The Click2NetFPGA Toolchain

Teemu Rinta-aho
NomadicLab
Ericsson Research
Jorvas, Finland

Mika Karlstedt
NomadicLab
Ericsson Research
Jorvas, Finland

Madhav P. Desai
Department of Electrical Engineering
Indian Institute of Technology (Bombay)
Mumbai, India

Abstract

High Level Synthesis (HLS) is a promising technology where algorithms described in high level languages are automatically transformed into a hardware design. Although many HLS tools exist, they are mainly targeting developers who want to use a high level programming language to design hardware modules. They are not designed to automatically compile a complete software system, such as a network packet processing application, into a hardware design.

In this paper, we describe a compiler toolchain that automatically transforms existing software in a limited domain to a functional hardware design. We have selected the Click Modular Router as the input system, and the Stanford NetFPGA as the target hardware platform. Our toolchain uses LLVM to transform Click C++ code into a form suitable for hardware implementation and then uses AHIR, a high level synthesis toolchain, to produce a VHDL netlist.

The resulting netlist has been verified with actual hardware on the NetFPGA platform. The resulting hardware can achieve 20-50 % of the performance compared to version handwritten in Verilog. We expect that improvements on the toolchain could provide better performance, but for the first prototype the results are good. We feel that one of the biggest contribution of this work is that it shows some new principles of high-level synthesis that could also be applied to different domains, source languages and targets.

1 Introduction

Writing packet processing applications in software offers a flexible and easy way for experimentation and product development. Hardware-based implementations, on the other hand, characterised by parallel operations and inflexibility, result in better energy efficiency and higher operational speed. At the same time producing hardware is orders of magnitude more expensive in terms of both design and manufacturing than producing software.

Click modular router [12] is a popular tool for writing software routers. Stanford NetFPGA [14] is a similarly flexible platform for developing hardware routers. While programming packet processing applications in Click C++ is easy and flexible, describing even a simple application in VHDL/Verilog (for the NetFPGA) is a relatively major undertaking. Combining the good sides from both of the two could be an enabler for rapid implementation of efficient packet processing applications.

An ideal software-to-hardware toolchain would allow one to express designs in a widely familiar programming language, such as C or C++, and convert these into a hardware implementation that could be tested in real life, e.g. using the NetFPGA. Unfortunately, our limited testing of the existing HLS (High-Level Synthesis) tools indicated that they are definitely not ready to take an existing software system and transform that into hardware [15].

In this paper, we describe an experimental toolchain that is able to transform existing, software-oriented C++ algorithms—within the limited domain of Click-based packet processing—into a hardware description. We have chosen to work with the LLVM compiler toolkit [13], as there is a wide variety of tools and an active development community working on LLVM. This allows us to take advantage of the existing LLVM-based parallelising optimisations. LLVM modularity also allows us to easily write our own transforming compiler passes, and to add our own back end, a non-commercial HLS toolchain called AHIR [17].

The Click2NetFPGA Toolchain has a different approach than the existing HLS tools. While it also has some restrictions on what type of source code can be used as input, it transforms a complete software system into a hardware design – automatically. By selecting a restricted source domain, it was possible to have certain assumptions, and create a toolchain that could in practice do what hasn't been done before – translating a software router to a hardware router. Our current approach combines typical software-oriented optimisations and some parallelising optimisations together with compile-time generated constant data structures and constant propagation. Our compiler front end lowers the original Click programs into versions that are more suitable for generating VHDL with the AHIR backend.

The version of our toolchain presented earlier [15] was about exploring the possibilities of some commercially available HLS tools as the backend of our toolchain. In this paper, we describe the second version of the toolchain. We have switched from a commercial backend to AHIR and rewritten the front end from scratch, building on the knowledge gained when working with the previous version. Since the second report of our work [16], we have modified the interface between Click and NetFPGA code, improved the performance and usability of the toolchain and have been able to evaluate some Click configurations on a NetFPGA card and in the Modelsim simulator.

In Section 2 we give a brief look onto the essential background information, in Section 3 we outline the operation of the toolchain, in Section 4 we go into the details with a usage example, Section 5 is on evaluation and, finally, in Section 6 we conclude the paper.

2 Background

In this section, we describe some existing related work and then we go into the major components that we have used to build our toolchain and its input and target systems.

2.1 Related Work

Over the years, several academic and commercial high-level synthesis research results and tools have been introduced. As already mentioned, most put some restrictions on the input programming language, or are solving a specific optimisation problem.

Trident is using LLVM to generate parallel hardware circuits from floating point algorithms. It doesn't allow e.g. recursion, `malloc` or `free` calls, function arguments or returned values [18].

LegUp introduces creating a soft MIPS processor and hardware accelerators that communicate through a bus interface. It uses LLVM to profile running software to identify candidate parts for hardware acceleration. The rest of the code is run as software on the MIPS processor, including code using dynamic memory, floating point and recursion [6].

The UCLA xPilot project developed a high level synthesis toolchain using LLVM [8] which evolved into a product, the AutoPilot from AutoESL, later bought by Xilinx [1]. The AutoESL High-Level Synthesis Tool accepts synthesisable ANSI C, C++ and SystemC as input.

The Click2NetFPGA Toolchain has one major difference to those listed above. It transforms a complete software system into a hardware design – automatically. Previous works either hardware accelerate certain parts of a software program, or completely synthesise only smaller units, such as single functions, that can then be used as parts of a hardware design. Click2NetFPGA takes a complete Click software router and transforms the packet processing functionality into the NetFPGA environment. By having this restricted source domain, it was possible to have certain assumptions (such as Click code having no recursion), and create the prototype toolchain.

Our toolchain is more of a "system-to-system" compiler, than yet another HLS tool. Besides the source software, the toolchain takes the system description, i.e. the Click router configration as an input. While many of the existing HLS tools could be used to compile single Click C++ elements into Verilog or VHDL, they wouldn't be able to connect these elements together nor to interface them correctly on the target system – the NetFPGA. Some of the existing HLS tools could be added to the Click2NetFPGA toolchain, either to perform a certain new optimisation, or then the toolchain could have been completely built upon some other tool than the AHIR that we chose. Before selecting AHIR, we did evaluate a number of commercial HLS tools [15]. AHIR suited our purposes better, as it is an open source project and we were free to modify it along our project.

