Towards Software-defined Silicon: Experiences in Compiling Click to NetFPGA

Pekka Nikander, Benny Nyman, Teemu Rinta-aho
Ericsson Research Nomadic Lab
Hirsalantie, 02420 Jorvas, Finland
first.last@ericsson.com

Sameer D. Sahasrabuddhe
Indian Institute of Technology Bombay
Powai, Mumbai
400076, India
sameerds@it.iitb.ac.in

James Kempf
Ericsson Research
200 Holger Way
San Jose, CA, 95314
James.Kempf@ericsson.com

ABSTRACT

Commercial C-to-silicon compilers, such as Catapult C and AutoPilot, are able to compile statically defined C and C++ programs into hardware definitions in VHDL or Verilog. However, they typically fail when they are fed anything more complex, such as virtual function calls, pointers to pointers, or dynamic memory management. Hence, generating hardware out of anything as complex as Click Modular Router elements, which represent a complex mix of quite dynamic C++, simply doesn't work out-of-the-box with these tools.

In this paper, we present our early results towards producing a tool chain for generating static, synthesisable C code out of Click elements. Our approach includes using the LLVM clang compiler to compile Click elements into an LLVM intermediate representation, combining the Click configuration front end with clang libraries, thereby generating a "Click compiler", and adding a few new link time optimisations to LLVM. With this approach, we are currently able to generate hardware out of a few simple Click elements, including the standard elements PushNull and SimpleIdle, and run those elements as a part of the Stanford NetFPGA reference router.

Categories and Subject Descriptors

B.5.2 [**RTL Implementation**]: Design Aids, D.3.4 [**Software Engineering**]: Processors — Translators

General Terms

Algorithms, Experimentation, Languages

Keywords

Click Modular Router, Stanford NetFPGA, LLVM, High-level Synthesis, C++, Verilog.

1. Introduction

Since its introduction 10 years ago, the Click modular router [1] has proven a popular platform for researching software routers and other packet processing applications. Written in C++, Click provides a collection of basic elements important in packet processing applications (e.g. queues, demultiplexers, etc.) and a framework for connecting these elements into packet processing pipelines. The Click platform allows students to build basic applications and researchers to extend the platform with new applications using C++.

The NetFPGA platform [2] provides a similar capability for researchers who are interested in investigating line speed packet processing applications. 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. 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 two systems seem to have complimentary strengths and weaknesses. The weakness of the Click platform is that the applications do not run at line speed. Since an important part of validating new research ideas in packet processing applications is measuring their performance, the lack of ability to run Click applications at line speed hinders the assessment of the research ideas behind them. Similarly, because NetFPGA is only programmable in Verilog, it is much less accessible to the network research community than a platform like Click. While developing hardware will always require some familiarity with basic hardware concepts, the learning curve behind Verilog is much steeper than many researchers are willing to climb. One approach to ease this learning curve would be to define a new (domain specific) high level language that can be transformed to a hardware description by a compiler. One example of this approach is the 'G' language [3]. However, we feel that the ideal research platform would allow researchers to express their designs for packet processing applications in a familiar language, such as C or C++ within the Click framework, then compile that to Verilog for synthesis into the NetFPGA.

Within the last several years, several CAD vendors have begun to offer tools that allow C or C++ code to be synthesised into hardware. These tools compile code describing a hardware design into Verilog or VHDL, which is then synthesised into hardware for an ASIC or for programming an FPGA. Examples of such tools are Catapult C from Mentor Graphics [4], C-to-Silicon from Cadence [5], and AutoPilot from AutoESL [6].

In this paper, we describe an experiment to investigate whether commercial high-level synthesis tools can support synthesis of Click modular router configurations into hardware. We chose to work with the LLVM compiler toolkit [7], as there are a wide variety of tools and an active development community working on LLVM. This allows us to take advantage of parallelising optimisations, originally developed for supporting multi-core processors, that are well supported in LLVM.

The rest of the paper is organised as follows. First, in Section 2, we provide some background on the Click modular router, high-level synthesis, and the LLVM compiler toolkit. Section 3 contains a description of our overall approach, while Section 4 provides more details on the experiment. In Section 5, a couple of

examples with early evaluation results are described. Section 6 discusses possible future work, while Section 7 concludes the paper.

2. Background

In order to understand our approach, it is necessary to understand the basic nature of the Click Modular Router, what high-level hardware synthesis involves in general, what kind of transformations are required on a software program in order to be able to generate hardware out of it, and what tools the LLVM Compiler Toolkit provides.

2.1 Click Modular Router

Click was introduced by Eddie Kohler [8] 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 in the graph. Click provides a few 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 [9] and prototyping a new architecture for large enterprise networks [10].

Click modules use the full power of C++ as an object-oriented programming language. That includes 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 code. A major part of the task in building the Click hardware synthesis tool chain was to figure out how the high-level synthesis tools dealt with these language constructs and to develop specific optimisations to get around them.

