# Developers Workshop

## August 13-14, 2009

## Stanford, CA

http://NetFPGA.org/DevWorkshop



- Program Chairs
  - John W. Lockwood: Stanford University
  - Andrew W. Moore: Cambridge University
- Program Committee
  - Satnam Singh, Microsoft Research Cambridge
  - David Miller, Cambridge University Computer Laboratory
  - Gordon Brebner, Xilinx
  - Martin Žádník, Brno University of Technology
  - Glen Gibb: Stanford University
  - Adam Covington: Stanford University
  - David Greaves: Cambridge University
  - Eric Keller: Princeton University
- NetFPGA Sponsors

## *Welcome from the Program Chairs*

- State of the NetFPGA Program
  - John W. Lockwood (Stanford University)
    (Slides) pp. 5-16

- NetFPGA at Cambridge
  - Andrew W. Moore (University of Cambridge)
    (Slides) pp. 17-23

## *Session 1: Packet Forwarding*

- zFilter Sprouter - Implementing zFilter based forwarding node on a NetFPGA
  - J. Keinänen, P. Jokela, K. Slavov (Ericsson Research)
    - (Wiki) and (Paper) pp. 24-31

- IP-Lookup with a Blooming Tree Array
  - Gianni Antichi, Andrea Di Pietro, Domenico Ficara, Stefano Giordano, Gregorio Procissi, Cristian Vairo, Fabio Vitucci (University of Pisa)
    - (Wiki) and (Paper) pp. 32-38

## *Session 2: Payload Processing*

- URL Extraction
  - M. Ciesla, V. Sivaraman, A. Seneviratne (UNSW, NICTA)
    - (Wiki) and (Paper) and (Slides) pp. 39-44

- DFA-based Regular Expression Matching Engine on NetFPGA
  - Y. Luo, S. Li, Y. Liu (University of Massachusetts Lowell)
    - (Wiki) and (Paper) pp. 45-49

## *Session 3: High-Level Programming*

- High-level programming of the FPGA on NetFPGA
  - M. Attig, G. Brebner (Xilinx)
    - (Wiki) and (Paper) pp. 50-55

- NetThreads: Programming NetFPGA with Threaded Software
  - M. Labrecque, J. Steffan, G. Salmon, M. Ghobadi, Y. Ganjali (University of Toronto)
    - (Wiki) and (Paper) and (Slides) pp. 56-61

- NetFPGA-based Precise Traffic Generation
  - G. Salmon, M. Ghobadi, Y. Ganjali, M. Labrecque, J. Steffan (University of Toronto)
    - (Wiki), (Paper) and (Slides) pp. 62-68

## *Session 4: Applications I*

- AirFPGA: A software defined radio platform based on NetFPGA
  - James Zeng, Adam Covington, and John Lockwood (Stanford University); Alex Tutor (Agilent)
    - (Wiki) , (Paper), and (Slides) pp. 69-75

- Fast Reroute and Multipath
  - Mario Flajslik, Nekhil Handigol, James Zeng (Stanford University- CS344 Project)
    - (Wiki) and (Paper) pp. 76-79

## *Session 5: Testbeds*

- NetFPGAs in the Open Network Lab (ONL)
  - Charlie Wiseman, Jonathan Turner, John DeHart, Jyoti, Parwatikar, Ken Wong, David Zar (Washington University in St. Louis)
    - (Wiki), (Paper), and (Slides) pp. 80-85

- Implementation of a Future Internet Testbed on KOREN based on NetFPGA/OpenFlow Switches
  - Man Kyu Park, Jae Yong Lee, Byung Chul Kim, Dae Young Kim (Chungnam National University)
    - (Wiki) and (Paper) pp. 86-89

## *Session 6: Applications II*

- RED - Random Early Detection
  - Gustav Rydstedt and Jingyang Xue (Stanford University - CS344 Project)
    - (Wiki)

- A Fast, Virtualized Data Plane for the NetFPGA
  - M. Anwer, N. Feamster (Georgia Institute of Technology)
    - (Wiki) and (Paper) pp. 90-94

- A Windows Support Framework for the NetFPGA 2 Platform
  - C. Tian, D. Zhang, G. Lu, Y. Shi, C. Guo, Y. Zhang
    - (Wiki) and (Paper) 95-101

# *Photographs from the event*

**Participants at the NetFPGA Developers Workshop on August 13, 2009 at Stanford University**

**Presentations**



**Questions & Answers**

**Live Demonstrations**

# State of the NetFPGA Program


## John W. Lockwood

**and the NetFPGA team at Stanford University**



Hardware and tools available for university programs thanks to grants, donations, and/or partnerships from:




# Many Thanks

- **Developers Workshop Program Chairs**
  - John W. Lockwood, Stanford University
  - Andrew W. Moore, Cambridge University
- **Developers Workshop Program Committee**
  - Satnam Singh, Microsoft Research Cambridge
  - David Miller, Cambridge University Computer Laboratory
  - Gordon Brebner, Xilinx
  - Martin Žádník, Brno University of Technology
  - Glen Gibb: Stanford University
  - Adam Covington: Stanford University
  - David Greaves: Cambridge University
  - Eric Keller: Princeton University
- **NetFPGA Team at Stanford University**

- **The Worldwide NetFPGA Developer Community**

# Where are NetFPGAs?

– Over 1,000 users with ~1,000 cards deployed
at ~150 universities in 17 Countries worldwide




# NetFPGA Hardware in North America

**Locations of Deployed NetFPGA Hardware**

**USA - Jan 2009**

# NetFPGA Hardware in Europe

**Locations of Deployed NetFPGA Hardware** EU - Jan 2009




# NetFPGA Hardware in Asia

**Locations of Deployed NetFPGA Hardware**


China, Korea, Japan, Taiwan - Jan 2009

# NetFPGA's Defining Characteristics

- **Line-Rate**
  - Processes back-to-back packets
    - Without dropping packets
    - At full rate of Gigabit Ethernet Links
  - Operating on packet headers
    - For switching, routing, and firewall rules
  - And packet payloads
    - For content processing and intrusion prevention
- **Open-source Hardware**
  - Similar to open-source software
    - Full source code available
    - BSD-Style License
  - But harder, because
    - Hardware modules must meeting timing
    - Verilog & VHDL Components have more complex interfaces
    - Hardware designers need high confidence in specification of modules



# Test-Driven Design

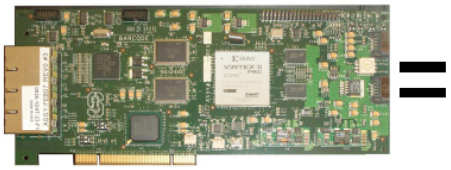
- **Regression tests**
  - Have repeatable results
  - Define the supported features
  - Provide clear expectation on functionality
- ***Example:* Internet Router**
  - Drops packets with bad IP checksum
  - Performs Longest Prefix Matching on destination address
  - Forwards IPv4 packets of length 64-1500 bytes
  - Generates ICMP message for packets with TTL <= 1
  - Defines how packets with IP options or non IPv4

    … and dozens more …

    ***Every feature is defined by a regression test***

# NetFPGA = Networked FPGA

**A line-rate, flexible, open networking platform for teaching and research**



Network Interface Card

Hardware Accelerated Linux Router

IPv4 Reference Router

Traffic Generator

Openflow Switch

List of All Projects

Add Your Project



# NetFPGA Designs

| Project (Title & Summary) | Base | Status | Organization | Docs. |
|---|---|---|---|---|
| IPv4 Reference Router | 2.0 | Functional | Stanford University | Guide |
| Quad-Port Gigabit NIC | 2.0 | Functional | Stanford University | Guide |
| Ethernet Switch | 2.0 | Functional | Stanford University | Guide |
| Hardware-Accelerated Linux Router | 2.0 | Functional | Stanford University | Guide |
| Packet Generator | 2.0 | Functional | Stanford University | Wiki |
| OpenFlow Switch | 2.0 | Functional | Stanford University | Wiki |
| DRAM-Router | 2.0 | Functional | Stanford University | Wiki |
| NetFlow Probe | 1.2 | Functional | Brno University | Wiki |
| AirFPGA | 2.0 | Functional | Stanford University | Wiki |
| Fast Reroute & Multipath Router | 2.0 | Functional | Stanford University | Wiki |
| NetThreads | 1.2.5 | Functional | University of Toronto | Wiki |
| URL Extraction | 2.0 | Functional | Univ. of New South Wales | Wiki |
| zFilter Sprouter (Pub/Sub) | 1.2 | Functional | Ericsson | Wiki |
| Windows Driver | 2.0 | Functional | Microsoft Research | Wiki |
| IP Lookup w/Blooming Tree | 1.2.5 | In Progress | University of Pisa | Wiki |
| DFA | 2.0 | In Progress | UMass Lowell | Wiki |
| G/PaX | ?.? | In Progress | Xilinx | Wiki |
| Precise Traffic Generator | 1.2.5 | In Progress | University of Toronto | Wiki |
| Open Network Lab | 2.0 | In Progress | Washington University | Wiki |
| KOREN Testbed | ?.? | In Progress | Chungnam-Korea | Wiki |
| RED | 2.0 | In Progress | Stanford University | Wiki |
| Virtual Data Plane | 1.2 | In Progress | Georgia Tech | Wiki |
| Precise Time Protocol (PTP) | 2.0 | In Progress | Stanford University | Wiki |
| Deficit Round Robin (DRR) | 1.2 | Repackage | Stanford University | Wiki |

.. And more on http://netfpga.org/netfpgawiki/index.php/ProjectTable

# Highlights of this Workshop

- **Packet Forwarding**
  - zFilter based forwarding node on a NetFPGA
  - IP-Lookup with a Blooming Tree Array
- **Payload**
  - URL Extraction
  - DFA-based Regular Expression Matching
- **High-level programming**
  - G / Packet Express (PAX)
  - Programming with Network Threads
  - Traffic Generation with NetThreads
- **Future Directions**
  - Panel Discussion with industry
- **Applications I**
  - AirFPGA: Software Defined Radio (SDR) platform
  - Fast Reroute & Multipath
- **Testbeds**
  - NetFPGAs in the Open Network Lab (ONL)
  - KOREN (Korea)
- **Applications II**
  - Random Early Detection
  - Virtualized Data Plane for the NetFPGA



# Cards,

# Systems

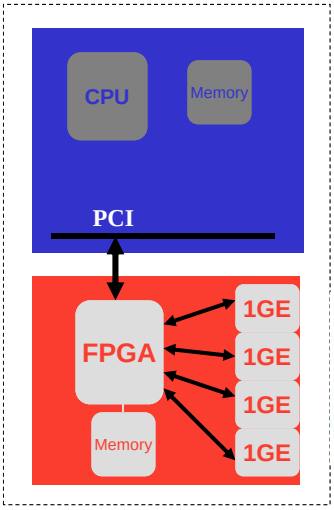
# Clusters

# NetFPGA System

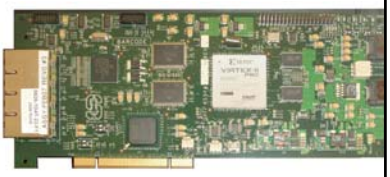
**Software running on a standard PC**

**+**

**A hardware accelerator built with Field Programmable Gate Array driving Gigabit network links**




PC with NetFPGA



NetFPGA Board



# NetFPGA Systems

- **Pre-built systems available**
  - From 3rd Party Vendor
- **PCs assembled from parts**
  - Integrates into standard PC
- **Details are in the Guide**
  - http://netfpga.org/static/guide.html

# Rackmount NetFPGA Servers

2U Server
(Dell 2950)

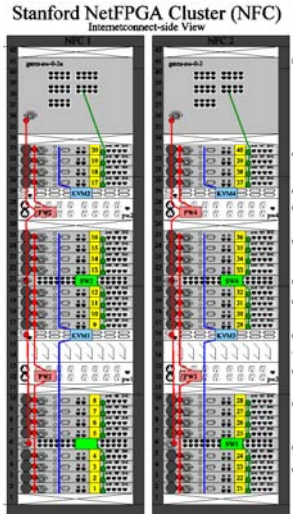
NetFPGA inserts in
PCI or PCI-X slot

1U Server
(Accent Technology, Inc)

Thanks: Brian Cashman for providing machine



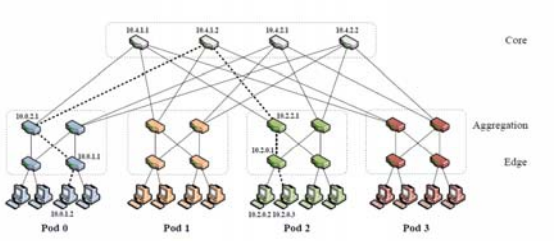
# Stanford NetFPGA Cluster

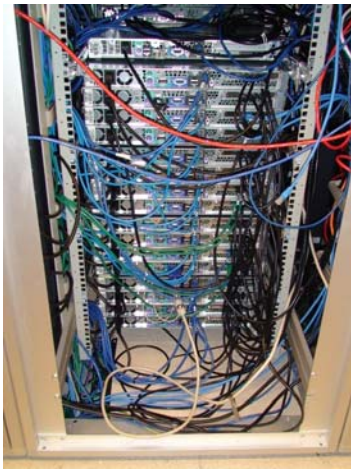



Statistics

- Rack of 40
  - 1U PCs
  - NetFPGAs
- Manged
  - Power,
  - Console
  - VLANs
- Provides 160 Gbps of full line-rate processing bandwidth

# UCSD-NetFPGA Cluster





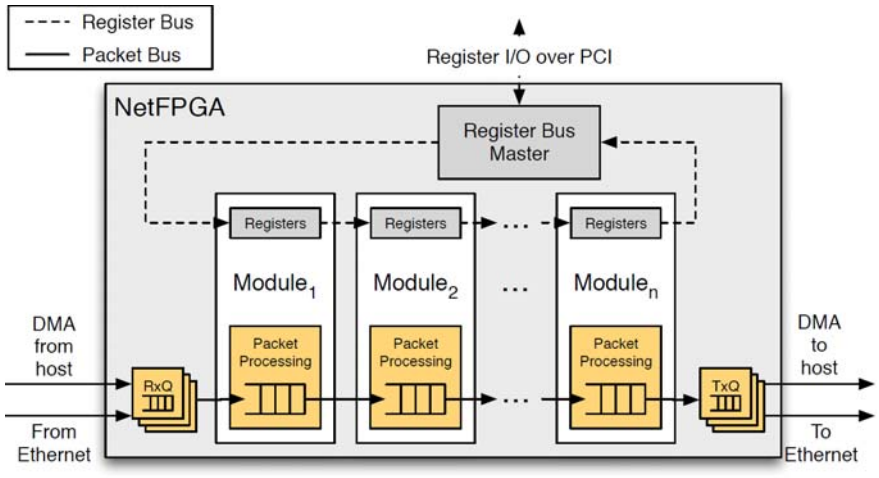
# Going Forward

# The New 2.0 Release

- **Modular Registers**
  - Simplifies integration of multiple modules
    - Many users control NetFPGAs from software
  - Register set joined together at build time
    - Project specifies registers in XML list
- **Packet Buffering in DRAM**
  - Supports Deep buffering
    - Single 64MByte queue in DDR2 memory
- **Programmable Packet Encapsulation**
  - Packet-in-packet encapsulation
    - Enables tunnels between OpenFlowSwitch nodes

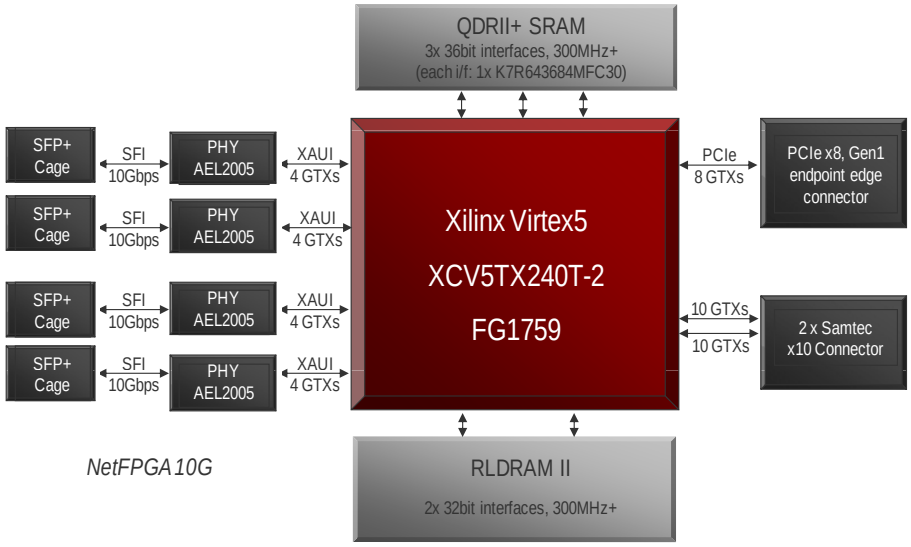


# Module Pipeline




From: Methodology to Contribute NetFPGA Modules, by G. Adam Covington, Glen Gibb, Jad Naous, John Lockwood, Nick McKeown; IEEE Microelectronics System Education (MSE), June 2009.
on : http://netfpga.org/php/publications.php

# NetFPGA 10G: (Coming in 2010)






# Going Forward

- **NSF Funding at Stanford**
  - Supports program at Stanford for next 4 years
    - Workshops, Tutorials, Support
- **Academic Collaborations**
  - Cambridge, NICTA, KOREN, ONL, …
    - Academic Tutorials
    - Developer Workshops
- **Industry Collaborations**
  - AlgoLogicSystems.com
    - Designs algorithms in Logic
    - Creates systems with open FPGA platforms
    - Uses and contributes to open-source cores
    - Provides customized training to industry

# Conclusions

- **NetFPGA Provides**
  - Open-source, hardware-accelerated Packet Processing
  - Modular interfaces arranged in reference pipeline
  - Extensible platform for packet processing
- **NetFPGA Reference Code Provides**
  - Large library of core packet processing functions
  - Scripts and GUIs for simulation and system operation
  - Set of Projects for download from repository
- **The NetFPGA Base Code**
  - Well defined functionality defined by regression tests
  - Function of the projects documented in the Wiki Guide



# Final Thoughts for Developers

- **Build Modular components**
  - Describe shared registers (as per 2.0 release)
  - Consider how modules would be used in larger systems
- **Define functionality clearly**
  - Through regression tests
  - With repeatable results
- **Disseminate projects**
  - Post open-source code
  - Document projects on Web, Wiki, and Blog
- **Expand the community of developers**
  - Answer questions in the Discussion Forum
  - Collaborate with your peers to build new applications

UNIVERSITY OF CAMBRIDGE
800 YEARS
1209 - 2009

# NetFPGA in Cambridge

**Andrew W. Moore**

**Computer Laboratory**

- Cambridge: not exactly network newcomers
- NetFPGA: right tool / right time
- Teaching
  - Masters course (similar to CS344)
  - Masters dissertation vehicle (6 month piece of work)
  - Undergraduate project vehicle (e.g., TOE implementation)
- Research
  - network emulation elements
  - implementation vehicle for middlebox ideas
  - testing new ideas for a revamped Ethernet
  - new MACs for new networks (SWIFT) and a prototype vehicle
  - target platform for better development toolchains
- Dissemination
  - Tutorials and workshops

# Cambridge? never heard of them

- But you may have heard of some of our more successful projects (some have changed name):

Autonomy alphamosaic ARM azuro TenisonEDA BBC MICRO Virata Level 5 networks ANSA Xen Source™

- And some of our not so successful projects:

ATM

**Cambridge Backbone Ring**
**1 Gb/s LAN/WAN in 1995**

(we didn't want 48 byte payloads either – so very silly)

Sun's sunray

# NetFPGA Teaching in Cambridge

- **Coursework**
  - P33 "Building an Internet Router"
    - based upon Stanford cs344
- **Graduate Dissertations**
  - A new Masters course means 6 month dissertations
    - (think of them as "PhD qualifiers")
- **Undergraduate Projects**
  - Smallish "Computer Science complete" projects
    - 2008/9: Peter Ogden implemented a TOE on NetFPGA

# P33: "Building an Internet Router" A Cambridge course from October

- A module in a new single-year Masters degree
  MPhil (Advanced Computer Science)
  - a "pre-PhD" entry programme.
- Lecturer: me
- TAs: Phil Watts and David Miller
- Ideally 3 groups of 3, current expressions of interest is 22(!)… but many will fall short of prerequisite requirements.
- Principally a pass-fail subject (with the "project competition reward"), BUT the subject is on offer to other Masters has a 0-100 mark scale (60=pass).