2.2 LLVM

LLVM [13] is a collection of modular components for building compiler toolchains. The LLVM components operate on an intermediate language, called the LLVM Intermediate Representation (LLVM IR). The LLVM core consists of a compiler driver, a number of analysis and code optimisation passes, and a debugger. Several front ends and backends are using LLVM: clang [2] is intended as a replacement for GCC. The Dragonegg plugin [3] allows to replace the GCC optimisers with those from LLVM, thereby enabling all the GCC supported languages and targets to be optimised with LLVM.

LLVM has been used to implement a variety of language toolchains, including previous approaches to generate hardware [18] and bit-level optimisation of HLS data flows [19]. TCE [9] is a set of tools for designing processors based on Transport Triggered Architecture. TCE uses the LLVM clang [2] compiler as the front end for compiling hardware designs written in C and C++.

2.3 AHIR

AHIR [17] is a backend for LLVM that transforms LLVM bytecode to VHDL. It is a toolchain on its own, consisting of several different tools with intermediate result files. AHIR is still a work in progress and several of its shortcomings were discovered and fixed during this project.

The LLVM IR is first translated to an AHIR assembly language program (an **Aa** program). The **Aa** programming language corresponds to a hierarchically constructed Petri net (the type II Petri net), in which parallelism as well as complex control flow can be expressed in a direct manner. In an **Aa** program, storage variables and first-in-first-out pipes are natural objects which can be used for communication between different parts of the program. Several **Aa** programs can be linked together using an **Aa** linker.

A linked **Aa** program is then transformed to a *virtual circuit* or **vC** program, in which the control-flow, data-flow and storage aspects of the source **Aa** program are separated out (and optimised). The primary optimisations possible at this stage are: resource sharing, dependency analysis, and clustering of storage into disjoint memory spaces. The **vC** program is further translated to a VHDL description, which can be directly synthesised to target an FPGA or ASIC implementation.

The current version of AHIR supports a wide set of LLVM IR—the few notable exceptions are function pointers and recursion.

2.4 Click Modular Router

Click was introduced by Eddie Kohler [11] as a platform for developing software routers and packet processing applications. A packet processing application is assembled from a collection of simpler elements that implement basic functions, such as packet classification, queuing, or interfacing with other network devices. The elements are assembled into a directed graph using a configuration language and packets flow along the links of the graph. Click provides some features to simplify writing complex applications, including pull connections to model packet flow driven by hardware and flow-based contexts to help elements locate other relevant elements. Since its introduction, Click has been used as a tool for research into a wide variety of packet processing applications. Some representative examples are multiprocessor routers [7] and prototyping a new architecture for large enterprise networks [10].

Click modules use the full power of C++ as an object-oriented programming language, including virtual functions and dynamically allocated memory. While these language constructs facilitate code reuse and ease of programming, they complicate the task of synthesising hardware from the software.

2.5 NetFPGA

The NetFPGA platform [14] provides a platform for researchers who are interested in investigating line speed packet processing applications. It is similar to Click in the sense that it provides a set of building blocks and the user can extend the system by introducing new building blocks as well. Within this framework, students and researchers can write code to implement a variety of routing and packet processing applications that are then synthesised into hardware. The NetFPGA board can be plugged into a PC providing control plane support, and the resulting application can be tested at line speed in actual networks.

The first version of the NetFPGA platform provides a hardware board with a Xilinx Virtex-II Pro FPGA and a Verilog framework supporting hardware with a PCI bus and four 1 Gbps Ethernet ports. There is also a newer version of the NetFPGA available with four 10 Gbps Ethernet ports, a faster and bigger FPGA and more memory. We are currently using the first version of the card for this project.

3 Overview of the Toolchain

We call this prototype software a "toolchain", although it could also be called or thought of as a "compiler", as it compiles a Click router into a NetFPGA compatible hardware design. However, it is more a chain of tools: Click2LLVM, LLVM, AHIR, NetFPGA SDK, Xilinx ISE and several shell scripts, bound together by a couple of Makefiles. We are running the toolchain on Linux, but it could be ran on e.g. FreeBSD or Mac OS X with minor modifications, at least until the synthesis (Xilinx) stage, as Xilinx ISE is only available for Linux and Windows.

The Click2NetFPGA toolchain consists of five main stages:

1. Compile Click elements
2. Link required files into an LLVM Module
3. Run transformations
4. Convert to VHDL
5. Create the netlist

This process is depicted in Figure 1. The goal of the first step is to compile all Click elements and library classes into linkable object files as well as LLVM IR

