented using stacks. Sklyarov et al.[10] show the difference between a recursive and iterative solution of some algorithms and the difference in performance. Research on this topic is discussed in the Related Work section.

Programming an FPGA is done in a Hardware Description Language (HDL) like VHDL or Verilog. A relatively new HDL is CAES Language for Synchronous Hardware (CLaSH), this is a functional HDL created by the CAES-group [1]. CLaSH does not support recursion yet, this paper will give more insight in how to add recursion to this compiler. There are some HDL compilers available that can cope with recursion like the C to HDL compiler by Maruyama et al.[4], but C is not a functional language.

2. RELATED WORK
Maruyama et al.[5] implement recursion using stacks. The authors show as an example that an algorithm to solve the knapsack problem can be split in 2 parts, a recursive part and a loop instruction (tail recursion optimization). The study shows that this loop part of the algorithm can be put on a stack and executed parallel to the recursive part using multiple threads. In that way a greater performance was achieved because multiple calculations are executed at the same time. In the example of the knapsack problem this shows a speedup of 6.7 times using an FPGA in stead of normal sequential execution, even when the clock speed is much lower than the processor (35Mhz compared to 200Mhz).

Sklyarov suggests an implementation of recursive algorithms using Hierarchical Graph Scheme (HGS) [9] . His idea is to divide the algorithm to a discrete number of modules where each module is subdivided with a number of states. (See Figure 1). This is a Hierarchical Finite State Machine (HFSM). This idea uses 3 stacks, a Modulestack where the different models (Zn in Figure 1) are stored, a Statestack (an in Figure 1) and the Datastack where the results are stored.

Figure 1. Modules and states of HGS
Every time a recursive module is called it is put on the stack. Therefore the stack can grow out of bounds if a function has to many recursive calls. Pipelining like [5] could decrease the stacks by multi-threaded execution.

Ferizis et al.[2] tried a different approach, no stacks in this case. This PhD study divides the recursive function in to a base part and a recursive part. The recursive part is then divided in a pre- and post recursive part. These are defined as the part before and after the recursive call is made. Due to dividing the algorithm in these parts, they are able to place recursive parts in a pipeline. One downside of this implementation is that this approach needs a runtime configuration of the hardware, which is not possible on all types and takes precious time. The pipeline takes up a lot of space on the FPGA so the algorithm can’t be larger than the FPGA itself. The results they achieved where good, a quick-sort algorithm of 8000 items took only 17% of the clock cycles it would take on a stack-based solution.

Ninos et al.[7] used the previous proposals and suggests a data-oriented approach. This basically is an algorithm that converts recursion to iteration with a stack (the same as it is executed using software), called recursion simplification operation. This doesn’t include parallel computing.

The methods above are compared by Skliarova et al.[8]. The conclusion of this study was that there is no optimal solution to go from recursion to reconfigurable hardware. All solutions require that the designer produces the hardware compatible code, the solutions only provide the theory (with exception to the C to HDL compiler of Maruyama et al.[4]).

Kegel [3] implemented the method of Sklyarov[9] using Haskell, as a start of adding recursion to the C\alpha\text{SH} compiler. This still requires the designer to implement the method of Sklyarov[9] if recursion is used. Kegel suggests that the framework code should be generated by the compiler so the designer should not have to write the error-prone code. This research uses a simplified framework which could be extended to Kegel’s framework as Future Work.

2.1 CLaSH Compiler

The C\alpha\text{SH} compiler takes a subset of the Haskell language and converts this to basic VHDL. This VHDL code can be run on for example an FPGA. As the circuit descriptions, simulation code, and test input are also valid Haskell, complete simulations can be done by a Haskell compiler or interpreter, allowing high-speed simulation and analysis. The specifics are described in [1]. This is still a work in progress. One of the missing elements is the support for recursion. This research gives insight on how this could be added and a simple example is given.

3. THEORY

Mapping a recursive function on hardware is not a simple task. There are several methods on how to implement this; the methods are discussed in the Related Work section. The most straight-forward method is the method of Sklyarov, this research uses a simplified version of his method. The method uses a stack to store the results in between of recursive calls. This framework can handle only simple recursion functions with 1 recursive call in the recursive case and 1 base case. It should give insight in how recursion can be handled and the possibility of more complex functions will be future work. This mimics the way recursion is solved in software.

