Chimpp: A Click-based Programming and Simulation Environment for Reconfigurable Networking Hardware

Erik Rubow, Rick McGeer, Jeffrey C. Mogul, Amin Vahdat

HP Laboratories
HPL-2010-25

Keyword(s):
FPGAs, modular programming, Click

Abstract:

Reconfigurable network hardware makes it easier to experiment with and prototype high-speed networking systems. However, these devices are still relatively hard to program; for example, the NetFPGA requires users to develop in Verilog. Further, these devices are commonly designed to work with software on a host computer, requiring the co-development of these hardware and software components. We address this situation with Chimpp, a development environment for reconfigurable network hardware, modeled on the popular Click software modular router system. Chimpp employs a modular approach to designing hardware-based packet-processing systems, featuring a simple configuration language similar to that of Click. We demonstrate this development environment by targeting the NetFPGA platform. Chimpp can be combined with Click itself at the software layer for a highly modular, mixed hardware and software design framework. We also enable the integrated simulation of the hardware and software components of a network device together with other network devices using the OMNeT++ network imulator. In contrast to some prior work, Chimpp focuses on making experimentation easy, rather than on optimizing hardware performance. Chimpp also avoids unnecessary restrictions on communication patterns and design styles such as were imposed by prior approaches. We describe our design and implementation of Chimpp, and provide initial evaluations showing how Chimpp makes it easy to implement, simulate, and modify a variety of packet-processing systems on the NetFPGA platform.

External Posting Date: March 6, 2010 [Fulltext] Approved for External Publication
Internal Posting Date: March 6, 2010 [Fulltext]

Chimpp: A Click-based Programming and Simulation Environment for Reconfigurable Networking Hardware

Erik Rubow[+] erubow@cs.ucsd.edu
Rick McGeer[*] Rick.McGeer@hp.com
Jeffrey C. Mogul[*] Jeff.Mogul@hp.com
Amin Vahdat[+] vahdat@cs.ucsd.edu

[*]*HP Labs, Palo Alto, CA 94304* [+]*UC San Diego*

Abstract

Reconfigurable network hardware makes it easier to experiment with and prototype high-speed networking systems. However, these devices are still relatively hard to program; for example, the NetFPGA requires users to develop in Verilog. Further, these devices are commonly designed to work with software on a host computer, requiring the co-development of these hardware and software components.

We address this situation with Chimpp, a development environment for reconfigurable network hardware, modeled on the popular Click software modular router system. Chimpp employs a modular approach to designing hardware-based packet-processing systems, featuring a simple configuration language similar to that of Click. We demonstrate this development environment by targeting the NetFPGA platform. Chimpp can be combined with Click itself at the software layer for a highly modular, mixed hardware and software design framework. We also enable the integrated simulation of the hardware and software components of a network device together with other network devices using the OMNeT++ network simulator.

In contrast to some prior work, Chimpp focuses on making experimentation easy, rather than on optimizing hardware performance. Chimpp also avoids unnecessary restrictions on communication patterns and design styles such as were imposed by prior approaches.

We describe our design and implementation of Chimpp, and provide initial evaluations showing how Chimpp makes it easy to implement, simulate, and modify a variety of packet-processing systems on the NetFPGA platform.

1 Introduction

Reconfigurable network hardware, based on FPGAs, makes it much easier to experiment with and rapidly develop new ideas in networking, while achieving packet and data rates unsustainable with software-based prototypes. Designing with FPGAs is considerably easier than designing with ASICs, but unfortunately still more cumbersome than writing software, especially since the typical tools for designing with FPGAs are unfamiliar to many network experts.

The NetFPGA project [6] in particular has made major steps towards simplifying the development of FPGA-based network hardware, by providing an accessible network-specific reconfigurable hardware platform together with an open-source collection of useful designs, as well as an active developer community. However, the NetFPGA requires users to develop using Verilog, which is unfamiliar to most network developers and researchers, and which can be cumbersome to use. Furthermore, it can be difficult to reuse portions of contributed projects for integration into other designs.

We address this situation with Chimpp, a development environment for the NetFPGA modeled on the popular Click software modular router [2]. Starting with the Click approach gives us a well-understood, modular approach to designing packet-processing systems. Through finer-grained modularity of packet-processing elements, we simplify and encourage code reuse. By using a high-level Click-like configuration language, we decrease the effort associated with integration and extension, and at the same time make it possible to build a variety of interesting designs without the need to use Verilog. We call this "Chimpp" (Click for reconfigurable-Hardware Implementations of Modular Packet Processing).

It is often the case that packet processing devices are made up of of some combination of hardware and software components. We explore the combined use of Chimpp hardware elements and Click software elements. The Click portion can operate as both a control plane and as a slow path for handling uncommon traffic. Click elements can be associated with Chimpp elements to manage the operation and internal state of hardware elements. This approach results in highly modular and easily extensible designs at both the hardware and the software level. It also allows us to shift the boundary between hardware and software; for example, an element initially implemented in software can be re-implemented in hardware with a similar interface, if more speed is needed. Conversely, a hardware module can be shifted

to software if it becomes too complex, or if its performance proves to be non-critical.

We do not attempt to automatically convert Click elements into Chimpp elements, and we do not automatically generate element implementations, but rather we assume that Chimpp elements are either developed by hand or possibly generated by some other tool. Instead, we focus on providing a toolbox of useful elements with defined module interfaces that can be automatically instantiated and interconnected from a high-level description. We avoid any single standard interface but instead allow new interfaces and elements to be easily defined within any number of add-on packages.

We also provide support for mixed simulation of hardware and software within a simulated network. This can be particularly useful where a design consists of a single logical entity that spans the hardware/software boundary. When the co-development of hardware and software requires several changes to hardware, this kind of simulation can significantly decrease the development cycle by avoiding the time-consuming synthesis process. It also provides greater visibility into a system while undergoing dynamic and realistic hardware/software interactions and traffic patterns. For example, TCP connections and dynamic routing protocols can be simulated in this environment.

In this paper, we describe our design and implementation of Chimpp, as well as some initial evaluations that show how Chimpp makes it easy to implement, simulate, and modify NetFPGA-based packet-processing systems.

Our specific contributions include:

- Generation of a top-level Verilog file from a high-level script.
- Simple XML-based definitions of elements and their interfaces.
- A framework for designing systems with Click software elements and Chimpp hardware elements.
- Integration of combined hardware and software simulation within a network simulator.

2 Related Work

Several previous systems have provided high-level programming of reconfigurable network hardware, mostly using the Click model. We start with a brief summary of the essential features of Click, and then describe the other prior work.

2.1 Click

The Click Modular Router [2] is a widely-used software architecture for the flexible creation of software-based routers (and similar packet-processing systems). Click's modules are called *elements*, which can be connected into a graph. Elements have input and output ports, and connections between elements can either be *push*, in which earlier elements call later ones to move a packet through the graph, or *pull*, in which the calls are made by the consuming element. Connections between elements are typed, so that (for example) one cannot improperly connect a pull port to a push port. Also, certain kinds of ports can be connected only once. Queues are explicit elements that connect a pull subgraph to a push subgraph.