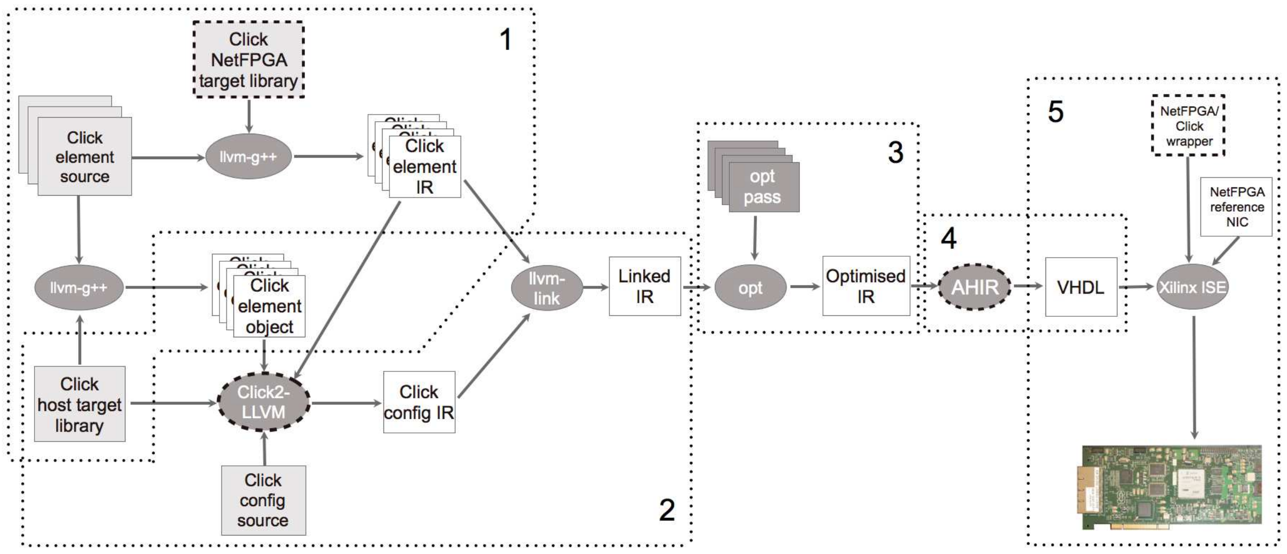

Figure 1: The Click-to-NetFPGA toolchain

source files. This stage needs to run only initially and when Click source code is modified.

In the second stage, `click2llvm` — the front end of the toolchain — reads the user-provided Click configuration file where elements and their connections are defined. The corresponding linkable object files are then loaded by the `click2llvm` tool with a Click library function.

After loading and initialising the Click router in the `click2llvm` process memory space, `click2llvm` reads the initialised values from memory and writes them out as constants in the LLVM IR format using LLVM library routines. `click2llvm` also generates some wrapper code and imports the library functions that the elements are calling. All the required code is inserted to an LLVM Module that is written out as a single LLVM IR file (Linked IR).

The resulting LLVM Module needs several transformations and optimisations so that AHIR can transform it to VHDL. First trick is to replace the *this* argument in functions with a constant variable (the Click element of the same type). For this, we have written our own LLVM transformation pass. This makes most functions constant and enables better constant propagation optimisation later. Then we run LLVM `opt` with a number of optimisation passes, including constant propagation and inlining. For inlining, we use high inline treshold to get as much inlining as possible. Finally, we run a script on the LLVM module that replaces calls to LLVM intrinsics (e.g. memcpy) with calls to our own implementations of those functions.

Finally, there will be no function calls, loops or other non-synthesisable constructs left, partly due to our several transformations, partly due to the nature of Click code. It might be possible to write e.g. recursive code in a Click element, but in practice most Click elements can be transformed into a synthesisable form. The AHIR toolchain reads this transformed LLVM IR file and creates a VHDL file.

In the last phase the toolchain takes the VHDL generated by the AHIR and combines it with a VHDL wrapper and Verilog files from the NetFPGA SDK and uses the Xilinx toolchain to create a netlist. The netlist can then be loaded to the NetFPGA. If the resulting netlist is too big to load, it will be only found out at the very end. It would be good to add a checking stage earlier in the toolchain for the required vs. available FPGA resources. This would require adding some analysis steps into the toolchain and is left for further study.

4 Implementation Details

In this section we will take a closer look on the implementation of the toolchain. First we will describe the resulting hardware architecture, and then go into more details. We'll describe the changes to Click, the step from Click to LLVM by the front end and transformations on the LLVM IR Module. Then we will explore the creation of the netlist by AHIR. At the end of the section we have a usage example on how a specific Click configuration gets compiled by the toolchain.

4.1 Resulting Hardware Architecture

The final system generated by AHIR has a single pair of input data/control pipes and a single pair of out-

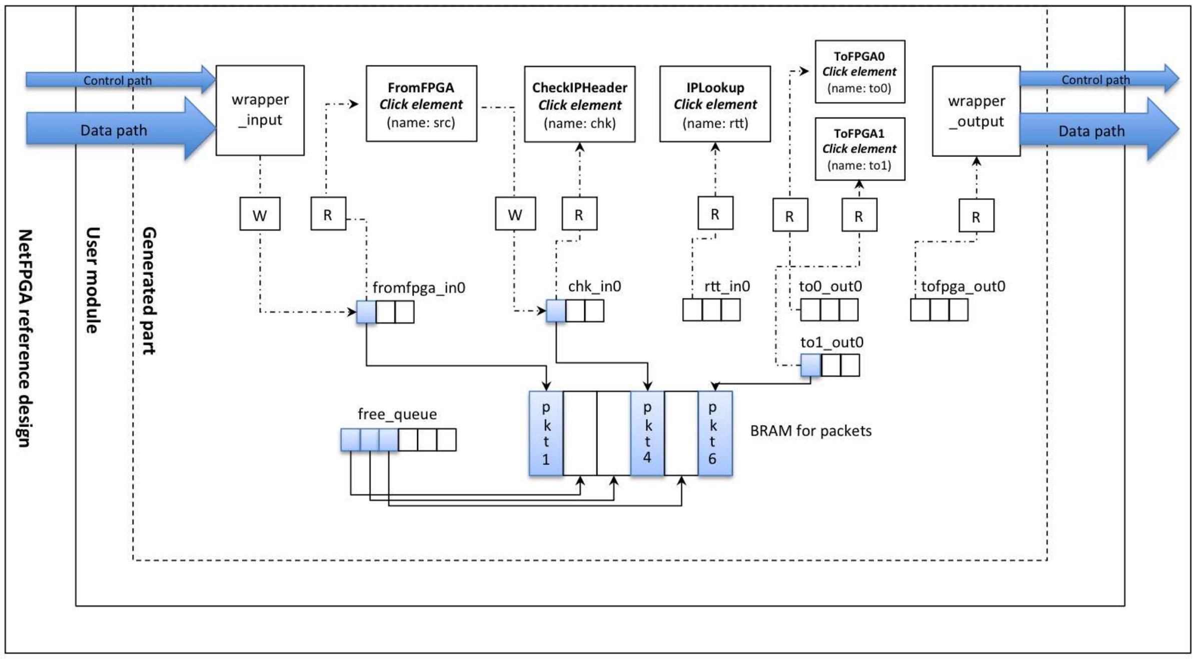

Figure 2: The resulting architecture

put data/control pipes which interface to the NetFPGA pipeline: Internally, the modules inside the AHIR system are of two types:

- Those that are "always on": these modules listen on input pipes, process data, and send processed data out on output pipes (essentially, these are pipe-line stages). Pipe accesses are blocking in nature, that is, a read from a pipe blocks until there is something in the pipe available to be read, and a write blocks until the pipe has room for new data.
- Those that are "called": these are invoked by input control signals, and normally have input and output arguments. When the module "finishes", it indicates this to the invoker, who can then use the output arguments produced by the module. These modules correspond to sub-routines invoked during system operation.

