Towards Software-defined Silicon: Applying LLVM to Simplifying Software

Teemu Rinta-aho
Ericsson Research
teemu.rinta-aho@ericsson.com

Adnan Ghani
Ericsson Research
adnan.hassan.ghani@gmail.com

Sameer D. Sahasrabuddhe
IIT Bombay
sameerds@it.iitb.ac.in

Pekka Nikander
Ericsson Research
pekka.nikander@ericsson.com

ABSTRACT

High-Level Synthesis (HLS) is a result of similar development in hardware design as there has been with software. Writing an efficient program does not require the programmer to understand machine language or the target CPU — it is the modern compiler that knows how to transform a high level language into efficient machine code. Similary, current HLS tools raise the abstraction level from Verilog or VHDL all the way to C or C++.

The current HLS tools are mainly targeted to hardware designers to improve their productivity. We, however, would like to take a different approach and target the network researchers — who usually have more competence in writing software than designing hardware. In this paper, we describe an experimental tool chain that is able to transform existing, software-oriented C++ — within the limited domain of Click-based packet processing — into a hardware description.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Compilers, Optimization

Keywords

High-Level Synthesis, LLVM, FPGA

1. INTRODUCTION

Hardware-based implementations of algorithms are typically characterised by parallel operations and relative inflexibility, resulting in better energy efficiency and higher operational speed than comparable software ones. At the same time, producing hardware both in terms of design and manufacturing is orders of magnitude more expensive than producing software, causing a major obstacle for research and exploration in areas where high processing speed and energy efficiency are desirable, such as packet communication networks.

Today, writing packet processing applications in software offers a flexible and easy way for students and researchers for experimentation. One popular tool for this is the Click modular router [9]. On the other hand, the Stanford NetFPGA platform [11] allows for flexible hardware designs for those who are interested in investigating line speed packet processing. Comparing these two; while programming packet processing applications in Click C++ is easy and flexible, describing even a simple application in Verilog (for the NetFPGA) is a relatively major undertaking, taking easily a few months for an average student.

Based on our limited experiences so far, we surmise that an ideal research platform would allow one to express their designs in a widely familiar software programming language, such as C or C++, and then be able to convert that into a hardware design that could be tested in real life, e.g. using the NetFPGA. Unfortunately, our limited testing of the existing high level synthesis (HLS) tools indicate that their target group is the hardware designers who use a high level language to increase productivity in hardware design. They are definitely not ready to take an existing piece of software and transform that into hardware [12].

In general, it is impossible to translate an arbitrary software program into a hardware description without adding some kind of a general purpose controller at the hardware side. At the extreme, any `eval`-like construction requires the ability of running arbitrary programs. More down to the earth, unpredictably bound recursion, function pointers, or virtual methods can only be supported if the notion of a "function call" is first mapped to a suitable mechanism in hardware. A lot depends on the way in which high-level concepts are modelled in hardware.

Well-known software transformations may in many cases be used to convert such problematic constructions into more hardware-friendly ones. However, sometimes the transformations must be used in a way that would not make any sense from software-optimisation point of view. For example, whenever the entire executable program is available to the optimisation and static enough, function pointers can be replaced with conditional branch instructions.

In this paper, we describe an experimental tool chain that is able to transform existing, software-oriented C++ algorithms — within the limited domain of Click-based packet

WISH-3 2011 Chamonix, France

processing — into a hardware description. We have chosen to work with the LLVM compiler toolkit, as there are a wide variety of tools and an active development community working on LLVM. This allows us to take advantage of the existing parallelising optimisations. LLVM modularity also allows us to easily write our own transforming compiler passes, and to add our own back end, for which we use AHIR [13].

Many of the parallelising optimisations, 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 [5], as well as loop peeling and unrolling [4].

