NetFPGA: Reusable Router Architecture for Experimental Research

Jad Naous
Stanford University
California, USA
jnaous@stanford.edu

Glen Gibb
Stanford University
California, USA
grg@stanford.edu

Sara Bolouki
Stanford University
California, USA
sbolouki@stanford.edu

Nick McKeown
Stanford University
California, USA
nickm@stanford.edu

ABSTRACT

Our goal is to enable fast prototyping of networking hardware (e.g. modified Ethernet switches and IP routers) for teaching and research. To this end, we built and made available the NetFPGA platform. Starting from open-source reference designs, students and researchers create their designs in Verilog, and then download them to the NetFPGA board where they can process packets at line-rate for 4-ports of 1GE. The board is becoming widely used for teaching and research, and so it has become important to make it easy to re-use modules and designs. We have created a standard interface between modules, making it easier to plug modules together in pipelines, and to create new re-usable designs. In this paper we describe our modular design, and how we have used it to build several systems, including our IP router reference design and some extensions to it.

Categories and Subject Descriptors

B.6.1 [**Logic Design**]: Design Styles—*sequential circuits, parallel circuits*; C.2.5 [**Computer-Communication Networks**]: Local and Wide-Area Networks—*ethernet, high-speed, internet*; C.2.6 [**Computer-Communication Networks**]: Internetworking—*routers*

General Terms

Design

Keywords

NetFPGA, modular design, reuse

1. INTRODUCTION

The benefits of re-use are well understood: It allows developers to quickly build on the work of others, reduces time to market, and increases the scrutiny (and therefore the quality) of code. Software re-use is widely practiced and wildly successful, particularly in the open-source community; as well as in corporate development practices and commercial tools.

The key to software re-use is to create a good API—an interface that is intuitive and simple to use, and useful to a large number of developers. Indeed, the whole field of networking is built on the re-use of layers interconnected by well-documented and well-designed interfaces and APIs.

On the other hand, the history of re-use in hardware design is mixed; there are no hugely successful open-source hardware projects analogous to Linux, mostly because complicated designs only recently started to fit on FPGAs. Still, this might seem surprising as there are very strong incentives to re-use Verilog code (or other HDLs)—the cost of developing Verilog is much higher (per line of code) than for software development languages, and the importance of correctly verifying the code is much higher, as even small bugs can cost millions of dollars to fix. Indeed, companies who build ASICs place great importance on building reusable blocks or macros. And some companies exist to produce and sell expensive IP (intellectual property) blocks for use by others. To date, the successes with open-source re-usable hardware have been smaller, with Opencores.org being the most well-known.

Re-using hardware is difficult because of the dependencies of the particular design it is part of; e.g. clock speed, I/Os, and so on. Unlike software projects, there is no underlying unifying operating system to provide a common platform for all contributed code.

Our goal is to make *networking* hardware design more re-usable for teachers, students and researchers—particularly on the low-cost NetFPGA platform. We have created a simple modular design methodology that allows networking hardware designers to write re-usable code. We are creating a library of modules that can be strung together in different ways to create new systems.

NetFPGA is a sandbox for networking hardware—it allows students and researchers to experiment with new ways to process packets at line-rate. For example, a student in a class might create a 4-port Gigabit Ethernet switch; or a researcher might add a novel feature to a 4-port Gigabit IP router. Packets can be processed in arbitrary ways, under the control of the user. NetFPGA uses an industry-

PRESTO'08, August 22, 2008, Seattle, Washington, USA.

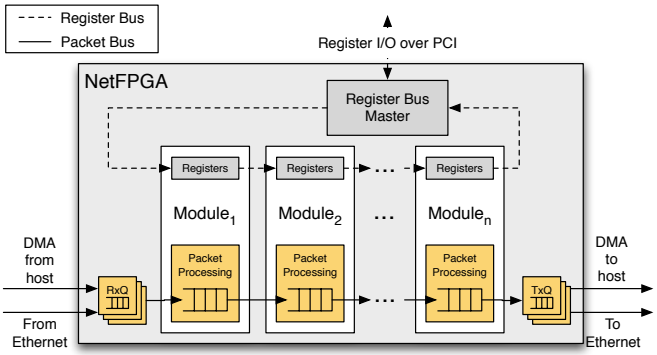

Figure 1: Stages in the modular pipeline are connected using two buses: the Packet Bus and the Register Bus.