This was planned to be a "clone" of cs344

# P33: "Building an Internet Router" (how well will we translate?)

Well not a clone, more a translation:

- *Arnie* becomes *Sean*

- Stanford Terms ≠ Cambridge Terms
  - so not quite enough weeks… solutions include:
    - cut the extension weeks
    - bigger groups (classic Brookes law (Mythical Man-Month) failure)
    - do less (e.g. drop the CLI requirement)
    - start with more:
      (start with CLI and static Ethernet switch)
- A lot more Lecturer contact time (a function of this being a new module and not having as many helpers as Nick, yet…)
- Entry criteria (Stanford and Cambridge have ECAD (Verilog))
  - most of the UK/EU does not (or has VHDL)
    Our solution is to seed with a few Cambridge ECAD backgrounded people

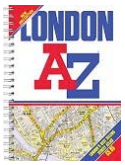
# NetFPGA-enabled Research

- network emulation elements
- implementation vehicle for middlebox ideas
- testing new ideas for a revamped Ethernet
- new MACs for new networks (SWIFT) and
  - a prototype vehicle for networks that don't exist
- target platform for better development toolchains (C# -> kiwi -> (bluespec) -> Verilog)





# Middlebox: AtoZ




- AtoZ implements an application-aware traffic manager on NetFPGA
  - Application-detection technology is the "magic in the box" but the implementation was challenging and noteworthy
- NetFPGA allows handcrafting to suite test deployments

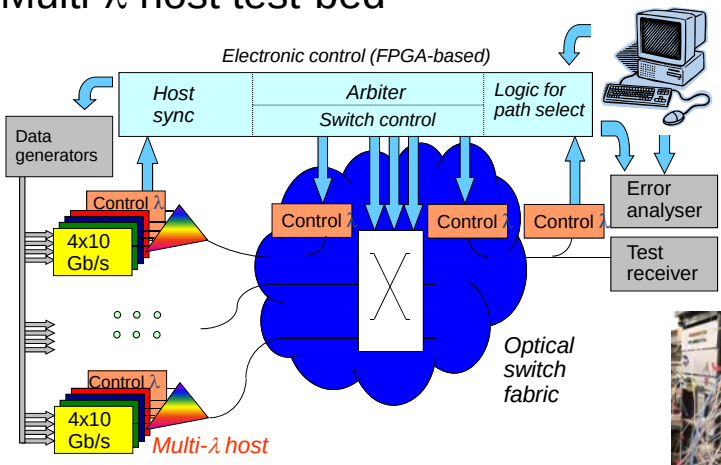
*Look for our paper in ANCS 2009 in Princeton*

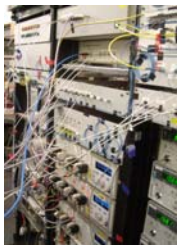
## MOOSE: Addressing the Scalability of Ethernet

- An approach to Ethernet that blurs the boundary of Layer-2 and Layer-3, through:
  - improved routing
  - mitigating broadcast/multicast data and
  - none of the DHT complexity of SEATTLE
- Currently a software prototype with a NetFPGA implementation in progress.
- (Solves similar problems to the "Floodless in SEATTLE" approach, but in a different/better way…)

## Building a new PCI

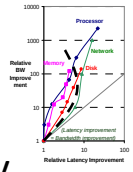
### Multi-λ host test-bed



Sequential multi-host testbed for multi-wavelength packets with FPGA control

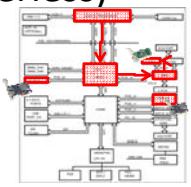
# Building a new PCI




- NetFPGA used as a test target in a latency study of PCI (old and new)

*Look for our paper in ANCS 2009 in Princeton*

- NetFPGA-based prototype network is the basis of a test network for a new (bufferless) PCI approach



# NetFPGA 2-Day workshop in Cambridge


s (full house)

nodation for non-locals

- 30% commercial attendees

Next Cambridge workshop: March'10

- (tutorial, workshop or camp…
to be decided)

# How might YOU use NetFPGA?

- Build an accurate, fast, line-rate NetDummy/nistnet element
- A flexible home-grown monitoring card
- Evaluate new packet classifiers
  - (and application classifiers, and other neat network apps….)
- Prototype a full line-rate next-generation Ethernet-type
- Trying any of Jon Crowcrofts' ideas (Sourceless IP routing for example)
- Demonstrate the wonders of Metarouting in a different implementation (dedicated hardware)
- Provable hardware (using a C# implementation and kiwi with NetFPGA as target h/w)
- Hardware supporting Virtual Routers
- Check that some brave new idea actually works
  - e.g. Rate Control Protocol (RCP), Multipath TCP,
- toolkit for hardware hashing
- MOOSE implementation
- IP address anonymization
- SSL decoding "bump in the wire"
- Xen specialist nic
- computational co-processor
- Distributed computational co-processor
- IPv6 anything
- IPv6 – IPv4 gateway (6in4, 4in6, 6over4, 4over6, ….)
- Netflow v9 reference
- PSAMP reference
- IPFIX reference
- Different driver/buffer interfaces (e.g. PFRING)
- or "escalators" (from gridprobe) for faster network monitors
- Firewall reference
- GPS packet-timestamp things
- High-Speed Host Bus Adapter reference implementations
  - Infiniband
  - iSCSI
  - Myranet
  - Fiber Channel
- Smart Disk adapter (presuming a direct-disk interface)
- Software Defined Radio (SDR) directly on the FPGA (probably UWB only)
- Routing accelerator
  - Hardware route-reflector
  - Internet exchange route accelerator
- Hardware channel bonding reference implementation
- TCP sanitizer
- Other protocol sanitizer (applications… UDP DCCP, etc.)
- Full and complete Crypto NIC
- IPSec endpoint/ VPN appliance
- VLAN reference implementation
- metarouting implementation
- virtual <pick-something>
- intelligent proxy
- application embargo-er
- Layer-4 gateway
- h/w gateway for VoIP/SIP/skype
- h/w gateway for video conference spaces
- security pattern/rules matching
- Anti-spoof traceback implementations (e.g. BBN stuff)
- IPtv multicast controller
- Intelligent IP-enabled device controller (e.g. IP cameras or IP powerm
- DES breaker
- platform for flexible NIC API evaluations
- snmp statistics reference implementation
- sflow (hp) reference implementation
- trajectory sampling (reference implementation)
- implementation of zeroconf/netconf configuration language for route
- h/w openflow and (simple) NOX controller in one…
- Network RAID (multicast TCP with redundancy)
- inline compression
- hardware accelerator for TOR
- load-balancer
- openflow with (netflow, ACL, ….)
- reference NAT device
- active measurement kit
- network discovery tool
- passive performance measurement
- active sender control (e.g. performance feedback fed to endpoints fo
- Prototype platform for NON-Ethernet or near-Ethernet MACs
  - Optical LAN (no buffers)

# Next…


- You can do it too….
  (Many of you have done it already!)
  - Research (even the smallest scale)
  - Teaching using the NetFPGA
  - Dissemination of the NetFPGA project…

# Implementing zFilter based forwarding node on a NetFPGA

Jari Keinänen, Petri Jokela, Kristian Slavov
Ericsson Research, NomadicLab
02420 Jorvas, Finland
firstname.secondname@ericsson.com

## ABSTRACT

Our previous work has produced a novel, Bloom-filter based, forwarding fabric, suitable for large-scale topic-based publish/subscribe [8]. Due to very simple forwarding decisions and small forwarding tables, the fabric may be more efficient than the currently used ones. In this paper, we describe the NetFPGA based forwarding node implementation for this new, IP-less, forwarding fabric. The implementation requires removing the traditional IP forwarding implementation, and replacing it with the Bloom-filter matching techniques for making the forwarding decisions. To complete the work, we provide measurement results to verify the forwarding efficiency of the proposed forwarding system and we compare these results to the measurements from the original, IP-based forwarding, implementation.

## 1. INTRODUCTION

While network-level IP multicast was proposed almost two decades ago [5], its success has been limited due to the lack of wide scale deployment. As a consequence, various forms of application-level multicast have gained in popularity, but their scalability and efficiency have been limited. Hence, a challenge is how to build a multicast infrastructure that can scale to, and tolerate the failure modes of, the general Internet, while achieving low latency and efficient use of resources.

In [8], we propose a novel multicast forwarding fabric. The mechanism is based on identifying links instead of nodes and uses in-packet Bloom filters [2] to encode source-route-style forwarding information in the packet header. The forwarding decisions are simple and the forwarding tables fairly small, potentially allowing faster, smaller, and more energy-efficient switches than what today's switches are. The proposed (inter-)networking model aims towards balancing the state between the packet headers and the network nodes, allowing both stateless and stateful operations [16].

The presented method takes advantage of "inverting" the Bloom filter thinking [3]. Instead of maintaining Bloom filters at the network nodes and verifying from incoming packets if they are included in the filter or not, we put the Bloom filters themselves in the packets and allow the nodes on the path to determine which outgoing links the packet should be forwarded to.

In this paper, we present the implementation of a forwarding node on a NetFPGA. At the first stage, we have implemented the basic forwarding node functions enabling packet delivery through the network using the described forwarding mechanism. At the same time we have been developing a FreeBSD-based end-host implementation, based on publish/subscribe networking architecture, described in [8]. The end-host implements the packet management, as well as networking related functions. The present environment supports only simple networks, but once the first release of the end-host implementation is ready, larger scale networks can be created and tested.

We selected NetFPGA as the forwarding node platform because it offers a fast way to develop custom routers. It provides a way easy to move implementations directly on hardware by taking advantage of reprogrammable FPGA circuits enabling prototype implementations that can handle high speed data transmission (1Gbps/link). We can also avoid time consuming and expensive process of designing new physical hardware components.

The rest of this paper is organized as follows. First, in Section 2, we discuss the general concepts and architecture of our solution. In Section 3, we go into details of the implementation. Next, in Section 4, we provide some evaluation and analysis of our forwarding fabric Section 5 contrasts our work with related work, and Section 6 concludes the paper.

## 2. ARCHITECTURE

Our main focus in this paper is on describing the forwarding node implementation of the Bloom-filter based forwarding mechanism referred to as zFilters. In this section, we describe the basic zFilter operations, and for more detailed description, we refer to [8].

### 2.1 Forwarding on Bloomed link identifiers

The forwarding mechanism described in this paper

is based on identifying links instead of nodes. In the basic operation, the forwarding nodes do not need to maintain any state other than a Link ID per interface. The forwarding information is constructed using these Link IDs and including them in the packet header in a Bloom filter fashion. For better scalability, we introduce an enhancement that inserts a small amount of state in the network by creating virtual trees in the network and identifying them using similar identifiers as the Link IDs. In this section we describe the basics of such forwarding system, and more detailed information can be found from [8].

### 2.1.1 The basic Bloom-filter-based forwarding

For each point-to-point link, we assign two identifiers, called Link IDs, one in each direction. For example, a link between the nodes $A$ and $B$ has two identifiers, $\overrightarrow{AB}$ and $\overleftarrow{AB}$. In the case of a multi-point link, such as a wireless link, we consider each pair of nodes as a separate link. With this setup, we don't need any common agreement between the nodes on the link identities – each link identity may be locally assigned, as long as the probability of duplicates is low enough.

Basically, a Link ID is an $m$-bit long name with just $k$ bits set to one. In [8] we discuss the proper values for $m$ and $k$, and what are the consequences if we change the values; however, for now it is sufficient to note that typically $k \ll m$ and $m$ is relatively large, making the Link IDs statistically unique (e.g., with $m = 248$, $k = 5$, # of Link IDs $\approx m!/(m-k)! \approx 9 * 10^{11}$).

The complete architecture includes a management system that creates a graph of the network using Link IDs and connectivity information, without any dependency on end-point naming or addressing (creating the "topology map" or "routing table"). Using the network graph, the topology system can determine a forwarding tree for any publication, from the locations of the publisher and subscribers [16]. In this paper, however, we assume that such topology management exists and refer to [8] for more detailed discussion about the complete architecture.

When the topology system gets a request to determine a forwarding tree for a certain publication, it first creates a conceptual delivery tree for the publication using the network graph. Once it has such an internal representation of the tree, it knows which links the packets need to pass, and it can determine when to use Bloom filters and when to create state. [16]

In the default case, we use a source-routing based approach which makes forwarding independent from routing. Basically, we encode all Link IDs of the delivery tree into a Bloom filter, forming the forwarding zFilter for the data. Once all Link IDs have been added to the filter, a mapping from the data topic identifier to the zFilter is given to the node acting as the data source, which now can create packets that will be delivered along the tree.

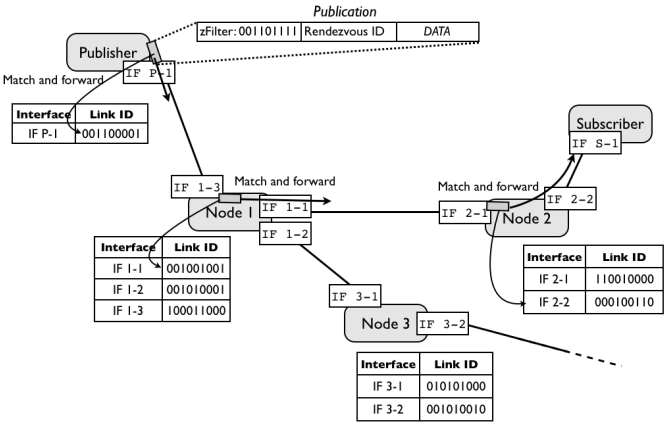



**Figure 1: Example of Link IDs assigned for links, as well as a publication with a zFilter, built for forwarding the packet from the Publisher to the Subscriber.**

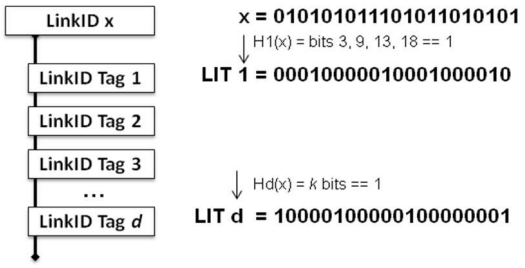



**Figure 2: An example relation of one Link ID to the $d$ LITs, using $k$ hashes on the Link ID.**

Each forwarding node acts on packets roughly as follows. For each link, the outgoing Link ID is ANDed with the zFilter found in the packet. If the result matches with the Link ID, it is assumed that the Link ID has been added to the zFilter and that the packet needs to be forwarded along that link.

With Bloom filters, matching may result with some false positives. In such a case, the packet is forwarded along a link that was not added to the zFilter, causing extra traffic. While the ratio of false positives depends on the number of entries added to the filter, we get a practical limit on how many link names can be included into a single zFilter.

Our approach to the Bloom filter capacity limit is twofold: Firstly, we use recursive layering [4] to divide the network into suitably-sized components and secondly, the topology system may dynamically add *virtual links* to the system (see Section 2.2.1).

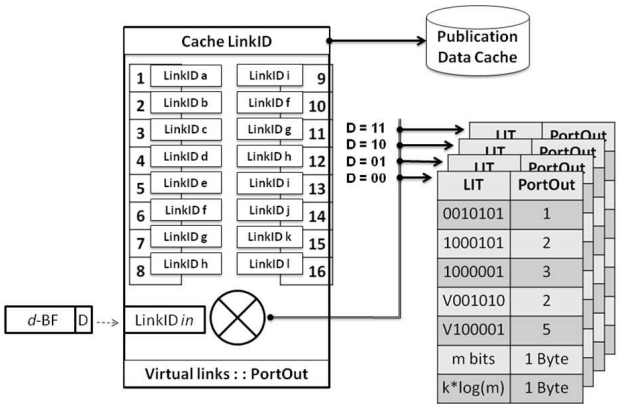



**Figure 3: Outgoing interfaces are equipped with $d$ forwarding tables, indexed by the value in the incoming packet.**

### 2.1.2 Link IDs and LITs

To reduce the number of false positives, we introduced [8] *Link ID Tags* (LITs), as an addition to the plain Link IDs. The idea is that instead of each link being identified with a single Link ID, every unidirectional link is associated with a set of $d$ distinct LITs (Fig. 2). This allows us to construct zFilters that can be optimized, e.g., in terms of the false positive rate, compliance with network policies, or multi path selection. The approach allows us to construct different candidate zFilters and to to select the best-performing Bloom filter from the candidates, according to any appropriate metric.

The forwarding information is stored in the form of $d$ forwarding tables, each containing the LIT entries of the active Link IDs, as depicted in Fig. 3. The only modification of the base forwarding method is that the node needs to be able to determine which forwarding table it should perform the matching operations; for this, we include the index in the packet header before zFilter.

The construction of the forwarding Bloom filter is similar to the one discussed in single Link ID case, except that for the selected path from the publisher to the subscriber, we calculate $d$ candidate filters, one using each of the $d$ values, which are each equivalent representations of the delivery tree.

As a consequence, having $d$ different candidates each representing the given delivery tree is a way to minimise the number of false forwardings in the network, as well as restricting these events to places where their negative effects are smallest. [8]

## 2.2 Stateful operations

In the previous, we presented the basic, single link, based forwarding solution. A forwarding node does not maintain any connection or tree based states, the only information that it has to maintain is the outgoing Link IDs. In this section, we discuss some issues that enhance the operation with the cost of adding small amount of state on the forwarding nodes.

### 2.2.1 Virtual links

As discussed in [8], the forwarding system in its basic form, is scalable into metropolitan area networks with sparse multicast trees. However, in case of dense multicast trees and larger networks, increasing the number of Link IDs in the Bloom filter will increase the number of false positives on the path. For more efficient operations, the topology layer can identify different kinds of delivery trees in the network and assign them virtual Link IDs that look similar to Link IDs described earlier.

Once a virtual link has been created, each participating router is configured with the newly created virtual Link ID information, adding a small amount of state in its forwarding table. The virtual link identifier can then be used to replace all the single Link IDs needed to form the delivery tree, when creating zFilters.

### 2.2.2 Link failures - fast recovery

All source routing based forwarding mechanisms are vulnerable when link failures occur in the network. While the packet header contains the exact route, the packets will not be re-routed using other paths.

In zFilters [8], we have proposed two simple solutions for this mentioned problem: we can use either pre-configured virtual links, having the same Link ID as the path which it is replacing or then we can use pre-computed zFilters, bypassing the broken link. The former method requires an additional signalling message so that the alternative path is activated, but the data packets can still use the same zFilter and do not need any modifications. The latter solutions requires that the alternative path is added to the zFilter in the packet header, thus increasing the fill factor of the zFilter, increasing the probability of false positives. However, the solution does not require any signalling when the new path is needed.

### 2.2.3 Loop prevention

The possibility for false positives means that there is a risk for loops in the network. The loop avoidance has also been discussed in [8] with some initial mechanisms for avoiding such loops. Locally, it is possible to calculate zFilter that do not contain loops, but when the packet is passed to another administrative domain, it is not necessarily possible. One other alternative is to use TTL-like field in the packet for removing looping packets. Work is going on in this area.

## 2.3 Control messages, slow path, and services

To inject packets to the slow path on forwarding nodes, each node can be equipped with a local, unique Link ID denoting the node-internal passway from the switching fabric to the control processor. That allows targeted control messages that are passed only to one or a few nodes. Additionally, there may a be virtual Link ID attached to these node-local passways, making it possible to multicast control messages to a number of forwarding nodes without needing to explicitly name each of them.

By default such control messages would be simultaneously passed to the slow path and forwarded to the neighboring nodes. The simultaneous forwarding can be blocked easily, either by using zFilters constructed for node-to-node communication, or using a virtual Link ID that is both configured to pass messages to the slow path and to block them at all the outgoing links.

Generalising, we make the observation that the egress points of a virtual link can be basically anything: nodes, processor cards within nodes, or even specific services. This allows our approach to be extended to upper layers, beyond forwarding, if so desired.

# 3. IMPLEMENTATION

In the project, we have designed a publish/subscribe based networking architecture with a novel forwarding mechanism. The motivation to choose NetFPGA as the platform for our forwarding node implementation was based on our requirements. We needed a platform that was capable for high data rates and has the flexibility that allows implementation of a completely new forwarding functionality.

The current implementation has roughly 500 lines of Verilog code, and it implements most of the functions described in the previous section. In this section, we will go deeper in the implementation and describe what changes we have made to the original reference implementation.