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 the toolchain presented earlier [12] was about exploring the possibilities of some commercially available HLS tools as the backend of our toolchain. The version presented in this paper is completely new work. We have switched from a commercial backend to AHIR, and have written the frontend completely from scratch, building on the knowledge gained when working with the previous version.

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. In section 5 we conclude the paper.

2. BACKGROUND

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 [2] and prototyping a new architecture for large enterprise networks [7].

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 try to find the correct code transformations for the language constructs that can not be directly mapped into a hardware description.

2.2 NetFPGA

The NetFPGA platform [11] 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.

2.3 LLVM

LLVM [10] 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 a compiler driver, a number of analysis and code optimization passes, and a debugger. Several frontends and backends are using LLVM: clang is intended as a replacement for GCC. However, 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 [14] and bit-level optimisation of HLS data flows [15].

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 [6] 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 [3] which later evolved into a product, the AutoPilot from AutoESL [1].

2.4 AHIR

AHIR [13] is a backend for LLVM that transforms LLVM bytecode to VHDL. It is developed at IIT Bombay and is to be published as open source. It is a toolchain on its own, consisting of five different tools with intermediate result files. The LLVM IR is first used to build a CDFG, which is then transformed into an AHIR representation. This can then be linked and optimised, and finally written out as VHDL modules. Current version of AHIR supports a wide set of LLVM IR - the few notable exceptions are function pointers and recursion.

3. OVERALL APPROACH

Our overall approach to preparing a Click application for hardware synthesis is depicted in Figure 1. The tool chain starts by compiling each Click element, written in C++ (both standard and user-supplied ones), into both a linkable object file and an LLVM IR file. The object files are linked to our Click2LLVM tool and loaded into memory by the Click library functions. The IR files are loaded and linked to form an LLVM Module, which can be handled with the LLVM API functions, also linked into Click2LLVM.

At the next step, we use Click2LLVM to generate LLVM IR out of a user-supplied Click configuration file. The compiler first parses the Click configuration and initialises the