To characterise the complexity of the task ahead of us, we note that Click consists of some 730 classes, and has altogether about 340,000 lines of code (including comments and empty lines). However, out of these only about 80 belong the essential "library" classes that implement the underlying functional logic and the Click-specific C++ coding conventions. Furthermore, only a small fraction of the code lines of these library classes are related to the packet processing. The rest of the code mostly deals with the extensive configuration flexibility.

In order to support the packet processing functions of Click configurations in hardware, we have to modify the related code of the library classes, and create a tool chain that compiles all the rest of the packet processing code (in the non-library classes) into code that can be synthesised with current tools. So far we have completed only small faction of the needed library-related work, being able to support only quite limited Click configurations. For example, we do not yet support creating new packets.

2.2 High-level Synthesis

High-level synthesis is the process of generating a hardware description starting from an executable specification, usually in the form of a program written in a high-level programming language. The use of high-level specifications lowers the domain expertise required to produce hardware. Also, if the compiler guarantees correctness, then the need to verify the resulting hardware is also reduced; only the original high-level specification must be verified, which can make use of existing software methods.

Hardware generated from such a high-level program is expected to be inefficient in all the relevant aspects such as size, performance and energy efficiency. High-level synthesis can be effective only when the inefficiencies are small enough to be offset by the advantages. The quality of the hardware produced is affected by the following factors:

1. The features provided by the programming language for expressing the design — in particular, the ability to expose the parallelism inherent in the behaviour being implemented.
2. The quality of the input program — in other words, the freedom available to the programmer in using the features provided by the language.

An effective high-level synthesis flow should allow the programmer to write "pure" specifications that are not influenced by the specific target platform. For this, the compiler should not restrict or reinterpret the input programming language in a way that exposes the details of the target platform.

The state of the art, represented by commercial tools, only supports a restricted subset of C and C++ constructs. In the typical case, the software has to be specifically written with hardware in mind. That is, the programmer has to be aware, at least at some level, that the code will be synthesised into hardware, when writing and optimising the code. However, the ability to design hardware in a widely known and easy-to-use language far outweighs the drawbacks of having to program in a particular way.

From this point of view, our approach goes much further, aiming at supporting the full Click usage of C++. This requires supporting a subset of the C++ language that is larger than what the current tools support. However, the task is made easier by the quality of the Click code, since it is a mature library designed to run within the Linux kernel.

2.3 Extending compilation into hardware

Compared to software, hardware is characterised by parallel operations and inflexibility, typically resulting in better energy efficiency and higher speed. But this also makes it difficult to bridge the gap from a software program to a hardware description, and a lot depends on the way in which high-level concepts are modelled in hardware. For example, function pointers or C++ virtual methods can be supported only if the notion of a "function call" is first mapped to a suitable mechanism in hardware. Also, when a program extensively uses pointers for accessing data, performance is greatly affected by the rate at which data can be accessed from memory.

Standard software optimisations can be effective in removing these obstacles in a large number of cases. For example, if the entire executable program is statically accessible, then function pointers can be replaced by branches in the control-flow. Similarly, a memory reference analysis of the program can replace memory accesses with data-flow, or at least provide sufficient information to remove bottlenecks encountered by the memory accesses.

In general, high-level synthesis aims to utilise the control and data parallelism, inherent to any design, to the highest possible degree. Many of the optimisations for parallel execution, including superscalar and VLIW processors, when applied to their extreme, result in code that has maximal parallelism and may therefore produce highest performing hardware with multiple parallel execution elements. Typical examples of such approaches include

duplicating basic blocks into superblocks [11], as well as loop peeling and unrolling [12]. In practice, however, the current commercial HLS tools have difficulties but with the simplest, statically defined loops. Dynamically bounded loops and variable sized arrays are both likely to cause difficulties.

Our current approach combines typical software-oriented optimisations and some parallelising optimisations together with compile-time generated constant data structures and very aggressive link-time constant propagation. This allows us to compensate for the limitations of the high-level synthesis process in general, and those in the current commercial tools in particular. Our customised compiler flow lowers the original Click programs into versions that are more suitable for hardware synthesis. The flow generates code that gives reasonable results when processed by the commercial synthesis tools.

2.4 LLVM Compiler Toolkit

LLVM [7] is a collection of modular components for building compiler tool chains. The LLVM components operate on an intermediate language, called the LLVM Intermediate Representation (LLVM IR). The LLVM core consists of the following components:

- A compiler for C, C++, and Objective-C, called *clang*, which compiles these source languages into LLVM IR.
- A number of code optimisers.
- Backends for many popular target hardware architectures.
- A debugger, LLVMDB, that operates on code compiled through LLVM.

LLVM also provides support for the GCC family of compilers, thereby including all their supported languages. LLVM has been used to implement a variety of language tool chains, including previous attempts to generate hardware [13] and bit-level optimisation of HLS data flows [14].