3.1 Compiler

The aim for this paper is to provide a method to implement recursion into the C\alpha\text{SH} compiler. The C\alpha\text{SH} compiler takes a C\alpha\text{SH} source file, runs it through the Haskell compiler and compiles this to VHDL. Inside the compiler several steps are needed to make this possible. One of these steps should be to convert recursive functions to hardware compatible code. In this research this step of converting recursive functions to hardware compatible code is done in a pre-compiler. This pre-compiler gets an Abstract Syntax Tree (AST) of the recursive function and converts it to a non-recursive Haskell code using the framework which is explained below.

3.2 Standard Form

Before a recursive function can be translated using the C\alpha\text{SH} compiler it has to be recognized. A simple recursive function can be described as below.

Listing 1. The Standard Form of a simple recursive function

<table>
<thead>
<tr>
<th></th>
<th>f n</th>
<th>p n</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>= a</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>otherwise = g n ( f (h n) )</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>g = main function</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>a = base case expression</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>p = base case condition of n</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>h = function applied on n</td>
<td></td>
</tr>
</tbody>
</table>

The g,a,p,h parameters have to be determined in order to convert it to the framework explained below. The simple format is limited to only 1 recursive call and 1 initial parameter. This limits the standard form to a few functions but it explains the basics. An example function is the factorial function which is used below. In order to use this on hardware, the different parameters have to be extracted from the function and have to be used in the framework explained below.

3.3 The Simple Framework

When the g,a,p,h parameters are determined they can be processed by the simple framework which will be explained here. Listing 2 shows the simple framework. It consists of a function ff which has 3 parameters.

- The parsed parameters (g,h,p,a)
- The state (n,v,s)
  - n is the input argument
  - v current value (during evaluation)
  - s is the stack
- The input, can be anything, this is only important as clock-tick

The result after 1 clock cycle

- output y
- new state (n1,v1,s1)

The workings of the framework is explained below. An example using the factorial function is given later on.

The function ff is called every clock cycle. When initiated, the first parameter is the tuple of parsed parameters, the second parameter is the state which has to be initiated, the third input is the clock input. The function ff will return a new state (s1) and the current output value(y), which
Listing 2. The stack based framework

```plaintext
ff (g, h, p, a) (n, v, s) = ( (n1, v1, s1), y )
where
(x, s0) = pop s0
(n1, v1, s1) | not (p n) = ( h n, a, push (n, s) )
| not (emptyS s) = ( n, g(x, v), s0 )
| otherwise = ( n, v, s )
y = v1
```

Figure 2. Framework graphical representation after 4 clock-cycles

is just a copy of the current value of the state (v1). The
next call, which most likely will happen the next clock-
cycle, calls the same function ff but the state(s1) from
the last call will be used as input for this call.

The output state(s1) will change depending on the base
case condition (p) or when there is nothing left to do (the
stack is empty). The output state(s1) will be as follows:

not (p n) The base condition (p) is not met, the input ar-
gument(n1) will be changed by function h, the cur-
rent value (v1) is the base case expression (a) and
the input value (n) is pushed onto the stack.

not (emptyS s) When the stack is not empty, the input
argument(n1) does not change, the current value(v1)
is the function g applied on parameters v and x where
x is a value popped from the stack.

otherwise The stack is empty and the base condition (p)
is met so there is nothing to work with and the out-
put state is copied from the input state.

A graphical representation of the framework is given in
Figure 2.

3.4 Simple Recursion Function (Factorial)

Listing 3 shows the recursive definition of the Factorial
function in the format of the standard form. This is re-
quired for the grammar to recognize it as a recursive func-
tion. For easier processing, the code will be converted
using \(\lambda\)-abstractions. This is a simple 1 on 1 process.

Listing 3. Factorial function

```plaintext
fac n | n==0 = 1
| otherwise = n * fac (n-1)
------
g : n * fac (n-1)
------
a : 1
------
p : (n==0)
------
h : (n-1)
```

For the factorial function this is done and the result is in
Listing 4.