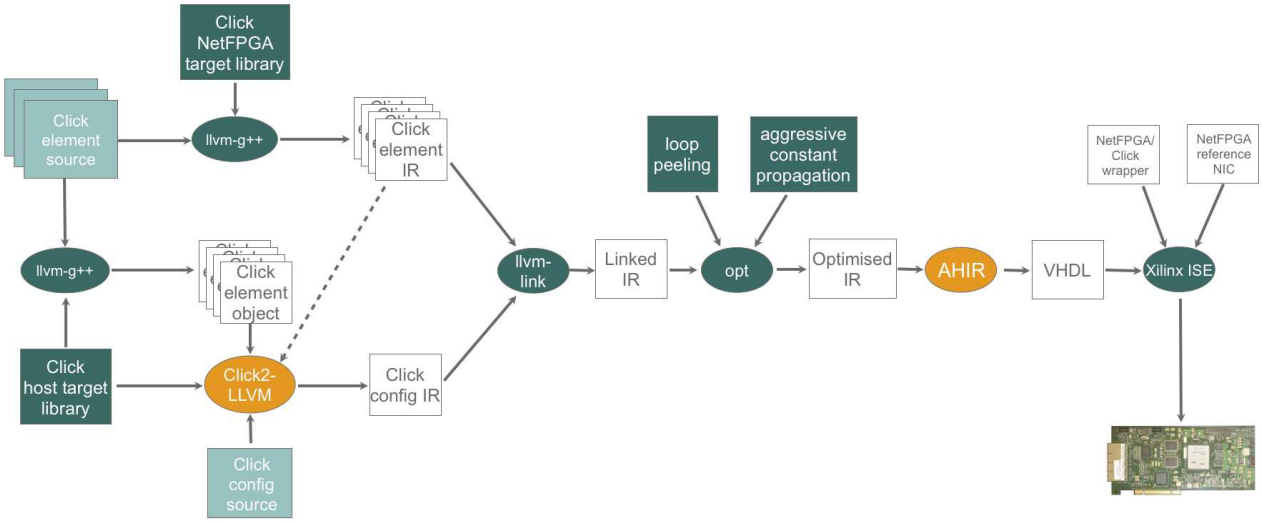

Figure 1: The Click-to-NetFPGA toolchain

resulting Click software router. We then "freeze" the initialised router and generate an LLVM IR description out of it, by directly reading the memory contents of initialised elements. The memory contents are transformed to LLVM constants, which are used as the initializers for the Click elements. These constant initialisers replace calls to C++ constructors and the accompanying runtime Click initialisation code. Click2LLVM also generates a wrapper function[1] for each Click element. This wrapper function continuously calls the packet processing code of the Click element on incoming packets, and is later used to form a VHDL module by AHIR. This new code is added to the LLVM Module which already has the Click element source code. The resulting IR module is then run through LLVM `opt`.

`opt` runs a set of optimising passes on the IR module. First, we use the `-internalize-public-api-file` parameter to pass `opt` a list of functions that need to be considered as the API, and thus preserved in the result. We list each of the generated wrapper function in that file. Now that 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 `simple_action()` function of the corresponding Click element inlined in the wrapper function. We also use standard passes Aggressive Dead Code Elimination, Dead Instruction Elimination, Lower atomic intrinsics, Lower invoke, Lower switch, Interprocedural Sparse Conditional Constant Propagation and Dead Argument Elimination. The optimised IR module is then run through AHIR, which transforms the LLVM IR into VHDL. The resulting VHDL, together with the static VHDL wrapper code (see below) can then be synthesized to a bitstream file that can be uploaded to the NetFPGA card and run.

We are currently extending the NetFPGA reference NIC design with our own module, and need to have compatible module interface. The data path for packets in Click differs significantly from that of the NetFPGA reference NIC design (pointers to whole packets vs. 64-bit chunks at a time) and there are differences in memory management in software and hardware. In this version of the prototype we have solved this with a wrapper (written in VHDL, see Figure 2) that sits between the AHIR-generated "Click VHDL" and the rest of the (existing) NetFPGA design. The wrapper transforms the 64-bit chunks to complete packets on input (the Input Module), and vice versa on output (the Output Module). It also handles the calls by Click elements to allocate memory (e.g. when creating new packets on the fly).

4. IMPLEMENTATION DETAILS

4.1 New Target for Click

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. 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 target for Click: NetFPGA. The modifications are implenented with similar precompiler instructions and often in the same functions or methods which already made a difference between Linux kernel and userspace implementations. One example of such modification is the creation of a new packet through the `Packet::make()` method. In Linux kernel an `skb` is referenced from `Class Packet`, while in userspace Click, memory is allocated dynamically from process memory space. With NetFPGA, we request a memory address for a block of pre-allocated BRAM via the function call `read_uintptr("free_queue")`[2].

[1]The wrapper functions are named `ahir_glue_xxx()`, where `xxx` is the name of the Click element

[2]This function is later recognized by AHIR and transformed to VHDL which reads a "pointer to an unsigned integer" from a FIFO named "free_queue".

While we have implemented a new target for Click by modifying some existing libraries and C++ classes, our goal has been to not require modifications to the existing Click elements or the guidelines for writing new Click elements. The goal is to let the programmer to concentrate on describing packet processing in C++ instead of thinking about the hardware target.

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 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 tool chain, the first two steps of this process are performed by our 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 the virtual methods `configure` and `initialize`.

Click2LLVM is essentially identical with the click userlevel 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 LLVM libraries.

After the Click router has been initialised, we link all necessary Click modules into a single LLVM module. By running the `TargetData` pass, we can later use the `StructLayout` API to find the memory locations for different fields of the Click elements. These memory locations represent the private variables of the C++ Click classes. We then iterate through each Click element of the Router, and write an LLVM Global Variable for each Click element into the LLVM Module.