One of the most interesting uses of LLVM is as an intermediate representation based on the Single Static Assignment (SSA) form. The approach has also some promise for high-level hardware synthesis. For example, TCE [15] is a set of tools for designing processors based on Transport Triggered Architecture. TCE uses the LLVM *clang* compiler as the front end for compiling hardware designs written in C and C++. As another example, the UCLA xPilot project developed a high level language hardware synthesis tool chain using LLVM [16]. While the goals of xPilot were quite ambitious, the basic high level hardware synthesis technology was successful enough that it evolved into a product, the AutoPilot product from AutoESL [6].

3. Overall approach

Our overall approach to preparing a Click application for hardware synthesis is shown in Figure 1. The tool chain starts with compiling each Click element (both standard and user-supplied ones) with both the normal Click libraries and with our adapted, hardware-friendly *Click hardware "runtime"* include files[1]; see Section 4.1. The former compilation creates a usual, "standard" Click module, which can be fed to the Click tools and executed in software. The latter compilation results in precompiled versions of the Click elements in the LLVM IR format.

At the next step, we use our click "compiler" to generate LLVM IR out of a user-supplied Click configuration file (see Section 4.2). The compiler first parses the Click configuration and initialises the resulting Click software router. We then "freeze" the initialised router and generate an LLVM IR description out of it. The LLVM IR description is then combined with the Click elements with LLVM ld (not really an object code linker, but more of a module combination tool). This forms a single large IR module that is then run through a set of optimisers to improve the hardware synthesis characteristics; see Section 4.3. Finally, the linked IR module is decompiled back into C by LLVM llc. This forms an optimised C source code file, which can then be submitted to a commercial HLS tool.

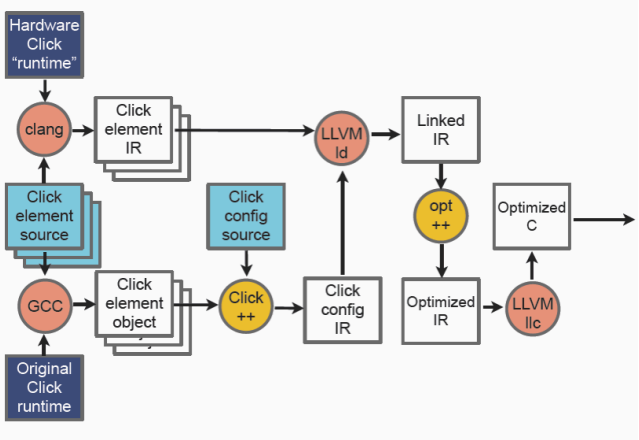

Figure 1: Click hardware synthesis tool chain

Hence, technically, our contribution consists of two new tools, together with the supporting libraries:

- Click++, a Click compiler that reads in a Click configuration file, instantiates a Click software router, and generates a static LLVM IR configuration, and
- Opt++, a set of LLVM IR optimisers that take a linked LLVM module and improve it, together with the standard LLVM optimisers, for hardware synthesis.

4. Implementation details

While the overall approach appears simple enough, the devil hides in the details, as usual. In this section, we describe some of the issues we have solved so far, and some of the aspects we are still trying to find solutions for.

4.1 A hardware-friendly Click "runtime"

Click, as distributed, 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. However, most elements use only a few of these classes directly, and even fewer in the actual packet processing methods.

A big problem from the hardware synthesis point of view is that many of the library classes are far too complex to be synthesised as such. For example, the `Element` class has almost 3000 lines of source code (including comments), out of which only some 30 lines are directly related to packet processing. Now, while in theory most of the non-packet-processing functionality could be dropped at the optimisation time by dead code elimination, in

[1] We are currently using both clang and GCC as our front ends, due to some clang bugs preventing us from using clang for compiling all Click code; however, we are planning to move solely to a clang as soon as feasible. Hence, we ignore this detail in the following.

practise the source code defines global constants and non-static constructors that would require explicit removal so that dead code elimination could fully exclude all unneeded code.

Another problem is that some of the essential classes, such as the `Packet` class, include features and fields that are not needed in a hardware implementation or that need to be implemented in a different way in order to allow efficient hardware synthesis.

Our initial approach to solve these problems has been to construct a simplified version of the Click runtime library, suitable for hardware synthesis. In most cases, we don't even need to implement the actual classes; it is sufficient to have suitable class declarations that allow the actual Click elements to compile into IR. Hence, our current runtime consists of only three class implementations and the declarations for about 30 classes.

The other "side" of the "runtime" (not shown in Figure 1) is then written in Verilog, and consists of modules that integrate the synthesised results out of our tool chain with the NetFPGA reference router. The current implementation simply buffers a packet in BRAM and gives a handle to it, through a FIFO, to the synthesised Click configuration, which in the end hands back the packet handle through another FIFO. Unless the packet was dropped, the Verilog runtime then feeds the (perhaps modified) packet further along the NetFPGA reference router user-data pipeline.

