Building an Heterogeneous NoC with the CλaSH Hardware Description Language

Jesse Kleinhesselink
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
jjkleinhesselink@gmail.com

ABSTRACT

This paper discusses the current limitations of traditional SoC designs and traditional Hardware Description Languages (HDLs) and proposes the use of the functional HDL CλaSH to describe future SoCs. The traditional SoC design methodologies cannot keep up with the increased need for performance and imperative HDLs lack certain abstraction mechanisms to deal with the increased complexity that comes with that need. CλaSH will be tested with a few NoC topologies to see if it is suited for this new SoC paradigm. The findings of these tests will be presented and conclusions will follow about CλaSH as implementation languages for NoCs.

Keywords
System on Chip, Network on Chip, heterogeneous NoC, CλaSH, Haskell, topology

1. INTRODUCTION

As the number of transistors that can be contained on a given chip continues to grow, old System on Chip design methods cannot keep up with the increasing complexity anymore. Therefore, new design methods conforming to the NoC paradigm have been suggested in multiple papers. NoC is a paradigm in which different resources on a chip are all physically interconnected through a network of switching mechanisms and resources. Such a resource can be any logic-, storage- or communication device that fits on the NoC, such as a processor, DSP or memory module. All components use a protocol stack (similar to TCP/IP) to communicate with each other so logic and control of the complete system is fully distributed. This paradigm enables us to build scalable intra- or inter System on Chip networks that have high throughput and relatively low hardware costs. Also, system designs can possible be reused.

In addition to the design methods, the increased complexity makes high-level abstraction mechanisms, that hide the complexity, more important. Those abstractions are lacking in traditional HDLs such as VHDL. CλaSH is a HDL that provides these abstraction mechanisms. It is a functional hardware description language based on Haskell. Functional languages lend themselves very good for hardware description because they excel at describing mathematical constructions, of which functions are especially important. Signal manipulating circuits are in fact functions on incoming signals so the use of a functional language comes naturally. The abstraction mechanisms allow the programmer to think functionally and semantically about the system and, more importantly, allow him to hide details. The language is still in development and it lacks the support of certain language constructs such as recursion[1]. It can, however, already be used to describe a range of circuits[2], including a non-trivial dataallow processor [9].

The goal of this paper is to present the possibilities of CλaSH to describe NoCs as they will probably play a meaningful role in the field of chip design. Separating the network from the topology allows us to configure a network with multiple topologies and test them. Three important network topologies will be discussed. The network is also heterogeneous, meaning, it may contain any resource as long as it’s interface conforms with the interface of the network. In the next section, the research questions will be presented. Section 3 handles the research methods. Then, attempts will be made to answer these questions. First there are some considerations that need to be made, after which the Haskell descriptions of an NoC will be discussed including a 2D-mesh like structure, ring and binary tree topology. In section 6, the discussion will be extended with CλaSH as the CλaSH compiler poses extra constraints on the code.

2. RESEARCH QUESTIONS

There are a few questions that come to mind when considering a heterogeneous NoC in CλaSH. These questions relate to the use of Haskell and CλaSH as implementation languages for NoCs. When building a network one needs to consider the different topologies that are possible. Every topology has it’s positive and negative aspects regarding the performance, hardware costs and other criteria. Also, some topologies might lend themselves better for implementation in a functional language. Three topologies will be selected that will be put to the test together with the network traversal algorithms. The questions are:

1. Can the network algorithms be expressed in CλaSH?
2. Are the selected topologies suited for implementation in CλaSH?

The questions will first be discussed with reference to Haskell, then conclusions regarding CλaSH will follow.
3. METHODS
As the number of available topologies is overwhelming, there is a need for reduction. Criteria for the selection and the topologies themselves will be based on literature in the area of NoC. The actual selection will be loosely based on the criteria. Then, this selection will be implemented in Haskell as a first step towards synthesis. The main cores will, initially, be simple black boxes that only generate testdata. The switches will also be kept as simple as possible. After implementation in Haskell, the implementations will be tested and evaluated. First, the tests only involves the dummy black box resources. The internal representation shall be checked for correctness for the first three clock cycles. The second round of testing involves the router of another author in this field that substitutes all the original dummy switches. It is important that, assuming that the router operates correctly, all messages must reach their destinations. As states must be manually checked, only a few use cases will be implemented. When testing succeeds, attempts will be made to translate the Haskell codes into synthesizable C. If a testing succeeds, attempts will be made to translate the Haskell codes into synthesizable C. Any structural changes that might be needed in this step may lead to differences in evaluation compared to previous testing so this code must also be tested. The CλaSH code is compiled to VHDL so simulations must be run on top the code. Final testing results will then be presented. Additionally, there might be the need for improvements in CλaSH.