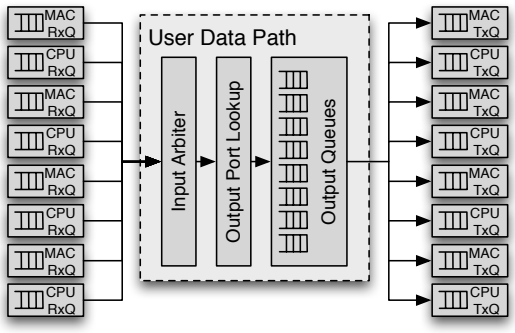

Figure 2: The IPv4 Router is built using the "Reference Pipeline" - a simple canonical five stage pipeline that can be applied to a variety of networking hardware.

standard design flow (users write program in Verilog, synthesize them, and then download to programmable hardware). Designs typically run at line-rate, allowing experimental deployment in real networks. A number of classes are taught using NetFPGA, and it is used by a growing number of networking researchers.

NetFPGA is a PCI card that plugs into a standard PC. The card contains an FPGA, four 1GigE ports and some buffer memory (SRAM and DRAM). The board is very low-cost[1] and software, gateware and courseware are freely available at http://NetFPGA.org. For more details, see [9].

Reusable modules require a well-defined and documented API. It has to be flexible enough to be usable on a wide variety of modules, as well as simple enough to allow both novice and experienced hardware designers to learn it in a short amount of time.

Our Approach — like many that have gone before — exploits the fact that networking hardware is generally arranged as a pipeline through which packets flow and are processed at each stage. This suggests an API that carries packets from one stage to the next along with the information needed to process the packet or results from a previous stage. Our interface does exactly that — and in some ways resembles the simple processing pipeline of Click [6] which allows a user to connect modules using a generic interface. One difference is that we only use a push interface, as opposed to both push and pull.

NetFPGA modules are connected as a sequence of stages in a pipeline. Stages communicate using a simple packet-based synchronous FIFO push interface: Stage $i+1$ tells Stage i that it has space for a packet word (i.e. the FIFO is not full); Stage i writes a packet word into Stage $i+1$. Since processing results and other information at one stage are usually needed at a subsequent stage, Stage i can prepend any information it wants to convey as a word at the beginning of a packet.

Figure 1 shows the high-level modular architecture. Figure 2 shows the pipeline of a simple IPv4 router built this way.

Our Goal is to enable a wide variety of users to create new systems using NetFPGA. Less experienced users will reuse entire pre-built projects on the NetFPGA card. Some others will modify pre-built projects and add new functionality to them by inserting new modules between the available modules. Others will design completely new projects without using any pre-built modules. This paper explains how NetFPGA enables the second group of users to reuse modules built by others, and to create new modules in a short time.

The paper is organized as follows: Section 2 gives the details of the communication channels in the pipeline, Section 3 describes the reference IPv4 router and other extensions, Section 4 discusses limitations of the NetFPGA approach, and Section 5 concludes the paper.

2. PIPELINE INTERFACE DETAILS

Figure 1 shows the NetFPGA pipeline that is entirely on the Virtex FPGA. Stages are interconnected using two point-to-point buses: the *packet bus* and the *register bus*.

The packet bus transfers packets from one stage to the next using a synchronous FIFO packet-based push interface, over a 64-bit wide bus running at 125MHz (an aggregate rate of 8Gbps). The FIFO interface has the advantage of hiding all the internals of the module behind a few signals and allows modules to be concatenated in any order. It is arguably the simplest interface that can be used to pass information and provide flow control while still being sufficiently efficient to run designs at full line rate.

The register bus provides another channel of communication that does not consume Packet Bus bandwidth. It allows information to travel in both directions through the pipeline, but has a much lower bandwidth.