## 3.1 Basic forwarding method

**Algorithm 1**: Forwarding method of LIPSIN

**Input**: Link IDs of the outgoing links; zFilter in the packet header

```
foreach Link ID of outgoing interface do
    if zFilter & Link ID == Link ID then
        Forward packet on the link
    end
end
```

The core operation of our forwarding node is to make the forwarding decision for incoming packets. With zFilters, the decision is based on a binary AND and comparison operations, both of which are very simple to implement in hardware. The forwarding decision

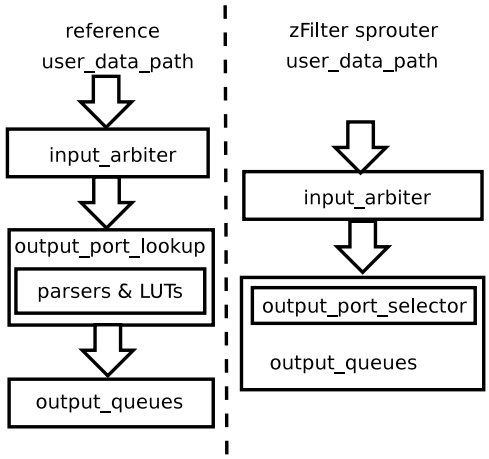



**Figure 4: Reference and modified datapaths**

(Alg. 1) can be easily parallelized, as there are no memory or other shared resource bottlenecks. The rest of this section describes the implementation based on this simple forwarding operation.

## 3.2 Forwarding node

For the implementation work we identified all unnecessary parts from the reference switch implementation and removed most of the code that is not required in our system (Figure 4). The removed parts were replaced with a simple zFilter switch.

The current version implements both the LIT and the virtual link extensions, and it has been tested with four real and four virtual LITs per each of the four interface. We are using our own *EtherType* for identifying zFilter packets. The implementation drops incoming packets with wrong ethertype, invalid zFilter, or if the TTL value has decreased down to zero.

The *output_port_lookup* module and all modules related to that are removed from the reference switch design. The zFilter implementation is not using any functions from those modules.

Our prototype has been implemented mainly in the new *output_port_selector* module. This module is responsible for the zFilter matching operations, including binary AND operation between the LIT and zFilter, and comparing the result with the LIT, as well as placing the packets to the correct output queues based on the matching result. The new module is added in *output_queueus*. Detailed structure of the *output_port_selector* module is shown in Figure 5.

### 3.2.1 Packet forwarding operations

All incoming data is forwarded, straight from the input arbiter, to the *store_packet* module, that stores it into the SRAM and to the *output_port_selector* mod-

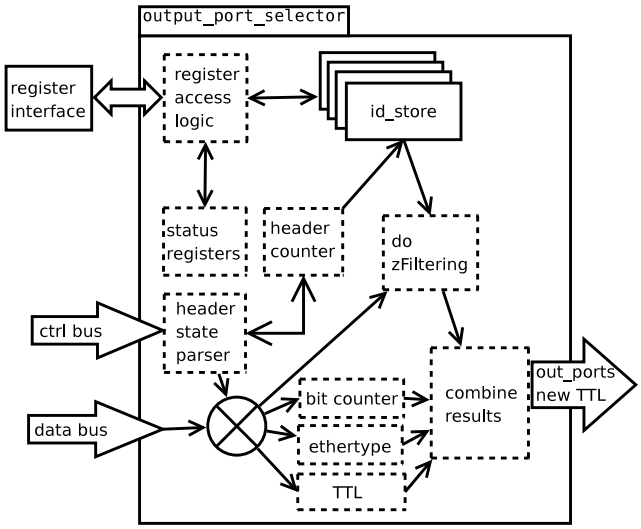



**Figure 5: Structure of the output port selector module**

ule for processing. Packets arrive in 64-bit pieces, one piece on each clock cycle. The packet handling starts with initiating processing for different verifications on the packet as well as on the actual zFilter matching operation.

The incoming packet processing takes place in various functions, where different kinds of verifications are performed to the packet. The three parallelized verification operations, *bit_counter*, *ethertype*, and *TTL*, make sanity checks on the packet, while the *do zFiltering* makes the actual forwarding decisions. In practice, for enabling parallelization, there exists separate instances of logic blocks that do zFiltering, one for each of the Link IDs (both for ordinary and virtual links). In the following, we go through the functions in Figure 5 function-by-function and in Figure 6, the operations are shown in function of clock cycles.

In *do zFiltering*, we make the actual zFilter matching for each 64-bit chunk. First, we select the correct LITs of each of the interfaces based on the d-value in the incoming zFilter. For maintaining the forwarding decision status for each of the interfaces during the matching process, we have a bit-vector where each of the interfaces has a single bit assigned, indicating the final forwarding decision. Prior to matching process, all the bits are set to one. During the zFilter matching, when the system notices that there is a mismatch in the comparison between the AND-operation result and the LIT, the corresponding interface's bit in the bit-vector is set to zero.

Finally, when the whole zFilter has been matched with the corresponding LITs, we know the interfaces where the packet should be forwarded by checking from the bit-vector, which of the bits are still ones. While the forwarding decision is also based on the other verifications on the packet, the *combine results* collects the information from the three verification functions in addition to the zFilter matching results. If all the collected verification function results indicate positive forwarding decision, the packet will be put to all outgoing queues indicated by the bit vector. The detailed operations of the verification functions are described in 3.2.2.

### 3.2.2 Support blocks and operations

To avoid the obvious attack of setting all bits to one in the zFilter, and delivering the packet to all possible nodes in the network, we have implemented a very simple verification on the zFilter. We have limited the maximum number of bits set to one in a zFilter to a constant value, which is configurable from the user space; if there are more bits set to one than the set maximum value, the packet is dropped. The bit counting function calculates the number of ones in a single zFilter and it is implemented in the *bit_counter* module. This module takes 64 bits wide input and it returns the amount of ones on the given input. Only wires and logic elements are used to calculate the result and there are no registers inside, meaning that block initiating the operation should take care of the needed synchronization.

The *Ethertype* of the packet is checked upon arrival. At the moment, we are using 0xacdc as the ethertype, identifying the zFilter-based packets. However, in a pure zFilter based network, the ethernet is not necessarily needed, thus this operation will be obsolete. The third packet checking operation is the verification of the TTL. This is used in the current implementation to avoid loops in the network. This is not an optimal solution for loop prevention, and better solutions are currently being worked on.

The *id_store* module implements Dual-Port RAM functionality making it possible for two processes to access it simultaneously. This allows modifications to LIT:s without blocking forwarding functionality. The *id_store* module is written so that it can be synthesized by using either logic cells or BRAM (Block RAM). One of the ports is 64 bits wide with only read access and it is used exclusively to get IDs for zFiltering logic. There is one instance of the *id_store* module for each LIT and virtual LIT. This way the memory is distributed and each instance of the filtering logic have access to *id_store* at line rate. The other port of the *id_store* module is 32 bits wide with a read and write connection for the user space access. This port is used by the management software (cf. Section 3.2.3) to configure LIT:s on the interfaces. One new register block is reserved for this access.

One additional register block is added for module control and debug purposes. It is used to read information that is collected into the status registers during forwarding operations. Status registers contain constants,

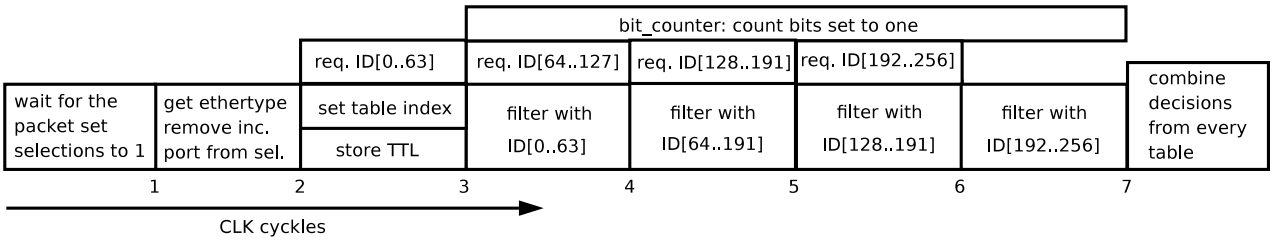



**Figure 6: Dataflow diagram**

amount of links, maximum amount of LITs and virtual LITs per link and also the LIT length. In addition, information about the last forwarded packet is stored together with the result of the bit count operation, d, TTL, and incoming port information. This block is also used to set the maximum amount of ones allowed in a valid zFilter.

### 3.2.3 *Management software*

For configuration and testing purposes, we have developed a specialized management software. When the system is started, the management software is used to retrieve information from the card and, if needed, to configure new values on the card. The information that the software can handle, includes the length of the LITs, the maximum $d$ value describing the number of LITs used, as well as both link and virtual link information from each of the interfaces.

Internally, the software works by creating chains of commands that it sends in batch to the hardware, gets the result and processes the received information. The commands are parsed using specific, for the purpose generated, grammar. The parsing is done using byacc and flex tools, and is therefore easily extendable.

For testing purposes, the software can be instructed to send customizable packets to the NetFPGA card, and to collect information about the made forwarding decisions. The software supports the following features:

- Selecting the outgoing interface
- Customizing the delay between transmitted packets
- Varying the sizes of packets
- Defining the Time-to-live (TTL) field in packet header
- Defining the $d$ value in packet header
- Defining the zFilter in the packet header
- Defining the ethernet protocol field

| # of NetFPGAs | Average latency | Std. Dev. | Latency/ NetFPGA |
|---|---|---|---|
| 0 | $16\mu s$ | $1\mu s$ | N/A |
| 1 | $19\mu s$ | $2\mu s$ | $3\mu s$ |
| 2 | $21\mu s$ | $2\mu s$ | $3\mu s$ |
| 3 | $24\mu s$ | $2\mu s$ | $3\mu s$ |

**Table 1: Simple latency measurement results**

## 4. EVALUATION

The basic functionality is tested by running simple scripts that use control software (cf. Section 3.2.3) to set Link IDs and to generate and send traffic. In practise, two network interfaces of the test host are connected to the NetFPGA of which one is used to send packets to the NetFPGA and the other one to receive forwarded packets. Forwarding decisions are also followed by tracking status registers. The results of the tests show that the basic forwarding functions work on the NetFPGA, also when using LITs. In addition, the packet verification operations, counting set bits, TTL verification, as well as ethertype checking were working as expected.

### 4.1 Performance

To get some understanding of the potential speed, we measured packet traversal times in our test environment. The first set of measurements, shown in Table 1, focused on the latency of the forwarding node with a very low load. For measurements, we had four different setups, with zero (direct wire) to three NetFPGAs on the path. Packets were sent at the rate of 25 packets/second; both sending and receiving operations were implemented directly in FreeBSD kernel.

The delay caused by the Bloom filter matching code is $64ns$ (8 clock cycles), which is insignificant compared to the measured $3\mu s$ delay of the whole NetFPGA processing. With background traffic, the average latency per NetFPGA was increased to $5\mu s$.

To get some practical reference, we also compared our implementation with the Stanford reference router. This was quantified by comparing ICMP echo requests' processing times with three setups: using a plain wire,

| Path | Avg. latency | Std. Dev. |
|---|---|---|
| Plain wire | $94\mu s$ | $28\mu s$ |
| IP router | $102\mu s$ | $44\mu s$ |
| LIPSIN | $96\mu s$ | $28\mu s$ |

**Table 2: Ping through various implementations**

using our implementation, and using the reference IP router with five entries in the forwarding table. To compensate the quite high deviation, caused by sending and receiving ICMP packets and involving user level processing, we averaged over 100 000 samples. Both IP router implementation and our implementation were run on the same NetFPGA hardware. The results are shown in Table 2.

While we did not directly measure the bandwidth due to the lack of test equipment for reliably filling up the pipes, there are no reasons why the implementation would not operate at full bandwidth. To further test this we did send video stream through our implmentation. During the streaming we did send random data through same NetFPGA but with different ports at almost 1Gbs datarate. Both the stream and was forwarded without a problem. The code is straightforward and should be able to keep the pipeline full under all conditions.

IP routers need increasing amount of states, which increases latency and resource consumption, when the size of the network increases. On the other hand, in our implementation, the latency and resource consumption for each node will remain same, independent of the amount of nodes in the network. Because of that, the results we got for one node should remain same even when large amount of nodes are connected to a same network.

### 4.2 Resource consumption:

To get an idea how much our implementation consumes resources we did syntetize design with 4 real and 4 virtual LITs per interface. With this configuration, the total usage of NetFPGA resources for the forwarding logic is 4.891 4-input LUTs out of 47.232, and 1.861 Slice Flip/Flops (FF) out of 47.232. No BRAMs are reserved for the forwarding logic. Synthetisizer saves BRAM blocks and uses other logic blocks to create registers for LITs. For the whole system, the corresponding numbers are 20.273 LUTs, 15.347 FFs, and 106 BRAMs. SRAM was used for the output queues in the measured design. We also tested to use BRAMs for output queue and the design works. However, we don't have measurement results from that implementation.

### 4.3 Forwarding table sizes:

Assuming that each forwarding node maintains $d$ distinct forwarding tables, each containing an entry per interface, where an entry further consists of a Link ID and the associated output port, we can estimate the amount of memory needed by the forwarding tables:

$$FT_{mem} = d \cdot \#Links \cdot [size(LIT) + size(P_{out})] \quad (1)$$

Considering $d = 8$, 128 links (physical & virtual), 248-bit LITs and 8 bits for the outport, the total memory required would be $256Kbit$, which easily fits on-chip.

Although this memory size is already small, we can design an even more efficient forwarding table by using a *sparse representation* to store just the positions of the bits set to 1. Thereby, the size of each LIT entry is reduced to $k \cdot log_2(LIT)$ and the total forwarding table requires only $\approx 48Kbit$ of memory, at the expense of the decoding logic.

## 5. RELATED WORK

OpenFlow [11] [12] provides a platform for experimental switches. It introduces simple, remote controlled, flow based switches, that can be run on existing IP switches. The concept allows evaluation of new ideas and even protocols in Openflow-enabled networks, where the new protocols can be run on top of the IP network. However, as high efficiency is one of our main goals, we wanted to get rid of unnecessary logic and decided for a native zFilter implementation.

There are not yet many publications where NetFPGA is used, in addition to OpenFlow and publications about implementing the NetFPGA card or reference designs. However one technical report were available [10], about implementing flow counter on NetFPGA. Authors of that work used NetFPGA succesfully to demonstrate that their idea can be implemented in practice.

In addition to NetFPGA, there are also other reconfigurable networking hardware approaches. For instance, [9] describes one alternative platform. There are also other platforms that could work for this type of development, for example Combo cards from Liberouter project [1]. However, NetFPGA provides enough speed and resources for our purposes, but Combo cards might become a good option later on if we need higher line speeds.

In the following we briefly discuss some work in the area of forwarding related to our zFilter proposal.

**IP multicast:** Our basic communication scheme is functionally similar to IP-based source specific multicast (SSM) [6], with the IP multicast groups having been replaced by the topic identifiers. The main difference is that we support stateless multicast for sparse subscriber groups, with unicast being a special case of multicast. On the contrary, IP multicast typically creates lots of state in the network if one needs to support a large set of small multicast groups.

**Networking applications of Bloom filters:**

For locating named resources, BFs have been used to bias random walks in P2P networks [3]. In content-based pub/sub systems [7], summarized subscriptions are created using BFs and used for event routing purposes. Bloom filters in packet headers were proposed in Icarus [14] to detect routing loops, in [15] for credentials-based data path authentication, and in [13] to represent AS-level paths of multicast packets in a 800-bit shim header, `TREE_BF`. Moreover, the authors of [13] use Bloom filters also to aggregate active multicast groups inside a domain and compactly piggyback this information in BGP updates.

## 6. CONCLUSIONS

Previously, we have proposed a new forwarding fabric for multicast traffic. The idea was based on reversing Bloom filter thinking and placing a Bloom filter into the delivered data packets. Our analysis showed that with reasonably small headers, comparable to those of IPv6, we can handle the large majority of Zipf-distributed multicast groups, up to some 20 subscribers, in realistic metropolitan-sized topologies, without adding any state in the network and with negligible forwarding overhead. For the remainder of traffic, the approach provides the ability to balance between stateless multiple sending and stateful approaches. With the stateful approach, we can handle dense multicast groups with a very good forwarding efficiency. The forwarding decisions are simple, potentially energy efficient, may be parallelized in hardware, and have appealing security properties.

To validate and test those claims, we implemented a prototype of a forwarding node and tested its performance. As described in Chapter 4, we ran some measurements on the forwarding node and concluded that the whole NetFPGA processing for a zFilter creates a $3\mu s$ delay. This delay could most likely be reduced, because the forwarding operation should take only $64ns$ (8 clock cycles). Comparison with the IP router implementation was done by using ICMP echo requests, showing zFilter implementation being slightly faster than IP-based forwarding, running on the same NetFPGA platform.

Our simple implementation still lacks some of the advanced features described in [8], for example reverse path creation and signaling. However, it should be quite straightforward to add those features to the existing design. Also, early studies indicate that it should be possible to start adding even more advanced features, like caching, error correction or congestion control to the implementation.

## 7. REFERENCES


[1] Liberouter. http://www.liberouter.org/.

[2] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[3] A. Z. Broder and M. Mitzenmacher. Survey: Network applications of Bloom filters: A survey. *Internet Mathematics*, 1:485–509, 2004.

[4] J. Day. *Patterns in Network Architecture: A Return to Fundamentals*. Prentice Hall, 2008.

[5] S. E. Deering and D. Cheriton. Multicast routing in datagram internetworks and extended lans. *ACM Trans. on Comp. Syst.*, 8(2), 1990.

[6] H. Holbrook and B. Cain. Source-specific multicast for IP. RFC 4607. Aug 2006.

[7] Z. Jerzak and C. Fetzer. Bloom filter based routing for content-based publish/subscribe. In *DEBS '08*, pages 71–81, New York, NY, USA, 2008. ACM.

[8] P. Jokela, A. Zahemszky, C. Esteve, S. Arianfar, and P. Nikander. LIPSIN: Line speed publish/subscribe inter-networking. Technical report, www.psirp.org, 2009.

[9] J. W. Lockwood, N. Naufel, J. S. Turner, and D. E. Taylor. Reprogrammable network packet processing on the field programmable port extender (FPX). In *Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays*, 2001.

[10] J. Luo, Y. Lu, and B. Prabhakar. Prototyping counter braids on netfpga. Technical report, 2008.

[11] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, J. Turner, and S. Shenker. Openflow: Enabling innovation in campus networks. In *ACM SIGCOMM Computer Communication Review*, 2008.

[12] J. Naous, D. Erickson, G. A. Covington, G. Appenzeller, and N. McKeown. Implementing an openflow switch on the netfpga platform. In *Symposium On Architecture For Networking And Communications Systems*, 2008.

[13] S. Ratnasamy, A. Ermolinskiy, and S. Shenker. Revisiting IP multicast. In *Proceedings of ACM SIGCOMM'06*, Pisa, Italy, Sept. 2006.

[14] A. C. Snoeren. Hash-based IP traceback. In *SIGCOMM '01*, pages 3–14, New York, NY, USA, 2001. ACM.

[15] T. Wolf. A credential-based data path architecture for assurable global networking. In *Proc. of IEEE MILCOM*, Orlando, FL, Oct 2007.

[16] A. Zahemszky, A. Csaszar, P. Nikander, and C. Esteve. Exploring the pubsub routing/forwarding space. In *International Workshop on the Network of the Future*, 2009.

# IP-Lookup with a Blooming Tree Array: A New Lookup Algorithm for High Performance Routers

Gianni Antichi gianni.antichi@iet.unipi.it
Andrea Di Pietro andrea.dipietro@iet.unipi.it
Domenico Ficara domenico.ficara@iet.unipi.it
Stefano Giordano s.giordano@iet.unipi.it
Gregorio Procissi g.procissi@iet.unipi.it
Cristian Vairo cristian.vairo@iet.unipi.it
Fabio Vitucci fabio.vitucci@iet.unipi.it

## ABSTRACT

Because of the rapid growth of both traffic and links capacity, the time budget to perform IP address lookup on a packet continues to decrease and lookup tables of routers unceasingly grow. Therefore, new lookup algorithms and new hardware platform are required. This paper presents a new scheme on top of the NetFPGA board which takes advantage of parallel queries made on perfect hash functions. Such functions are built by using a very compact and fast data structure called Blooming Trees, thus allowing the vast majority of memory accesses to involve small and fast on-chip memories only.

## Keywords

High Performance, IP Address Lookup, Perfect Hash, Bloom Filters, FPGA

## 1. INTRODUCTION

The primary task of a router is the IP-address lookup: it requires that a router looks, among possibly several thousands of entries, for the best (i.e., the longest) rule that matches the IP destination address of the packet. The explosive growth of Internet traffic and link bandwidth forces network routers to meet harder and harder requirements. Therefore, the search for the Longest Prefix Match (LPM) in the forwarding tables has now become a critical task and it can result often into the bottleneck for high performance routers. For this reason a large variety of algorithms have been presented, trying to improve the efficiency and speed of the lookup.