Over the years, a large library of Click elements have been built, allowing designers to re-use the work of others.

2.2 NP-Click

The NP-Click project [8] developed a version of Click tailored specifically for the Intel IXP1200 architecture, a popular network processor with six RISC cores and a StrongARM central core, and a multi-tiered memory architecture. NP-Click enabled efficient use of these resources in realizing a Click design. To that end, NP-Click used the classic Click element architecture. Elements were implemented in IXP-C, a somewhat restricted subset of the C language supported by the IXP architecture. In order to efficiently use the six "microengine" cores of the IXP, multiple elements could be instantiated on the IXP board.

NP-Click also addressed efficient use of the memory hierarchy. The IPX1200 has large, slow memories shared between microengines, and small, fast microengine-specific memories. NP-Click keywords allowed the programmer to annotate data to specify where a particular object could be stored. NP-Click supported programmer-controllable arbitration of shared resources between parallel elements. The NP-Click authors estimated a four-fold productivity increase using NP-Click, and demonstrated performance between 60% and 99% of native IXP-C coded designs.

2.3 Cliff

Cliff (Click for FPGAs) [3] was the first attempt to implement Click configurations directly on an FPGA base. As with NP-Click, Cliff made the Click element the center of the design. The heart of the Cliff approach was a standard communication protocol among elements. Each element implemented a memory interface, and an inter-element communication protocol consisting of two signalling wires per port. The wires implemented a three-way handshake between connected ports. Each Cliff element implemented a simple, user-extensible finite-state machine (FSM) with three states: waiting for input, ready to send, or processing a packet.

The Cliff system offered no simulation tools, and the Cliff driver simply instantiated and connected predefined Cliff elements. The predefined Cliff elements were responsible for implementing the protocols given above. Effectively, Cliff was a netlisting tool for predefined Verilog modules.

There were several notable features of Cliff de-

signs [7]. Data was passed between elements on a 344-bit-wide bus that obviated the need for memory storage of packet headers, though packet payloads were stored in memory. However, the full bus width had to be filled before packet processing could begin, resulting in a latency penalty of several tens of cycles. Also, only one Cliff element was active at any time in processing a given packet; this failure to exploit the parallelism available in hardware implementations significantly limited performance.

2.4 CUSP

Click Utilizing Speculation and Parallelism (CUSP) [7] extended the Cliff work, but with several changes. CUSP used a much narrower (16-bit) data bus, and provided explicit support for parallel and speculative execution of blocks of elements. It also provides some automatic generation of the communication between these blocks.

Parallelism in CUSP is entirely at the block level: the blocks in a design execute separately or in sequence, but within a block the elements operate in parallel as described above.

CUSP demonstrated a 2x performance improvement over Cliff in FPGA space, and came within 20% to a hand design, and processed packets at near line rate on two sample designs.

2.5 G

G [1] represents another recent attempt to realize high-level designs in a NetFPGA environment. A G description consists of a set of *rules* for packet rewriting and state manipulation. Each rule can be guarded (executed conditionally). As in CUSP, G descriptions are automatically clustered into optimized parallel microarchitectures; G uses automatic dependence analysis among rules to optimize scheduling.

2.6 NetThreads

NetThreads [4] takes a different approach to the problem. Rather than compiling to raw hardware (as Verilog), the NetThreads team took a two-step approach: building a number of "soft" processors from lookup tables (LUTs) on the FPGA, and then compiling a multithreaded program to run on the soft processors. The advantage of this approach over the use of hardware-based approaches is that C code can be compiled directly onto the NetFPGA platform, and run on essentially a standard processor. In order to make programming easy, NetThreads permits a static number of threads (a fixed number of threads/processor and a fixed number of processors/FPGA) and uses a static round-robin thread schedule.

In preliminary results [4], performance was not dramatically better than could be expected from Click running on a modern x86. The paper reported packet rates for three designs: a DHCP implementation, a packet classifier, and a NAT, respectively as 500, 1800, and 4600 packets/sec. (At 1500 bytes/packet, this works out to 6 Mb/sec – 59 Mb/sec.) We note that these were preliminary experiments, reported in a workshop paper, and that better performance may follow subsequent optimization.

2.7 Summary of related work

We summarize the work in this area in Table 1, attempting to highlight the various features of the work in this area. The table shows that previous work essentially split into two groups: NP-Click and NetThreads, focussed on ensuring that multithreaded Click code ran correctly on embedded processors; and G, CUSP and Cliff, focussed on realizing high-level Click designs efficiently in a hardware-only implementation. CUSP and Cliff both relied on pre-designed hardware blocks, and imposed a rigid protocol and design style on the library elements to ensure correctness and composability. The contribution of CUSP over Cliff was the insertion of automatically-generated communication blocks to permit parallelism and speculative execution. G has only been briefly and recently described, and its input format is sufficiently high-level that it is difficult to determine exactly the implementation details. However, it seems likely that it is effectively a layer sitting on top of a CUSP base.

Like Cliff and CUSP (and, probably, like G as well) Chimpp relies on a library of pre-designed hardware blocks, and focusses on integrating them into a design. Unlike Cliff and CUSP, Chimpp explicitly supports mixed hardware/software designs, since typical NetFPGA designs have some components that reside on the host. Further, Chimpp does not impose a rigid protocol on the hardware side of the design. Chimpp elements may communicate in any fashion they desire; all that is forbidden is cycles in the communication graph. In this sense, Chimpp's hardware synthesis may be regarded as a lower-level primitive than either Cliff's or CUSP's.

Indeed, we believe that it would be straightforward to re-implement either Cliff or CUSP to generate Chimpp designs. The advantages and disadvantages of Chimpp's approach are the usual ones of a lower-level vs. a higher-level framework. By not imposing a specific policy, a la Cliff and CUSP, we permit designers more freedom and potentially more efficient implementations. However, designers must think about the communication protocol they want between blocks, something Cliff and CUSP spares them.

2.8 A Brief Discussion of Verilog

Since this paper is targetted to a networking audience, it is likely that some readers will be unfamiliar with the Verilog language and what it is like to develop hardware using Verilog. We briefly describe it here. Readers interested in a fuller description should consult[9]. Readers who are familiar with Verilog may skip this section.

System	Approach	Hardware datapath	Simulation supported	Integration of HW/SW elements?
NetFPGA	User-written Verilog	Virtex-II FPGA	Verilog	No
NP-Click [8]	Compiling high-level code to multicore network processors	IXP1200 NP	N/A	N/A
CLIFF [3]	Composing Verilog blocks representing Click elements on NetFPGA	Fixed Protocol	Verilog	No
CUSP [7]	Composing VHDL blocks representing Click elements on NetFPGA, optimizing parallelism and speculation	Fixed Protocol	VHDL	No
G [1]	Compiling high-level rules into a NetFPGA hardware datapath	(unknown)	gsim, High-level simulation (not cycle accurate)	No
NetThreads [4]	Compiling multi-threaded C code onto soft processors implemented on NetFPGA	Mutithreaded processor cores	N/A	N/A
Chimpp	Composing Verilog blocks representing Click elements on NetFPGA together with software Click elements	Designer's choice	Omnet++ and Verilog: multi-mode simulation	Yes

Table 1: Feature Comparison of NetFPGA Programming Environments

The syntax of Verilog is reminiscent of C with extensive bit operations, but the semantics are very different, reflecting the semantics of the implementation domain. A hardware circuit is typically composed of acyclic graphs of logic gates (`AND, OR, NOT`, etc), called *combinational logic blocks*, separated by banks of stateholding elements called *flipflops*. The flipflops are governed by clocks; they change value only on the rising, or falling, edge of the clock. Conversely, logic gates change state whenever one of their inputs changes state.