2.1 Entry and Exit Points

Packets enter and exit the pipeline through various Receive and Transmit Queue modules respectively. These connect the various I/O ports to the pipeline and translate from the diverse peripheral interfaces to the unified pipeline FIFO interface. This makes it simpler for designers to connect to different I/O ports without having to learn how to use each.

Currently there are two types of I/O ports implemented with a third planned. These are: the *Ethernet Rx/Tx queues*, which send and receive packets via GigE ports, the *CPU DMA Rx/Tx queues*, which transfer packets via DMA between the NetFPGA and the host CPU, and the *Multi-gigabit serial Rx/Tx queues* (to be added) to allow transfer-

[1]At the time of writing, boards are available for $500 for research and teaching.

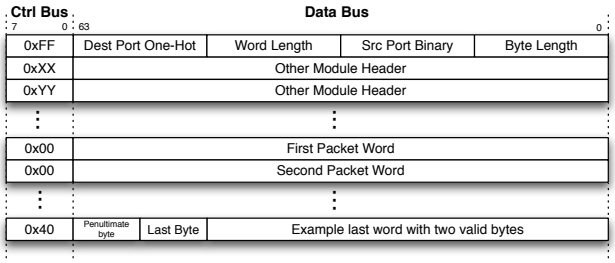

Figure 3: Format of a packet passing on the packet bus.

ring packets over two 2.5Gbps serial links. The multi-gigabit serial links allow extending the NetFPGA by, for example, connecting it to another NetFPGA to implement an 8-port switch or a ring of NetFPGAs.

2.2 Packet Bus

To keep stages simple, the interface is packet-based. When Stage i sends a packet to Stage $i+1$, it will send the entire packet without being interleaved with another. Modules are not required to process multiple packets at a time, and they are not required to split the packet header from its data — although a module is free to choose to do so internally. We have found that the simple packet-based interface makes it easier to reason about the performance of the entire pipeline.

The packet bus consists of a `ctrl` bus and a `data` bus along with a `write` signal to indicate that the buses are carrying valid information. A `ready` signal from Stage $i+1$ to Stage i provides flow control using backpressure. Stage $i+1$ asserts the `ready` signal indicating it can accept at least two more words of data, and deasserts it when it can accept only one more word or less. Stage i sets the `ctrl` and `data` buses, and asserts the `write` signal to write a word to Stage $i+1$.

Packets on the packet bus have the format shown in Figure 3. As the packet passes from one stage to the next, a stage can modify the packet itself and/or parse the packet to obtain information that is needed by a later stage for additional processing on the packet.

This extracted information is prepended to the beginning of the packet as a 64-bit word which we call a *module header* and is uniquely identified by its `ctrl` word from other module headers. Subsequent stages in the pipeline can identify this module header from its `ctrl` word and use the header to do additional processing on the packet.

While prepending module headers onto the packet and passing processing results in-band consumes bandwidth, the bus's 8Gbps bandwidth leaves 3Gbps to be consumed by module headers (4Gbps used by Ethernet packets and 1Gbps used by packets to/from the host). This translates to more than 64 bytes available for module headers per packet in the worst case. Compared with sending processing results over a separate bus, sending them in-band simplifies the state machines responsible for communicating between stages and leaves less room for assumptions and mistakes in the relative timing of packet data and processing results.

2.3 Register Bus

Networking hardware is more than just passing packets around between pipeline stages. The operation of these stages needs to be controllable by and visible to software that runs the more complex algorithms and protocols at a higher level, as well as by the user to configure and debug the hardware and the network. This means that we need to make the hardware's registers, counters, and tables visible and controllable. A common register interface exposes these data types to the software and allows it to modify them. This is done by memory-mapping the internal hardware registers. The memory-mapped registers then appear as I/O registers to the user software that can access them using ioctl calls.

The register bus strings together register modules in each stage in a pipelined daisy-chain that is looped back in a ring. One module in the chain initiates and responds to requests that arrive as PCI register requests on behalf of the software. However, any stage on the chain is allowed to issue register access requests, allowing information to trickle backwards in the pipeline, and allowing Stage i to get information from Stage $i+k$.