While this approach works well enough, it causes at minimum a packet-long buffering delay and in practise more. We are currently trying to identify how to generate more streamlined code, at least for the simplest Click configurations.

4.2 Compiling Click configurations

A Click *configuration* defines a particular assembly of Click elements, thereby constructing a packet processing application; in Click terms, a `Router`. In practise, when the user-level 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. (More recent versions of Click also support limited runtime reconfiguration. We do not plan to support such features, though.) In our tool chain, the first two steps of this process are performed by our Click 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 the virtual methods `configure` and `initialize`.

Our Click compiler is essentially identical with the `click` user-level tool up to this point. However, while the standard tool now initiates packet processing, our compiler writes out the resulting initialised router. For this, our compiler uses the clang libraries.

We have implemented a new clang `Action` that first compiles a single C++ source code file into the internal Abstract Syntax Tree (AST) representation, just as `clang++` would do. As the next step, our action inspects the initialised Click router and adds to the AST a set of constants that define the router structure. Finally, the LLVM libraries generate LLVM IR out of the AST representation.

With this approach, we drive the action first to read in the C++ declarations of the Click used elements, using the "hardware-friendly" library include files. This gives us the needed type definitions, as used in the synthesised hardware; we don't need to hard-code any type information into the tool itself. Using this type information, we can then generate constants that define the packet processing graph, and another set of constants and variables that define the element values and registers. The Click library, then, has enough of introspection facilities that we can relatively easily loop up the required information from the initialised software router.

4.3 Optimising dynamic C++ constructions

The simple combination of the LLVM IR representing Click element classes and the Click initialised router is still a relatively large and dynamic piece of code. Among other things, we have to deal with C++ constructors, virtual method calls, and flattening control flow.

Firstly, the IR data definitions generated from the AST-level constant instances rely on explicit calls to the class constructors, meant to be run when the resulting software process is loaded for execution. In our hardware implementation such explicit constructions would both mean unnecessary hardware and unnecessarily complex control logic. Hence, we want to make sure that all C++ constructors are either optimised away already during the compilation time or replaced with specialised, Click-specific circuits. We currently implement only the former, relying on aggressive constant propagation. However, we do recognise that we will also need to support the latter, e.g. in order to allow the elements to create new packets on the fly.

Secondly, each Click element may implement a couple of dozen virtual methods. Fortunately, most of these are meant to be used at configuration and initialisation time, with only a handful being executed at packet processing time in the typical case.

One of the major tasks in optimising the system for synthesis is to deal with the virtual method calls. In general, there are two practical ways to deal with them: either optimise them into static function calls, or figure out all the possible functions that the virtual method call may branch to, depending on the current object type, and replace the IR-level function pointers with explicit control structures. Again, at this stage we implement only the simple approach, where use repeated aggressive constant propagation phases, interleaved with our own optimisations, to replace all of the virtual method calls with their statically computed equivalents.

```
class Element {
  virtual void push(int port, Packet *p){
    p = simple_action(p);
    if (p) output(0).push(p);
  }
  Port& output(int port) {
    return port(true, port);
  }
  Port& port(bool isout, int port) {
    return _ports[isout][port];
  }
private:
  Port* _ports[2];
}
class Port {
  void push(Packet* p) { _e->push(_port, p); }
private:
  Element* _e;
  int _port;
}
```

Listing 1. Simplified Click packet pushing source code

Finally, the basic software-level flow control of Click is based on recursive virtual method calls from one element, through an intermediate class, to another element. The relevant source code is depicted in Listing 1. In there, the default implementation of the virtual `Element` method `push` finds the right output `Port` through the `output` and `port` methods, and calls the `push` method of the port. The `Port` in turn calls the `push` method of the *next* element in the graph.

From the optimisation point of view, there are two problematic aspects in this call path. First, the `_ports` instance field is a variable, and may change during execution; indeed, it is changed when the elements are assembled into the graph during the configuration phase. Second, the actual function called by the first virtual method call depends on the type of the Element.

With our separate hardware-friendly runtime, Click compiler, and constant propagation, these problems can be overcome relatively easily:

1. In the runtime, we define the `_ports[]` field as a constant pointer pointer, pointing to constant pointers, each pointing to a constant `Port`.
2. In the Click compiler, we emit readily assembled, constant `Ports`, and install the constant pointers pointing to them into the `Elements`. We also emit the first `push` invocation, from the runtime to the first `Element` in the graph.
3. Assuming that the first element in the graph does not redefine the `push` method, once the resulting LLVM IR is run through optimisation, the optimisers are able to correctly determine the type of the first `Element` in the graph and inline the code, all the way to the next `Element`, essentially producing IR equivalent to the code shown in Listing 2.

```
Element *e; Packet *p;
...
p = E1::simple_action(p); // inlined
if (!p) return;
e = e->_ports[true][0]->_e;
e->push(CONSTANT, p); // tail recursion
return;
```

Listing 2. Simplified Click packet pushing source code

Careful inspection shows that this can be further optimised, since the Element in the end is a known constant. Hence, we run the repeatedly optimisers again, until the code cannot be any further optimised. A typical result, hand-converted to C, is shown in Listing 3 (the `simple_action` methods are also inlined, but that is not shown.)