The algorithm here proposed is based on data structures called Blooming Trees (hereafter BTs) [8], compact and fast techniques for membership queries. A BT is a Bloom Filter based structure, which takes advantage of low false positive probability in order to reduce the mean number of memory accesses. Indeed, the number of required memory accesses is one of the most important evaluation criterion for the quality of an algorithm for high performance routers, given that it strongly influences the mean time required for a lookup process.

An array of parallel BTs accomplishes the LPM function for the entries of the forwarding table by storing the entries belonging to the 16–32 bit range. Every BT has been configured according to the Minimal Perfect Hash Function (MPHF) [1], a scheme conceived to obtain memory efficient storage and fast item retrieval. Shorter entries, instead, are stored in a very simple Direct Addressing (DA) logical block. DA module uses the address itself (in this case only the 15 most significant bits) as on offset to memory locations.

The implementation platform for this algorithm is the NetFPGA [12] board, a new networking hardware which proves to be a perfect tool for research and experimentation. It is composed of a full programmable Field Programmable Gate Array (FPGA) core, four Gigabit Ethernet ports and four banks of Static and Dynamic Random Access Memories (S/DRAM).

This work is focused on the data-path implementation of the BT-based algorithm for fast IP lookup. The software control plane has been also modified in order to accommodate the management and construction of the novel data structure. The software modifications merges perfectly in the preexistent *SCONE* (Software Component of the NetFPGA).

The rest of the paper is organized as follows: after the related work in address lookup area, section 3 illustrates the main idea, the overall algorithm and the data structures of our scheme. Then section 4 shows the actual implementation of our algorithm on NetFPGA while section 5 presents the modifications in the control plane software. Finally, section 6 shows the experimental results and section 7 ends the paper.

## 2. RELATED WORK

Due to its essential role in Internet routers, IP lookup is a well investigated topic, which encompasses trie-based schemes as well as T-CAM solutions and hashing

techniques. Many algorithms have been proposed in this area ([4][6][9][10][14][15]); to the best of our knowledge, the most efficient trie-based solutions in terms of memory consumption and lookup speed are Lulea and Tree Bitmap.

Lulea [4] is based on a data structure that can represent large forwarding tables in a very compact form, which is small enough to fit entirely in the L1/L2 cache of a PC Host or in a small memory of a network processor. It requires the prefix trie to be complete, which means that a node with a single child must be expanded to have two children; the children added in this way are always leaf nodes, and they inherit the next-hop information of the closest ancestor with a specified next-hop, or the undefined next hop if no such ancestor exists. In the Lulea algorithm, the expanded unibit prefix trie denoting the IP forwarding table is split into three levels in a 16-8-8 pattern. The Lulea algorithm needs only 4-5 bytes per entry for large forwarding tables and allows for performing several millions full IP routing lookups per second with standard general purpose processors.

Tree Bitmap [6] is amenable to both software and hardware implementations. In this algorithm, all children nodes of a given node are stored contiguously, thus allowing for using just one pointer for all of them; there are two bitmaps per node, one for all the internally stored prefixes and one for the external pointers; the nodes are kept as small as possible to reduce the required memory access size for a given stride (thus, each node has fixed size and only contains an external pointer bitmap, an internal next hop info bitmap, and a single pointer to the block of children nodes); the next hops associated with the internal prefixes kept within each node are stored in a separate array corresponding to such a node. The advantages of Tree Bitmap over Lulea are the single memory reference per node (Lulea requires two accesses) and the guaranteed fast update time (an update of the Lulea table may require the entire table to be almost rewritten).

A hardware solution for the lookup problem is given by CAMs, which minimize the number of memory accesses required to locate an entry. Given an input key, a CAM device compares it against all memory words in parallel; hence, a lookup actually requires one clock cycle only. The widespread use of address aggregation techniques like CIDR requires storing and searching entries with arbitrary prefix lengths. For this reason, TCAMs have been developed. They could store an additional *Don't Care* state thereby enabling them to retain single clock cycle lookups for arbitrary prefix lengths. This high degree of parallelism comes at the cost of storage density, access time, and power consumption. Moreover TCAMs are expensive and offer little adaptability to new addressing and routing protocols [2].

Therefore, other solutions which use tree traversal and SRAM-based approach are necessary. For example, the authors of [11] propose a scalable, high-throughput SRAM-based dual linear pipeline architecture for IP Lookup on FPGAs, named DuPI. Using a single Virtex-4, DuPI can support a routing table of up to 228K prefixes. This architecture can also be easily partitioned, so as to use external SRAM to handle even larger routing tables, maintains packet input order, and supports in-place nonblocking route updates.

Other solutions take advantage of hashing techniques for IP lookup. For instance, Dharmapurikar et al. [5] use Bloom Filters (BFs) [3] for longest prefix matching. Each BF represents the set of prefixes of a certain length, and the algorithm performs parallel queries on such filters. The filters return a yes/no match result (with false positives), therefore the final lookup job is completed by a priority encoder and a subsequent search in off-chip hash tables. Instead, in our scheme, we will use BF-like structures which have been properly modified in order to directly provide an index for fast search.

## 3. THE ALGORITHM

All the algorithms previously described remark the most important metrics to be evaluated in a lookup process: lookup speed, mean number of memory access and update time. Each of the cited solutions tries to maximize general performance, with the aim of be implemented on a high performance router and obtain *line–rate* speed. The main motivations for this work come from the general limitations for high–performance routing hardware: limited memory and speed. Specifically, because of the limited amount of memory available, we adopt a probabilistic approach, thus reducing the number of external memory accesses also.

Because of the large heterogeneity of real IP prefixes distribution (as shown in several works as [5] and [13]), our first idea is to divide the entire rule database into two large groups, in order to optimize the structure:

- the prefixes of length $\leq 15$, which are the minority of IP prefixes , are simply stored in a Direct Addressing array; this solution is easily implemented in hardware and requires an extremely low portion of the FPGA logic area;
- the prefixes of length $\geq 16$ are represented by an array of Blooming Trees (hereafter called BT-array).

In the lookup process, the destination address under processing is hashed and the output is analyzed by the BT-array and the DA module *in parallel* (see fig. 1). Finally, an *Output Controller* compares the results of both modules and provides the right output (i.e., the longest matching), which is composed of a next-hop ad-

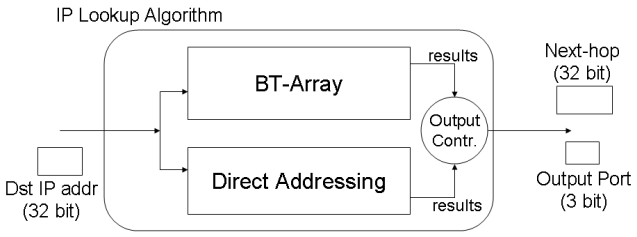



**Figure 1: The overall IP lookup scheme.**

dress (32 bits) and an output port number (3 bits, given that the NetFPGA has 8 output ports).

In the BT-array the prefixes are divided into groups based on their lengths and every group is organized in an MPHF structure (as shown in fig. 2). Therefore, the BT-array is an array where 17 parallel queries are conducted at the same time; at the end of the process, a bus of 17 wires carries the results: a wire is set to 1 if there is a match in the corresponding filter. Then a priority encoder collects the results of the BT-array and takes the longest matching prefix, while a SRAM query module checks the correctness of the lookup (since BTs are probabilistic filters in which false positives can happen).

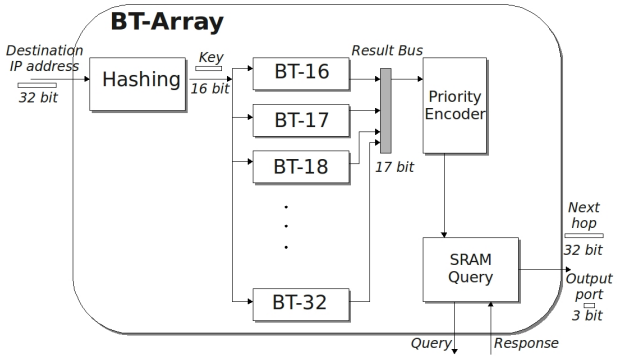



**Figure 2: BT-array schematic.**

## 3.1 Blooming Tree for MPHF

The structure we adopt to realize a Minimal Perfect Hashing Functions is a Blooming Tree [8], which is based on the same principles of Bloom Filters and allows for a further memory reduction. The idea of BT is constructing a binary tree upon each element of a plain Bloom Filter, thus creating a multilayered structure where each layer represents a different depth-level of tree nodes.

A Blooming Tree is composed of $L+1$ layers:

- a plain BF ($B_0$) with $k_0$ hash functions $h_j$ ($j = 1 \dots k_0$) and $m$ bins such that $m = nk_0/\ln 2$ (in order to minimize the false positive probability);
- $L$ layers ($B_1 \dots B_L$), each composed of $m_i$ ($i = 1 \dots L$) blocks of $2^b$ bits.

Just as a BF, $k_0$ hash functions are used. Each of them provides an output of $\log_2 m + L \times b$ bits: the first group of $\log_2 m$ bits addresses the BF at layer 0, while the other $L \times b$ bits are used for the upper layers. The lookup for an element $\sigma$ consists of a check on $k_0$ elements in the BF (layer 0) and an exploration of the corresponding $k_0$ "branches" of the Blooming Tree.

"Zero-blocks" (i.e., blocks composed of a string of $b$ zeros which are impossible to be found in a naive BT for construction) are used to stop the "branch" from growing as soon as the absence of a collision is detected in a layer, thus saving memory. This requires additional bitmaps to be used in the construction process only. For more details about BTs, refer to [8].

Our MPHF on BT is based on the statement that, taken the BT as ordering algorithm (with $k_0$=1), a MPHF of an element $x \in S$ ($S$ is a set of elements) is simply the position of $x$ in the BT:

$$\mathrm{MPHF}(x) = \underset{S,BT}{\mathrm{position}}(x) \qquad (1)$$

All we need to care when designing this structure is that, in the construction phase, all the collisions vanish, in order to achieve a perfect function.

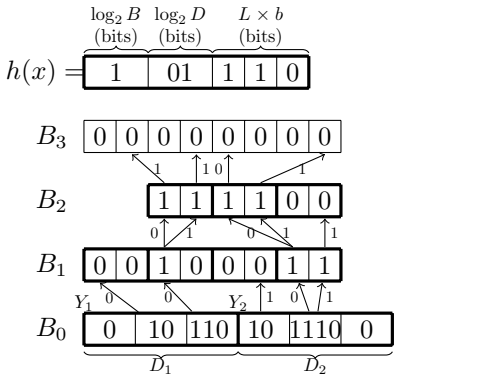
Instead, as for the lookup, the procedure that finds the position of an element $x$ is divided into two steps:

- find the tree which $x$ belongs to (we call it $T_x$) and compute the number of elements at the $T_x$'s left;
- compute the leaves at the left of $x$ in $T_x$.

In order to simplify the process, we propose the HSBF [7] as the first level of the BT, instead of the standard BF. The HSBF is composed of a series of bins encoded by Huffman coding, so that a value $j$ translates into $j$ ones and a trailing zero. Therefore, the first step of the procedure is accomplished by a simple popcount in the HSBF of all the bins at the left of $x$'s bin. As for the second step, we have to explore (from left to right) the tree $T_x$ until we find $x$, thus obtaining its position within the tree. The sum of these two components gives the hash value to be assigned. For more details about a MPHF realized by means of BT, refer to [1].

A simple example (see fig. 3) clarifies the procedure: we want to compute the MPHF value of the element $x$. In order to simplify the search in the HSBF, this filter is divided into $B$ sections of $D$ bins, which are addressed through a lookup table. Let us assume $B = 2$, $D = 3$, and $b = 1$: hence, the hash output is 6-bits long.

Let us suppose $h(x) = 101110$. The first bit is used to address the lookup table: it points to the second entry. We read the starting address of section $D_2$ and




Lookup Table

| Start addr. | Prev.elements | Prev. "10"s |
|---|---|---|
| $D_1$ | 0 | 0 |
| $D_2$ | 3 | 2 |

**Figure 3: Example of hash retrieval through BT.**

that 3 elements are in the previous sections (i.e., already assigned by the MPHF). Now we use the next two bits of $h(x)$ to address the proper bin in section $D_2$: "01" means the second bin. The popcount on the previous bins in section $D_2$ indicates that another element is present (so far the total number of elements at $T_x$'s left is 4).

Then we focus on $T_x$: to move up to the next layer, we both use the third information in the table (the number of '10's in previous sections, which is 2) and count the number of "10"s in the previous bins of this section (that is 1). The sum shows that, before our bin, 3 bins are not equal to 0, so we move to the fourth block in layer $B_1$.

Here, the fourth bit of $h(x)$ allows to select the bit to be processed: the second one. But we want to know all the $T_x$'s leaves at $x$'s left, hence we have to explore all the branches belonging to the bin under processing. So we start from the first bit of the block and count the number of zero-blocks we find: 2, at layer $B_3$. Now the counter reads 6.

Regarding the second bit of the block (which is "the bit of $x$"), a popcount in layer $B_1$ indicates the third block in layer $B_2$: it is a zero-block, so we have found the block representing our element only: $x$ is the 7-th element in our ordering scheme. Then $\text{MPHF}(x) = 6$.

# 4. IMPLEMENTATION

## 4.1 MPHF Module

As above mentioned, the main component of the algorithm is the BT-array, which is composed of a series of MPHFs realized through BTs. Because of the large difficulties in allocating a variable–sized structure in hardware and for the sake of simplicity, in our implementation we simplify the scheme proposed in [1] and adopt a fixed-size structure. In details, the implemented structure presents 3 layers:

- Layer 0: a *Counting Bloom Filter* (CBF) composed of 128 *sections* and with 16 *bins* for every section;
- Layer 1: a simple bitmap that contains *two* bits for every bin of the level 0;
- Layer 2: another bitmap with *two* bits for every bit of the level 1; its size is then of 8192 bits.

These parameters (in terms of number of bins, sections and layers) are chosen in order to allocate, with a very low false positives probability, up to 8192 prefixes per prefix length, which implies that the total maximum number of entries is 128 thousands. Therefore, this implementation can handle even recent prefix rules databases and largely overcome the limitations of the simple (linear-search-based) scheme provided with the standard NetFPGA reference architecture.

Every bin of the CBF, according to the original idea in [1] and as shown in [7], is Huffman–encoded. Again, in order to simplify the hardware implementation, each bin consists of 5 bits and its length is fixed. Thus a maximum of 4 elements are allowed at level 0 for the same bin (i.e.: a trailing zero and max 4 bits set to 1). Since the probability of having bins with more than 4 elements is quite small (around $10^{-2}$) even if the structure is crowded, this implementation allows for a large number of entries to be stored.

Moreover, a lookup table is used to perform the lookup in the layer 0, which is composed of 128 rows containing the SRAM initial address for each section of the CBF. We place the entire BT and the lookup table in the fast BRAM memory: the CBF at layer 0 occupies a block of $2048 \times 5$ bits, while the lookup table has a BRAM block of $128 \times 20$ bits.

## 4.2 H3 Hash Function

The characteristics of the hash function to be used in the MPHF are not critical to the performance of the algorithm, therefore a function which ensures a fast hash logic has been implemented: the *H3* class of hardware hash functions.

Define $A$ as the *key* space (i.e. inputs) and $B$ as the *address* space (i.e. outputs):

- $A = 0, 1, ..., 2^i - 1$
- $B = 0, 1, ..., 2^j - 1$

where $i$ is the number of bits in the key and $j$ is the number of bits in the address. The H3 class is defined as follows: denote $Q$ as the set of all the $i \times j$ boolean matrices. For a given $q \in Q$ and $x \in A$, let $q(k)$ be the

$k$–th row of the matrix $q$ and $x_k$ the $k$–th bit of $x$. The hashing function $h_q(x) : A \rightarrow B$ is defined as:

$$h_q(x) = x_1 \cdot q(1) \oplus x_2 \cdot q(2) \oplus \ldots \oplus x_i \cdot q(i) \quad (2)$$

where $\cdot$ denotes the binary AND operation and $\oplus$ denotes the binary exclusive OR operation. The class H3 is the set $\{h_q | q \in Q\}$.

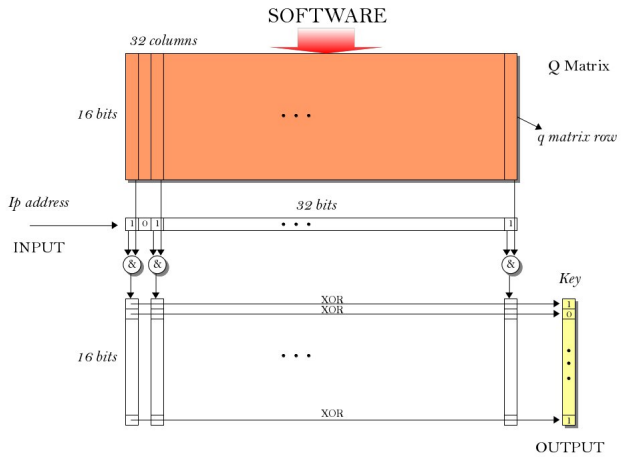



**Figure 4: The scheme of the hash function belonging to the H3 class. The Q matrix is stored in a BRAM block.**

Figure 4 shows our H3 design. The process is divided into two steps: first the 32-bit input value (i.e. the IP address) is processed by a block of AND, then the resulting 32 vectors of $n$ bits are XOR-ed by a matrix of logical operations and a resulting string of $n$ bits is provided as output.

The length of the output is chosen to meet the requirements of the following BT-array. The $q$ matrix is pre–programmed via software, passed through the PCI bus, and stored in a BRAM block.

## 4.3 Managing false positives

As already stated, a BT provides also a certain amount of false positives with probability $f$. Thus every lookup match has to be confirmed with a final lookup into SRAM. Then, intuitively, the average number of SRAM accesses $\overline{n}$ increases as $f$ grows. More formally, assuming all BTs in the BT-array have the same false positive probability $f$, we can write:

$$\overline{n} \leq 1 + \sum_{i=1}^{16} f^i \leq \frac{1}{1-f} \quad (3)$$

This equation takes into account the probability of the worst case that happens when all BTs provide false positives and are checked in sequence. As one can easily verify, even if $f$ is quite large, the average number of memory accesses is always close to 1 (less than 1.11 for $f = 0.1$).

# 5. CONTROL PLANE

The MPHF IP lookup algorithm needs a controller that manages the building of the database, the setup of the forwarding table, and the potential updates of the prefixes structures. All these functions have been implemented in C/C++ and integrated into the Software SCONE.

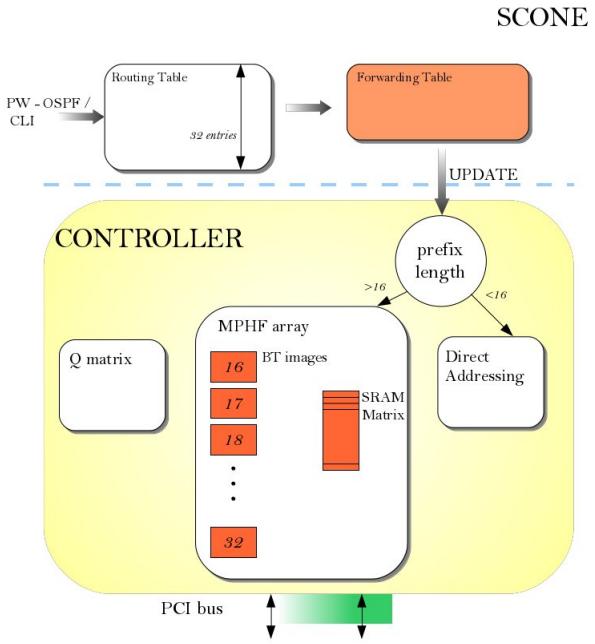



**Figure 5: In this figure the different functions of the software control plane can be seen.**

This software adopts PW-OSPF as routing protocol, which is a greatly simplified link–state routing protocol based on OSPFv2 (rfc 1247). Every time an update in the routing table occurs, a new forwarding table is created and passed entry by entry through the PCI bus to the NetFPGA. An entry in the forwarding table is composed of destination IP address, subnet mask, gateway, and output port. The original behavior in SCONE was to retransmit all the entries of the forwarding table for each update, while we transmit the modified (new, updated or deleted) entries only. Then the developed Controller analyzes these entries and modifies the proper structures. This communication takes advantage of the register bus.