The daisy-chain architecture is preferable to a centralized arbiter approach because it facilitates the interconnection of stages as well as limits inter-stage dependencies.

Requests on the bus can be either a register read or a register write. The bus is pipelined with each transaction consuming one clock cycle. As a request goes through a stage, the stage decides whether to respond to the request or send the request unmodified to the next stage in the chain. Responding to a request means modifying the request by asserting an acknowledge signal in the request and if the request is a read, then also setting the data lines on the register bus to the value of the register.

3. USAGE EXAMPLES

This section describes the IPv4 router and two extensions to this router that are used for research: Buffer Monitoring and OpenFlow. Two other extensions, Time Synchronization and RCP, are described in the Appendix.

3.1 The IPv4 Router

Three basic designs have been implemented on NetFPGA using the interfaces described above: a 4-port NIC, an Ethernet switch, and an IPv4 router. Most projects will build on one of these designs and extend it. In this section, we will describe the IPv4 router on which the rest of the examples in this paper are based.

The basic IPv4 router can run at the full 4x1Gbps linerate. The router project includes the forwarding path in hardware, two software packages that allow it to build routes and routing tables, and command-line and graphical user interfaces for management.

Software: The software packages allow the routing tables to be built using a routing protocol (PeeWee OSPF [14]) running in user-space completely independent of the Linux host, or by using the Linux host's own routing table. The software also handles slow path processing such as generating ICMP messages, handling ARP, IP options, etc. More information can be found in the NetFPGA Guide [12]. The rest of this subsection describes the hardware.

Hardware: The IPv4 hardware forwarding path lends itself naturally to the classic five stage switch pipeline shown in Figure 2. The first stage, *Rx Queues*, receives each packet from the board's I/O ports (such as the Ethernet ports and the CPU DMA interface), appends a module header indicating the packet's length and ingress port, and passes it

using the FIFO interface into the *User Datapath*. The User Datapath contains three stages that perform the packet processing and is where most user modifications would occur. The Rx Queues guarantee that only good link-layer packets are pushed into the User Datapath, and so they handle all the clock domain crossings and error checking.

The first stage in the User Datapath, the *Input Arbiter*, uses packetized round-robin arbitration to select which of the Rx Queues to service and pushes the packet into the *Output Port Lookup* stage.

The Output Port Lookup stage selects which output queue to place the packet in and, if necessary, modifies the packet. In the case of the IPv4 router the Output Port Lookup decrements the TTL, checks and updates the IP checksum, performs the forwarding table and ARP cache lookups and decides whether to send the packet to the CPU as an exception packet or forward it out one of the Ethernet ports. The longest prefix match and the ARP cache lookups are performed using the FPGA's on-chip TCAMs. The stage also checks for non-IP packets (ARP, etc.), packets with IP options, or other exception packets to be sent up to the software to be handled in the slow path. It then modifies the module header originally added by the Rx queue to also indicate the destination output port.

After the Output Port Lookup decides what to do with the packet, it pushes it to the *Output Queues* stage which puts the packet in one of eight output buffers (4 for CPU interfaces and 4 for Ethernet interfaces) using the information that is stored in the module header. For the IPv4 router, the Output Port Lookup stage implements the output buffers in the on-board SRAM. When output ports are free, the Output Queues stage uses a packetized round-robin arbiter to select which output queue to service from the SRAM and delivers the packet to the final stage, the destination *Tx Queue*, which strips out the module headers and puts the packet out on the output port, to go to either the CPU via DMA or out to the Ethernet.

The ability to split up the stages of the IPv4 pipeline so cleanly and hide them under the NetFPGA's pipeline interface allows the module pipeline to be easily and efficiently extended. Developers do not need to know the details of the implementation of each pipeline stage since its results are explicitly present in the module headers. In the next few sections, we use this interface to extend the IPv4 router.

Commercial routers are not usually built this way. Even though the basic stages mentioned, here do exist, they are not so easily or cleanly split apart. The main difference, though, stems from the fact that the NetFPGA router is a pure output-queued switch. An output-queued switch is work conserving and has the highest throughput and lowest average delay.