A Click configuration consists of a set of Click elements that exchange packet pointers over port connections. The pointers are used by the elements to access packets from the main memory. We have translated an input Click configuration to a set of "always on" AHIR modules where each module implements the behaviour of one Click element and the connections between Click elements are mapped to writing and reading from a named pipe that connects the elements (see Figure 2, where a "W" stands for `write_uintptr()` and an "R" for `read_uintptr()` – note that the figure represents a "snapshot of the system in time", where 3 different packets are being processed by different modules – it is not a description of the hardware with all wires drawn to be visible.). The result is a collection of interacting AHIR modules (a pipeline) which implements the original Click configuration.

The Click framework defines a packet as a distinct in-memory object that can be created, copied, and deleted. The size of the packet may change during its lifetime, which can also affect the space it occupies in the memory. In our implementation, the available memory for packets is divided into a set of fixed-size buffers. A packet inside the system is then identified by the pointer to the buffer which it occupies.

The packet buffers are managed using queues. A "free queue" (see Figure 2) contains pointers to currently free slots for packets. In addition, queues are provided between elements to hold packets that are in transit. The modules `wrapper_input()` and `wrapper_output()` are static pieces of C++ and Aa code which provide the functionality needed to obtain a free packet buffer, fill it with incoming packet data, pass the incoming packet pointer to the Click modules, receive the outgoing packet pointer from the Click pipeline, send out the outgoing packet and release the packet buffer. These are linked into the Click-based modules by the AHIR toolchain (see Figure 2). The linked design is then used to replace the output port lookup module of the NetFPGA reference NIC design.

4.2 New Target for Click

Click consists of a runtime library and a large set of standard elements. The library implements the essential components, such as the Element, Packet, and Router classes, altogether some 80 classes. Click can be compiled as a userspace program on e.g. Linux, FreeBSD or Mac OS X, or as a kernel module for Linux. Certain parts of the library differ between these targets, e.g. in Linux kernel native `skb` structures are used to represent packets. When compiled as a userspace program, packets are stored in the memory area of the running `click` process.

We have added yet another compilation target for Click: NetFPGA. The modifications are implemented with similar precompiler instructions and often in the same functions or methods which already differed between Linux kernel and userspace implementations. One example of such a modification is the creation of a new packet through the `Packet::make()` method. In the Linux kernel an `skb` is referenced from `Class Packet`, while in userspace Click, memory is allocated dynamically from process memory space. In NetFPGA, we request a memory address for a block of pre-allocated BRAM. This BRAM is defined in C++ as an array of bytes, that is transformed to VHDL by AHIR. By default, we have allocated 32 KB of BRAM that can hold 16 packets, each having maximum size of 2 KB. A "free queue" is used to record which BRAM slots are in use and which ones are free. Access to the free queue is managed by AHIR primitives so that only one request is served at a time.

While we have modified some existing Click libraries and C++ classes, such as the Element and Packet Classes to map method implementations to AHIR primitives instead of Linux kernel or user space, our goal has been to require no modifications to the existing Click elements and to require no new guidelines for writing new Click elements. The goal is to let the programmer concentrate on describing packet processing in Click C++ without having to consider that it might be compiled to hardware instead of software—as it is of no concern whether the target platform for a software compilation would be Linux or FreeBSD.

4.3 Compiling Click Configurations

A Click configuration defines a particular assembly of Click elements, thereby constructing a packet processing application or, in Click terms, a Router. In practise, when the userlevel `click` tool is used to execute the router, the tool first parses the configuration, then initialises the router, and finally starts packet processing. In the typical case, the packet processing phase then continues until the user terminates it. In our toolchain, the first two steps of this process are performed by the `click2llvm` compiler. The last, actual packet processing step is then performed by the synthesised hardware.

When Click parses and initialises a configuration, it also instantiates all the elements defined in the configuration and invokes the initialisation methods of the resulting element instances. In practise, the elements are either statically compiled to the tool itself, or the `click` tool dynamically loads the elements into its address space from a dynamically linked shared library. The tool then instantiates the C++ classes representing the elements and invokes their virtual methods to configure and initialise them.

`click2llvm` is essentially identical with the `click` userlevel tool up to this point. While the standard tool would now initiate packet processing, our compiler writes out the resulting initialised router. For this, `click2llvm` uses LLVM libraries to link all necessary Click elements as pre-compiled LLVM modules into a single LLVM module. It then uses the LLVM `StructLayout` API to find the memory locations for different fields of the Click elements. These memory locations represent the instance variables of the C++ Click classes. It then iterates through each Click element of the Router, and writes an LLVM Global Variable for each Click element into the LLVM Module. Each element becomes a global constant struct in the resulting module.

Normally, when a packet is received in `click`, it calls the `push()` or `simple_action()` of the first element in the configuration. After finishing with the packet processing tasks, the current element then either drops the packet or calls the `push()` or `simple_action()` method of the next element in the configuration and so on. When the final element has finished the call stack returns and the first element reads the next packet. While this is a flexible way to operate in software, in hardware all Click elements will be continuously running in parallel, and we need a different approach.

In Click, connections between elements are C++ pointers. For each packet (that is not simply dropped) a Click element needs to decide the outgoing port. During the initialisation of the router, the port table of each element is populated with pointers to appropriate next elements based on the Click configuration file. In our solution, we replace the pointers with names of AHIR pipes. Instead of calling the function pointer from the port, we send the memory address of the packet through a pipe linked to the aforementioned port. We have implemented this by creating a NetFPGA specific implementation of the `Element::Port::push()`. Instead of calling the methods of the next element through pointers, we call function `write_uintptr()` which AHIR will later interpret as a write to a named pipe. The name of the pipe is stored in the `Element::Port` Class. This way we get rid of the C++ call stack and are able to create separate hardware modules for each Click element that will run in parallel, connected by named pipes. We didn't have to implement a NetFPGA counterpart for the C++ call stack, as AHIR takes care of passing the arguments and return values for function calls.