Verilog models this behavior with two separate forms of assignment operation. The *continuous assignment* operation, denoted by the keyword `assign`, operates as a logic gate – whenever a value on the right-hand side of the assignment statement changes, the left-hand side is re-evaluated. This means that there is no flow of control in a Verilog program; every `assign` statement is conceptually evaluated in parallel, all the time.

The second assignment operation is *sequential assignment*. There are two forms, *blocking assignment*, denoted by the $=$ operator, and *non-blocking assignment*, denoted by the $<=$ operator. These statement both represent flipflops, and thus only appear when guarded by a clock, represented by the `always` guard, of the form:

```
always @(posedge clk) c <= d
```

which indicates that `c` is a flipflop, clocked on the rising edge of `clk`, input `d`. This statement is only executed when `clk` has a rising edge.

Similarly, `module`s in Verilog have a superficial similarity to C routines, but the semantics are very different. A `module` in Verilog represents a chunk of hardware. Its parameters are physical wires running into and out of the module. As with continuous assignment, all modules in Verilog conceptually execute in parallel.

Note that while Verilog has syntactic similarities with high-level software languages, it does not hide the messy details of hardware design from the developer. Synthesizable Verilog code must be designed with an idea of what the synthesized hardware will look like. Timing considerations are particularly important to be mindful of. The developer must consider, for example, at which clock cycle a value will become available, and how to manage the storage of other values in the meantime, particularly if pipelining is required to maintain throughput.

Such being the case, it is clear that being able to reuse Verilog code is very important.

3 Design and implementation

In this section, we describe the design and implementation of Chimpp.

3.1 Mapping Click concepts to hardware

The Click Modular Router is a software-based platform. While Chimpp aims to map the Click approach to hardware, we borrowed only those concepts that made sense in the context of hardware and which yielded practical benefit without unduly limiting the designer's flexibility.

The element is the most fundamental Click concept. In Click, elements are C++ objects, belonging to a particular element class. Click elements allow per-instance customization via configuration strings. This is similar to modules in Verilog, where parameters may be used for per-instance customization.

Click elements have some number of ports by which

they connect to other elements. These connections correspond to paths through which packets may flow. Packet flow is implemented in software as a virtual function call from one element to another, where one element passes a pointer to a packet as a parameter to a method invocation on another element. This is the natural, intuitive way to implement packet flow in software.

In hardware, there are many ways a designer might choose to have a packet flow from module to module, each with its advantages and disadvantages. In the case of the NetFPGA reference framework, the datapath for packet flow consists of a 64-bit data bus and 8-bit control bus, with upstream and downstream flow control signals. The entire contents of a packet flow through this bus, preceded by some per-packet information (referred to as "module headers") as indicated by the control bus.

However, one might prefer to use a different method, perhaps due to a different set of goals or constraints. For example, one might prefer to have a different bus width, or to extract the header fields for processing while storing the payload to memory, or to divide packets into fixed-sized cells. A common interface for packet flow between hardware elements is needed, yet there is no single "right" interface. Note that Click's push and pull concepts are specific to procedural implementation; in hardware with bidirectional flow control, there is no such distinction.

As a packet flows through Click elements, some per-packet information may be attached to it. These are called annotations, and provide a way for elements to communicate to downstream elements information associated with a packet, but external to the packet itself. A hardware implementation of an annotation would depend on the packet transfer mechanism. In the case of the NetFPGA reference framework, the so-called "module headers" that precede each packet are the equivalent of Click annotations.

Communication between Click elements is not restricted to these connections for packet flow. Elements may also export arbitrary method interfaces accessible by other elements. This communication is not represented by port connections. Elements may locate each other either explicitly by a name in the configuration string, or implicitly by proximity in the graph. The latter method is referred to as "flow-based router context". A hardware equivalent of these arbitrary method interfaces would be an arbitrary set of signals. We view flow-based router context as a way to work around the fact that method interface associations are not directly indicated in the language, and as it will be made clear later, we eliminate the need for it in our framework.

Handlers provide a way for other applications to interact with Click elements. A Click handler can be read from or written to through the Linux proc file system. Analogously, a hardware design may provide mechanisms for software interaction and control of hardware elements. In the case of the NetFPGA reference framework, this interaction takes the form of register reads and writes initiated by software on the host system.

The fact that Click configurations form a directed graph, with potentially complex interconnections between elements, can introduce certain complexities for hardware implementation. In software these connections between elements are implemented as a function call. A complex graph poses no particular difficulty for a software implementation – for example, when Click paths merge during push processing, nothing special needs to be done.

However, there is an important difference between software and hardware datapaths. A software datapath operates on abstractions such as function calls, objects, and pointers. A hardware datapath operates on data flowing between fixed physical modules over physical wires.

Complex graphs, as used in Click, therefore create problems for hardware-based designs. In particular, they require arbiters wherever paths merge. Also, in an FPGA-based design, a complex graph could make it more difficult for the development tools to do placement and routing.

Another difference arises from the fact that, while Click is single-threaded, hardware elements all operate simultaneously in parallel. Suppose we have multiple parallel paths which begin and end at the same element. In Click, as long as there is no switch between push and pull processing in between, packet order would be maintained. In hardware, the parallel paths can have different processing delays, so a packet could be forced to wait for another packet which arrived after it did. If a packet must pass through multiple arbiters during processing, then processing time could be highly variable.

In section 3.7 we discuss one design strategy that can be used to mitigate these problems.

3.2 The Chimpp framework

In this section, we describe the abstractions and framework of Chimpp, by which we generate a top-level Verilog file from an element library and a user-supplied script.

In consideration of the fact that there is no single "right" interface for packet flow in hardware and that it is desirable to allow other forms of communication between elements than packet flow, we chose not to require Chimpp elements to conform to any single standard interface. Instead, Chimpp provides a general method to specify connections between compatible element interfaces, defined in a library of Chimpp hardware elements. These connections include not only those by which packets flow from element to element, but also any sort of inter-element communication. This eliminates the need

for something like Click's flow-based router context, and also makes the design easier to understand, as there are no hidden element interactions. Furthermore, there are no required features that all elements must implement, and no restrictions on the structure of internal implementations.