Organizations building routers with many more ports than NetFPGA cannot afford (or sometimes even design) memory that that has enough bandwidth to be used in a pure output-queued switch. So, they resort to using other tricks such as virtual input queueing, combined input-output queueing ([1]), smart scheduling ([10]), and distributed shared memory to approximate an output queued switch. Since the NetFPGA router runs at line-rate and implements output queueing, the main difference between its behavior and that of a commercial router will be in terms of available buffer sizes and delays across it.

3.2 Buffer Monitoring Router

The Buffer Monitoring Router augments the IPv4 router with an Event Capture stage that allows monitoring the output buffers' occupancies in real-time with single cycle accuracy. This extension was needed to verify the results of resarch on using small output buffers in switches and routers [4]. To do this, the Event Capture stage timestamps when each packet enters an output queue and when it leaves, as well as its length. The host can use these event records to reconstruct the evolution of the queue occupancy from the series.

Since packets can be arriving at up to 4Gbps with a minimum packet length of 64 bytes (84 including preamble and inter-packet gap), a packet will be arriving every $168ns$. Each packet can generate two events: Once going into a queue, and once when leaving. Since each packet event record is 32-bits, the required bandwidth when running at full line rate is approximately 32 bits$/168/2ns = 381$Mbps!

This eliminates the possibility of placing these timestamps in a queue and having the software read them via the PCI bus since it takes approximately $1\mu s$ per 32-bit read (32 Mbps). The other option would be to design a specific mechanism by which these events could be written to a queue and then sent via DMA to the CPU. This, however, would require too much effort and work for a very specific functionality. The solution we use is to collect these events into a packet which can be sent out an output port — either to the CPU via DMA or to an external host via 1 Gbps Ethernet.

To implement the solution, we need to be able to timestamp some signals from the Output Queues module indicating packet events and store these in a FIFO. When enough events are collected, the events are put into a packet that is injected into the router's pipeline with the correct module headers to be placed in an output queue to send to a host, whether local via DMA or remote.

There are are mainly two possibilities for where this extension can be implemented. The first choice would be to add the Event Capture stage between the Output Port Lookup stage and the Output Queues stage. This would allow reusing the stage to monitor signals other than those coming from the Output Queues as well as separate the monitor*ing* logic from the monitor*ed* logic. Unfortunately, since the timestamping happens at single cycle accuracies, signals indicating packet storage and packet removal have to be pulled out of the Output Queues into the Event Capture stage violating the FIFO discipline and using channels other than the packet and register buses for inter-module communication.

Another possibility is to add the buffer monitoring logic into the Output Queues stage. This would not violate the NetFPGA methodology at the cost of making it harder to re-use the monitoring logic for other purposes. The current implementation uses the first approach since we give high priority to re-use and flexibility. This design is shown in Figure 4.

The *Event Capture* stage consists of two parts: an *Event Recorder* module and a *Packet Writer* module. The Event Recorder captures the time when signals are asserted and serializes the events to be sent to the Packet Writer, which aggregates the events into a packet by placing them in a buffer. When an event packet is ready to be sent out, the Packet Writer adds a header to the packet and injects it into the packet bus to be sent into the Output Queues.

While not hard, the main difficulties encountered while

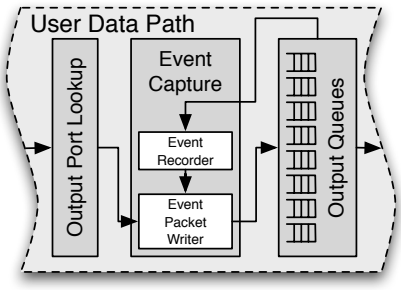

Figure 4: The event capture stage is inserted between the Output Port Lookup and Output Queues stages. The Event Recorder generates events while the Event Packet Writer aggregates them into packets.