Finally, we write out a single LLVM Module in the LLVM IR language that contains the source code for the Click elements, required parts of the Click library (such as the `Element` and `Packet` classes), constants that represent initialised elements, and a wrapper function for each element. The wrapper function maps the element to a separate program that reads packets from input ports, processes those packets and writes them to appropriate output ports (see Program 3).

Some Click elements use C++ library functions like `memcpy()`, `ntohs()` or `clock_gettime()`. The C++ compiler leaves these in the IR as "llvm intrinsics", which means that the compiler backend needs to map these calls to the target-specific implementations (on a Unix based system it would be the C++ runtime library). To achieve this, we insert LLVM instructions that implement the same functionality. For some, we currently do not have a counterpart, like for `clock_gettime()`. So far, we have implemented the required functions on-demand. It should be noted, however, that to support any (future) Click element, all system calls should be implemented on the NetFPGA. This is left as future work.

In the current prototype, we transfer all code into the NetFPGA. In practice, it would make more sense to analyse the software, and leave parts of the code to be run as software on the host CPU. This would also require automatic generation of interface code between software and hardware. However, with this prototype we have concentrated on the problem of software to hardware translation, and we leave this analysis and divisioning problem for further study.

4.4 Optimisation Phase

The LLVM Module created by the `click2llvm` tool is still unoptimised and contains constructs such as function pointers that don't easily map to hardware. We use a number of LLVM passes to transform these constructs in a more suitable form.

The first pass replaces the `this` argument in the C++ originated methods with the global variable representing the Click element. As a result the method becomes a constant function and can be inlined later. Although this means that we can have only one instance of each Click element class, the limitation can be removed, either with the trivial approach of replicating the methods and naming them differently, or writing a pass that splits the `this` argument to two: one pointing to the constant part of the class and another to the part holding instance-specific variables. The latter approach requires splitting the types in two as well. We feel this is all doable and could benefit optimising C++ software in general.

Next we pass the LLVM optimiser a list of the wrapper functions that need to be considered as the "API", and thus preserved in the output. "API" here means the same as the `main()` function in regular C/C++ programs – a starting point for program execution that is not called from within the program – that shouldn't be optimised away as "dead code". Temporary helper function definitions (`element_push()`, see Program 3) and other dead code gets optimised away and is not preserved in the output. Since the Click elements are constants, we get good results with the constant propagation and inlining passes. We use `-inline-threshold=10000` to get all packet processing code, e.g. the `push()` or `simple_action()` function of the corresponding Click element inlined in the wrapper function. We also use several other standard passes provided by the LLVM project.

4.5 Creating a Netlist

The collection of (optimised) LLVM modules produced by `click2llvm` is run through the AHIR toolchain which produces an AHIR system (described in VHDL).

Each LLVM module is implemented as an AHIR module (described as a VHDL entity/architecture pair). The AHIR module itself implements the control and data flow in the LLVM module with some optimisations; dependency analysis is used to extract parallelism from sequential statement blocks, and expensive operators (such as multipliers) are shared by multiple operations. Storage variables described in the LLVM Module collection are implemented as declared. In the AHIR system, storage variables are organised into disjoint memory spaces based on a static alias-analysis of the source program In addition, the AHIR system implements the concept of pipes (finite depth first-in-first-out queues) for inter-module communication and synchronisation. Thus, two modules in an AHIR system can communicate either through storage variables or through pipes.

The translation process in the AHIR tool-chain itself consists of three steps. In the first step, the LLVM IR is translated to an AHIR assembly level (**Aa**) program. **Aa** is an imperative and block-structured language which supports a large variety of types. The flow of control in an **Aa** program block can be specified to be sequential or parallel. During this translation:

- Each LLVM module is translated to an equivalent

Aa module. All blocks in the resulting **Aa** program are sequential in nature.

- Declared storage variables in the LLVM IR are mapped to declared storage variables in the **Aa** program.

- Pipes are inferred from the LLVM IR by keying off the special functions `uintptr read_uintptr(const char* pname)` (this is translated as a read from a pipe with the specified name) and `void write_uintptr(const char* pname, uintptr ptr)` (translated as writing the value ptr into the pipe with the specified name).

The second step is the conversion of the **Aa** program to a virtual circuit in which the control flow, data flow and storage aspects of the **Aa** program are separated. At this stage, dependency analysis is used to extract the maximum amount of parallelism that is possible from sequential statement basic-blocks. Storage variables are segregated into disjoint memory spaces whenever possible (disjoint spaces reduce load/store dependencies, and further, are accessible in parallel). The virtual circuit itself consists of distinct modules which communicate with each other through pipes or through shared memory spaces.

The final step is to generate the VHDL netlist from the virtual circuit. This translation uses a VHDL library of predesigned components such as operators, pipes, memory spaces, arbiters etc. Virtual circuit modules are translated to VHDL entities (currently, each such entity is instantiated once in the final system). Pipes are modeled in a direct manner, as are the memory spaces. Concurrency analysis is carried out in the modules to determine operations which can be mapped to the same operator without the need of arbitration. Further, depending on command line switches, the generated VHDL can be optimised for clock-period, and/or for area, or for cycle-count etc. We optimise the netlist to obtain the minimum area (given the constraints of the NetFPGA card) with a primary objective and the minimum clock period a secondary objective (in order to meet the 8 nanosecond clock period requirement of the FPGA on the NetFPGA card).

4.6 Usage Example

To illustrate the operation of the toolchain, we will go through the steps leading from the Click configuration to the optimised LLVM IR. We have created a configuration (see Program 1) which does packet switching based on the destination IPv4 address. The configuration contains seven different Click elements. FromFPGA and ToFPGA* elements in the configuration come from our own `minimal-package`, thus the `require` declaration on the first line. We currently require the elements to be introduced and given names, which is done on the next seven lines. The last five lines describe the flow of packets between the elements.