## 5.1 Updates

When a modification in the forwarding table occurs, it may happen that the new elements lead to collisions in one of the MPHF structures (because they are limited to 2 layers only). In this case, the hash function has to be modified in order to avoid these collisions. This requires the change of the $Q$ matrix and its re-

transmission to the H3 hashing module, which stores the matrix in a BRAM block. The same communication protocol used for the construction is adopted, indeed a simple registers call has to be made. Therefore, a new hash function is used and all the data-structures for lookup are updated, thus obtaining MHPFs with no collisions.

## 6. RESULTS

In this section, the simulative results about the implementation of our algorithm are shown. In details, we focus on resource utilization, in terms of slices, 4-input LUTs, flip flops, and Block RAMs. We compare our results with those of the NetFPGA reference router where a simple linear search is implemented for IP lookup.

Table 1 shows the device utilization (both as absolute and relative figures) for the original NetFPGA lookup algorithm. It provides a simple lookup table which allows to manage 32 entries only to be looked for through a linear search. Instead we implement a more efficient and scalable algorithm, which is capable of handling up to 130000 entries (by assuming a uniform distribution for entries prefix length). This complexity is obviously paid in terms of resource consumption (see tab. 2): in particular, our lookup module uses 41% of the available slices on the Xilinx Virtex II pro 50 FPGA and 29% of the Block RAMs. However, as for the synthesis of the project, it is worth noticing that even though we use a wide number of resources, the timing closure is achieved without any need to re-iterate the project flow.

**Table 1: Resource utilization for the original lookup algorithm.**

| Resources | XC2VP50 Utilization | Utilization Percentage |
|---|---|---|
| Slices | 935 out of 23616 | 3% |
| 4-input LUTS | 1321 out of 47232 | 2% |
| Flip Flops | 343 out of 47232 | 0% |
| Block RAMs | 3 out of 232 | 1% |

**Table 2: Utilization for our algorithm.**

| Resources | XC2VP50 Utilization | Utilization Percentage |
|---|---|---|
| Slices | 9803 out of 23616 | 41% |
| 4-input LUTS | 10642 out of 47232 | 22% |
| Flip Flops | 19606 out of 47232 | 41% |
| Block RAMs | 68 out of 232 | 29% |

Tables 3 and 4 list the specific consumption of the main modules composing our project. In particular, the final synthesis summary is reported for both a single MPHF and the hashing module. It is interesting to observe the number of slices required for the hashing compared to that of the table 3: the hashing module ends up to be bigger than the whole MPHF module because of its complexity, since it has to provide 17 different hash values for each of the 17 MPHF blocks.

**Table 3: Utilization for a single MPHF.**

| Resources | XC2VP50 Utilization | Utilization Percentage |
|---|---|---|
| Slices | 398 out of 23616 | 1% |
| 4-input LUTS | 561 out of 47232 | 1% |
| Flip Flops | 618 out of 47232 | 1% |
| Block RAMs | 6 out of 232 | 2% |

**Table 4: Utilization for the hashing module.**

| Resources | XC2VP50 Utilization | Utilization Percentage |
|---|---|---|
| Slices | 1179 out of 23616 | 4% |
| 4-input LUTS | 1293 out of 47232 | 2% |
| Flip Flops | 1841 out of 47232 | 3% |

Finally, table 5 presents the overall device utilization for the reference router including our lookup algorithm and highlights the extensive use of the various resources. In particular we use 94% of the available Block Rams and 74% of slices and LUTs.

## 7. CONCLUSIONS

This paper presents a novel scheme to perform longest prefix matching for IP lookup in backbone routers. By following the real IP prefixes distributions, we divide the entire rule database into two large groups, in order to optimize our scheme. For prefixes of length $< 16$, which are the minority of IP prefixes, we use a simple Direct Addressing scheme, while for the others we use an array of Blooming Trees.

The implementation platform for this work is the NetFPGA board, a new networking tool which proves to be very suitable for research and experimentation.

**Table 5: Utilization for our overall project.**

| Resources | XC2VP50 Utilization | Utilization Percentage |
|---|---|---|
| Slices | 17626 out of 23616 | 74% |
| 4-input LUTS | 32252 out of 47232 | 74% |
| Flip Flops | 31512 out of 47232 | 66% |
| Block RAMs | 220 out of 232 | 94% |
| External IOBs | 360 out of 692 | 52% |

NetFPGA, originally, provides a simple lookup scheme which allows to manage 32 entries only by means of linear searches. Instead, our scheme is capable of handling up to 130000 entries at the cost of a bigger resource consumption. Anyway, the timing closure is achieved without any need to re-iterate the project flow.

## 8. ADDITIONAL AUTHORS

## 9. REFERENCES


[1] G. Antichi, D. Ficara, S. Giordano, G. Procissi, and F. Vitucci. Blooming trees for minimal perfect hashing. In *Global Telecommunications Conference, 2008. IEEE GLOBECOM 2008. IEEE*, pages 1–5, 30 2008-Dec. 4 2008.

[2] F. Baboescu, S. Rajgopal, L. B. Huang, and N. Richardson. Hardware implementation of a tree-based ip lookup algorithm for oc-768 and beyond. In *DesignCon*, 2006.

[3] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[4] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink. Small forwarding tables for fast routing lookups. In *Proc. of the ACM SIGCOMM '97*, pages 3–14, New York, NY, USA, 1997. ACM.

[5] S. Dharmapurikar, P. Krishnamurthy, and D. Taylor. Longest prefix matching using bloom filters. In *SIGCOMM 2003*.

[6] W. Eatherton, Z. Dittia, and G. Varghese. Tree bitmap: Hardware/software ip lookups with incremental updates. In *ACM SIGCOMM Computer Communications Review*, 2004.

[7] D. Ficara, S. Giordano, G. Procissi, and F. Vitucci. Multilayer compressed counting bloom filters. In *Proc. of IEEE INFOCOM '08*.

[8] D. Ficara, S. Giordano, G. Procissi, and F. Vitucci. Blooming trees: Space-efficient structures for data representation. In *Communications, 2008. ICC '08. IEEE International Conference on*, pages 5828–5832, May 2008.

[9] P. Gupta and N. Mckeown. Packet classification using hierarchical intelligent cuttings. In *in Hot Interconnects VII*, pages 34–41, 1999.

[10] T. V. Lakshman and D. Stiliadis. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. In *Proc. of SIGCOMM*, pages 203–214, 1998.

[11] H. Le, W. Jiang, and V. K. Prasanna. Scalable high-throughput sram-based architecture for ip-lookup using fpga. In *International Conference on Field Programmable Logic and Applications*, 2008.

[12] 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.

[13] *Route Views 6447, http://www.routeviews.org/.*

[14] S. Singh, F. Baboescu, G. Varghese, and J. Wang. Packet classification using multidimensional cutting, 2003.

[15] V. Srinivasan, S. Suri, and G. Varghese. Packet classification using tuple space search. In *Proc. of SIGCOMM*, pages 135–146, 1999.

# URL Extraction on the NetFPGA Reference Router

Michael Ciesla and Vijay Sivaraman
School of Electrical Engineering and Telecommunications
University of New South Wales, Sydney NSW 2052, Australia.
Emails: m.ciesla@student.unsw.edu.au, vijay@unsw.edu.au

Aruna Seneviratne
National ICT Australia (NICTA)
Sydney, Australia.
Email: aruna.seneviratne@nicta.com.au

***Abstract*— The reference router implementation on the NetFPGA platform has been augmented for real-time extraction of URLs from packets. URL extraction can be useful for application-layer forwarding, design of caching services, monitoring of browsing patterns, and search term profiling. Our implementation modifies the gateware to filter packets containing a HTTP GET request and sends a copy to the host. Host software is implemented to extract URLs and search terms. The software integrates with a database facility and a GUI for offline display of web-access and search term profiles. We characterise the link throughput and CPU load achieved by our implementation on a real network trace, and highlight the benefits of the NetFPGA platform in combining the superior performance of hardware routers with the flexibility of software routers. We also demonstrate that with relatively small changes to the reference router, useful applications can be created on the NetFPGA platform.**

## I. Introduction

The ability to view Uniform Resource Locators (URLs) corresponding to traffic flowing through a network device enables many diverse and interesting applications ranging from application layer switching and caching to search term visibility. Application-layer switching uses URLs to direct HTTP protocol requests to specialised servers that store less content, allowing for higher cache hit rates, and improved server response rates [1]. The basis of most caching system architectures is the interception of HTTP requests [2]. HTTP requests identify the objects that the system must store and are also used as index keys into storage databases. URLs also contain user search engine terms (popular search engines such as Google and Yahoo embed the search terms in the URLs). Through appropriate means, ISPs can partner with marketing companies to harvest search terms in order to generate new revenue streams from targeted advertising [3].

To our knowledge, there currently is no deep packet inspection (DPI) functionality, specifically for URL extraction, available on the NetFPGA platform. This project augments the NetFPGA reference router to analyse HTTP packets and extract URL and search term data. The NetFPGA user data path parses packet headers and payloads to identify HTTP GET packets and sends a copy to the host (in addition to forwarding the packet along its normal path). Software on the host displays the URLs and search terms on-screen in real-time. It also logs the URLs and search terms to a database, and a graphical user interface (GUI) has been developed to graphically profile web-page accesses (e.g. top-20 web-sites accessed) and search terms (e.g. to identify potential illegal activity).

## II. Architecture

Our hardware accelerated URL extraction system consists of two main components: hardware and software. The hardware component is an extended NetFPGA IPv4 reference router that filters packets containing a HTTP GET request method in hardware and sends a copy to the host. The software component is composed of three parts: URL Extractor (urlx), database, and a graphical user interface. The URL Extractor parses HTTP GET packets, extracts the contained URLs and search terms, and then stores them into a database. The GUI queries the database for top occurring URLs and search terms, and displays them on-screen. Fig. 1 shows a system diagram.

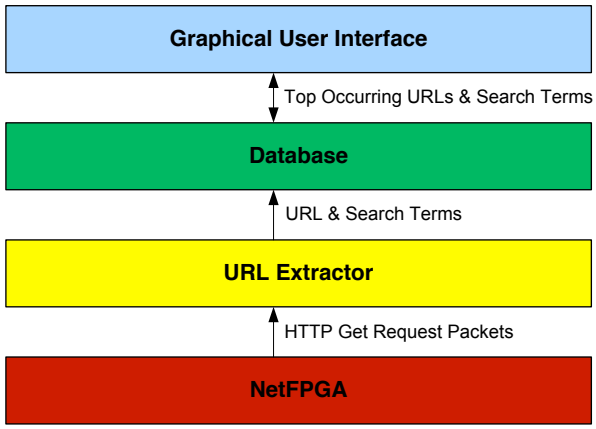



Fig. 1. System Diagram

### A. IPv4 Reference Router Modification

Our design modifies the Output Port Lookup module of the reference router. Fig. 2 shows the layout of the module with the addition of the new *http_get_filter* submodule. The *output_port_lookup.v* file has been altered to include the definition of the *http_get_filter* and its wire connections to the *preprocess_control* and *op_lut_process_sm* submodules.

The *http_get_filter* functions as a new preprocess block with the responsibility of identifying packets containing URLs. The HTTP protocol uses the GET request method to send URL requests to a server. Packets containing a GET request are

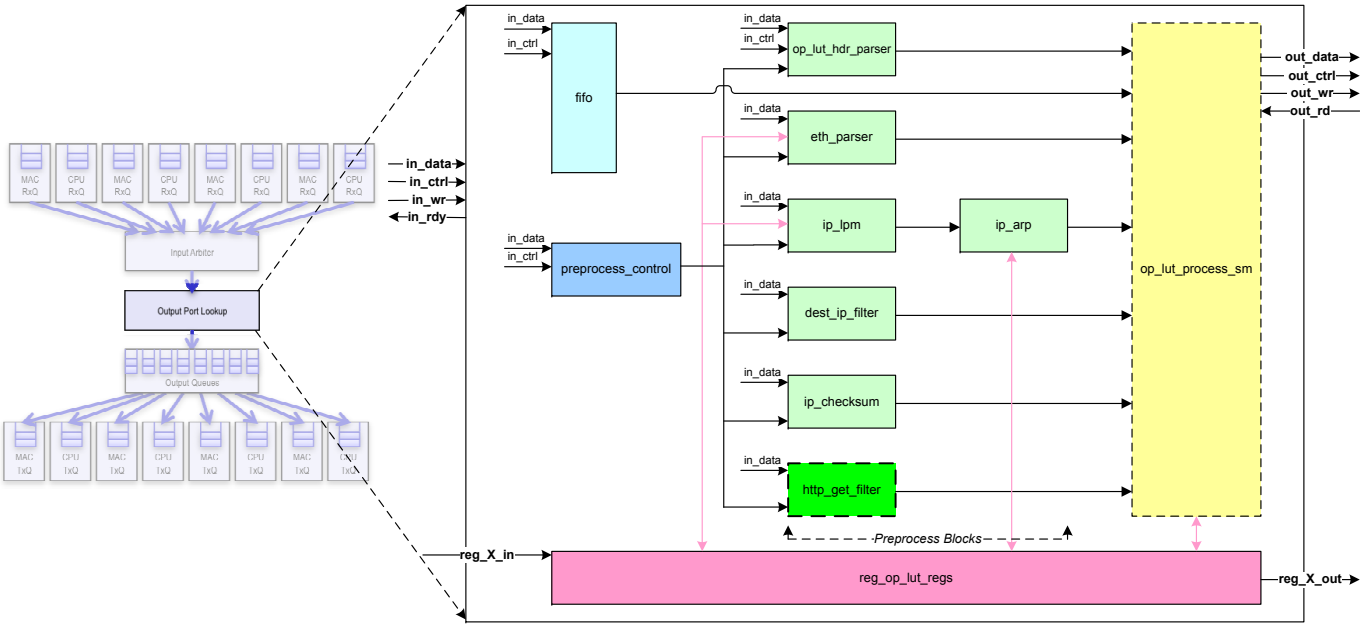



Fig. 2. Submodule layout of the modified Output Port Lookup. The *http_get_filter* is a new submodule and the *op_lut_process_sm* has been altered.

| | User Data Path (in_data) Register Bits | | | |
|---|---|---|---|---|
| **Words** | 63:48 | 47:32 | 31:16 | 15:0 |
| 1 | eth da | | | eth sa |
| 2 | eth sa | | type | ver, ihl,tos |
| 3 | total length | identification | flags, foff | ttl, proto |
| 4 | checksum | src ip | | dst ip |
| 5 | dst ip | src port | dst port | sequence |
| 6 | sequence | ack | | doff, flags |
| 7 | win size | checksum | urgent pointer | options |
| 8 | options | | | |
| 9 | | | **HTTP "GET"** | |

Fig. 3. NetFPGA word alignment for Unix GET packets. Fields shaded in red are inspected for GET packet identification.

distinguished by containing the "GET" string at the beginning of the TCP payload. In addition to checking for this string, identifying GET packets involves inspecting four other header fields (refer to Fig. 3). First, the packet length is checked to ensure its large enough to contain the "GET" string. Second, we check for the TCP protocol, which is used to transport HTTP. Third, the destination port is inspected for HTTP port numbers (our current implementation only checks for port 80). Fourth, the TCP header length is checked since it varies in size for GET packets originating from different operating systems. For example, Linux TCP headers include a 12-byte Option field that Windows does not. Consequently, this changes the location of the "GET" string, and extra state must be maintained to track whether the current packet being processed is potentially a Windows or Unix GET packet.

The identification of GET packets is implemented by the state machine shown in Fig. 4. By checking the above mentioned protocol header fields and for the occurrence of the "GET" string at the beginning of the TCP payload, the state machine carries out a seven stage elimination process of

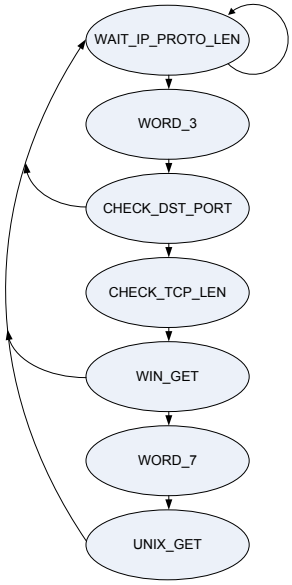



Fig. 4. Diagram of State Machine Used to Identify GET Packets

identifying a GET packet. The state machine initially idles in the *WAIT_IP_PROTO_LEN* state waiting for the IP packet length and protocol fields of the IP header to be present on the data bus. The *preprocess_control* signals the *http_get_filter* when this data is on the bus, and the elimination process is started. If any of the checks fail, the state machine resets to the *WAIT_IP_PROTO_LEN* state, and waits for a new packet. A FIFO is used to store the result of the GET packet check, which is later used by *op_lut_process_sm*.

The method used to check for the "GET" string varies between Windows and Unix packets. For Unix packets (we've tested Linux and Mac OS X operating systems), the string is located in bits 8 to 31 of the 9th word. This is shown in Fig. 3. Conversely, for Windows packets, there are no TCP Options and the string spans multiple words on the NetFPGA data bus. The letters "GE" are located in the first two bytes of the 7th word and the remaining letter "T" is stored in the first byte of the 8th word. To simplify the implementation, the *WIN_GET* state only checks for the "GE" letters and the *UNIX_GET* state checks for the whole "GET" string.

The *WORD_3* and *WORD_7* states do no processing and are used to skip packet headers that are on the data bus.

The role of the *op_lut_process_sm* submodule is to use data gathered by the preprocess blocks to determine the correct output port(s) a packet should take and then forward the packet to the Output Queues module. Fig. 5 shows a state diagram of the submodule. The initial state is *WAIT_PREPROCESS_RDY*, which waits until all the preprocess blocks, i.e. *eth_parser*, *ip_lpm*, *http_get_filter*, etc. have completed their processing roles for the current packet on the data bus. The next state for error free packets is *MOVE_MODULE_HDRS*. This state controls which output port(s) a packet will be sent out on by modifying the one-hot encoded output port field in the IOQ packet header.

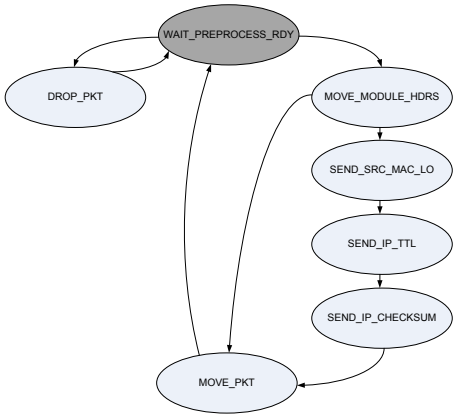



Fig. 5. State diagram of the *op_lut_process_sm* submodule. Code in the *WAIT_PREPROCESS_RDY* state has been altered.

Code changes have been made in the *WAIT_PREPROCESS_RDY* state to update the IOQ header output port field so that GET packets are duplicated up to the host system through one of the CPU transmit queues.

The rest of the states take care of updating the correct Ethernet MAC addresses, time-to-live field, and IP checksum as packets get passed to the Output Queues module. No other changes have been made in these states.

The extended reference router is configured using the software tools provided in the NetFPGA base package, i.e. the cli, Java GUI, or SCONE.

### B. Software

The URL Extractor is written in C, and reads packets from the first NetFPGA software interface, i.e. nf2c0, using raw sockets. Raw sockets have been used because they allow packets to bypass the Linux TCP/IP stack and be handed directly to the application in the same form they were sent from the NetFPGA hardware.

Uniform Resource Locators consists of two parts: Host, and Uniform Resource Identifier (URI). Both these fields are contained within a GET packet, as shown in Fig. 6. The URL Extractor parses GET packets for these fields, and then extracts and concatenates the data before storing it in the database. The URLs are also checked to contain Google search terms, and if found, are also entered into the database. Extracted URLs and search terms are printed on-screen in real-time.


Fig. 6. Image of a HTTP GET Request Packet

As previously stated, the location of the "GET" string in HTTP packets varies with the client operating system. In addition to this, the format of GET packets is heterogeneous across different browsing software, with variations in the offset between the URI and Host fields. This is due to the fact that the HTTP/1.1 RFC only states that certain fields must be included in a GET request but not their specific location [4]. The URL Extractor has been designed to handle both these variations. Furthermore, GET requests can be large and span across multiple TCP segments. Currently the software only deals with GET requests confined to a single packet.

A MySQL database [5] is used to store the extracted URLs and search terms. The database is composed of two tables: one for URLs and the other for search terms.

The GUI queries the database for the top-20 occurring URLs and search terms. It has also been written in C using the GTK+ API [6].

## III. Experimentation

We first verified our implementation in the simulation platform by using the Perl testing library. The library allowed us to create packets with specific TCP payloads. This was accomplished by first capturing GET packets using Wireshark and then exporting the TCP header and payload using the "Export as C Arrays" feature. These packet bytes were then imported into our simulation scripts. Once verified in simulation, we

created regression tests that mirrored our simulation tests. The regression tests were also created using the Perl testing library and allowed us to verify the operation of our design in real hardware. Furthermore, the regression tests of the reference router were also used to ensure that our modifications did not break the standard functionality of the reference router. The availability of these tests greatly reduced the time required to test our design.