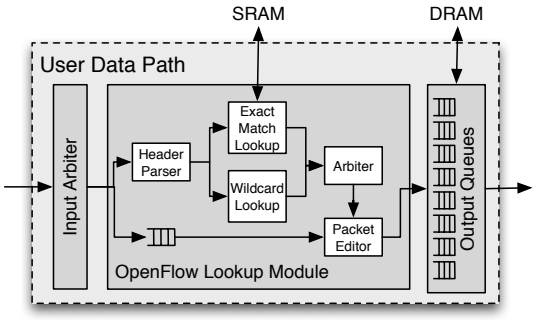

Figure 5: The OpenFlow switch pipeline implements a different Output Port Lookup stage and uses DRAM for packet buffering.

implementing this system were handling simultaneous packet reads and writes from the output queues, and ordering and serializing them to get the correct timestamps. The system was easily implemented over a few weeks time by one grad student, and verified thoroughly by another.

3.3 OpenFlow Switch

OpenFlow is a feature on a networking switch that allows a researcher to experiment with new functionality in their own network; for example, to add a new routing protocol, a new management technique, a novel packet processing algorithm, or even eventually alternatives to IP [11]. The OpenFlow Switch and the OpenFlow Protocol specifications essentially provide a mechanism to allow a switch's flow table to be controlled remotely.

Packets are matched on a 10-tuple consisting of a packet's ingress port, Ethernet MAC destination and source addresses, Ethernet type, VLAN identifier (if one exists), IP destination and source addresses, IP protocol identifier, and TCP/UDP source and destination ports (if they exist). Packets can be matched exactly or using wildcards to specify fields that are *Don't Cares*. If no match is found for a packet, the packet is forwarded to the remote controller that can examine the packet and decide on the next steps to take [13].

Actions on packets can be forwarding on a specific port, normal L2/L3 switch processing, sending to the local host, or dropping. Optionally, they can also include modifying VLAN tags, modifying the IP source/destination addresses, and modifying the UDP/TCP source/destination addresses. Even without the optional per-packet modifications, implementing an OpenFlow switch using a general commodity PC would not allow us to achieve line-rate on four 1Gbps ports; therefore, we implemented an OpenFlow switch on NetFPGA.

The OpenFlow implementation on NetFPGA replaces two of the IPv4 router's stages, the Output Port Lookup and Output Queues stages, and adds a few other stages to implement the actions to be taken on packets as shown in Figure 5. The *OpenFlow Lookup* stage implements the flow table using a combination of on-chip TCAMs and off-chip SRAM to support a large number of flow entries and allow matching on wildcards.

As a packet enters the stage, the *Header Parser* pulls the relevant fields from the packet and concatenates them. This forms the flow header which is then passed to the *Wildcard Lookup* and *Exact Lookup* modules. The Exact Lookup module uses two hashing functions on the flow header to index into the SRAM and reduce collisions. In parallel, the Wildcard Lookup module performs the lookup in the TCAMs to check for any matches on flow entries with wildcards.

The results of both wildcard and exact match lookups are sent to an arbiter that decides which result to choose. Once a decision is reached on the actions to take on a packet, the counters for that flow entry are updated and the actions are specified in new module headers prepended at the beginning of the packet by the *Packet Editor*.

The stages between the Output Port Lookup and before the Output Queues, the *OpenFlow Action* stages, handle packet modifications as specified by the actions in the module headers. Figure 5 only shows one OpenFlow Action stage for compactness, but it is possible to have multiple Action stages in series each doing one of the actions from the flow entry. This allows adding more actions very easily as the specification matures.

The new Output Queues stage implements output FIFOs that are handled in round-robin order using a hierarchy of on-chip Block RAMs (BRAMs) and DRAM as in [8]. The head and tail caches are implemented as static FIFOs in BRAM, and the larger queues are maintained in the DRAM.

Running on the software side is the OpenFlow client that establishes a SSL connection to the controller. It provides the controller with access to the local flow table maintained in the software and hardware and can connect to any OpenFlow compatible controller such as NOX[5].