This code is now simple enough to be synthesised by the commercial tools, since it simply contains a single basic block, with simple exit points.

In practise the process is somewhat more involved than indicated above, though. For example, in certain cases the standard optimisers are able to turn the recursive calls into a loop, requiring loop peeling for the repeated constant propagation to work. As another aspect, the code must be slightly transformed to explicitly return the resulting packet back to the hardware-friendly runtime so that the optimisers don't optimise everything away.

```
Element *e; Packet *p;
...
// First element
p = E1::simple_action(p); // inlined
if (!p) return;
// Second element
p = E2::simple_action(p); // inlined
// Never drops packets
...
// Last element
p = EN::simple_action(p); // inlined
// Always returns a NULL
return;
```

Listing 3. Optimised Click packet pushing

At this moment, we are able to compile only very simple Click configurations, such as linear graphs, into synthesisable C. Anything more complex typically results in constructions that cannot be synthesised directly, requiring more manual tweaking with the optimisations. However, we expect the situation to grow progressively better as we keep enhancing the system components.

5. Examples and early evaluation results

While we so far support only a small fraction of the Click packet handling APIs, the obstacles encountered and solutions to them have been quite instructive. In this section, we first give a couple of detailed examples, illustrating the current tool chain in action. Then, we summarise the current status of the commercial HLS tools, in the light of our still quite limited experience, and discuss the lessons we've learned so far.

5.1 The tool chain in action

We first focus on how the tool chain works in practise. For this purpose, we use a very simple configuration, in Listing 4. In this configuration, the generated hardware reduces to essentially nothing, since this `SimpleIdle` element in the configuration simply eats the packets, without doing anything.

```
FromFPGA -> SimpleIdle;
```

Listing 4. A very simple Click configuration file

We first run our Click compiler with the given Click configuration file. The compiler generates an LLVM IR file, named `configuration.ir`. This file contains the class definitions for Click elements `FromFPGA` and `SimpleIdle`, as well as `Element`, `Element::Port`, and `Packet`.

```
define zeroext i8 @veriglue(i64* %memory, i8 zeroext %pi, i8* %pkt_dropped) nounwind ssp {
bb5:
  %0 = zext i8 %pi to i64
  %1 = shl i64 %0, 8
  %2 = getelementptr inbounds i64* %memory, i64 %1
  %3 = load i64* %2, align 8
  %4 = or i64 %3, 0x800000000000000
  store i64 %4, i64* %2, align 8
  store i8 1, i8* %pkt_dropped, align 1
  ret i8 0
}
```

Listing 5. The final `veriglue()` function, corresponding to the configuration in Listing 4, in LLVM IR

```
unsigned char veriglue(unsigned long long llvm_cbe_memory[16][256],
                       unsigned char llvm_cbe_pi, unsigned char *llvm_cbe_pkt_dropped) {
  unsigned long long *llvm_cbe_tmp__1;
  unsigned long long llvm_cbe_tmp__2;
  llvm_cbe_tmp__1
     = (&llvm_cbe_memory[((signed long long)((((unsigned long long)llvm_cbe_pi)) << 8ull))]);
  llvm_cbe_tmp__2 = *llvm_cbe_tmp__1;
  *llvm_cbe_tmp__1 = (llvm_cbe_tmp__2 | 0x800000000000000ull);
  *llvm_cbe_pkt_dropped = ((unsigned char )1);
  return ((unsigned char )0);
}
```

Listing 6. The final `veriglue()` function, corresponding to the configuration in Listing 4, in C

We also separately compile the Click element classes into LLVM IR files. A separate C++ source file, `glue.cc`, is compiled into `glue.ir`. This file contains a function called `veriglue`, the C++-side interface to the existing NetFPGA user data path. Once all the source files have been compiled into IR, we apply `llvm-link` to mix them together into a single `linked.ir` file (currently 679 lines of code).

After having all the needed code in a single LLVM IR source file, we can then run a number of optimisations to the code, as described earlier in the paper. We first run standard `-O3` and `-std-link-opts` once, then iteratively perform loop peeling and constant propagation optimisations, followed by the standard `-O3` and `-std-link-opts` until the code doesn't optimise any further. The resulting code is in a file called `veriglue.ir` (19 lines of code) (see Listing 5 for the function `veriglue()`).

The final step is to convert the LLVM IR back into C, with `llc -march=c`, because the HLS tool we are using does not read LLVM IR. We also need to perform some modifications (using `awk`) to the C code, due to some bugs and limitations in the HLS tool. Eventually, we have a `veriglue.c` which we can then use as a source file for the HLS tool (see Listing 6). The `veriglue()` and its enclosing function are used together the top level module in the synthesis.