Having verified the correctness of our implementation, we ran three experiments to profile the system resource utilization of our URL extraction system with and without the NetFPGA platform. The first two experiments both utilised the NetFPGA but used different hardware designs; one filtered HTTP GET packets while the other filtered all packets with a TCP destination port equal to 80 (HTTP). We refer to these designs as the GET and SPAN filters respectively, from here on. The third experiment used a software router (a PC configured as a router). Our experiments were based on a host computer system running the CentOS 5.2 operating system and contained an AMD dual core processor running at 3.0 GHz, 2 GB RAM, an Intel Pro/1000 Dual-port NIC, and an ASUS M2N-VM DVI motherboard with an on-board gigabit NIC. Ideally we would have liked to deploy our implementation in a live network, but this raised practical concerns from the system administrators at our University. We therefore had to take the next best option, by which the network administrators collected a trace of the entire department's traffic to/from the Internet over a 24-hour period, and gave us the trace, after some sanitization (to remove clear-text passwords etc.), as a set of PCAP files. We then used tcpreplay software [7] to play the PCAP files at the maximum rate, using the --topspeed flag (the size of the PCAP files were too large for use on the NetFPGA packet generator [8]). Two PCs were used to pump traffic into the URL extraction system in order to increase the throughput to gigabit speeds. The total input rate by both PCs was approximately 1.3 Gb/s into the NetFPGA platform and 800 Mbps into the software router (both inputs into the software router were connected to the Intel NIC). The network trace contained 13 GB of traffic and was replayed in three continuous iterations in each experiment. These input rates and traffic volume presented a reasonable "stress-test" under which the performance of the system was profiled.

Performance profiling was conducted with a lighter version of the URL extraction software that did not include the database and GUI. URLs and search terms were extracted to a text file instead. This produced results that focused more on the hardware component of the system as the higher level software had not yet been optimised for performance.

Whilst conducting the experiments we monitored 4 system variables: throughput into the router (measured at the output of the senders), throughput on the interface that the urlx software was binded to, the host CPU utilization, and the throughput on the input interface of the adjacent router (as a measure of the router's forwarding rate).

Figs. 7 and 8 show the input and output rates of the routers. The rates for both NetFPGA designs (SPAN and GET filters) are identical as the filtering level does not affect the data plane forwarding rate of the reference router. Their output rate is slightly below 1 Gb/s because the output measurements were taken on a PC that could not keep up with the line rate. As the NetFPGA is capable of forwarding at the line rate [9], the output rate would be 1 Gb/s had the measurements been taken directly from the NetFPGA output port. Hence, it is fair to assume that our design can perform URL extraction at the gigabit line rate. Due to the dumbbell experimentation topology, the output of the NetFPGA is a bottle neck point, and the difference between the input and output graphs (Fig. 7) represents dropped packets at the output queue.

The input rate into the software router (Fig. 8) is substantially lower than that of the NetFGPA platform, even though the tcpreplay settings were identical (using the –topspeed flag). This is due to the flow control mechanism in Gigabit Ethernet [10] kicking in and reducing the rate as the software router's buffers become full. The slower input rate led to an increased transmission time. The software router also drops packets. This is most likely caused by the processor not being able keep up since it is at near maximum utilization, as shown in Fig. 10. The average forwarding rate for the software router was 450 Mbps. Overall, the NetFPGA forwarding rate for this topology is more than 2 times faster than that of the software router.

Fig. 9 shows the throughput on the urlx receiving interface. The GET and SPAN hardware filters transmit an average of 4K and 48K packets per second up to the host respectively. The GET filter transmits 12 times less traffic up to the host than the SPAN filter. This result is in-line with the protocol hierarchy analysis performed on the network trace that showed 1.62% of packets contain a GET request and 19.20% were destined for port 80. The resulting ration of these two numbers is 11.85.

In both the SPAN and GET filter implementations, duplicated packets are sent up to the host through the nf2c0 interface. During experimentation, we ensured that nothing was connected on the MAC-0 port of the NetFPGA. This prevented packets not part of the filtering process from being sent up to the nf2c0 interface since the reference router sends exception packets up to the host to be handled by software. It allowed accurate collection of data from the interface.

As the software router has no filtering capability, the urlx software is required to inspect every packet that enters the router, and hence the high throughput level in Fig. 9.

Fig. 10 shows the CPU utilization of the host system. The NetFPGA GET filter reduces the utilization by a factor of 36 over the software router and a factor of 5.5 over the SPAN router. The reductions are to due to fewer packets being processed since filtering takes places in hardware. In addition, the NetFPGA performs forwarding in hardware. This is in contrast to the software router which has to process every single packet.

The three distinct repetitions of the SPAN and GET curves in Fig. 10 represent tcpreplay being looped through three iterations. The 4th smaller repetition is most likely caused by one of the two senders being out of sync and finishing later.

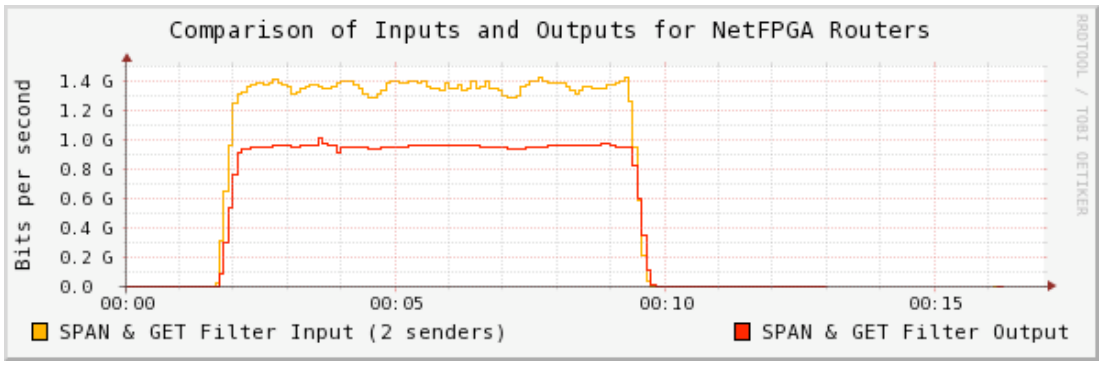



Fig. 7. Input and Output Throughputs for NetFPGA Routers

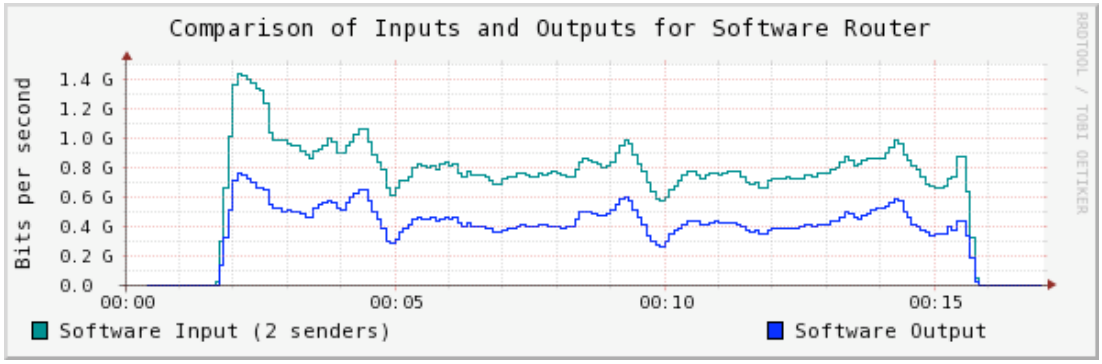



Fig. 8. Input and Output Throughput for Software Router

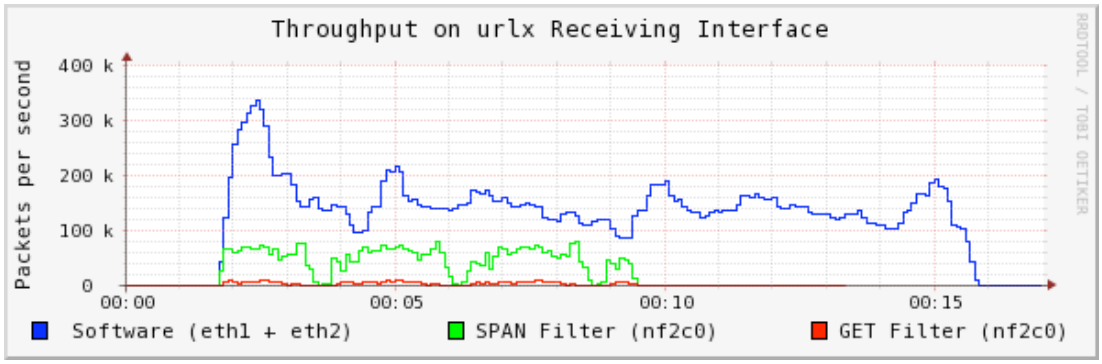



Fig. 9. Throughput on urlx Receiving Interface(s)

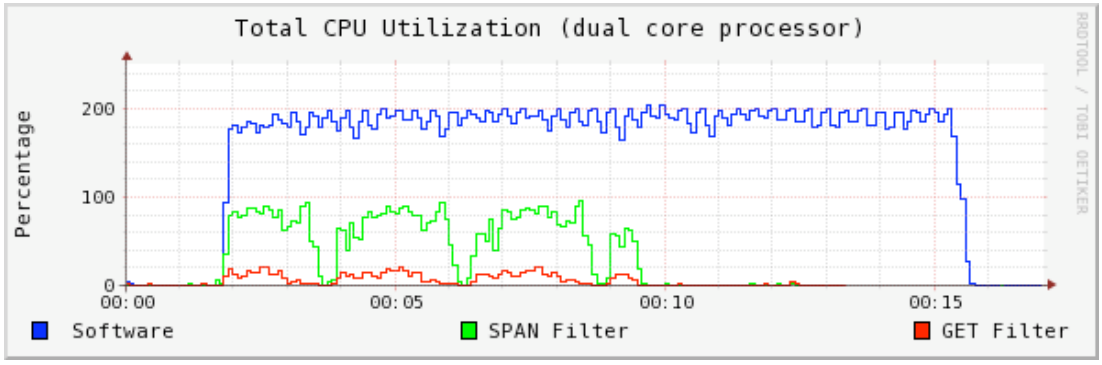



Fig. 10. Total CPU Utilization for Dual Core Processor

Our network trace spanned 13 PCAP files, each 1 GB in size.

The system performance data was gathered using collectd [11] and the graphs were created using drraw [12].

## IV. Device Utilization

The device utilization of the hardware component of our URL extraction system (the GET filter) is almost identical to that of the reference router design and is displayed in table I.

TABLE I
DEVICE UTILIZATION FOR THE GET FILTER

| Resources | XC2VP50 Utilization | Utilization Percentage |
|---|---|---|
| Slices | 9731 out of 23616 | 41% |
| 4-input LUTS | 14394 out of 47232 | 30% |
| Flip Flops | 8238 out of 47232 | 17% |
| Block RAMs | 27 out of 232 | 11% |
| External IOBs | 360 out of 692 | 52% |

## V. Conclusion

Our hardware accelerated URL extraction system is implemented on the NetFPGA platform. It performs filtering of HTTP GET packets in hardware and extraction, storage, and display of URLs and search terms in software. We believe this mix of hardware (high performance) and software (high flexibility) makes the NetFPGA platform very suitable for URL extraction: the filtering of HTTP GET packets in hardware reduces the load on the host system's processor, whilst still maintaining packet forwarding at gigabit line-rate speeds. We have shown that full URL extraction in a software-based router consumes substantially more CPU cycles when compared to the NetFPGA platform. On the other hand, a fully hardware-based implementation of URL extraction, say in a commercial router, would involve long development time; simple solutions such as configuring port mirroring (e.g. SPAN port on a Cisco router [13]) do not provide hardware filtering of traffic and therefore still require a host system to filter the traffic in software.

The implementation process of the GET filter was simplified by the pipelined architecture of the reference router. Only the operating details of the Output Port Lookup stage were required in order to achieve our goal of filtering GET packets in hardware. Furthermore, by reusing the reference router design, the development time of the GET filter was greatly reduced as we did not have to start from scratch.

Our code has been released, following the guidelines in [14], to the larger community for re-use, feedback, and enhancement. It can be downloaded from [15].

## References


[1] G. Memik, W. H. Mangione-Smith, and W. Hu.Netbench, "A benchmarking suite for networkprocessors," in *International Conference on Computer Aided Design (ICCAD)*, San Jose, CA, 2001.

[2] G. Barish and K. Andobraczka, "World wide web caching: Trends and techniques," *IEEE Communications*, vol. 38, no. 5, pp. 178–184, 2000.

[3] P. Whoriskey, "Every click you make: Internet providers quietly test expanded tracking of web use to target advertising," 2008, http://www.washingtonpost.com/wp-dyn/content/article/2008/04/03/AR2008040304052.html.

[4] R. Fielding *et al.*, "RFC 2616: Hypertext Transfer Protocol – HTTP/1.1," 1999, http://www.ietf.org/rfc/rfc2616.txt.

[5] MySQL, "MySQ website," http://www.mysql.com/.

[6] Gtk, "The Gtk+ Project," http://www.gtk.org/.

[7] tcpreplay developers, "tcpreplay website," http://tcpreplay.synfin.net/trac/wiki/tcpreplay.

[8] G. A. Covington, G. Gibb, J. Lockwood, and N. McKeown, "A Packet Generator on the NetFPGA Platform," in *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Apr 2009.

[9] G. Gibb, J. W. Lockwood, J. Naous, P. Hartke, and N. McKeown, "Netfpga: An open platform for teaching how to build gigabit-rate network switches and routers," in *IEEE Transactions on Education*, August 2008.

[10] *IEEE Standard for Information technology–Telecommunications and information exchange between systems–LAN/MAN–Part 3: CSMA/CD Access Method and Physical Layer Specifications - Section Two*, IEEE Std. 802.3, 2008.

[11] F. Forster, "collectd website," http://collectd.org/.

[12] C. Kalt, "drraw website," http://web.taranis.org/drraw/.

[13] Cisco Systems, "Cisco SPAN Configuration," http://www.cisco.com/en/US/products/hw/switches/ps708/products_tech_note09186a008015c612.shtml#topic5.

[14] G. A. Covington, G. Gibb, J. Naous, J. Lockwood, and N. McKeown, "Methodology to contribute netfpga modules," in *International Conference on Microelectronic Systems Education (submitted to)*, 2009.

[15] M. Ciesla, V. Sivaraman, and A. Seneviratne, "URL Extraction Project Wiki Page," http://netfpga.org/netfpgawiki/index.php/Projects:URL.

# A DFA-Based Regular Expression Matching Engine on A NetFPGA Platform

Yan Luo Sanping Li Yu Liu
Department of Electrical and Computer Engineering
University of Massachusetts Lowell
yan_luo@uml.edu,{sanping_li,yu_liu}@student.uml.edu

## ABSTRACT

Regular expression matching plays an important role in deep packet inspection that searches for signatures of virus or attacks in network traffic. The complexity of regular expression patterns and increasing line rate challenge the high speed matching. In this paper, we present a high speed regular expression matching engine implemented on a NetFPGA platform. We describe our design considerations, the architecture and a number of verification test cases.

## 1. INTRODUCTION

Regular expression is powerful in expressing groups of complex patterns. Such patterns are often used in Deep Packet Inspection (DPI) that searches network packet payloads to detect threats, such as intrusions, worms, viruses and spam. The increasing number and complexity of patterns (aka rules) in rule sets (e.g. Snort[11]), coupled with constantly increasing network line rates, make DPI at line rate a challenging task.

Most of the pattern matching methods rely on state machines or finite automata, including Non-deterministic Finite Automata (NFA) or Deterministic Finite Automata (DFA), to search for patterns. In addition, DFA is usually chosen over NFA for its deterministic performance. The pattern matching based on DFA is essentially a memory intensive task. However, the size of the DFAs can be too large (often at least tens of megabytes) to be stored in on-chip memory modules. As a result, the searching on the automata incurs a large number of off-chip memory accesses that lead to unsatisfactory performance. Therefore, it is important to minimize the storage requirements of state machines and put as many states as possible into on-chip or other fast memory modules to achieve high-speed pattern matching.

In this paper, we present a DFA based regular expression matching engine that takes advantage of the architectural features of FPGAs to perform fast pattern matching. Specifically, we store the DFA states along with their transitions in a compact fashion by eliminating redundant information. We organize the DFA states in parallel on-chip memory banks to ensure the state lookup and transition to be completed in a constant time. We pipeline the design to further improve the system throughput.

The paper is organized as follows: Section 2 reviews related work briefly. Section 3 elaborates our design of a regular expression matching engine on a NetFPGA platform, and the verification results are presented in Section 4. Finally, the paper is concluded in Section 5.



## 2. RELATED WORK

Variable pattern length and location, and increasingly large rule sets make pattern matching a difficult task. Many string matching algorithms exist, such as those due to Boyer-Moore [3], Wu-Manber [13], and Aho-Corasick [2], and there have been numerous studies on regular expression matching [4, 14, 8, 1]. Different platforms have been used to perform DPI, including ASICs [12], FPGAs [5] and network processors [10]. Recently there are works such as [9, 6] that apply NFA-based pattern matching methods on FPGAs.

## 3. THE DESIGN

### 3.1 Design Considerations

The design goal of the Regular Expression (RegEx) pattern engine is to achieve line rate pattern matching speed with a single modern FPGA chip. To achieve the goal, we face several challenges:

- The intractable size of DFA generated from Perl Compatible Regular Expressions (PCRE) and the expanding PCRE pattern set. Due to the complexity of PCRE syntax, the number of DFA states generated from a PCRE can be exponential. The Snort rule set as of Sept 11, 2007 consists of 1783 unique PCREs and the set keeps expanding. As a result, the memory required to store the DFA states are often beyond the capacity of available memory devices.
- Single-cycle DFA transition determination. This is one of the key factors to achieve high pattern matching rate. The comparison of current input byte with the current DFA state is made complex by the requirement of supporting advanced features of PCREs such as back references.
- Off-chip memory bandwidth constraints. The target development board (and other single board designs)

places limitations on the support off-chip memory channels. The board supports up to two SRAM channels. This could be a bottleneck of supporting more than two matching engines when portions of the DFA have to be stored in off-chip SRAM.

- On-chip memory size constraint. The size of on-chip memory gives an upper bound of the DFA states stored on-chip.
- On-chip logic resource constraint. The number of LUTs (logic elements) dictates the number of matching engines that can be instantiated. The number of matching engines directly scales the throughput.

We have proposed in [7] a method to achieve single-cycle state lookup/transition time while storing some DFA states in a compact fashion. In a baseline design, the state transition is done by using the input character as a direct index to the lookup table to determine the next state. This means every DFA state has to have a 256-entry lookup table (Fig. 1(a)), which easily uses up the on-chip memory space. In our proposal, a DFA state is stored in either on-chip memory or off-chip memory (SRAM), depending on the number of transitions it has (we count only the *valid* transitions that do not go back to the starting state of the DFA.) If the number of valid transitions is less than the number of on-chip memory banks, we store the state transitions along with the transition value in those banks, as illustrated in Fig. 1(b). Since these on-chip memory banks can be accessed in parallel, we are able to compare the input character with all the transition values in one cycle to determine the next state. If the number of valid transitions is more than the number of on-chip banks accessed in parallel, we store a 256-entry lookup table in SRAM and the transition lookup uses the input character as the index.

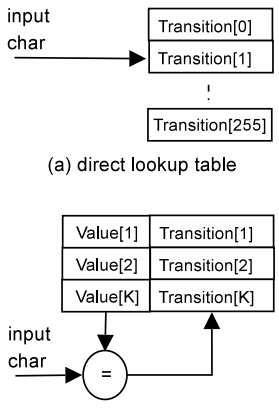



**Figure 1: DFA states stored in on-chip memory.**

## 3.2 The Overview

We have designed a regular expression compiler to generate DFAs from PCREs. We use the rules from the Snort rule set as our test rules. Snort rules defines both the 5-tuple headers to match incoming packets and the PCRE patterns to search for when a packet matches the 5-tuple header. In our study, we focus on the PCRE matching and assume that a header matching unit is available to determine which PCRE patterns the packet should be matched against.