Pipelining two exact lookups to hide the SRAM latency turned out to be the most challenging part of implementing the OpenFlow Output Port Lookup stage. It required modifying the SRAM arbiter to synchronize its state machine to the Exact Lookup module's state machine and modifying the Wildcard Lookup module to place its results in a shallow fifo because it finishes earlier and doesn't need pipelining. The Match Arbiter has to handle the delays between the Exact Lookup and Wildcard Lookup modules and delays between when the hit/miss signals are generated and when the data is available. To run at line-rate, all lookups had to complete in 16 cycles; so, another challenge was compressing all information needed from a lookup to be able to pull it out from the SRAM with its limited bandwidth in less than 16 cycles.

The hardware implementation currently only uses BRAM for the Output Queues and does not implement any optional

packet modifications. It was completed over a period of five weeks by one graduate student, and can handle full-line rate switching across all ports. The DRAM Output Queues are still being implemented. Integration with software is currently taking place. The exact match flow table can store up to 32,000 flow table entries, while the wildcard match flow table can hold 32. The completed DRAM Output Queues should be able to store up to 5500 maximum-sized (1514 bytes) packets per output queue.

4. LIMITATIONS

While the problems described in the previous section and in the appendix have solutions that fit nicely in NetFPGA, one has to wonder what problems do not. There are at least three issues: latency, memory, and bandwidth.

The first type of problems cannot be easily split into clean computation sections that have short enough latency to fit into a pipeline stage and allow the pipeline to run at line-rate. This includes many cryptographic applications such as some complex message authentication codes or other public key certifications or encryptions.

Protocols that require several messages to be exchanged, and hence require messages to be stored in the NetFPGA an arbitrary amount of time while waiting for responses also do not lend themselves to a simple and clean solution in hardware. This includes TCP, ARP, and others.

We already saw one slight example of the third type of problems in the Buffer Monitoring extension. Most solutions that need too much feedback from stages ahead in the pipeline are difficult to implement using the NetFPGA pipeline. This includes input arbiters tightly coupled the output queues, load-balancing, weighted fair queueing, etc.

5. CONCLUSION

Networking hardware provides fertile ground for designing highly modular and re-usable components. We have designed an interface that directly translates the way packets need to be processed into a simple clean pipeline that has enough flexibility to allow for designing some powerful extensions to a basic IPv4 router. The packet and register buses provide a simple way to pass information around between stages while maintaining a generic enough interface to be applied across all stages in the pipeline.

By providing a simple interface between hardware stages and an easy way to interact with the software, NetFPGA makes the learning curve for networking hardware much gentler and invites students and researchers to modify or extend the projects that run on it. By providing a library of re-usable modules, NetFPGA allows developers to mix and match functionality provided by different modules and string together a new design in a very short time. In addition, it naturally allows the addition of new functionality as a stage into the pipeline without either having to understand the internals of previous or past stages, or having to modify any of them. We believe that we have been able to achieve the goal we have set for NetFPGA of allowing users of different levels to easily build powerful designs in a very short time.

6. ACKNOWLEDGMENTS

We wish to thank John W. Lockwood for leading the NetFPGA project through the rough times of verification and testing, as well as helping to spread the word about NetFPGA. We also wish to thank Adam Covington, David Erickson, Brandon Heller, Paul Hartke, Jianying Luo, and everyone who helped make NetFPGA a success. Finally we wish to thank our reviewers for their helpful comments and suggestions.

7. REFERENCES

[1] S.-T. Chuang, A. Goel, N. McKeown, and B. Prabhakar. Matching Output Queueing with a Combined Input Output Queued Switch. In *INFOCOM (3)*, pages 1169–1178, 1999.

[2] N. Dukkipati, G. Gibb, N. McKeown, and J. Zhu. Building a RCP (Rate Control Protocol) Test Network. In *Hot Interconnects*, 2007.

[3] N. Dukkipati, M. Kobayashi, R. Zhang-Shen, and N. McKeown. Processor Sharing Flows in the Internet. In *Thirteenth International Workshop on Quality of Service (IWQoS)*, 2005.

[4] M. Enachescu, Y. Ganjali, A. Goel, N. McKeown, and T. Roughgarden. Routers With Very Small Buffers. In *IEEE Infocom*, 2006.