Looking into the details of final `veriglue()`, we can see that there are three function arguments: `%memory`, `%pi` and `%pkt_dropped`. The first argument is bound to the BRAM memory in our hand-written Verilog wrapper, the second one is the packet index, which tells where the incoming packet is stored in the BRAM memory, and the third one is used to signal if any of the Click elements decided to drop the packet.

In this first example, the actual operation of the function is very simple. As the Click element `SimpleIdle` simply drops each packet by calling the standard Click method `Packet::kill()`, the resulting code is ultimately optimized into simply storing a "killed" flag into the status word of the packet and setting `%pkt_dropped` to 1. Storing the flag seems unnecessary in this case, but the compiler cannot infer that storing the flag is actually unnecessary. This can potentially be optimised away in the future.

The next step is to compile this C source into Verilog with the HLS tool, together with the hand written `process_packet.c` file, which interfaces with the NetFPGA user data path. The result is a single Verilog module that is finally used as a part of the NetFPGA user data path, and the operation can be verified with the existing test scripts of the NetFPGA project and simulating in e.g. ModelSim. Finally, the NetFPGA card can be flashed with the synthesized hardware image and used with real network traffic.

5.2 Minimally modifying packets

In the previous example we saw how the tool chain works. In this example, we now focus on actually doing something (almost) useful.

```
FromFPGA -> Minimal -> ToFPGA;
```

Listing 7. Click configuration to minimally modify packets

The Click configuration file for this example is shown in Listing 7. The `Minimal` element adds a one to the first word of each packet passing by; the relevant `simple_action` method is illustrated in Listing 8.

```
Packet *
Minimal::simple_action(Packet *p) const {
  unsigned long long *data
     = p->uniqueify()->data();
  data[0]++;
  return p;
}
```

Listing 8. The `simple_action` method for Minimal

We run exactly the same tool chain as in the first example. The resulting code is slightly longer, and the resulting `veriglue()` is illustrated in Listing 9. The bolded lines contains the code that increments the first word of the packet data by one.

```
unsigned char veriglue(unsigned long long llvm_cbe_memory[16][256],
                       unsigned char llvm_cbe_pi, unsigned char *llvm_cbe_pkt_dropped) {
  ...
  llvm_cbe_tmp__1 = (((unsigned long long )(unsigned char )llvm_cbe_pi)) << 8ull;
  llvm_cbe_tmp__2 = (&llvm_cbe_memory[((signed long long )llvm_cbe_tmp__1)]);
  llvm_cbe_tmp__3 = *llvm_cbe_tmp__2;
 *llvm_cbe_tmp__2 = (llvm_cbe_tmp__3 & 0xF7FFFFFFFFFFFFFFull);
  llvm_cbe_tmp__4 = (&llvm_cbe_memory[((signed long long )(llvm_cbe_tmp__1 | 1ull))]);
  llvm_cbe_tmp__5 = *llvm_cbe_tmp__4;
 *llvm_cbe_tmp__4 = (((unsigned long long )(((unsigned long long )llvm_cbe_tmp__5)
                     + ((unsigned long long )1ull))));
  *llvm_cbe_pkt_dropped = ((unsigned char )0);
  return llvm_cbe_pi;
}
```

Listing 9. The resulting C code for the Click configuration in Listing 7 (local variable declarations removed)

5.3 Limitations in commercial HLS tools

So far we have real experience of only one commercial HLS tool, denoted as "HLS A" in this paper. In addition to that, we have had the possibility to briefly test our system with another commercial tool ("HLS B") and with the few years old, prototype level xPilot [16] tool from UCLA. Hence, what we describe here reflects almost solely our experience with the tool that we have had most experience with, HLS A; however, as of now we have little reason to believe that there are really substantial differences between the available commercial tools. Of course, in the light of our results outlined below, we hope that our further testing would prove the situation to be otherwise.

In HLS A, the number of bugs and limitations has been quite high, considering that the tool has been available for a number of years already. Apparently we are using the tool in a very different way than what most people do, e.g., as we feed the tool with machine generated C code, which sometimes is quite convoluted.

However, not all bugs or limitations are explained by this; instead, the vendor of HLS A has clearly simply decided not to support some C / C++ constructions. For example, the tool explicitly does not support unions, goto, or pointers to pointers. It disallows conversion of certain data types to void pointers, and this makes void pointers almost useless in practise. Even in the case of explicitly mapping arrays to memory, it has difficulties in accessing the arrays through pointers, sometimes *silently* failing to generate RTL for certain constructions at all. Furthermore, it has problems in handling dynamically bounded loop exit conditions; in many of our cases, it simply has not been able to generate RTL at all.

With both HLS A and xPilot, the number of C / C++ constructions that simply failed to compile was quite high. In many cases, trying to feed them with Click source code, as such, caused either assertion failures, silent crashes (core dumps), or explicit error messages about unsupported C or C++ features. Even with the commercial tool HLS A, the number of assertion failures and silent crashes was surprisingly high in our initial testing.