With this approach, we can write out a single LLVM Module in 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 initialized elements, and the wrapper function for each element.

4.3 Optimisation Phase

The resulting LLVM Module constructed by the Click2LLVM tool is still unoptimised and contains constructs not suitable for generating hardware from, such as function pointers. The next phase is to run a set of transforming passes to the module.

First, we run a pass that replaces the `this` argument in the C++ originated methods with the global variable representing the Click element. After this, the method becomes constant and can be e.g. inlined later. Currently this approach brings a limitation to our tool, we can only have one instance of each Click element class. However, the limitation can be removed, either with the trivial approach of duplicating 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 will benefit optimizing C++ software in general, but the work on this is ongoing as of writing this report.

Next step is to run LLVM opt with the standard compile optimisations; this includes loop unrolling, argument promotion, dead code elimination and other well known optimisations. Then we run inlining with a more aggressive than default threshold to get as much as possible inlined in the wrapper functions. This helps reducing the number of resulting VHDL modules in the end.

4.4 Usage Example

To illustrate the usage of the toolchain, we will go through the steps leading from the Click configuration file until the resulting VHDL. We have selected a minimal configuration (see Program 1) with only a single actual packet processing element: `Minimal`. All the three elements in the configuration come from our own `minimal-package`, therefore the `require` declaration on the first line. We currently require the elements to be introduced and given names, which is done on the next three lines. The last line describes the flow of packets between the elements.

Program 1 test.click

```
require(package "minimal-package");

src :: FromFPGA;
min :: Minimal;
dst :: ToFPGA;

src -> min -> dst;
```

Program 2 Minimal::simple_action()

```
Packet *
Minimal::simple_action(Packet *p) {
  unsigned char *data =
      p->uniqueify()->data();
  data[0] ^= 0x0F;
  return p;
}
```

Elements `FromFPGA` and `ToFPGA` are special elements that interface the Click/NetFPGA wrapper. They both 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 (written in VHDL) needs no modifications if Click itself is updated. ToFPGA does the reverse, mapping Click-specific fields into NetFPGA control flags.

Program 2 shows the C++ source code for the packet processing code of Click element `Minimal`. `simple_action()` is part of the Click API, and is called for the packet if the element has defined it. In our `Minimal` element, we simply XOR the first byte of the `p->data` with `0x0F` and pass the packet further.

Program 3 Generated ahir_glue_min()

```
@1 = constant [8 x i8] c"min_in0\00"
@min = constant %struct.Minimal { ... }    ; Struct contents not shown here

define void @ahir_glue_min() {
  %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.Minimal* @min, i32 0, i32 0), i32 0, %struct.Packet* %1)
  ret void
}
```

Program 4 Optimized ahir_glue_min()

```
@1 = internal constant [8 x i8] c"min_in0\00"

@GV_3 = constant [1 x %"struct.Element::Port"] [%"struct.Element::Port" {
  [16 x i8] c"dst_in0\00\00\00\00\00\00\00\00", %struct.Element* undef, i32 undef }]

define void @ahir_glue_min() ssp {
  %tmp = tail call i32 @read_uintptr(i8* getelementptr inbounds ([8 x i8]* @1, i32 0, i32 0))
  %tmp3 = inttoptr i32 %tmp to %struct.Packet*
  %tmp4 = getelementptr inbounds %struct.Packet* %tmp3, i32 0, i32 3
  %tmp5 = load i8** %tmp4, align 4
  %tmp6 = load i8* %tmp5, align 1
  %tmp7 = xor i8 %tmp6, 15
  store i8 %tmp7, i8* %tmp5, align 1
  %tmp8 = icmp eq i32 %tmp, 0
  br i1 %tmp8, label %_ZN7Element4pushEiP6Packet.exit, label %bb.i

bb.i:
  tail call void @write_uintptr(i8* getelementptr inbounds (
    [1 x %"struct.Element::Port"]* @GV_3, i32 0, i32 0, i32 0, i32 0), i32 %tmp)
  ret void

_ZN7Element4pushEiP6Packet.exit:
  ret void
}
```