4. CONSIDERATIONS
Because CλaSH cannot handle recursion yet, this must be taken into account when writing the code. So, recursion should be avoided whenever possible. The entry point of the program, which eventually will be called every clock cycle, preferably executes each node's operation in parallel because sequential code runs much slower.

For successful communication there is a need for common interfaces for resources and communication devices. The external part of the resource network interface (RNI) needs to be considered in the design so any device with the correct external RNI can be placed into the network. The internal workings are not of any interest for the network implementation because they are abstracted away in the network design. The width of the node-to-node signals puts some constraints on the interfaces and the lower layer protocols. Also, variable signal widths are possible as in Fat Trees[8] where links become fatter as they go up in tree level. Dataflow must be synchronous in a way that no data is lost because of overwritten data or the fact that some resources run faster than others. Also all devices must be able to differentiate between old- and new data. This means that there must be a global clock, or other extra signals that devices can use to control dataflow. It is stated that future SoCs should use the GALS model of operation[7]. So, the traffic between nodes needs to be asynchronous from the nodes themselves. This reduces complexity, mainly because of the heterogeneous nature of the NoC we are discussing, and reduces wiring and power consumption. It might prove to be difficult to describe a GALS network in Haskell, however, so a global clock is a simple alternative. Considerations about topologies will be handled in the next section.

5. DESCRIPTIONS IN HASKELL
The design is modular such that the network topology is separated from the algorithms that run on it. These are in turn separated from the actual cores and switches. So, in theory any routing protocol and any core can be integrated into the network. The topologies were also programmed in a generic way to support various sizes. Because CλaSH cannot handle recursion yet and it was not necessary, recursion was omitted.

A network, whatever the topology may be, is defined as a List of the Node type. The Node type is specified in Figure 1. For all adjacent nodes, the buffer for incoming traffic is specified as a mapping from Node to Integer List. The mapping attribute of a node defines it’s operations. The mapping takes an abstract state and a buffer, does calculations and returns a new abstract state and output buffers. The abstract state or UnitCore acts as a container for the internal state of the resource. A UnitCore can be made for any possible resource so it actually is an abstraction mechanism that hides the internal workings of a resource. To support this an RNI must be made for a resource to run in the network. The network does not, however, support packet sizes greater than 64 because packets are represented as Int64. They could be represented more generically, however, which eliminates this constraint.

5.1 Topologies
From the popular topologies, two were taken from literature and one was added for completeness (the ring). A 2D-mesh, a ring and a fat tree topology were initially considered. Due to time constraints, a binary tree was implemented instead of a fat tree. All topologies can be laid out in a two dimensional plane, which is a favorable property for SoCs [6] because less die material can be used. Important selection criteria for performance are diameter, connectivity, bandwidth, latency[3] and scalability. Bandwidth and latency are not measurable for Haskell implementations so they are omitted. The diameter refers to the maximum number of nodes between any source-destination pair. Connectivity is the number of direct neighbors of a node.

All of the topologies are implemented with each switch being the neighbor of exactly one core, so the number of switches equals the number of cores. The communication channels consist of two one-directional links between two nodes. Note, that in this section, n is used as the number of switches, or switch/core pairs in the topology.

5.1.1 2D-Mesh
The mesh discussed in this subsection uses the CLICHE approach (Chip-Level Integration of Communicating Heterogeneous Elements) [7]. The term mesh describes the layout of the channels, but it actually is a torus-like topology. Nevertheless, the terms mesh and 2D-mesh will be used for this topology unless stated otherwise. The original paper also uses these terms. Note that a torus is in fact a mesh with the endpoints connected to each other.

All resources are embedded in a m * n grid and interconnected via switches. Every core communicates with exactly one switch, that is also connected with the four other neighboring switches (see Figure 2).