5.4 Lessons learned

Hardware acceleration of software applications is still more an art than an engineering discipline. While the marketing material for the commercial HLS tools lead one to think that they are able to produces synthesisable RTL out of most C / C++ programs, the reality seems to be that they only support programs that have been written with the hardware in mind. They are, understandably, still quite far from allowing one to take any existing software-oriented C or C++ code and to generate hardware out of that with only minimal or no modifications.

Conversely, when starting from an existing legacy code base (like the Click code base in our case), one has to understand the application semantics of the software, be imaginative in figuring out how to map those semantics into hardware, and build at least some custom tools that embed this domain-specific knowledge.

In our case, and we imagine this to be fairly common, the features of the software can be divided into start-up time operations and actual "run-time" operations. The former are performed when a software process is started, and are often guided by a number of configuration files, settings, command line flags, and other such external pieces of information. In essence, the software parses those external data and adjusts its run-time operation accordingly.

During the actual "run-time" processing, the software performs a relatively repetitive task on some data set. In our case, the packet handling operations are applied on the incoming packets. In some other case, the software may process data coming out of a file or a set of sensors. In the typical case, the same piece of software is able to perform a large and often essentially unbounded set of operations on the input data, depending on the configuration. For hardware synthesis, what is typically wanted is less flexibility and higher performance. Hence, instead of supporting all the options the legacy software permits, the desired hardware configuration typically would just perform a relatively fixed set of operations on the input data.

An old trick is to "freeze" a software process immediately after it has performed the start-up time configurations but but before it starts the "run-time" processing. This allows the specific configuration of the software to start up much faster, enhancing user experience. This has been used at least since the 70's, for example, in the GNU Emacs editor to precompile and dump lisp code during the startup phase [17], and also in modern operating systems to improve boot-up times.

For hardware synthesis, such "configuration freezing" is essential in order to reduce flexibility and produce more efficient, more fixed hardware. While the "freezing" process itself could be essentially automated, domain-specific knowledge is needed for knowing when to freeze and how to interpret the frozen memory configuration. For hardware synthesis, especially the latter is tricky, since the memory dump itself does not contain any information of which parts of the memory will remain unchanged and which will change during the "run-time" processing.

In our work, we have tried to apply the high-level hints in the C++ source code (`const` fields in classes, `const` instances, etc.) to figure out the relevant data. Unfortunately, both the LLVM GCC and LLVM clang compilers are buggy in this respect; they do not always faithfully represent the `const` information in the generated IR but sometimes simply drop it, especially in the case of constant instances of more complex C++ classes. For now, we have fixed this mostly with simple scripts that modify the textual IR based on our domain-specific knowledge. Our plan is to move most of these "hacks" into clang proper as we proceed further.

Another related problem is that even the modern compilers, such as GCC and clang, do not try hard enough with constant propagation when facing complex control flow. For typical software, they have little incentive, as such transformations would considerably increase the code size. However, for hardware generation, such aggressive constant propagation is absolutely essential, since each variable-turned-into-a-constant means less hardware, and thereby higher speed and less energy consumption. For now, we simply manually force LLVM to run its optimisation phases multiple times, interspersed with our domain-specific transformations, in order to force it to do all constant propagation.

A final problem we have faced is that LLVM does not try to optimise loops at the IR level very hard. Hence, we have used manual loop-unrolling options and repeated iterations to make LLVM to try harder, and to peel or unroll the loops as far as is possible.

6. Future work

As of now, we have only started our journey. However, by now it has become apparent that the current HLS tool we use (HLS A) starts to be more a nuisance than genuine help. Hence, we are trying to move away from it, testing some other commercial HLS tools and university research results, such as AHIR [18]. We hope to get rid of many of the present hacks with such (hopefully) better back-end tools, allowing us to focus on enhancing the Click compiler, our domain-specific LLVM transformations, and the Click hardware-friendly "run-time" library. With this approach we

hope to be able to support maybe half of the existing Click elements.

However, we expect that some of the existing Click elements do things that are simply not possible to implement in hardware, at least not with the current HLS tools. For example, some elements communicate with user-level manement entities through sockets, and some other elements communicate with other elements by ad hoc C++-level mechanisms that rely on dynamic memory management and packet-processing-time usage of Click metadata. In such cases, we plan to punt part of the code into a built-in co-processor. (Dual PowerPC in the case of current NetFPGA.) While we do not expect any real progress in this space for some time, so far we have been able to push packets to the NetFPGA PPC for processing, and make the PPC to perform minimal manipulation on the packets[2].

7. Conclusions

In this paper, we have described our early efforts in generating NetFPGA RTL out of the Click Modular Router software configurations. Our approach is based on using the LLVM Compiler Toolkit to build a new compiler front end that reads in a Click configuration file and produces a corresponding static configuration in LLVM IR, representing the Click elements as constant C++ class instances. In addition to that, we have written an early version of a Click hardware "run-time" library, consisting of modified version of the Click library classes, and a few LLVM transformations, applying domain-knowledge-embedding optimisations to the LLVM IR. With this approach, we are able to produce synthesisable C code, and using a commercial HLS tool, Verilog out of that C code.