After running the `click2llvm` tool with the `test.click` configuration, we get function `ahir_glue_min()` generated in the resulting LLVM IR Module (see Program 3). First there is a call to `read_uintptr()` with pointer to constant `@1`. This is a blocking call, i.e. it returns when there is something written in queue `min_in0` first. Writing to this queue is done by the `FromFPGA` element, as described in the `test.click`. Next, the read pointer is cast to type `struct.Packet`, which represents the `Class Packet` of Click. Then a helper function `element_push()` is called with two arguments: a pointer to the Click element (`@min`) and the current packet (`%1`).

Then we run the optimizations to the LLVM Module, and we can see that the optimised version of `ahir_glue_min()` (see Program 4 is longer, but it has everything inlined. The only calls to external functions are `read_uintptr()` and `write_uintptr()`, which actually won't result in function calls in the resulting hardware, but AHIR will recognize them as I/O calls for this module, and treat them in a special way. They will be reads and writes to named FIFOs in the VHDL.

As we have constant arguments to `element_push()`, constant propagation and inlining passes have succesfully reduced the helper function away, leaving the contents of the original `simple_action()` inlined in this wrapper function. The core operation, XOR to a data byte, is visible in the LLVM representation. The final step to produce the VHDL is to run AHIR on this generated LLVM Module.

4.5 Resulting Hardware Architecture

An AHIR "system" consists of hardware modules that are "always on" — the module has an input control signal to process data. On completion, it sends an output control signal. The control signals may be accompained by arguments while additional data and results may be stored in the main memory. In addition, AHIR modules also support blocking I/O ports for directly exchanging data with each other or with other non-AHIR components in the system.

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 translate an input Click configuration to a set of AHIR modules where each module implements the behaviour of one Click element and the connections between

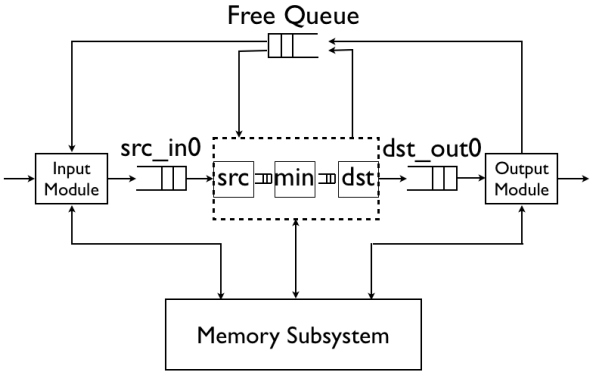

Figure 2: The Click/NetFPGA Wrapper

Click elements are mapped to I/O interaction between modules (see Figure 2).

We generate a separate wrapper function for each element in the Click configuration, whose overall effect is to map the element to a separate program that reads packets from input ports, process those packets and writes them to appropriate output ports.

The wrapper glues an AHIR I/O read operation to the each input port of the element. When a packet pointer is read from an input port, the wrapper calls `push()` method for that port, which in turn, executes the corresponding actions defined for that element. This behaviour terminates with a call to the `push()` method on an appropriate output port, which in fact triggers a corresponding AHIR I/O write operation.

Each "wrapped element" is compiled and optimised separately by our toolchain. The aggressive optimisation results in a single function that contains calls to AHIR I/O operations along with a highly optimised implementation of the original element behaviour. The AHIR toolchain processes each element separately to create a corresponding AHIR module.

I/O ports on the AHIR module are inferred from the port names used in the I/O function calls in the body of the wrapper. The port names are also used to automatically create I/O connections between the AHIR modules that correspond to the port connections in the original Click configuration. The result is a set of interacting AHIR modules that together implement 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. Our implementation reinterprets the packet as a fixed-size region of the available address-space. The address space is divided *a priori* into a set of pre-defined packet locations, where each location is mapped to a different memory bank to improve performance.