Program 1 router.click

```
require(package "minimal-package");
src :: FromFPGA;
to0 :: ToFPGA0;
to1 :: ToFPGA1;
to2 :: ToFPGA2;
to3 :: ToFPGA3;
chk :: CheckIPHeader(14);
rtt :: LinearIPLookup(172.16.0.0/24 0,
                      172.16.1.0/24 1,
                      172.16.2.0/24 2,
                      172.16.3.0/24 3);
src -> chk -> rtt;
rtt[0] -> to0;
rtt[1] -> to1;
rtt[2] -> to2;
rtt[3] -> to3;
```

Program 2 CheckIPHeader::simple_action()

```
Packet *
CheckIPHeader::simple_action(Packet *p)
{
  const click_ip *ip =
    reinterpret_cast<const click_ip *>
    (p->data() + _offset);
  unsigned plen = p->length() -
                  _offset;
  unsigned hlen, len;

  if ((int)plen <
      (int)sizeof(click_ip))
    return drop(MINISCULE_PACKET, p);
  ...
  return(p);
}
```

Elements `FromFPGA` and `ToFPGA*` are special elements that interface the Click/NetFPGA wrapper. They are for convenience (to have static wrapper code) and also as placeholders for code to transform packets between the NetFPGA and Click worlds. FromFPGA calculates the Click-specific packet lengths and offsets and stores them in the packet—this way the wrapper needs no modifications if Click itself is updated. ToFPGA does the reverse, mapping Click-specific fields into NetFPGA control flags. We have created a separate ToFPGA element for each physical NetFPGA port: ToFPGA0 sends the packets to port 0, ToFPGA1 to port 1, and so on.

To illustrate the mapping from Click C++ code and the Click configuration file to LLVM IR, we take a closer look at one of the used elements and parts of its packet processing code. Program 2 shows a part of the C++ source code for the packet processing code of Click element `CheckIPHeader`. Method `simple_action()` is part of the Click API, and is called for the packet if the element has defined it. In the `CheckIPHeader` element, various standard checks are performed on an IP packet. A valid packet is forwarded to the next element, while packets failing a test will be dropped. The `simple_action()` function of CheckIPHeader consists of several checks, but we focus here on the first test, where the size of the IP packet is checked.

After running the `click2llvm` tool with the `router.click` configuration, we generate a wrapper function named `ahir_glue_chk()` in the resulting LLVM IR Module (see Program 3). First there is a call to `read_uintptr()` with the first argument being a pointer to the constant `@1`. This maps to a blocking read of an AHIR pipe and returns when there is something written in queue `chk_in0`. Writing to this queue is done by the `FromFPGA` element, as described in the `router.click`. Next, the read pointer is cast to type `struct.Packet`, which represents the C++ `Class Packet` of Click. Then a temporary helper function `element_push()` is called with two arguments: a pointer to the Click element (`@chk`) and the current packet (`%1`). This `ahir_glue_chk()` function becomes an "always on" AHIR hardware module – its software counterpart would be a loop that never terminates.

After optimisations on the LLVM Module, the optimised version of `ahir_glue_chk()` (see Program 4) is longer, but it has everything inlined[1]. The only calls to external functions are `read_uintptr()` and `write_uintptr()`, which are only keywords for AHIR – they will not result as function calls in hardware.

Because we have constant arguments to `element_push()`, constant propagation and inlining passes have succesfully removed it, leaving the contents of the original `simple_action()` inlined in the wrapper function. The core operation, checking that the IP packet is at least 20 bytes long, is visible in the LLVM representation before the first branch instruction (`br`). In case the packet is too short, `ahir_packet_free()` is called to free the memory slot storing the packet and `ahir_glue_chk()` returns. Otherwise, at the end of the function, `write_uintptr()` is called, leading to the `LinearIPLookup` element, as per the `router.click` configuration (see Program 1).

[1]The whole function is not presented in the listing due to space constraints.

5 Evaluation

We have compared the performance of the Click-based design vs. the Stanford reference switch implementation that is distributed with the NetFPGA software package. The former is generated with our toolchain while the latter is handwritten in Verilog. While we can see that we cannot yet reach the same performance with our toolchain, the results prove that our toolchain runs. We would like to remind that this is a proof-of-concepts prototype, yet we feel that it could be possible to reach better performance levels by further optimising our toolchain, e.g. by rewriting some Click library implementations to better match the packet processing model of the NetFPGA.

Tests performed were PPS (Packets Per Second for 98 and 1442 byte packets), Ping (average round-trip time for an ICMP Echo request/reply message pair) and maximum bandwidth (TCP for 60 seconds).

For bandwidth tests we have used iperf v2.0.4 [4] and for the ping test the standard Ubuntu Linux `ping` command. For packets per second test we have used tcpreplay v3.4.3 [5] on the sending host and iptables on the receiving host to find the approximate maximum number of packets per second before NetFPGA starts to drop packets. As the results show, we did not need more accurate measurement tools to find differences at this stage.

The test setup consisted of two standard PCs with gigabit ethernet interfaces running Ubuntu. The PCs were connected to two ports of the NetFPGA card in the third Linux machine running CentOS. With our PCs, we were able to send maximum of about 415,000 packets per second when the packet size was 98 bytes (equals 325 Mbps), and 84,000 packets when size was 1442 bytes (equals 969 Mbps).

We compared the Stanford reference switch to two different Click configurations: "router" and "pipe". Router is the configuration presented in Section 4, performing IP header checking and destination port selection based on the destination IP address. Pipe is the simplest configuration possible, it only connects NetFPGA ports in two pairs, i.e. when sending a packet to port 0, it is forwarded to port 1 and vice versa. The same handling is present for ports 2 and 3. There is no actual Click packet processing, the only elements used are FromFPGA and ToFPGA.