The workflow of the regular expression matching is illustrated in Fig. 2. We first use a regular expression compiler to convert Snort rules (the PCRE portion) to DFAs. We divide the PCREs (1784 unique PCREs in Snort 2007 rule set) into groups of N-PCREs where N can be 1 through a reasonable number $k$ depending on the header classification. Such group is feasible as the number of PCREs to match is limited after header matching. The generated DFAs are analyzed to determine whether a DFA state should be stored in on-chip memory banks or off-chip ones. The memory images are generated subsequently to initialize the memory components in the system.

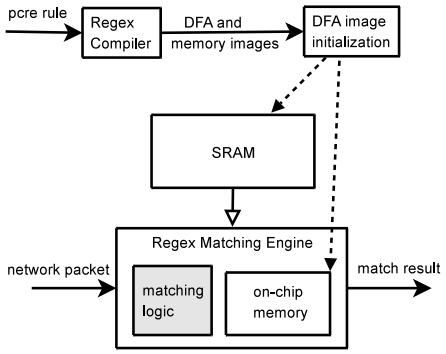



**Figure 2: Workflow of Regular Expression Matching.**

We apply the techniques proposed in [7] in the design of regular expression matching engine. The information of a DFA state and its transitions are encoded and stored in $M$ on-chip memory banks, if the number of transitions is less than a threshold (six in our design). The on-chip memory banks are read in parallel and the input byte from the packet is compared against the encoded DFA state to determine the next state. The DFA states with larger number of transitions than the threshold are placed in off-chip memory units and the lookup of the next state is simply indexing the memory with the input byte.

The packet content inspection engine is depicted in Fig. 3. The design contains regular expression matching engines (MEs) and a DFA update controller. Each ME is connected to a Triple Speed Ethernet (TSE) IP core that receives and sends packets. There is a packet scheduler in a ME that interleaves incoming packets for the pipeline in ME. A ME also contains on-chip memory banks that store encoded DFAs and matching logic that looks up next states to perform state transition. The DFA image update controller initializes or updates the memory components with DFA images received from a dedicated Ethernet interface. In our design, we choose to use one Ethernet port to receive DFA images, instead of updating over PCI bus, due to two reasons: (1) fast prototyping using Ethernet interface to receive DFA images. We are more familiar with TSE than PCI bus transactions; and (2) to allow DFA being updated remotely by a control PC. Therefore, on the NetFPGA platform we instantiate three RegEx MEs and one DFA update controller. It is worthy noting that our design is not based on the NetFPGA reference designs which has well-established FIFO and packet processing pipelines Instead, we build our design di-

rectly on Xilinx IP cores (ISE, onchip memory, etc.) which we are familiar at the time when we start this work. It will be our future work to integrate our design with NetFPGA reference design framework.

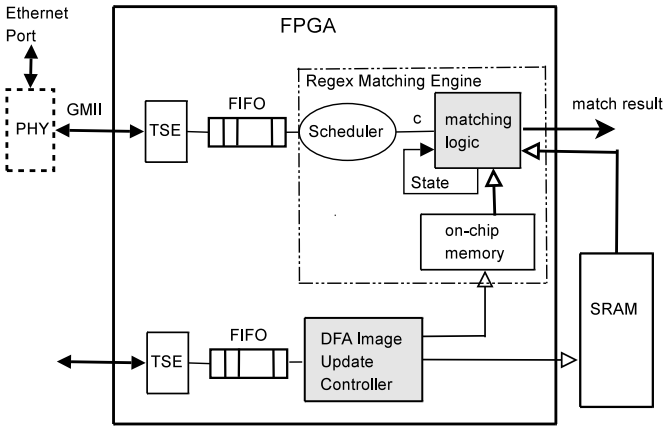



**Figure 3: Deep Packet Inspection Engine on NetFPGA**

The DPI engine works as follows. The TSE core receives Ethernet frames and keeps them in the FIFO unit. The FIFO unit supplies packet content (including both header and payload) to the scheduler by providing byte streams and frame start and stop signals. The packet scheduler puts the incoming packets to three internal packet buffers and records the size of each frame. The scheduler then sends the bytes from the three buffered frames to the pipelined matching engine in a round-robin fashion. Thus, the MEs see interleaved bytes from the three packets. The reason of such a design with a scheduler and interleaved packets is that a ME is implemented as a three-stage pipeline to improve the throughput of the system. The MEs compare the input byte with the transitions of the current DFA state to determine the next state. They then perform the state transition by updating the current state register until a matching state is reached. In the next subsections, we describe our design in more details.

## 3.3 The RegEx Matching Engine

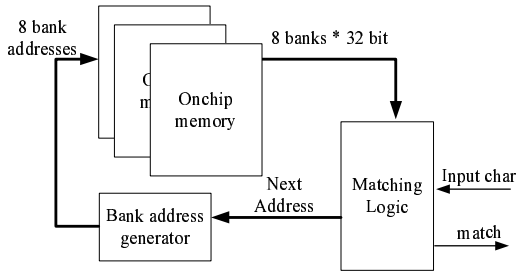



**Figure 4: High Level Description of Matching Engine**

The design of matching engine is made up of 3 modules, the Matching Logic, Bank Addresses Generator and Onchip Memory. The block diagram of architecture is shown in Figure 4.

The Matching Logic module is the core module in the design. Its input signals consist of the input characters, DFA image data read from on-chip memory bank and additional control signals. The module consumes input character streams; and also gets the 8 banks of memory data and processes them in parallel to generate next memory address for on-chip memory. Then the specified matching transition data is selected according to the current input character. If the current input character makes the state transition happen, the module continues to process the next input character with the new transition data from the on-chip memory. Repeat this process and finally the matching logic produces the signal "match" to tell the upper level module that current input character stream matches the current PCRE rule stored in the on-chip memory.

The Bank Address Generator module is used to calculate the next address for on-chip memory banks and off-chip memory. In the module On-chip Memory, 8 single-port RAMs are instantiated.

The delays introduced by combinational logics in the modules limit the maximum frequency of the whole matching engine design. In order to improve the frequency performance, the design of three stages pipeline is introduced. The basic idea is to make use of two levels registers to store temporary combinational signals during the signal flow. The on-board testing shows that the frequency performance has been improved significantly. The pipeline design, however, involves that the 3-character data streams which have to be interleaved to feed into matching engine.

## 3.4 DFA Image Update

It is known that the DFAs generated from regular expression rules can be prohibitively large and the number of rules in a set keeps increasing. The DFA states generated from *all* the rules cannot be accommodated in on-chip memory at once, even though some of states can be stored as on-chip states. It is necessary to dynamically load DFA states so that at any time the on-chip memory stores only the states that are necessary in matching the current packet payload. Therefore, we design a DFA update module to initialize and dynamically write DFA states into the on-chip memory.

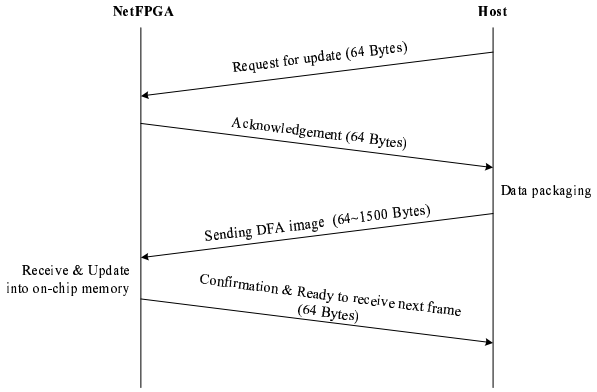



**Figure 5: DFA Image Update Protocol**

The module of DFA update implements the function of writing the DFA image data into FPGA on-chip memory. The updating process executed by the FPGA device includes the following steps, as shown in the Figure.5, (1)

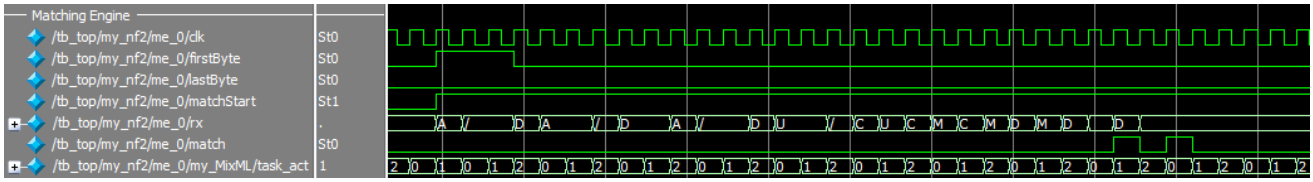



**Figure 6: Verification of test case "\x2FAD\x2FU?CMD/smi"**

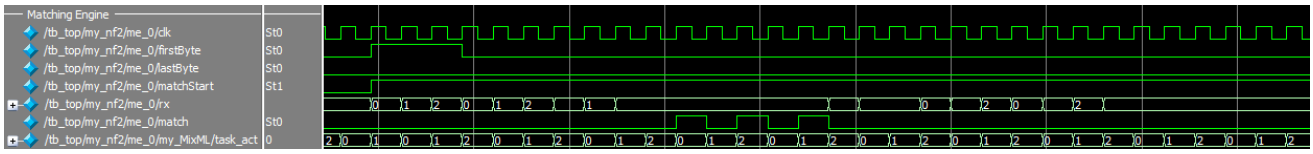



**Figure 7: Verification of test case "^[0-2]{1,3}\x00"**

receive the request from the host for updating DFA image data into FPGA on-chip memory; (2) send the acknowledgement packet to the host and be ready to receive; (3) receive packets comprising the DFA image data; (4) at the same time, write DFA image data into the on-chip memory; (5) send the acknowledgement for successfully receiving. If the packets sending to the FPGA are of mal-format or invalid during the updating process, the FPGA sends the specified acknowledgement to notify the host to re-send the corresponding packets.

While receiving packets containing DFA image data from the host byte by byte, the update module groups each 4-byte into one word which will be written into the on-chip memory bank; meanwhile the memory address is also generated, including the bank number and the row address in the bank. So when receiving the entire image data for one memory bank, the update module completes the operation of writing DFA data into the memory bank.

## 4. VERIFICATION WITH CASE STUDIES

The functional correctness of the matching engines is verified using ModelSim. We build test cases that exercise individual features. The testing results show that the matching engine supports all PCRE features, including anchor mark, dollar mark, multiple lines mode, word boundary, back reference and look ahead.

We use the NetFPGA development board to perform on-board testing. The on-board testing is the most comprehensive and strict testing method that ensures all the modules functional correctly and no timing violations. The test cases cover mixed memory matching engine, accessing both on-chip and SRAM memory units. We also tested DFA update module by sending on-chip memory image from the PC to the port 3 on NetFPGA board. The DFA update module on the FPGA receives the memory images and subsequently writes image data to eight on-chip memory banks. The SRAM writing module is done via a separate SRAM writing state machine in the design. The experiment was performed by sending raw Ethernet packets from a PC to the port 0, 1, 2 of the four Ethernet ports on NetFPGA board at 1Gbps rate. When there is a match, a special packet will be sent out by the NetFPGA. The testing results show that our mixed memory matching engine passed the test cases correctly.

**Table 1: Test Case 1**

| PCRE rule | \x2FAD\x2FU?CMD/smi |
|---|---|
| Input stream | Testing Result |
| AD/AD/CMD | Match |
| /AD/UCMD | Match |
| /AD/UUCMD | No Match |

**Table 2: Test Case 2**

| PCRE rule | ^[0-2]{1,3}\x00 |
|---|---|
| Input stream | Testing Result |
| 00^@ | Match |
| 111^@ | Match |
| 22^@ | Match |
| ^@00^@ | No Match |

In the following we present the verification of the RegEx ME using several example PCRE rules. For example, we select two PCRE rules, "\x2FAD\x2F U?CMD/smi" and "^[0-2]{1,3}\x00", as the testing cases. The corresponding input character streams and testing results are shown in Table 1 and Table 2. In Table 2, the symbol "^@" in the input character stream represents the character of 0x00. Fig. 6 and Fig. 7 are the corresponding testing waveforms generated by ModelSim XE.

In both figures of waveform, the input signal of "firstByte" asserts for one cycle when the first byte of each packet arrives; the input signal of "lastByte" asserts for one cycle when the last byte of each packet arrives; the signal "matchStart" is an active-high enable signal for the matching logic module; the input signal of "rx" is the characters sequence arranged in the interleaved order of three incoming packets. The output signal "match" asserts when the input character stream match the current PCRE rule. All signals are synchronized at the rising edge of the system clock.

## 5. CONCLUSION

We present in this paper a DFA based regular expression matching engine implemented on a NetFPGA platform. We have implemented in the design a packet scheduler, and pipelined matching engines and DFA image update module. We also show two test cases used to verify our design. Currently, our implementation is not based on the framework

provided by NetFPGA reference designs which have well established packet processing pipeline and interface with the software on the host computer. In the near future, we plan to port our design into the NetFPGA framework so that it can be available for a wide range of interested users.

## Acknowledgment

The authors thank Justin Latham, Chris Hayes and Ke Xiang for their earlier work on this topic. The authors also thank Jian Chen and Hao Wang for their valuable comments. This project is supported in part by NSF Grant No. CNS 0709001 and a grant from Huawei Technologies.

## 6. REFERENCES

[1] TRE: POSIX Compliant Regular Expression Matching Library. http://laurikari.net/tre/.

[2] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

[3] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[4] B.C. Brodie, R.K. Cytron, and D.E. Taylor. A Scalable Architecture for High-Throughput Regular-Expression Pattern Matching. In *ISCA*, Boston, MA, June 2006.

[5] S. Dharmapurikar and J. Lockwood. Fast and scalable pattern matching for network intrusion detection systems. *IEEE Journal on Selected Areas in Communications*, 24(10):1781–1792, October 2006.

[6] M. Faezipour and M. Nourani. Constraint Repetition Inspection for Regular Expression on FPGA. In *IEEE Symposium on High Performance Interconnects*, Palo Alto, CA, August 2008.

[7] C. Hayes and Y. Luo. Dpico: A high speed deep packet inspection engine using compact finite automata. In *ACM Symposium on Architecture for Network and Communication Systems*, Orlando, FL, December 2007.

[8] S. Kumar, S. Dharmapurikar, P. Crowley, J. Turner, and F. Yu. Algorithms to accelerate multiple regular expression matching for deep packet inspection. In *SIGCOMM*, Pisa, Italy, September 2006.

[9] Abhishek Mitra, Walid Najjar, and Laxmi Bhuyan. Compiling pcre to fpga for accelerating snort ids. In *ACM Symposium on Architecture for Network and Communication Systems*, Orlando, FL, December 2007.

[10] P. Piyachon and Y. Luo. Efficient memory utilization on network processors for deep packet inspection. In *ACM Symposium on Architecture for Network and Communication Systems*, San Jose, CA, December 2006.

[11] Snort. http://www.snort.org/, 2003.

[12] L. Tan and T. Sherwood. Architectures for Bit-Split String Scanning in Intrusion Detection. *IEEE Micro: Micro's Top Picks from Computer Architecture Conferences*, January-February 2006.

[13] S. Wu and U. Manber. Fast text searching: Allowing errors. *Communications of the ACM*, 35(10):83–91, 1992.

[14] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz. Fast and memory-efficient regular expression matching for deep packet inspection. In *ACM Symposium on Architecture for Network and Communication Systems*, San Jose, CA, December 2006.

# High-level programming of the FPGA on NetFPGA

Michael Attig and Gordon Brebner
Xilinx Labs
2100 Logic Drive
San Jose, CA 95124
{mike.attig,gordon.brebner}@xilinx.com

## ABSTRACT

The NetFPGA platform enables users to build working prototypes of high-speed, hardware-accelerated networking systems. However, one roadblock is that a typical networking specialist with a software-side background will find the programming of the FPGA to be a challenge because of the need for hardware design and description skills. This paper introduces G, which is a high-level packet-centric language for describing packet processing specifications in an implementation-independent manner. This language can be compiled to give high-speed FPGA-based components. An extension has been produced that allows these components to be dropped easily into the data path of the standard NetFPGA framework. This allows a user to write and debug packet processing functions at a high-level in G, and then use these on the NetFPGA alongside other components designed in the traditional way.

## 1. INTRODUCTION

Diversity and flexibility are increasingly important characteristics of the Internet, both to encourage innovation in services provided and to harness innovation in physical infrastructure. This means that it is necessary to provide multiple packet processing solutions in different contexts. At the application level, there is a need to implement deep packet inspection for security or packetization of video data, for example. At the transmission level, there is a need to keep pace with rapidly-evolving technologies such as MPLS and carrier Ethernet. This is in addition to support for evolution in protocols like IP and TCP that mediate between application and transmission.

Recent research in Xilinx Labs has led to four main contributions which, taken together, offer a flexible *and high-performance* solution to the problems posed by allowing increasing diversity in packet processing:

- Introducing G, which is a domain-specific high-level language for describing packet processing. G focuses on packets and is protocol-agnostic, thus supporting diversity and experimentation. In addition, G is concerned with the 'what' (specification) not the 'how' (implementation). In other words, G describes the problem, not the solution.
- Using the rich — but raw — concurrent processing capabilities of modern Field Programmable Gate Array (FPGA) devices to enable the creation of tailored virtual processing architectures that match the individual needs of particular packet processing solutions. These provide the required packet processing performance.
- Demonstrating a fast compiler that maps a G description to a matching virtual architecture, which is then mapped to an FPGA-based implementation. This facilitates experimentation both with options for the packet processing itself and with options for the characteristics of the implementation. It also removes existing barriers to ease of use of FPGAs by non-hardware experts.
- Generating modules with standard interfaces that are harmonious with the Click [8] modular router framework. This facilitates the assembly of complete network node capabilities by smoothly integrating varied FPGA-based components, and by interfacing processor-based components.

The technology that has been developed is scaleable. The initial target of the research was networking at 10Gb/s and above rates, with current experiements addressing 100Gb/s rates. However, going in the other direction, the G technology is also applicable to the current NetFPGA platform, for single 1Gb/s rate channels, or an aggregate 4Gb/s rate channel. In order to enable seamless integration with NetFPGA, wrappers have been developed that allow the generated modules to be dropped into the standard packet processing pipeline used in NetFPGA designs.

In this paper, we first provide a short overview of G and the compilation of G to FPGA-based modules. We then describe how these modules can be integrated and tested on the NetFPGA platform. This is illustrated by an example where G is used to describe VLAN header manipulation, and this is added into the reference router datapath.

## 2. OVERVIEW OF G

The full version of the G language is targeted at specifying requirements across network processing. The initial implementation has focused on a subset of G that is concerned with expressing rules for packet parsing and packet editing, the area which has been most neglected in terms of harnessing FPGA technology to its best effect. Other aspects of G beyond this subset are still the subject of continuing research.

Although independent of particular implementation targets, a G description is mapped onto an abstract 'black box' component that may be integrated with other components to form complete packet processing — or more general processing — systems. The setting for such components is that of the Click modular router framework [8] (an aspect not covered in this paper). In Click, an individual system component is called an element, and each element has a number of input and output ports. This enclosing Click context means that a G description is regarded as describing the internal behavior of an element, and also declaring the external input and output ports of that element.

A G element description consists of three parts. The first part is concerned with declaring the nature of the element's external interactions, that is, its input and output ports. The second part is concerned with declaring packet formats, and the format of any other data structures used. Finally, the third part contains packet handling rules.

Input and output ports in G have types, two of which will be introduced here. The packet type is one on which packets arrive (input) or depart (output), and this corresponds to the standard kind of (untyped) Click port. The access type is one on which read and/or write data requests can be made (output) or handled (input). This is provided to allow interaction between elements, separately from the passing of packets.

G has no built-in packet formats, since it is deliberately protocol-agnostic to give maximum flexibility. Packet formats and other data formats may either be declared directly in a G description or be incorporated from standard libraries through included header files. A format consists of an ordered sequence of typed fields. Types are scalar or non-scalar. The scalar types were selected to be those deemed of particular relevance to the needs of the packet processing domain. They are bit vector, boolean (true or false), character (ISO 8859-1), and natural number (in range 0 to $2^{32} - 1$). For example, negative integers and real numbers were not included. Non-scalar type fields may be of another declared format, or be a list of sub-fields, or be a set of alternative fields. G itself does not explicitly differentiate between uses of packet fields (e.g., header, trailer, payload) — this is expressed through the actual packet handling rules.

The substantive portion of a G description provides the rules to be applied for packet handling. These refer as appropriate to the preceding declarations of ports and formats. In particular, the handler heading for the set of rules includes the name of the packet input port and the name of the input packet format. Prior to the rules themselves, local variables can be declared. These have a lifetime corresponding solely to a particular packet being handled. Thus, if several packets are being handled simultaneously, each has its own unique set of the local variables. In general, concurrency is maximized, here across multiple packets.