The Click language provides an easy and concise way to instantiate and customize elements from an existing element library, and interconnect them to form a directed graph. The idea of instantiating and interconnecting elements, rather than writing sequential code, actually maps much more naturally to hardware than to software. We have created a simple language to concisely instantiate and interconnect Chimpp elements at a high level. A script written in this language, together with a library of existing elements, generates the top-level Verilog for a given design. This approach enables the construction of designs without the need to change any Verilog, provided that all required elements exist in the library. Even for those comfortable with coding in Verilog directly, this provides a valuable productivity benefit. Furthermore, the approach encourages the development of modular, reusable elements. Our language does not attempt to preserve the syntax of the Click language, particularly because it has different semantics; for example, unlike Click, Chimpp connections do not have any notion of direction.

3.3 Bus and element types

The Chimpp framework is built on the concepts of typed *buses*, *elements*, and *interfaces*.

A bus consists of a set of signals. A signal corresponds to a single "wire" declaration in Verilog, which may have a width of one or more. An element is implemented as a Verilog module, which may optionally contain submodules. Bus instances belong to named bus types, and element instances belong to named element types. Bus types specify a set of named signals, and list some set of named interface types that may connect to that bus type. Different interfaces may have different directions (input, output, or inout) for each signal in the bus. Element types specify a set of named interfaces, each belonging to a type specified by the corresponding bus type, along with a mapping between the local signal names and the bus signal names. Bus types carry with them an understanding of how the bus is to be used by the various elements which interface with them.

The specific operational semantics of a particular bus type are not enforced by this framework, but it is expected that element designers will take care that the operational semantics and assumptions about the meanings of signal values are consistent across elements that use a common bus. For example, if there is a signal that indicates that there is data to be read, or that a module is not yet ready to receive data, then individual elements are expected to respond appropriately.

3.4 Connection rules

A few simple rules are used to validate connections between elements (or rather, connections of element interfaces to a common bus instance):

- The interfaces must belong to the same bus type.
- Each signal must have exactly one "output" driver or one or more "inout" drivers.

Bus types could be used to prevent a connection between element interfaces which have different operational assumptions, even if the signals themselves are compatible. For example, it might not be desirable to allow an interface expecting IPv4 packets to be connected to an interface which emits IPv6 packets. It is not a requirement that every output signal have an input signal to drive. However, a warning is presented to the user if this is the case.

Any useful design must have some connection to the "outside world." In Click, these connections are embedded within certain elements. For example, the "ToDevice" and "FromDevice" elements send packets to, and receive packets from, a given network interface. In a Verilog design, the interface with the outside world consists of the interface of the top-level module. This interface is platform-specific. This "top-level" module could be the same top-level module used for synthesis, or it may be some submodule within the design which is intended to implement the core user logic. In any case, Chimpp represents this interface to the "outside world" as a special type of element which we refer to as an *environment*. An environment element is instantiated in the same way as other elements, and uses interfaces in the same way. However, it is treated differently during the Verilog generation process, as it provides a skeleton for the top-level file rather than a module instance within the file. Exactly one environment element must be present in a design.

3.5 Libraries and packages

Chimpp has no standard or built-in element or bus types. All types are library-defined, and it is easy to add new types. The library is structured as a collection of packages. A package contains a group of related bus and element definitions. For a particular design, a user must specify which packages are to be used. This provides some protection against naming conflicts, and also makes it easier to distribute and share library elements. Bus and element types are defined in simple XML files, which can be easily written and modified by hand. Figures 1 and 2 show examples of these XML formats. Element types also come with one or more Verilog files that implement the element. Our tool does not generate the Verilog that implements elements, only the Verilog that instantiates and interconnects them. However, these element Verilog files could be generated by another tool

```
<?xml version="1.0"?>
<bus name="nf2_pkts">
  <interface name="in">
    <signal name="data" type="input"  width="64"/>
    <signal name="ctrl" type="input"  width="8"/>
    <signal name="wr"   type="input"  width="1"/>
    <signal name="rdy"  type="output" width="1"/>
  </interface>
  <interface name="out">
    <signal name="data" type="output"/>
    <signal name="ctrl" type="output"/>
    <signal name="wr"   type="output"/>
    <signal name="rdy"  type="input"/>
  </interface>
  <interface name="spy">
    <signal name="data" type="input"/>
    <signal name="ctrl" type="input"/>
    <signal name="wr"   type="input"/>
    <signal name="rdy"  type="input"/>
  </interface>
</bus>
```

Figure 1: An example bus definition

```
<?xml version="1.0"?>
<element name="ipv4_decrement_ttl">
  <interface name="clk"      type="clock.in">
    <signal name="clk"/>
  </interface>
  <interface name="reset"    type="reset.in">
    <signal name="reset"/>
  </interface>
  <interface name="pkts_in"  type="nf2_pkts.in">
    <signal name="in_data"/>
    <signal name="in_ctrl"/>
    <signal name="in_wr"/>
    <signal name="in_rdy"/>
  </interface>
  <interface name="pkts_out" type="nf2_pkts.out">
    <signal name="out_data"/>
    <signal name="out_ctrl"/>
    <signal name="out_wr"/>
    <signal name="out_rdy"/>
  </interface>
</element>
```

Figure 2: An example element definition

from some high-level specification (such as G), and used together with hand-written elements in the same design.

3.6 Language syntax

We use a very simple language for specifying a particular design. There are three kinds of statements:

1. Those that specify which packages to use
2. Those that instantiate elements
3. Those that connect element interfaces to each other

A short example will quickly illustrate these statements. The example design in Fig. 3 contains three element instances: two of type some_elem and one of type debug_elem. The environment, instantiated with the name env, is of type some_environment. The only package used in this example is called some_pkg, so all bus and element types must be specified in that package. An element instantiation statement (as indicated by the "::" token) may or may not include a list of Verilog parameters enclosed in parentheses. The string enclosed in parentheses is not interpreted by this tool but is passed along to the generated Verilog instantiation.

```
// some comment
use some_pkg;

env   :: some_environment;
elem1 :: some_elem(.SOME_PARAM(1));
elem2 :: some_elem(.SOME_PARAM(2));
debug :: debug_elem(.STR("debug"));

env.clk <=> *.clk;
env.reset <=> *.reset;

env.data_in <=> elem1.in;
elem1.out <=> elem2.in <=> debug.spy;
elem2.out <=> env.data_out;
```

Figure 3: A simple Chimpp example

The most notable difference between this language and the Click language is that connections in the Click language are directed, whereas here they are not. The "<=>" token indicates that the indicated interfaces are connected to the same bus. Multiple such connections may be made with a single statement. For example, the line "elem1.out <=> elem2.in <=> debug.spy" indicates that interface *out* of element *elem1*, interface *in* of element *elem2*, and interface *spy* of element *debug* are all connected to the same bus. While their implementations are specified here, it could be that the debug element is there for the purpose of monitoring and perhaps printing the communications between elem1 and elem2 during simulation.

Another way to connect multiple interfaces together with a single statement is to use "*" as the element name, as in the line "env.clk <=> *.clk". This would cause all interfaces with the name clk to be connected to env.clk. Note that when we say there is no standard or built-in interface, we don't make any exception for clock and reset signals. Some designs may require multiple clocks, so it is up to the designer to specify these connections as well.