Perhaps the most important step in our approach is the "freezing" of a Click configuration, once it has been instantiated, configured, and initialised. This approach allows us to use the out-of-the-box Click code for the lexer and parser of our Click compiler. We then implement our own logic to construct a C++ Abstract Syntax Tree (AST) out of the "frozen" Click configuration, letting the LLVM clang libraries to produce LLVM IR out of that.

Our efforts so far indicate that the currently available commercial HLS tools are quite restricted and even buggy. They currently don't support high-level C++ constructions, such as virtual method calls, pointers to pointers, or dynamic memory management. Hence, an essential part of our tool chain is to use domain-specific knowledge to transform the quite dynamic and flexible Click C++ code into configuration-specific, fixed C code, which then can be fed to a HLS tool.

8. References

[1] Kohler, E., Morris, R., Chen, B., Jannotti, J., and Kasshoek, F., "The Click Modular Router", *Operating Systems Review,* **34**(5), pp 217{231, December, 1999.

[2] Lookwood, 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", IEEE International Conference on Microelectronics Education, June 3-4, 2007, San Diego, CA.

[3] Michael Attig and Gordon Brebner, "High-level programming of the FPGA on NetFPGA", NetFPGA Developers Workshop 2009, Palo Alto, CA.

[4] Mentor Graphics Catapult C: http://www.mentor.com/products/esl/high_level_synthesis/catapult_synthesis/

[5] Cadence C-to-Silicon: http://www.cadence.com/products/sd/silicon_compiler

[6] AutoESL AutoPilot: http://www.autoesl.com/autopilot_fpga.html.

[7] Lattner, C., and Adve, V., "The LLVM Compiler Framework and Infrastructure Tutorial", LCPC'04 Mini Workshop on Compiler Research Infrastructures, West Lafayette, Indiana, Sep. 2004.

[8] Kohler, E., "The Click Modular Router", Ph.D. dissertation, MIT, November, 2000.

[9] Morris, R., and Chen, B., "Flexible Control of Parallelism in a Multiprocessor PC Router", Proceedings of the USENIX 2001 Annual Technical Conference, June 2001.

[10] Kim, C., Caeser, M., Rexford, J., "Floodless in Seattle: A Scalable Ethernet Architecture for Large Enterprises" Proceedings of SIGCOMM, Seattle Washington, August, 2008.

[11] W.M.W. Hwu, S.A. Mahlke, W.Y. Chen, P.P. Chang, N.J. Warter, R.A. Bringmann, R.G. Ouellette, R.E. Hank, T. Kiyohara, G.E. Haab, J.G. Holm, and D.M. Lavery, "The superblock: an effective technique for VLIW and super-scalar compilation," Journal of Supercomputing, Vol. 7, pp 229–248, 1993.

[12] Jeannette Ferrante, Karl J Ottenstein, Joe D Warren, "The Program Dependence Graph and Its Use in Optimization," rogramming Languages and Systems, Vol. 9 (3), pp. 319-349, 1987.

[13] Justin L. Tripp, Maya B. Gokhale, and Kristopher D. Peterson, "Trident: From High-Level Language to Hardware Circuitry", IEEE Computer, Mar. 2007.

[14] Jiyu Zhang, Zhiru Zhang, Sheng Zhou, Mingxing Tan, Xianhua Liu, Xu Cheng, Jason Cong, "Bit-Level Optimization for High-Level Synthesis and FPGA-Based Acceleration", Proc. of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays , Monterey, CA, February 2010.

[15] P. Jääskeläinen, V. Guzma, A. Cilio and J. Takala, "Codesign Toolset for Application-Specific Instruction-Set Processors", in Multimedia on Mobile Devices, San Jose, California, USA, Jan 2007, Proc. SPIE Vol. 6507, 65070X.
See also `http://tce.cs.tut.fi/`.

[16] D. Chen, J. Cong, Y. Fan, G. Han, W. Jiang, and Z. Zhang, "xPilot: A Platform-Based Behavioral Synthesis System" SRC TechCon'05, Portland, OR, Nov, 2005.

[17] Jonathan M. Smith and Gerald Q. Maguire, Jr., "Effects of copy-on-write memory management on the response time of UNIX fork operations", Computing Systems, Vol. 1, Number 3, pp 255–278, Summer 1988, Usenix Association.

[18] Sameer D. Sahasrabuddhe, Hakim Raja, Kavi Arya and Madhav P. Desai, "AHIR: A Hardware Intermediate Representation for Hardware Generation from High-level Languages", 20th International Conference on VLSI Design, January 2007.

[2] The kudos for our ability to use the NetFPGA PPCs goes mostly to Erik Rubow, who first figured out how to do that in practice.