This topology was chosen because it scores good on performance, complexity and hardware costs. The 2D-mesh also scales well. The structure is rather simple and straightforward. It uses more wires than the other two, however (3n), but this is just a linear factor. The diameter of this topology is \(\sqrt{n} - 1\) but with the switch-core configuration this boils down to \(\sqrt{n} - 1 + 2 = \sqrt{n} + 1\) because there is always an overhead of two switches. The connectivity of the combined switch and core is 4.
5.1.2 Ring
All switches in this structure are connected in a ring with, again, the cores being only attached to their neighboring switches. The diameter of this topology is \(\lfloor \frac{n}{2} \rfloor\), taking cores into account this is \(\lfloor \frac{n}{2} \rfloor + 2\). The combined connectivity is 2. This topology has long shortest paths and has bad scalability but the wire costs are relatively low (proportional to \(2^{n}\)).

5.1.3 Binary Tree
The fat-tree structure was initially chosen because it has been formally proved that it is the most cost-efficient topology for VLSI realisations [8] (see Figure 3 for the SPIN fat-tree). Because of time constraints, however, the binary tree was chosen; the structures are related to each other. Tree topologies are defined recursively, so this poses some difficulty on the programmer. But the solution was quite simple. Binary trees, specifically, can be described in an array with the parent placed on position \(i\) and the left and right children placed on position \(2i\) and \(2i + 1\), respectively. The root is placed on position 1. Similar configurations can be made for trees with a constant branching factor greater than 2.

The diameter of the binary tree is \(2^d\), combined \(2^d + 2\), \(d = \lfloor \log_2 n \rfloor\) being the depth of the tree. The connectivity is 3. The binary tree has great performance and few wires (\(2^n - 1\)). Better performance of the fat tree is mainly expressed in bandwidth and latency as the bandwidth is distributed better at the top level nodes.

5.2 Synchronisation
The network uses a global clock to distribute messages. The operations of the resources are initiated by the clock but they can use multiple clock cycles as long as they produce output every cycle, this could be an empty packet for instance but this depends on the network protocol.

5.3 Code
5.3.1 Network algorithm
Figure 4 shows the network algorithm code at top level. See Figure 1 for the Node type definition. The function cycleClock takes a network, which is in fact a nodelist, and an integer \(i\) and executes every node's operation \(i\) times. traverse executes the resources once. It takes the network argument and calls the updateNodes function with all identifiers within the network, these are taken using the label attribute of all Nodes. The function updateNodes has the same functionality as updateNd, but with multiple nodes and the addition of clearing old data from buffers. Unfortunately, the use of foldl makes this network not executable in parallel since the fold functions introduce dependencies on the nodes. This sequential mode of operation is still a major performance problem but probably fixable by rewriting the algorithm. The folds are used because the updates that a given node makes to it's neighbors need to be available before those neighbors execute their logic. Otherwise, changes are overwriten by initial values.
5.3.2 Resource Network Interface

The RNI in Figure 5 is the interface for the router [5] that is used in the mesh topology. The RNI executes the actual routing function with state and inputs as parameters, which are taken from the corresponding node, and translates the output back to the generic network format that the node can later be updated. The UnitCore is discussed in the beginning of this section, it contains the actual internal state.

5.4 Testing

First, the initial neighbor configurations of all nodes were checked to make sure that the topologies were correctly described, then the network algorithms were put to the test. Unfortunately, only the 2D-mesh was tested with a working routing protocol [5] because it was specifically designed for a mesh-structure of dimensions at least 3 x 3. This is, however, not a problem because initial network configurations are checked. In addition, the routing protocol is a test for network traversal algorithms. The protocol was only implemented in the switches since there was no core implementation available. However, since every switch pairs with a core, the switches could forward the messages directly to their cores when the destination was reached. The cores were implemented as black boxes that merely act as hubs. Testing was done by placing two packets – one for the header and one for the data – in two corebuffers. All the other buffers were filled with null-packets – packets that initiate no actions. Then, clock cycle wise, the packets were monitored until they reached their destination. The other two topologies were tested using the dummy core mentioned previously and the dummy switch which simply copies it’s inputs.