When the script in Fig. 3 is processed, a top-level Verilog file is generated. All necessary signals are automatically named and declared, and all Verilog modules are instantiated with the proper connections. Additionally, the implementing Verilog files for all elements used in the design are copied into a working directory.

Notice that this framework is not specific to networking. In fact, it is not domain-specific at all, and can therefore be used in a wide variety of applications. The domain-specific and platform-specific aspects of this framework are contained within the element libraries themselves. Nevertheless, we have obtained a concise and high-level way to instantiate and interconnect elements, as in Click.

3.7 Design of a NetFPGA Element Library

Here we discuss a few of our design decisions relating to the initial development of a library of packet processing elements specifically for the NetFPGA platform.

The NetFPGA reference framework was designed with the intent that all application-specific user logic would be contained within a single module, which is named user_data_path. Within this module, the user logic is presented with interfaces to packet queues, to a register bus, and to off-chip memory, hiding such details as DMA transfers. Changes outside the user_data_path module are usually unnecessary. We decided to operate within this module, adopting it as our environment of choice and taking advantage of the existing framework.

We used the same interface for packet transfer between elements as defined in the reference framework, consisting of a 64-bit data bus and an 8-bit control bus, with upstream and downstream flow control. Additional interfaces for inter-element communication include the existing register bus and memory interfaces, as well as others that were defined as needed. The flexibility provided by our tool in defining new bus and interface types has proven itself to be a useful and desirable feature.

In section 3.1 we discussed the problem of requiring arbiters for every path merge. To address this, we came up with an alternative to having strictly dedicated interfaces for packet flow, which is to flatten the packet flow graph, to have all packets pass through all packet-processing elements, and to mark each packet with an indication of whether a particular element should process it or forward it unchanged. For this purpose we defined a special packet annotation which we refer to as a "PTAG". The PTAG is a 16-bit field that precedes packets as part of the first annotation word (or "module header"). It takes the place of a field that was previously defined in NetFPGA as the word length of the packet. The word length is used by the output_queues module, but can safely be overwritten and later reconstructed from the byte length field.

Each element which obeys the PTAG convention has a PTAG associated with it, and only processes a packet if its PTAG matches the one contained in the packet annotation. When an element processes a packet, it can change the PTAG annotation to match that of the next element which should process it. These elements only require a single input interface and a single output interface for packet flow. The need for arbiters for every path merger goes away, and processing delays become more consistent.

A side effect of this approach is that only acyclic graphs can be represented in this way. We consider this a good thing, because cycles in the graph can introduce deadlock due to a flow control loop, unless special care is taken to avoid it. Note that this approach does not prevent the use of other dedicated interfaces. The PTAG method may be freely combined with the method of having dedicated interfaces. This may be desirable, for example, if certain packet streams are to receive higher priority treatment.

More attention will be given to the particular elements that we implemented in section 6.

4 Modular design with hardware and software elements

There are many functions of a network device that are best implemented in software rather than hardware, either because hardware implementation is infeasible, too costly, or yields a negligible performance benefit. Therefore software is often written in support of the hardware datapath. With modular and easily modifiable hardware designs, it is desirable for the supporting software to also be modular and easily modifiable. One way to achieve this goal is to employ Click itself as the control plane of a hardware-based data plane. Click can not only operate as a control plane for the hardware, but also as an extension of the data plane for traffic that may require sophisticated treatment.

There are two mechanisms that are provided by the existing NetFPGA kernel module and reference framework for communication between the host software and the NetFPGA hardware. First of all, the NetFPGA presents itself as a NIC to the operating system. Thus packets are sent to and from the NetFPGA in the same way that they would be sent to and from a NIC. Therefore, Click can easily be configured to send packets to and receive packets from the NetFPGA simply by using the appropriate interfaces.

The other communication mechanism is for reading and writing 32-bit user-defined registers within the NetFPGA. These registers can be embedded within individual hardware elements. Each hardware element which contains software-accessible registers must be allocated some section of the register address space. Click software elements can be written to monitor and manipulate the state of one or more hardware elements.

For example, a Click element could manipulate a routing table contained within a hardware element. A software element can be associated with a hardware element by providing the base register address of the hardware element in the configuration string. The individual registers within that element could then be accessed by using a known offset from the base address.

A useful practice would be to have special read-only registers in each hardware element indicating the type of element and a version number that software elements can check to avoid possible configuration or consistency problems.

This modular approach to mixed hardware and soft-

ware design can make it much easier to develop and modify these kinds of systems. Hardware and software element pairs can be co-designed to implement specific features, and their insertion into or removal from a larger system can be done easily. Furthermore, this approach can make the overall design easier to understand at a high level.

5 Simulation Environment

The current practice for testing a NetFPGA design is to first simulate the hardware in isolation using manually generated stimuli. Then, if those tests are successful, the design is synthesized and loaded onto a NetFPGA, and tested with independently written software in a real network. When things do not interoperate correctly in a running system, it can be very difficult to debug. Incremental hardware changes can result in a long development cycle when synthesis is required after each change. Furthermore, the network topology scenarios that can be tested are limited to those which are physically available to the developer.

To address this, and to aid in the development of modular hardware and software designs, we created an environment in which the hardware and software components of a design can be simulated together within a simulated network topology. The foundation for this environment is OMNeT++ [10], a popular discrete-event network simulator.

5.1 Integrating the hardware simulator

In order to integrate Verilog simulation into the network simulator, it is necessary that the Verilog simulator be interactive: that is, the inputs can be determined dynamically at run-time and the outputs can be reported to the network simulation kernel as they are generated. To satisfy this requirement, we used Verilator[1], an open-source Verilog-to-C++ compiler. Verilator compiles a synthesizable Verilog design into a C++ class. This class provides a simple way to manipulate the top-level inputs, allow the stimuli to propagate within the design, and read the outputs. Aside from the fact that this enables the interactivity that we require, it also happens to be very fast, and the fact that it can be linked directly into the network simulator is valuable for performance reasons.

We use Verilator to compile the user_data_path module, which contains the application-specific packet processing logic. A wrapper around the resulting class provides an interface for injecting packets into the MAC and CPU queues, injecting register reads and write requests, advancing the clock, and receiving any generated output packets or responses to the register requests. Once a packet is injected into the wrapper object, it is fed into the Verilator-generated object one 64-bit word at a time. The injection of a packet into the wrapper corresponds to the completed receipt of a full frame by the MAC or by DMA transfer. Packets are likewise received from the output queues 64 bits at a time, subject to external flow control (for example, due to a simulated bandwidth limitation corresponding to that of a PCI bus). Once a packet has fully been received from an output queue, the wrapper reports the event. Another wrapper around this wrapper turns it into an OMNeT++ simple module.

We had to extend the OMNeT++ scheduler to support this integration. NetFPGA modules register themselves with the scheduler during initialization. Then, before each event is removed from the event heap, the clocks of all registered NetFPGA modules are advanced until either the time of the next scheduled event is reached, or until a NetFPGA emits a packet or register access response. If the latter occurs, then all such events emitted during that cycle are converted into objects which can be handled by the simulator, and inserted into the event heap, after which the new next event is removed and processed as usual.