Analysing the results, we can see that the reference switch handles the maximum load we can feed. From previous experiments, we know that it has been designed to be running at line rate. Even though we have continuously improved our toolchain and the wrapper libraries,

Program 3 Generated ahir_glue_chk()

```
@1 = internal constant [8 x i8] c"chk_in0\00"

define void @ahir_glue_chk() {
entry:
  %0 = call i32 @read_uintptr(i8* getelementptr inbounds ([8 x i8]* @1,
                                                          i32 0, i32 0))
  %1 = inttoptr i32 %0 to %struct.Packet*
  call void @element_push(%struct.Element* getelementptr inbounds
    (%struct.CheckIPHeader* @chk, i32 0, i32 0), i32 0, %struct.Packet* %1)
  ret void
}
```

Program 4 Optimised ahir_glue_chk()

```
@1 = internal constant [8 x i8] c"chk_in0\00"
@6 = internal constant [8 x i8] c"rtt_in0\00"

define void @ahir_glue_chk() {
entry:
  %tmp13 = tail call i32 @read_uintptr(
                i8* getelementptr inbounds ([8 x i8]* @1, i32 0, i32 0))
  %tmp14 = inttoptr i32 %tmp13 to %struct.Packet*
  %tmp15 = getelementptr inbounds %struct.Packet* %tmp14, i32 0, i32 3
  %tmp16 = load i8** %tmp15, align 4
  %tmp17 = getelementptr i8* %tmp16, i32 14
  %tmp18 = getelementptr inbounds %struct.Packet* %tmp14, i32 0, i32 4
  %tmp19 = load i8** %tmp18, align 4
  %tmp20 = ptrtoint i8* %tmp19 to i32
  %tmp21 = ptrtoint i8* %tmp16 to i32
  %tmp22 = sub nsw i32 %tmp20, %tmp21
  %tmp23 = add i32 %tmp22, -14
  %tmp24 = icmp slt i32 %tmp23, 20
  br i1 %tmp24, label %"3.i1", label %"4.i"

"3.i1":
  tail call void @ahir_packet_free(i32 %tmp13)
  br label %_ZN7Element4pushEiP6Packet.exit

"4.i":
  ...
  %tmp70 = ptrtoint %struct.Packet* %tmp14 to i32
  tail call void @write_uintptr(i8* getelementptr inbounds ([8 x i8]* @6,
                                                            i32 0, i32 0),
                                i32 %tmp70)
  br label %_ZN7Element4pushEiP6Packet.exit

_ZN7Element4pushEiP6Packet.exit:
  ret void
}
```

Measurement	Reference	Click	Percent
PPS 98 B	415K	178K	42.9
PPS 1442 B	84K	24K	28.6
Ping	105 us	115 us	109.5
Bandwidth	940 Mbps	215 Mbps	22.9

Figure 3: Reference switch vs. router.click

Measurement	Reference	Click	Percent
PPS 98 B	415K	225K	54.2
PPS 1442 B	84K	25.5K	30.4
Ping	105 us	114 us	108.6
Bandwidth	940 Mbps	227 Mbps	24.2

Figure 4: Reference switch vs. pipe.click

we lack somewhat behind. We can currently reach almost 1/3 of the 1 Gbps line rate with large packets. With smaller packets the performance is better – almost half of the line rate, as the wrapper_input needs to spend less time copying the packet to the memory subsystem. With larger packets our wrapper libraries are the bottlenecks and differences between the two Click configurations are not that large. With smaller packets, the router configuration is roughly 20 percent slower than the pipe configuration.

Based on an analysis of the pipeline stage latencies (using the Modelsim simulator and the NetFPGA verification scripts), we observe the following bottlenecks:

- The bottleneck which limits packet throughput for large packets is the interface between the NetFPGA datapath and the Click counterpart. Incoming data from the NetFPGA datapath is written into a shared packet memory, which is byte wide, single-ported and supports one access per clock cycle. Since the clock period in the NetFPGA board is set to 8 nanoseconds, this translates to an effective memory bandwidth of 1 Gbps, which is shared between reads and writes. Thus, the peak available throughput of the Click datapath is currently 500 Mbps, of which approximately 50% is actually being achieved.

- For small packets, the bottleneck (in the "router" configuration) is the latency of the IP header checking stage which was observed to be $4.2\mu s$. For short (98B) packets, this would limit the packet data rate to the 200 Mbps range, as observed. For longer packets, this bottleneck is not as serious, and the limit on the packet data rate would be higher. The latency of this stage needs to be reduced in order to achieve higher data rates (given the constraints of the NetFPGA card).

We did some experiments (using the NetFPGA verification environment together with the Modelsim simulator) to see the extent to which performance could be improved by targeting these bottlenecks. A universal method to get more performance is to use more resources, in our case FPGA slices. We could do this by replicating Click elements to parallelise some stages of the pipeline. It can be done without affecting the packet processing logic when the replicated Click element doesn't store state. As an example, we replicated the `CheckIPHeader` element to solve the first bottleneck. The result was that throughput improved by 18% over the baseline, with a corresponding increase in FPGA resource usage of 10%. It should be noted that if we create parallel paths for packets in an IP router, we need to add some packet reordering mechanism to our input and output modules, to ensure that the router operates similarly to the software version of the same Click router.

To target the packet memory bottleneck, we doubled the memory bandwidth by making it a two-banked system. With this modification together with the element replication, the throughput improvement over the baseline was 31%, with a 19% increase in FPGA resource usage. Thus, some simple bottleneck alleviation steps seem to pay good performance dividends. We have not yet tried this in actual hardware, as the FPGA chip on our NetFPGA card has limited resources, but resource replication could be useful in many applications which use larger FPGAs or ASICs.

The performance shortfall relative to hand-coded RTL can be tackled at two levels. The essential problem is to reduce the latency of the critical path through a code section. At the source-level, code transformations which enhance parallelism need to be further explored. The standard code transformations (e.g. loop unrolling, inlining) do help, but it is worth investigating if more is possible. As a fallback option, hand-optimised routines for critical code sections can make a substantial difference (analogous to the use of assembly language routines in performance critical embedded applications). To meet our goal of putting no extra burden to the software programmer, the insertion of these optimised routines should be done by the toolchain.

Further, the functionality of critical code sections can often be sub-divided into a sequence of simpler functions (pipelining the critical code segments) in order to improve the throughput of the final system. In our flow, critical code sections can be redone at three levels: at the C/C++ level, at the **Aa** level or at the VHDL level, with a corresponding performance/productivity trade-off as we descend from C/C++ to VHDL. At the hardware level, the AHIR toolchain in its current form is conservative in terms of adding register stages to meet clock period requirements (potentially adding needless cycles to the latency). A more optimised mechanism for adding such register stages is likely to improve the latency.