[5] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: Towards an Operating System for Networks. To appear.

[6] M. Handley, E. Kohler, A. Ghosh, O. Hodson, and P. Radoslavov. Designing Extensible IP Router Software. In *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, pages 189–202, Berkeley, CA, USA, 2005. USENIX Association.

[7] IEEE. IEEE 1588 - 2002, Precision Time Protocol. Technical report, IEEE, 2002.

[8] S. Iyer, R. R. Kompella, and N. McKeown. Designing Packet Buffers for Router Line Cards. Technical report, Stanford University High Performance Networking Group, 2002.

[9] J. W. 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 *MSE '07: Proceedings of the 2007 IEEE International Conference on Microelectronic Systems Education*, pages 160–161, Washington, DC, USA, 2007. IEEE Computer Society.

[10] N. McKeown. The iSLIP Scheduling Algorithm for Input-Queued Switches. *IEEE/ACM Trans. Netw.*, 7(2):188–201, 1999.

[11] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling Innovation in College Networks. Soon to appear in ACM Computer Communication Review.

[12] NetFPGA Development Team. NetFPGA User's and Developer's Guide. Can be found at http://netfpga.org/static/guide.html.

[13] OpenFlow Consortium. OpenFlow Switch Specification. Available at http://openflowswitch.org/documents.html.

[14] Stanford University. Pee-Wee OSPF Protocol Details. Can be found at http://yuba.stanford.edu/cs344_public/docs/pwospf_ref.txt.

APPENDIX

A. RCP ROUTER

RCP (Rate Control Protocol) is a congestion control algorithm which tries to emulate processor sharing in routers [2]. An RCP router maintains a single rate for all flows. RCP packets carry a header with a rate field, R_p, which is overwritten by the router if the value within the packet is larger than the value maintained by the router, otherwise it is left unchanged. The destination copies R_p into acknowledgment packets sent back to the source, which the source then uses to limit the rate at which it transmits. The packet header also includes the source's guess of the round trip time (RTT); a router uses the RTT to calculate the average RTT for all the flows passing through it.

RCP's goal is to decrease flow-completion times compared to TCP and XCP [3] while coexisting with legacy transport protocols. One of the requirements is minimal changes to the router hardware so most of the complexity is pushed to software. The hardware is responsible for collecting the required statistics, averaging the RTT and flow sizes, and stamping the rate in the packets if required, while the software is responsible for calculating the router's rate periodically based on the data it gathers from the hardware. The calculated rate is then passed to the hardware using the register interface.

Since per-packet modification might be necessary, it would be very difficult for a software based router to implement this functionality on four 1Gbps ports running at line rate. The NetFPGA router was therefore used to implement this extension. The IPv4 router is extended by adding an additional stage between the router Output Port Lookup and the Output Queues stages.

The additional stage, the *RCP Stage*, parses the RCP packet and updates R_p if required. It also calculates per-port averages and makes this information available to the software via the memory-mapped register interface. The router also includes the Buffer Monitoring stage from the Buffer Monitoring Router, allowing users to monitor queue occupancy evolution when RCP is being used and when it is not.

The design of the system took around two days and the implementation and testing took around 10 days.

B. PRECISION TIME SYNCHRONIZATION ROUTER

A design that is loosely based on the IEEE 1588 standard [7] was implemented to enable high precision synchronization of clocks between hosts over a network. Since very high precision synchronization requires timestamping packets as close as possible to the wire on egress and ingress, a modifiable hardware forwarding path was needed. The NetFPGA IPv4 router was modified to do these timestamps and to send them to the software along with any synchronization protocol packets.

The IPv4 router was extended to timestamp packets on ingress and egress before reaching the Ethernet Rx Queue and after the Tx Queues. Timestamps are then stored in registers to allow the software to extract them. A rate-controlled clock is implemented on the NetFPGA allowing changing the clock rate very precisely via registers. A software daemon implements the synchronization protocol that adjusts the clock to achieve synchronization. The design has been implemented and is currently being debugged and tested for accuracy.