To shed light on the internal operation of a NetFPGA module during simulation, it may optionally be configured to dump a waveform to a specified file. Additionally, Verilog's $display system task is available. For example, this can be used to implement hardware elements similar to Click's Print element, effective within the simulation environment.

5.2 Integration with simulated software

After integrating the Verilog simulator into the network simulator, our next step was to integrate the software component as well. We explored two alternative approaches to this integration.

The first approach, in line with our goal of supporting designs with modular hardware and software elements, consisted of integrating Click into OMNeT++. Because Click had previously been integrated into the ns-2 simulator[2], we were able to harness the existing nsclick interface, inserting the appropriate hooks and writing an appropriate wrapper to create an OMNeT++ simple module. A compound module could then be easily created by combining a NetFPGA module with a Click module. Bandwidth and delay characteristics of hardware/software communication can be represented by setting OMNeT++'s datarate and delay parameters on the connections between Click elements and hardware modules. This setup is enough to simulate packet flows spanning hardware and software. To add NetFPGA register access support to Click within the simulation environment, however, some special attention is needed. Special attention would also be needed to support elements which use user-level sockets.

Before discussing these issues for Click simulator integration, we will discuss an alternative framework for

[1] `http://www.veripool.org/wiki/verilator`

[2] `http://www.isi.edu/nsnam/ns/`

integrating software into a network simulation. For some developers or applications, Click might not be the preferred or ideal way to implement a software control plane. Therefore, we considered a more general approach: simulating a single-threaded program implemented as a C++ object.

OMNeT++ modules which represent user-level applications are typically created for the purpose of generating synthetic network traffic, with characteristics that resemble real applications. In contrast, we would like to execute real application code with real data in a simulated environment.

There are a number of challenges associated with trying to run real applications within a network simulator such as OMNeT++; Some of these are discussed in [5]. One is the simple problem of control flow and program structure. OMNeT++ uses a callback function, handleMessage(), to deliver messages to simple modules. However, real user-level applications are built on system calls, some of which may block until an event of interest occurs. If there is more than one potentially blocking system call within a given application, then this means that there should be multiple entry and exit points at which the control flow transfers between the application module and the rest of the simulation environment. When control is restored to the application module, the stack should be in the same state in which it was before the blocking call was made. The callback mechanism of OMNeT++ does not provide these semantics, and modifying an application to fit this structure could require significant code refactoring.

There is, however, a feature of OMNeT++ that can provide these semantics. Simple modules have the option either to use the handleMessage() callback function or to run as a coroutine in an infinite loop. In the latter case, modules export an activity() function, which is the entry point at the beginning of the simulation. At any point, modules can call a receive(), function which yields control to the simulation environment until the next message for that module is received, at which point the receive() function returns a pointer to the message. Each module which uses activity()/receive() is allocated its own stack, which enables this kind of behavior. All context switches between coroutines are cooperative, and only one executes at any given point in time.

The activity()/receive() approach is discouraged by OMNeT++ developers on the grounds that it requires more memory (for the stack) and that it can make for sloppy code in the hands of careless developers. However, it serves our goals quite well, and since we are interested in smaller-scale simulations, the memory requirements are not a significant concern.

We can use the activity()/receive() approach to simulate an I/O-bound process with potentially blocking system calls. We assume the process is I/O-bound and that CPU processing time is negligible because, to the process, simulation time only advances during a receive() call. We consider this to be a reasonable assumption to make. If a more accurate timing model is desired, we could actually measure the elapsed time between blocking calls and perform a simulated sleep for that amount of time. All receive() calls are made through a single wrapper function which is prepared to process any kind of message, and which buffers events appropriately. For a given blocking call, this wrapper function is called repeatedly until the desired condition is met: for example, some receive buffer is not empty, or the response to a NetFPGA register read has been received, or a timer has expired.

We have used this framework to enable real applications to operate in the simulated environment with very little special treatment in the source code. In particular, we can use the same source code for both implementation and simulation, requiring only recompilation.

Because of incompatibilities of OMNeT++ and multithreading, we limit our attention to single-threaded applications. To facilitate simulator integration, we implement the application as a C++ class. Applications can use the standard sockets API for network communication and a user-level NetFPGA library for register access. When compiled for use in a real environment, these functions operate unhindered. When compiled for simulation, these function calls are intercepted (by declaring class member functions with the same interface) and routed to the the associated application module in the simulated environment to handle. This enables protocols such as TCP to operate in simulation time rather than real time, since simulator implementations of TCP would use simulator time for things like round trip time and timeouts.

Now that we have discussed this in detail, I will briefly explain what is needed to provide register access or socket support to Click while in simulation. Basically, the same approach is needed because these are calls made within Click which must return control to the simulation environment. In the case of a register read, some small amount of time must pass for the NetFPGA to process the request and return the value. This requires that we use the activity() approach to enable this feature.

6 Experimental evaluation

In order to evaluate the effectiveness of our framework, and to begin development of a useful element library, we experimented with a few example designs. In particular, we attempt to validate several specific claims made in this paper:

1. Chimpp makes it easy to create and modify NetFPGA designs.
2. Chimpp hardware elements can be easily integrated

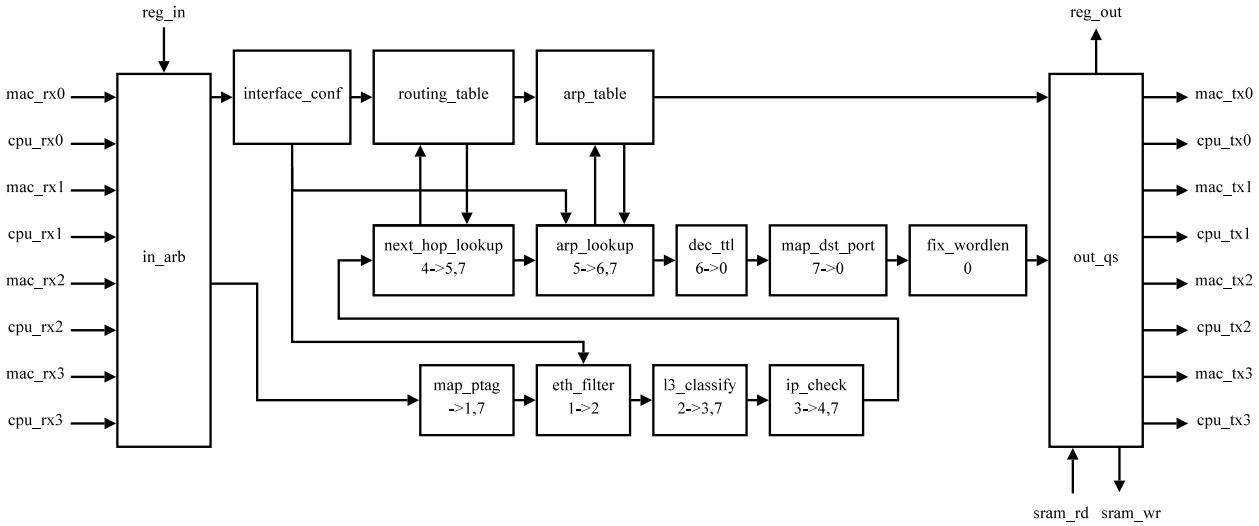

Figure 4: A Simple IPv4 Router Example

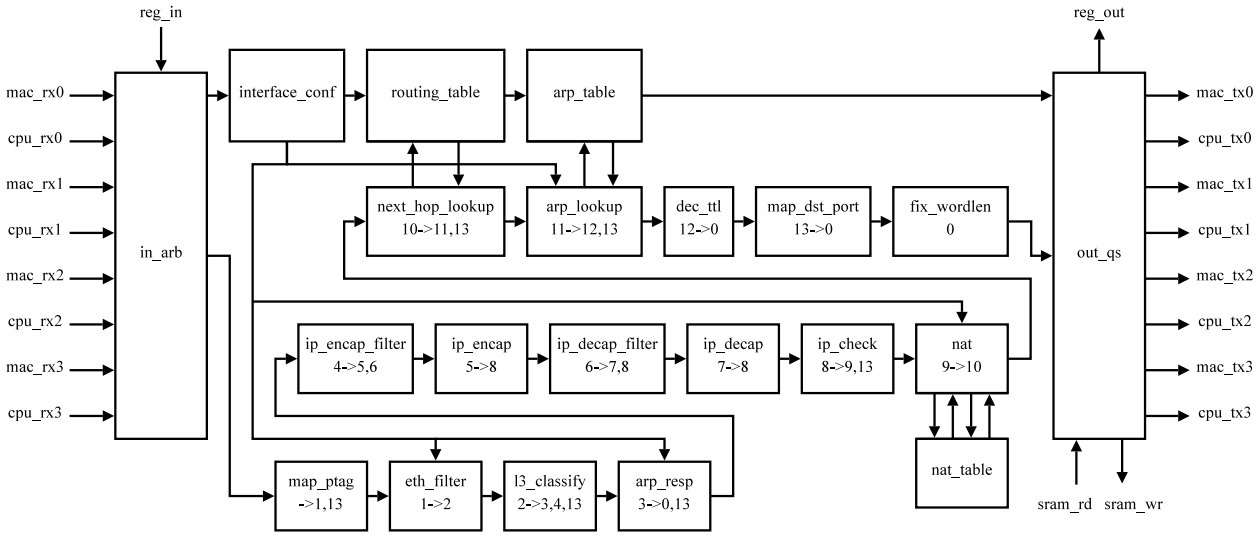

Figure 5: A Router enhanced with NAT, IP-in-IP encapsulation, and hardware-generated ARP responses

with Click software elements in the same system.

3. We can simulate these mixed hardware-software designs within a simulated network.
4. Performance in terms of packet-processing rate is competitive and FPGA resource consumption is comparable.

We developed several designs as part of our evaluation:

- **An IPv4 router**: to compare against the standard NetFPGA implementation.
- **Hardware-based ARP support**: as a modification of the IPv4 router.
- **NAT support**: also as a modification of the IPv4 router.
- **IP-in-IP encapsulation and decapsulation**: also as a modification of the IPv4 router.

6.1 IPv4 router design

Figure 4 provides a graphical depiction of the hardware portion of the simple IPv4 router composed of Chimpp elements. Several different bus types are represented by the lines interconnecting elements. The direction of the arrows reflects the direction of information flow across these buses. The directions are depicted for clarity; again, they are not represented in the language. The elements in the main packet pipeline have a notation indicating their behavior with respect to the PTAG convention discussed in Section 3. For example, the notation "2->3,7" indicates that the element pro-

cesses packets with PTAG equal to 2, and may then modify that PTAG to a value of 3 or 7. The elements along the top, which include "in_arb" (input arbiter), "interface_conf", "routing_table", "arp_table", and "out_qs" (output queues), each have a register interface. This allows the software to perform operations such as configuring the MAC and IP addresses of each interface and to insert routing table and ARP cache entries.

The routing table and ARP table are implemented as separate elements from those that perform the lookup and data transformation logic. The motivation for this separation was so that other tables could be easily plugged in. For example, it may be desired to use a different table lookup algorithm or structure, or to store the table in off-chip memory.

The "map_ptag" element sets the PTAG according to the source port, and the "map_dst_port" element sets the destination output port according to the source port. The purpose of both these elements is to send packets which originate from the host CPU directly out on the right network interface, and to send certain packets up to to host CPU (for example, non-IP packets or a lookup miss). The input arbiter and output queues elements were adopted from the NetFPGA reference library unmodified. All other elements were written from scratch.

This simple router design has essentially the same functionality as the NetFPGA reference router. However, the structure of it is very different. The reference router encapsulates all of the functionality between the input arbiter and the output queues into a single module called "output_port_lookup". It implements layer 2 and layer 3 processing in a very tightly coupled way. The operations of filtering Ethernet addresses, validating the IP header, performing the next hop lookup and ARP lookup, decrementing the TTL, recomputing the checksum, and managing packet flow to and from the host CPU are all performed within this module, coordinated by a single "mastermind" module. The advantage to this approach is that many of these things can be done in parallel. The disadvantage is that it is very difficult to add new functionality to this design, or to extract portions of the design for reuse elsewhere. In contrast, our design decouples these functions into finer-grained elements. Each element has a simple and well-defined operation to perform, and the task of determining the correct output port is distributed across these elements.

6.2 Quantitative comparisons

We can make a few quantitative comparisons between our Chimpp router and the NetFPGA reference router. It should be noted that during the development of the Chimpp router, there was no attempt to optimize any of these metrics, save that we required line rate operation.

1. Lines of Verilog: The only difference in Verilog code between the two router designs is between the input arbiter and the output queues. To implement this portion, the Chimpp router used 2,093 lines of Verilog, and the reference router used 3,096 lines of Verilog. Neither of these figures include the the user_data_path module itself.
2. FPGA Resources: The reference router uses 46% of the slices on the FPGA (a slice is a basic unit of reconfigurable logic) and 11% of the block RAMs. The Chimpp router uses 50% of the slices and 12% of the block RAMs. The higher slice utilization appears to be due to the use of more distributed RAM, though there are fewer slices used as logic.
3. Processing Latency: The processing latency through the main pipeline (the section between the input arbiter and the output queues) for the reference router is 18 cycles, or 144 ns. The processing latency for the Chimpp router is 26 cycles, or 208 ns. This is due to the fact that the reference router performs certain operations in parallel. The lower latency makes little difference, however; as the NetFPGA framework does not allow cut-through operation, large frames incur a store-and-forward delay of over 1500 cycles. The input arbiter introduces its own latencies due to contention, and the biggest source of latency overall is likely to be the output queues, even if buffers are relatively small.
4. Throughput: Both routers operate at line rate for all packet sizes.