Currently we initialise the Click router before hardware synthesis where we discard all configuration and initialisation related code. Therefore the resulting Click-on-NetFPGA router cannot be live reconfigured. The ideal toolchain would analyse the Click configuration and then separate parts that should be run as hardware and those that could be left as software, and then automatically create an interface between those two. That would allow parts of the Click router to run on a CPU and other parts on the FPGA, similar to the approach in [6].

6 Conclusion

In this paper we have shown that it is possible to implement a domain specific toolchain that converts high level language software to hardware. We have implemented a prototype toolchain that can transform Click routers written in C++ to a hardware description in VHDL, which can then be synthesised and run on a NetFPGA card as part of the Stanford reference NIC design.

We use the "initialise-freeze-dump" method to transform initialised C++ objects in memory to constants in the output file of the front end. This way we can run the initialisation code outside the hardware, and then use the essential parts of code directly related to packet processing to form the hardware parts. Using numerous LLVM optimisations we can then transform the code in a form that is suitable for AHIR to transform it further to VHDL.

The performance achieved by the hardware produced by this toolchain is a significant fraction of that achieved by a handwritten NetFPGA implementation. Although work remains regarding the performance and flexibility of this specific toolchain, the results in general are promising. The main performance bottlenecks are in the translation between the Click and NetFPGA packet data models. Changing the target, e.g. creating a packet processing ASIC from the Click code, and using more chip surface, would most likely bring far better performance.

7 Acknowledgments

The authors wish to thank Dr. Pekka Nikander and Dr. Sameer D. Sahasrabuddhe for their invaluable efforts and ideas in the early stages of this project.

This work has been partly funded by Tekes through its Future Internet research program.

References

[1] AutoESL High-Level Synthesis Tool. `http://www.xilinx.com/tools/autoesl.htm`.

[2] clang: a C language family frontend for LLVM. `http://clang.llvm.org/`.

[3] DragonEgg. `http://dragonegg.llvm.org/`.

[4] Iperf. `http://sourceforge.net/projects/iperf/`.

[5] Tcpreplay. `http://tcpreplay.synfin.net/`.

[6] CANIS, A., CHOI, J., ALDHAM, M., ZHANG, V., KAMMOONA, A., ANDERSON, J. H., BROWN, S., AND CZAJKOWSKI, T. LegUp: High-Level Synthesis for FPGA-based Processor/Accelerator Systems. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays* (New York, NY, USA, 2011), FPGA '11, ACM, pp. 33–36.

[7] CHEN, B., AND MORRIS, R. Flexible Control of Parallelism in a Multiprocessor PC Router. In *USENIX Annual Technical Conference, General Track* (2001), Y. Park, Ed., USENIX, pp. 333–346.

[8] CHEN, D., CONG, J., FAN, Y., HAN, G., JIANG, W., AND ZHANG, Z. xPilot: A Platform-Based Behavioral Synthesis System. In *Proc. of SRC TechCon'05* (2005).

[9] JÄÄSKELÄINEN, P., GUZMA, V., CILIO, A., AND TAKALA, J. Codesign Toolset for Application-Specific Instruction-Set Processors. In *Proc. of Multimedia on Mobile Devices 2007* (2007).

[10] KIM, C., CAESAR, M., AND REXFORD, J. Floodless in Seattle: a scalable ethernet architecture for large enterprises. In *SIGCOMM* (2008), V. Bahl, D. Wetherall, S. Savage, and I. Stoica, Eds., ACM, pp. 3–14.

[11] KOHLER, E. *The Click modular router*. PhD thesis, MIT, 2000.

[12] KOHLER, E., MORRIS, R., CHEN, B., JANNOTTI, J., AND KAASHOEK, M. F. The Click modular router. *ACM Trans. Comput. Syst. 18*, 3 (2000), 263–297.

[13] LATTNER, C., AND ADVE, V. S. The LLVM Compiler Framework and Infrastructure Tutorial. In *LCPC* (2004), R. Eigenmann, Z. Li, and S. P. Midkiff, Eds., vol. 3602 of *Lecture Notes in Computer Science*, Springer, pp. 15–16.

[14] LOCKWOOD, J., MCKEOWN, N., WATSON, G., GIBB, G., HARTKE, P., NAOUS, J., RAGHURAMAN, R., AND LUO, J. NetFPGA - An Open Platform for Gigabit-rate Network Switching and Routing. In *IEEE International Conference on Microelectronics Education* (June 2007).

[15] NIKANDER, P., NYMAN, B., RINTA-AHO, T., SAHASRABUDDHE, S. D., AND KEMPF, J. Towards Software-defined Silicon: Experiences in Compiling Click to NetFPGA. 1st European NetFPGA Developers Workshop, Cambridge, UK, 2010.

[16] RINTA-AHO, T., GHANI, A., SAHASRABUDDHE, S. D., AND NIKANDER, P. Towards Software-defined Silicon: Applying LLVM to Simplifying Software. WISH - 3rd Workshop on Infrastructures for Software/Hardware co-design, Chamonix, France, 2011.

[17] SAHASRABUDDHE, S. D., SUBRAMANIAN, S., GHOSH, K. P., ARYA, K., AND DESAI, M. P. A c-to-rtl flow as an energy efficient alternative to embedded processors in digital systems. In *Proceedings of the 2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools* (Washington, DC, USA, 2010), DSD '10, IEEE Computer Society, pp. 147–154.

[18] TRIPP, J. L., GOKHALE, M., AND PETERSON, K. D. Trident: From High-Level Language to Hardware Circuitry. *IEEE Computer 40*, 3 (2007), 28–37.

[19] ZHANG, J., ZHANG, Z., ZHOU, S., TAN, M., LIU, X., CHENG, X., AND CONG, J. Bit-level optimization for high-level synthesis and FPGA-based acceleration. In *FPGA* (2010), P. Y. K. Cheung and J. Wawrzynek, Eds., ACM, pp. 59–68.