5.5 Results

In the listing below, the initial buffer configurations are shown for the router test. All buffers that are not shown are filled with empty packets. So, this configuration emulates the behaviour that Core C(0, 0) and C(2, 2) have just sent two packages to their switches. The first package basically means that the number 88 is sent to the core at position (2, 1). The meaning of the second one is analog to the first. The keywords like Continue are not of any concern in this paper, they are used in the routing algorithms. It is important that the second packet in the buffer belongs to the same stream as the first one.

```
data UnitCore = Router {RouterState | DummyRouter}
 deriving (Show)

switch1 :: (UnitCore, [Int64]) -> (UnitCore, [Int64])
switch1 ((Router state), inputs) = (unit', output)
  where
    uni' = 'Router state'
    (state', packets') = router state packets
    output = map inputToPacket inputs
    output = map packetsToInt packets'

Figure 5. RNI Code of a switch.
```

Listing the test results of the other topologies are far less interesting because the dummy functions only copy the input to the output, which means that the packets are only bounced back and forth between every adjacent node pair. Initial neighbor configurations were correct for all topologies and the traversal algorithms seemed to work flawlessly in the 2D-mesh.

So, all three topologies were successfully implemented in Haskell without the use of recursion. Translation to C\(\lambda\)Sh is, therefore, feasible. Note that, although recursion was not necessary, it might be for other projects and code can be quite unreadable when omitting recursion.

6. TRANSLATION TO C\(\lambda\)Sh

Translations to C\(\lambda\)Sh have not been made. It is expected that those translations are not significantly complex because recursion was omitted and other translations largely include converting Haskell types to hardware friendly types. This means, that for a specific topology to be translated, the List types must converted to vectors of constant size.

7. RELATED WORK

Related work has been done in the area of NoCs and functional HDLs but not so much in both. The router discussed in 5.4 was ment for mesh NoCs and was written in Haskell. It is expected that the Haskell descriptions of this router can be translated to C\(\lambda\)Sh with few transormations.

The topologies that are described in this paper are based on topologies in other papers. The 2D-mesh was taken from [7]. This paper proposes a packet switched platform that includes the architecture and design methodology. The architecture includes the physical layer, the data link layer and the network layer of the OSI protocol stack. It describes two phases where phase one encompasses the design of a concrete architecture and phase two maps the specific application onto this architecture. The fat-tree topology was based on [4]. This paper discards the use of a bus for future On-Chip systems and presents a packet-switched interconnection template with the fat-tree as it’s topology. This paper also discusses the modularity in which services are built on top of a bare network. For other functional HDLs that have been made you are encouraged to read [2]. In the related work section, this paper briefly describes other functional HDLs that are available. HML, Hawk, ForSyDe, Lava and Bluespec are discussed. All these languages lack at certain points where C\(\lambda\)Sh doesn’t. The main reason is that C\(\lambda\)Sh supports choice elements like pattern matching and guards and can handle polymorphic Haskell types and higher-order functions[2].
8. CONCLUSIONS
Two topologies were selected from research papers, the fat-tree and the 2D-mesh. The ring was added for completeness. The network algorithms were written successfully in Haskell without recursion and with types that can be transformed to CλaSH, which answers question two. A fat-tree is a complete binary tree with bandwidth parameters so these results can be easily extended. The same goes for the configuration of the core locations; the cores in the fat tree are positioned at the leaves while the binary tree discussed here has cores at every switch. The code still needs some improvements to achieve acceptable performance, better flexibility and better readability. This will be discussed in the next section. Unfortunately, there was no time to create a GALS network. The network uses a global clock for the network and the resources. This works but it has poor performance.

9. FUTURE WORK
There is still a lot of work to be done when it comes to descriptions in functional HDLs. Not only is the work presented in this paper not complete, CλaSH is still a language in development mainly because of the lack of recursion. The research in this paper needs to be extended with the translation step to CλaSH. Also, the implementations are not tested thoroughly; only the mesh could be tested with a routing protocol because of the static nature of the protocol. The code needs some improvements to facilitate the integration of new topologies and resources, there is still a lot work in writing the interface part, partly due to the use of integer types instead of algebraic types. The network algorithms still operate on a sequential level, which needs to be transformed to parallel code so it can run more efficiently. The plan to write a GALS network has not succeeded.

CλaSH is still a very young language so complex and large systems have to be written in CλaSH to fully explore it’s potentials and limitations.

10. REFERENCES