6.3 Integration with Click

The Click portion of our router design essentially consists of a basic router with an overall structure which is similar to the hardware datapath. It looks like a plain Click router, but contains elements which are modified to reflect their state in their corresponding hardware elements. We modified (subclassed) two existing Click elements and created one new element. The new element, called NF2AddrInit, takes as parameters the register block tag of the interface_conf element and the MAC and IP addresses of those elements. This element simply initializes the appropriate registers when Click loads. The other two elements, NF2ARPQuerier and NF2LinearIPLookup are equivalent to their superclasses ARPQuerier and LinearIPLookup except that they push their ARP tables and routing tables into their associated hardware elements. This can be thought of as a hardware-accelerated Click router. We grouped these NetFPGA-specific elements into a separate Click package. By packaging up these specialized Click elements separately, they can easily be distributed along with packages of Chimpp hardware elements.

6.4 Adding features to the router

We will now discuss our experiments in extending the functionality of the example router.

ARP Responder: The ARP responder is simple: it consumes ARP requests, and when the request is for the configured IP address of the interface on which the packet was received, it generates an ARP response. This can easily be inserted after the l3_classify element, which should be configured to set the appropriate PTAG for ARP requests. Due to the relative infrequency of ARP requests, this will not yield a significant performance impact. However, it does demonstrate the ease with which the router can be extended.

This is an example of a feature which was already present in the Click configuration but which was pushed down into the hardware. Figure 5 depicts a modified version of the router which contains this ARP responder (labeled arp_resp), as well as other additional elements which we will now discuss.

Network Address Translation: The next enhancement to the IPv4 router that we implemented was a Network Address (Port) Translation element. This NAT implementation was implemented purely in hardware using simple mechanisms.

Outgoing packets are identified by having a private source IP address and a public destination IP address. For these packets, the source IP address and TCP/UDP port number are hashed to a 15-bit value which is used to replace the source port (We only use the upper half of the port space for NAT). The source IP address is replaced with the IP address of the external NetFPGA interface (for simplicity we assumed only one external interface). When the hash is performed, a reverse mapping of 15-bit port number to the internal IP address and port is installed in a directly indexed table.

Incoming packets destined to a port number for which such an entry exists will have their destination IP address and port number replaced with these internal entries.

The advantages to this implementation are that it is fast and that the table is small enough to fit in on-chip memory. The disadvantages are that external port collisions can occur, and it does not currently validate the source IP address for incoming traffic (this would be easy to do but would require more storage). Once addresses have been replaced, the packet can be routed normally, so this element should be placed before the next_hop_lookup element. However, this demonstrates the point that we were able to add this functionality simply by inserting an element into the design.

IP-in-IP encapsulation: The final enhancement to the IPv4 router that we have made thus far is simple IP-in-IP encapsulation and decapsulation. The desired behavior here is to conditionally insert a basic IP header with some configured source and destination IP address. At the terminating node of this IP tunnel, the header should be removed.

The decision about whether to apply encapsulation or decapsulation of a packet is currently made statically by generic classifiers (labeled ip_encap_filter and ip_decap_filter). These classifiers are provided with sets of bitmasks and values to match on for one or more of the first eight words of a packet. In this way, any combination of arbitrary field bits in the first 64 bytes of a packet may be matched against specific values. In the case of the encapsulator filter, it checked for IPv4 packets with a particular destination IP prefix. In the case of the decapsulator filter, it checked for IPv4 packets with a protocol field value of 4 (indicating IP-in-IP) and a specific destination IP address.

These generic classifiers are fairly flexible, and the implementation is trivial. Ideally, however, we would like these encapsulation and decapsulation elements to be configurable at runtime (for example, to correspond with a configured interface IP address). One possible way to achieve this would be to use dynamic generic classifiers, where the bitmasks and values to match against are contained in software-accessible registers.

During the development of all of these features, we used Verilator for individual unit tests and the mixed simulation environment based on OMNeT++ for multi-element simulations, before attempting to synthesize a design. The simulation environment proved to be quite useful, and the synthesized designs generally operated exactly as expected. The ability to simulate arbitrary C++ programs had other uses aside from acting as a control plane for the NetFPGA. For example we could write a program to transfer a file from one virtual node to another through the NetFPGA and write it to a file. We could do the same for multiple node instances, and verify that the file was transmitted correctly.

7 Summary and conclusions

In this paper, we presented Chimpp, a development environment for reconfigurable network hardware, and its implementation for NetFPGA. Chimpp differs from previous attempts in that it permits a wide variety of communication patterns and bus structures in the implemented design, handles communications with a Click software layer, and, as part of the Chimpp environment, incorporates common simulation tools. We demonstrated a sample design with Chimpp, the IPv4 router, and showed it had minor overhead in LUT count, latency, and no penalty in throughput compared to the hand-coded router design , and 33% less Verilog code. We were able to demonstrate easily adding enhancements to the router: Network Address Translation, ARP response, and IP-in-IP encapsulation.

Chimpp is primarily designed to be a vehicle for network researchers wishing to use the NetFPGA platform for experimentation. We believe that the value of an environment like Chimpp is determined by its user community. In particular, essential to Chimpp's success is

a broad range of pre-written modules to permit network experimenters to rapidly build novel and interesting designs on the NetFPGA platform. Therefore, we will release Chimpp and our library of Chimpp hardware modules on the NetFPGA website, and encourage network researchers and NetFPGA users to use the environment and contribute back any modules they have designed for use with Chimpp.

References

[1] M. Attig and G. Brebner. High-level programming of the FPGA on NetFPGA. In *Proc. NetFPGA Developers' Workshop*, August 2009.

[2] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *TOCS*, 18(3):263–297, 2000.

[3] C. Kulkarni, G. Brebner, and G. Schelle. Mapping a Domain Specific Language to a Platform FPGA. In *Proc. Design Automation Conference*, pages 924–927, June 2004.

[4] M. Labrecque, J. G. Steffan, G. Salmon, M. Ghobadi, and Y. Ganjali. NetThreads: Programming NetFPGA with Threaded Software. In *Proc. NetFPGA Developers' Workshop*, August 2009.

[5] C. P. Mayer, T. Gamer, and C. P. Mayer. Integrating real world applications into omnet++, 2008.

[6] J. Naous, G. Gibb, S. Bolouki, and N. McKeown. A Programming Model for Reusable Hardware in NetFPGA. In *Proc. PRESTO*, Aug. 2008.

[7] G. Schelle and D. Grunwald. CUSP: A Modular Framework for High Speed Network Applications on FPGAs. In *FPGA '05*, pages 246–257, February 2005.

[8] N. Shah, W. Plishker, and K. Keutzer. NP-Click: A Programming Model for the Intel IXP1200. In *Proc. 2nd Workshop on Network Processors*, February 2003.

[9] D. E. Thomas and P. R. Moorby. *The Verilog hardware description language*. Kluwer Academic Publishers, 101 Philip Drive, Assinippi Park, Norwell, MA, 02061, fifth edition, 2002.

[10] A. Varga. The OMNeT++ discrete event simulation system. In *Proc. European Simulation Multiconference*, pages 319–324, 2001.