The packet locations are managed using queues. A global "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. Figure 2 depicts the resulting design: the Click elements are inside the dashed box in the middle; the rest is a static construct described in VHDL. This whole design is then used to replace the output port lookup module of the NetFPGA reference NIC design.

5. CONCLUSION

In this paper we have presented results from our ongoing work towards a toolchain that can transform Click routers written in C++ to a hardware description in VHDL, which can then be synthesized and run on a NetFPGA card as part of the Stanford reference NIC design.

By using the "initialize-freeze-dump" method, 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.

In the future we plan to evaluate the performance of the Click router on a NetFPGA card and exploring further possibilities of optimisations on early stages of the toolchain. We foresee that some of these generic optimisations are useful also in other domains, e.g. compiling C++ programs to be run on a CPU.

6. REFERENCES

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

[2] B. Chen and R. Morris. Flexible Control of Parallelism in a Multiprocessor PC Router. In Y. Park, editor, *USENIX Annual Technical Conference, General Track*, pages 333–346. USENIX, 2001.

[3] D. Chen, J. Cong, Y. Fan, G. Han, W. Jiang, and Z. Zhang. xPilot: A Platform-Based Behavioral Synthesis System. In *Proc. of SRC TechCon'05*, 2005.

[4] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.

[5] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. O. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The Superblock: An Effective Technique for VLIW and Superscalar Compilation. *Journal of Supercomputing*, 7(1,2):229–248, March 1993.

[6] P. Jääskeläinen, V. Guzma, A. Cilio, and J. Takala. Codesign Toolset for Application-Specific Instruction-Set Processors. In *Proc. of Multimedia on Mobile Devices 2007*, 2007.

[7] C. Kim, M. Caesar, and J. Rexford. Floodless in Seattle: a scalable ethernet architecture for large enterprises. In V. Bahl, D. Wetherall, S. Savage, and I. Stoica, editors, *SIGCOMM*, pages 3–14. ACM, 2008.

[8] E. Kohler. *The Click modular router*. PhD thesis, MIT, 2000.

[9] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297, 2000.

[10] C. Lattner and V. S. Adve. The LLVM Compiler Framework and Infrastructure Tutorial. In R. Eigenmann, Z. Li, and S. P. Midkiff, editors, *LCPC*, volume 3602 of *Lecture Notes in Computer Science*, pages 15–16. Springer, 2004.

[11] J. Lockwood, N. McKeown, G. Watson, G. Gibb, P. Hartke, J. Naous, R. Raghuraman, and J. Luo.

NetFPGA - An Open Platform for Gigabit-rate Network Switching and Routing. In *IEEE International Conference on Microelectronics Education*, June 2007.

[12] P. Nikander, B. Nyman, T. Rinta-aho, S. D. Sahasrabuddhe, and J. Kempf. Towards Software-defined Silicon: Experiences in Compiling Click to NetFPGA. 1st European NetFPGA Developers Workshop, Cambridge, UK, 2010.

[13] S. D. Sahasrabuddhe, S. Subramanian, K. P. Ghosh, K. Arya, and M. P. Desai. A C-to-RTL flow as an energy efficient alternative to embedded processors in digital systems. In *13th Euromicro Conference on Digital System Design*, September 2010.

[14] J. L. Tripp, M. Gokhale, and K. D. Peterson. Trident: From High-Level Language to Hardware Circuitry. *IEEE Computer*, 40(3):28–37, 2007.

[15] J. Zhang, Z. Zhang, S. Zhou, M. Tan, X. Liu, X. Cheng, and J. Cong. Bit-level optimization for high-level synthesis and FPGA-based acceleration. In P. Y. K. Cheung and J. Wawrzynek, editors, *FPGA*, pages 59–68. ACM, 2010.