Listing 4. Factorial using \(\lambda\)-abstraction

```plaintext
fac n | (\x->x==0) n = 1
| otherwise = (\(x,y)->(x*y)) (a, b)
where
a = n
b = fac ((\x->x-1) n)
------
g = (\x,y)->(x*y)
------
a = 1
------
p = (\x->(x==0))
------
h = (\x->(x-1))
```
The conversions are as follows:

\[\text{n} == 0 \quad \text{The base condition, this is } (\lambda x \rightarrow x == 0)\text{n}.\]

\[\text{n} * \text{fac(n - 1)} \quad \text{This is a multiplication applied on } \text{n} \text{ and fac. This corresponds with the lambda abstraction } \lambda(x, y) \rightarrow x * y \text{ where } x = \text{n}, y = \text{fac(n - 1)}.\]

\[\text{n - 1} \quad \text{The next iteration of the input value, in lambda calculus this corresponds with } \lambda x \rightarrow x - 1\]

The g,h,p and a parameters will be as stated on lines[6..9] in Listing 4. When used in a simulation of the framework this will look something like Figure 2. This shows the factorial(5!) function with the first 4 cycles. During work this will look like Figure 2. This shows the g,h,p and a parameters will be as stated on lines[6..9] in Listing 4. When used in a simulation of the framework this will look something like Figure 2. This shows the factorial(5!) function with the first 4 cycles. During these 4 cycles the base condition is not met, \text{n} > 0 so \text{n} is pushed onto the stack. The output value(y) is the base case value. The output value of 10 clock cycles is [1, 1, 1, 1, 1, 2, 6, 24, 120], more clock cycles will just return 120 because the otherwise condition from the function ff is met.

4. IMPLEMENTATION

In this research a pre-compiler is used on a handmade AST. This is not the best solution but demonstrates how the grammar and framework works. Ideally the programmer does not have to do anything special, like special annotations in the code which has to be done in the C to HDL Compiler[4], to use recursion in C\text{a}SH. The pre-compiler will parse the parameters out of the AST and paste's them into the framework. The framework displayed in Listing 2 is in Haskell syntax. The framework uses lists which are not compatible with C\text{a}SH. These lists have to be converted to a vector. This conversion to C\text{a}SH has been tried but not with success this is because of lack of knowledge of C\text{a}SH. The output of this pre-compiler will consist of the framework code and the parsed functions (g,h,p,a). This output file is a valid Haskell file which can be compiled by the Haskell compiler. As mentioned above, this is not C\text{a}SH compatible. This is called a pre-compiler because it is run before the actual compilation.

4.1 Grammar

The grammar for this simple recursive standard form is shown in Listing 5. The functions that can be parsed by this grammar consist of a base-case and a recursive-case. This base-case consists of a condition and a value if this condition holds. The recursive-case consist of an expression on the input argument and the return value of the recursive call. As mentioned before, the grammar only accepts expressions in the \lambda-format.

5. CONCLUSION

A best method of implementing recursion has still to be determined, but the a straight forward way to make C\text{a}SH compatible with recursion is the method provided by Sklyarov [9] because it mimics the method of how recursion works in software. This research shows that a simplified version of this method is possible to implement in an automated way. There are still steps to be taken to add it completely to the C\text{a}SH compiler, these steps are mentioned in the future work section.

6. FUTURE WORK

The process of adding recursion to the C\text{a}SH compiler is not finished. What is missing will be mentioned here as future work. These missing features are:

- Extending the grammar to add support for all recursive functions, it currently only can handle simple recursive functions like factorial.
- Add it to the C\text{a}SH compiler, the current process of adding recursion to C\text{a}SH is to pre-compile a recursive function which then will be given to the C\text{a}SH compiler, this should be incorporated in the said compiler.
- In this research a handwritten AST is used, the input should come from either the Haskell compiler or the C\text{a}SH compiler.
- A simple framework is used in this research to explain the process, a framework made in Haskell is available[3] and should be ported to C\text{a}SH this then can be used with the extended grammar (which is also future work).

7. ACKNOWLEDGMENTS

Without the help of J. Kuper and C.P.R. Baaij this research wouldn’t be possible. J. Kuper for guiding me and C.P.R. Baaij for all my C\text{a}SH related questions.

8. REFERENCES

Listing 5. Appendix

```haskell
data FuncDef = FuncDef Identifier Var BaseCase RecCase

data BaseCase = BaseCase App Condition Var Expr

data RecCase = RecCase Expr

data Expr = EVar Var
            | EInt Int
            | OpExpr Operator Expr Expr
            | AppExpr App FuncExpr Expr
            | EFunc FuncExpr Expr
            | App2Expr App FuncExpr (Expr, Expr)

type Operator = String

type App = String

type Lam = Char

data FuncExpr = FIdentifier String
               | LamExpr Lam Var Expr
               | Lam2Expr Lam (Identifier, Identifier) Expr

type Var = Identifier


type Condition = FuncExpr

type Identifier = String
```