Packet handling rules implement the requirements of protocols for handling packets. When writing rules, the G user is liberated from having to think about when rules are applied, or how rules are applied. In particular, the G user does not have to be concerned with where fields occur within packets, or how packet data arrives and departs at input and output ports. Application of different rules is intended to be done as independently as possible.

In terms of the rules themselves, there are two categories: packet rules and external state rules. Packet rules are used to change the content or format of the packet, or to approve it for forwarding. There are four types of packet rules in the G subset being considered:

- **Set:** change packet fields
- **Insert:** insert one or more additional fields into the packet
- **Remove:** remove one or more fields from the packet
- **Forward:** indicate that the packet should be output after handling

For a simple forwarding function, combinations of set and forward rules are sufficient. For encapsulation or decapsulation, for example, insert or remove rules respectively are used also. External state rules are used to perform reading and writing of arbitrary data format values on external access ports. These rules can optionally specify additional read or write parameters (e.g., memory addresses, lookup keys, or function arguments, depending on the type of external element).

Guards provide for the conditional application of rules, and so form the basis for packet parsing and simple packet classification. The Boolean expression within a guard condition is always fully evaluated and tested for truth before the guarded rule will be applied. There can be a nest of disjoint guarded rules, in which each sequential guard condition is always fully evaluated and tested for falsity before a subsequent rule in the nest will be considered. In other words, the first rule in the nest with a true guard condition, or a final unguarded rule if none of the preceding rules have a true guard condition, is the one that is applied.

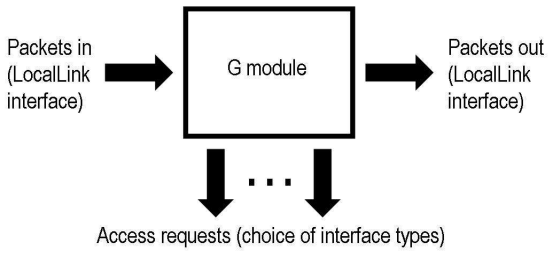



**Figure 1: Generated module**

# 3. COMPILATION OF G TO FPGA

## 3.1 Underlying principles

The compilation of G towards an FPGA implementation is founded upon three important principles: (i) an expected low level of dependencies between rules in G descriptions; (ii) the creation of *virtual architectures* to implement the packet handling rules; and (iii) the generation of modules with standard interfaces that can be plugged together to create complete systems.

The first principle is derived from observation of the nature of packet parsing and editing for protocols. There is considerable independence in how packet fields are treated with respect to other packet fields. This gives packet processing a distinctive flavor when compared with normal data processing or digital signal processing. The fruit of the independence is to provide greater opportunity for concurrent operations, with consequent benefits for throughput and latency. In turn, the huge amount of potential for concurrent processing in an FPGA provides the physical support for this. The G language itself is designed to encourage the expression of independence in descriptions, something attractive to its user but also avoiding the need for too much 'automatic discovery of parallelism' magic in the compiler.

The second principle reveals an analogy between virtual networks and virtual architecture. Just as it is beneficial to implement problem-specific networks on programmable physical infrastructure, so it is beneficial to implement problem-specific microarchitectures on the programmable physical infrastructure of an FPGA. The G compiler builds a processing microarchitecture that matches the needs of the particular G description. This contrasts with the usual need to contort the problem to fit a fixed microarchitecture, whether expressing it sequentially in C for a normal processor, or warping it to fit the properties of a specialized network processor.

The third principle is that each G description is compiled to generate a module targeted at an FPGA, and that this module has standard interfaces. This is so that it can be integrated with other modules, either also generated from G or from other sources, to build a complete FPGA-based system.

Figure 1 shows a high-level schematic of the generated module. For this research, Xilinx FPGA devices have been the target technology, so the module interfacing was chosen for compatibility with standard modules produced by Xilinx. Packet input and output ports in the G description are mapped to module interfaces that use the LocalLink standard [12] for signaling packets word-wise between modules. Access ports are mapped to interfaces that can follow several different standards for accessing modules that support read/write requests.

## 3.2 Compilation process

The generated module is not described directly in terms of the underlying FPGA resources: programmable logic gates, interconnect, and other embedded units. It is described in an intermediate form, expressed in a hardware description language, VHDL in the case of this compiler. The intent is to generate VHDL of a quality comparable with that achieved by hand design, in a much shorter time and a much more maintainable manner. The VHDL description is then processed by the standard design tools provided for FPGAs, which have had many thousands of person-years devoted to achieving high quality of results. In particular, these tools seek to harness all of the features of the specific target FPGA. Thus, the G compiler itself is decoupled from specific FPGA device detail, and works in partnership with an FPGA vendor's specific processing tools.

Since a G description is abstracted from any implementation detail, additional input is supplied to the compiler. This input concerns the nature of the interfaces to the module. It includes structural details, such as word width and minimum/maximum packet sizes, and temporal details of required throughput. At present, the compiler reports on achieved latency and achieved FPGA resource count; in the future, these will also be supplied as targets to the the compiler. The main steps carried out by the compiler are:

1. **Parsing** of the G description, and of the additional implementation-specific information.
2. **Analysis** to determine dependencies through multiple uses of packet fields or local variables, or through multiple uses of external access ports.
3. **Partitioning** of rules into dependency-respecting clusters to form an optimal parallel processing microarchiture.
4. **Scheduling** of rule applications within each cluster to respect arrival and departure times of packet data via word-wise interfaces, and the word-wise nature of access interfaces.
5. **Generating** a VHDL description of a module that implements the virtual packet-handling architecture, plus other inputs for the standard tools.

## 4. G DROP-IN MODULES

Generated G modules are a natural fit within the NetFPGA framework. Streaming packet interfaces enable G modules to be instantiated at various locations in the standard packet processing pipeline of the NetFPGA. However, to fully enmesh G modules into the NetFPGA environment, some infrastructure modifications were necessary. The goal was to make G modules indistinguishable from other NetFPGA libary components. This allows the current NetFPGA implmentation approach to remain unaltered, while enabling the inherent ability of the high-level simulation approach that G components bring. Essentially, development of G modules becomes an extension of the methodology for contributing NetFPGA modules [3].

A G component is packaged to appear as any other NetFPGA module, such as the *input arbiter* or *output port lookup* modules used in the reference router project. Figure 2 shows a wrapped G module. The G module is wrapped amongst various infrastructure pieces. An input-side protocol bridge converts from the NetFPGA bus signaling protocol to Xilinx's LocalLink signaling protocol [12]. G modules can operate as pipeline stages, as they support cut-through processing mode. The streaming LocalLink interfaces enable this desirable feature. Once the G module begins to emit the packet, it is sent through the output-side protocol bridge that converts back to the NetFPGA signaling protocol from LocalLink.

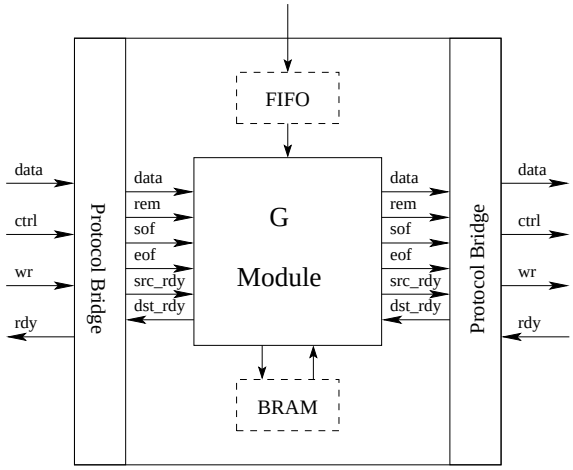



**Figure 2: G Drop-in Module**

The structure shown in Figure 2 is actually a top-level verilog wrapper. This wrapper instantiates the protocol bridges, the compiler-generated VHDL G module hierarchy, and the access components attached to the G module, such as registers, FIFOs, or BRAMs. This wrapper is auto-generated by a Perl script that scans the G module for interface signals.

## 5. EXAMPLE

The following section highlights the methodology in using a G module in the NetFPGA framework. For illustrative purposes, a simple VLAN extraction component will serve as the G component. The functionality of VLAN extraction is to remove VLAN tags attached to packet data that are no longer necessary to aid routing.

### 5.1 Step 1: Setup environment

The standard directory structure associated with the NetFPGA environment should be utilized. Users work out of the *projects* directory. Files that are developed for using G should be placed in a new directory, *gsrc*. This includes all *g*, *xml*, *fv*, testbench, and synthesis files. The expected structure is shown in Figure 3.

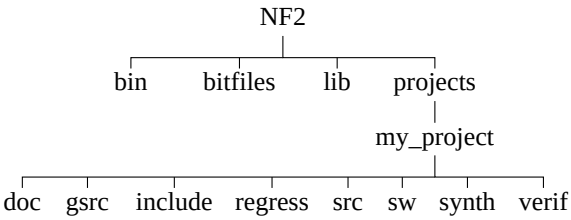



**Figure 3: Environment directory structure**

### 5.2 Step 2: Compose G

The first development step in utilizing G is to compose the G to describe the desired functionality. The entire G for the VLAN extraction example is shown below. A companion file is created to describe the characteristics of the interface ports, such as the width.

```
element vlan_extract{

  input  packetin  : packet;
  output packetout : packet;

  #define VLAN_TYPE     0x8100
  #define NON_VLAN_TYPE 0xabcd

  format MyPacket = (
    IOQHeader : (
      dst_port_one_hot   : 8,
      resv               : 8,
      word_length        : 16,
      src_port_binary    : 16,
      byte_length        : 16
    ),
    type        : 16,
    VLAN_header : 32,
    : *
  );
```

```
  handle MyPacket on packetin {

    [type == VLAN_TYPE]{
      remove VLAN_header;
      set type = NON_VLAN_TYPE;
      set IOQHeader.word_length =
             IOQHeader.word_length - 1;
      set IOQHeader.byte_length =
             IOQHeader.byte_length - 8;
    }

    forward on packetout;
  }
}
```

The packet format for this example defines the NetFPGA control header (*IOQHeader*), the *type*, and the *VLAN_header*. The remainder of the packet is indicated using the * field to mean that the *VLAN_header* is the last field of the packet this G module is interested in. The packet handler for this example is relatively simple. The *type* field is checked. If it is a VLAN type, the *VLAN_header* is removed, and the control header is updated to reflect that there is one less word in the packet. Finally, the packet is forwarded on the output packet interface.

## 5.3 Step 3: High-level G simulation

The functionality of the written G description must be verified. The G development environment comes with two tools to aid with high-level G description verification. The first is *gdebug*. This tool steps through a G description, allowing the user to select from a list of available actions to invoke. (Recall that G descriptions are declarative rather than imperative, so multiple actions could potentially be executed at any given debug step.) The second tool is *gsim*. This tool reads *packet instances* corresponding to input and produces the resulting output. The *gsim* tool is the main simulation vehicle for G descriptions.

Packet instances are created through use of the *gfv* language (G format value). This language is used to specify the values that packet fields can take. An example of this is shown below.

```
formatvalue ioq = (
  0x20    : 8,
  0       : 8,
  4       : 16,
  1       : 16,
  0x20    : 16
);

formatvalue test1 = (
  : ioq,
  [0x8100 | 0xaaaa]  : 16,
  0xfeedcafe         : 32,
  0x000011112222     : 48,
  0x1011121314151617 : 64,
  0x18191a1b1c1d1e1f : 64
);
```

This format value description will create two packet instances to input. One instance will fill the *type* field with $0x8100$ and the other instance with $0xAAAA$. The *gsim* tool reads these packet instances and simulates the expected output. The processed output for the first packet instance is shown below. Note the inclusion of field names matched to bit patterns in the output. This aids quick verification. Simulation with *gsim* is a fast process, enabling quick functional verification cycles.

```
formatvalue test1_1 = (
 : (
   0x20 : bit[8]  /* dst_port_one_hot */,
   0x00 : bit[8]  /* resv */,
   0x0003 : bit[16] /* word_length */,
   0x0001 : bit[16] /* src_port_binary */,
   0x0018 : bit[16] /* byte_length */
 )  /* IOQHeader */ ,
 0xabcd : bit[16]  /* type */,
 0x000011112222 : bit[48],
 0x1011121314151617 : bit[64],
 0x18191a1b1c1d1e1f : bit[64]
);
```

## 5.4 Step 4: Embed in NetFPGA pipeline

This G module fits between the input arbiter and the output port lookup modules in the reference router pipeline, as shown in Figure 4. G modules are currently limited to support single input and output packet streaming interfaces, so G modules can only be placed between the input arbiter and the output queues in the reference router pipeline.

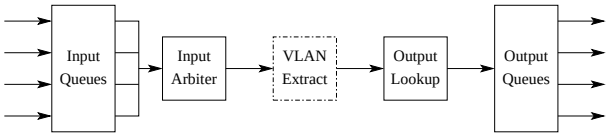



**Figure 4: VLAN extraction G module included in reference router pipeline.**

To include the G module, the *user_data_path* verilog top-level is modified to instantiate the G wrapper and wire it into the data path. The *user_data_path* file found in *NF2/lib/verilog/user_data_path/reference_user_data_path/src/* is copied into and modified within the *src* directory of the project. Recall that the G wrapper instantiates protocol bridges, the G module itself, and any access memories that the G module uses.

### 5.5 Step 5: Simulate system

The simulation environment for NetFPGA is at the HDL level. The G compiler produces synthesizable and simulation-ready VHDL, so the simulation scripts distributed with a standard NetFPGA distribution were modified to incorporate simulation of G components. These scripts are run in the same manner as any other NetFPGA project. Test data and regression tests are also created in a similar way.

### 5.6 Step 6: Implement

The build scripts have also been modified to accommodate the synthesis of G components. These scripts are run as if a standard NetFPGA project is being constructed.

## 6. RELATED WORK

Various domain-specific packet processing languages have been proposed. PacLang [4] is an imperative, concurrent, linearly typed language designed for expressing packet processing applications. FPL [2] is a functional protocol processing language for performing complex packet classification and analysis. PPL [7] is another functional programming language for packet processing, oriented towards TCP/IP as well as deep packet inspection. SNORT [10] is a well-known intrusion detection system based on complex rule sets that specify the processing of packet headers and the matching of patterns in packet payloads.

There has been considerable research and development on compiling C to FPGA (so-called 'C-to-gates'). In general, C subsets and/or C annotated with pragmas must be used. A focus has been on loop unrolling as a means of introducing concurrency, particularly for digital signal processing applications. Examples include Catapult C [9], Handel-C [1], SpecC [5], Streams-C (now Impulse C) [6], and Synfora C [11]. To date, there has been little evidence of this style of technology being of benefit for high-speed packet processing.

## 7. CONCLUSIONS AND FUTURE WORK

This paper has presented the G language and the efficient compilation of G to FPGA-based modules. In particular, it has presented a new framework which allows G modules to be easily incorporated into NetFPGA-based system designs. Taken together, these advances will ensure greater ease of use for networking experts who are not familiar with traditional FPGA design approaches.

The paper focused on just a subset of the full G language. Subsequent publications will explore other aspects including, for example: describing relationships between packets and hence functions like segmentation and reassembly; structuring of packet handlers and hence modularity between protocols; and support for more flexible packet formats. Ultimately, G is intended to cover the full spectrum of packet processing.

Future work will include seeking better integration between the standard software Click environment and an FPGA Click environment that has been developed around G. The linkage to the NetFPGA platform now offers the hardware contribution to developing an ecosystem around Click and G. It provides a convenient basis for interfacing software Click elements with FPGA-based elements described in G, thus enabling experiments where time-critical data plane functions benefit from the speed of the FPGA.

## 8. REFERENCES


[1] Celoxica. Handel-C. www.celoxica.com.

[2] D. Comer. *Network Systems Design Using Network Processors, Agere version*. Prentice Hall, 2004.

[3] G. A. Covington, G. Gibb, J. Naous, J. Lockwood, and N. McKeown. Methodology to contribute netfpga modules. In *International Conference on Microelectronic Systems Education (submitted to)*, 2009.

[4] R. Ennals, R. Sharp, and A. Mycroft. Linear types for packet processing. In *Proc. ESOP 2004*, pages 204–218, Barcelona, Spain, Mar. 2004. Springer-Verlag LNCS 2986.

[5] D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Methodology*. Kluwer, 2000.

[6] M. Gokhale, J. Stone, J. Arnold, and M. Kalinowski. Stream-oriented FPGA computing in the Streams-C high level language. In *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 49–56, Napa, CA, Apr. 2000.

[7] IP Fabrics. Packet Processing Language (PPL). www.ipfabrics.com.

[8] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[9] Mentor. Catapult C. www.mentor.com.

[10] M. Rosesch. SNORT — lightweight intrusion detection for networks. In *Proc. LISA 1999*, pages 229–238, Seattle, WA, Nov. 1999.

[11] Synfora. Synfora C. www.synfora.com.

[12] Xilinx. FPGA and CPLD solutions. www.xilinx.com.

# NetThreads: Programming NetFPGA with Threaded Software

Martin Labrecque, J. Gregory Steffan
ECE Dept., University of Toronto
{martinl,steffan}@eecg.toronto.edu

Geoffrey Salmon, Monia Ghobadi, Yashar Ganjali
CS Dept. University of Toronto
{geoff, monia, yganjali}@cs.toronto.edu

## ABSTRACT

As FPGA-based systems including soft processors become increasingly common, we are motivated to better understand the architectural trade-offs and improve the efficiency of these systems. The traditional forwarding and routing are now well understood problems that can be accomplished at line speed by FPGAs but more complex applications are best described in a high-level software executing on a processor. In this paper, we evaluate stateful network applications with a custom multithreaded soft multiprocessor system-on-chip—as an improvement on previous work focusing on single-threaded off-the-shelf soft processors— to demonstrate the features of an efficient yet usable parallel processing system along with potential avenues to improve on its main bottlenecks.

## 1. INTRODUCTION

The NetFPGA development platform [1] allows networking researchers to create custom hardware designs affordably, and to test new theories, algorithms, and applications at line-speeds much closer to current state-of-the-art. The challenge is that many networking researchers are not necessarily trained in hardware design; and even for those that are, composing packet processing hardware in a *hardware-description language* is time consuming and error prone.

Improvements in logic density and achievable clock frequencies for FPGAs have dramatically increased the applicability of *soft processors*—processors composed of programmable logic on the FPGA. Despite the raw performance drawbacks, a soft processor has several advantages compared to creating custom logic in a hardware-description language: (i) it is easier to program (e.g., using `C`), (ii) it is portable to different FPGAs, (iii) it is flexible (i.e., can be customized), and (iv) it can be used to manage other components/accelerators in the design. However, and most importantly, soft processors are very well suited to packet processing applications that have irregular data access and control flow, and hence unpredictable processing times.

### 1.1 NetThreads

In this paper we present NetThreads, a NetFPGA-based soft multithreaded multiprocessor architecture. There are several features of our design that ease the implementation of high-performance, irregular packet processing applications. First, our CPU design is multithreaded, allowing a simple and area-efficient datapath to avoid stalls and tolerate memory and synchronization latencies. Second, our memory system is composed of several different memories (instruction, input, output, shared), allowing our design to tolerate the limited number of ports on FPGA block-RAMs while supporting a shared memory. Third, our architecture supports multiple processors, allowing the hardware design to scale up to the limits of parallelism within the application.

We evaluate NetThreads using several parallel packet-processing applications that use shared memory and synchronization, including UDHCP, packet classification and NAT. We measure several *packet processing* workloads on a 4-way multithreaded, 5-pipeline-stage, two-processor instantiation of our architecture implemented on NetFPGA, and also compare with a simulation of the system. We find that synchronization remains a significant performance bottleneck, inspiring future work to address this limitation.

## 2. MULTITHREADED SOFT PROCESSORS

Prior work [2–6] has demonstrated that supporting *multithreading* can be very effective for soft processors. In particular, by adding hardware support for multiple thread contexts (i.e., by having multiple program counters and logical register files) and issuing an instruction from a different thread every cycle in a round-robin manner, a soft processor can avoid pipeline bubbles without the need for hazard detection logic [2, 4]: a pipeline with $N$ stages that supports $N-1$ threads can be fully utilized without hazard detection logic [4]. A multithreaded soft processor with an abundance of independent threads to execute is also compelling because it can tolerate memory and I/O latency [5], as well as the compute latency of custom hardware accelerators [6]. Such designs are particularly well-suited to FPGA-based processors because (i) hazard detection logic can often be on the critical path and can require significant area [7], and (ii) using the block RAMs provided in an FPGA to implement multiple logical register files is comparatively fast and area-efficient.

The applications that we implement require (i) synchroniza-

tion between threads, resulting in synchronization latency (while waiting to acquire a lock) and (ii) *critical sections* (while holding a lock). To fulfil these requirements in a way that can scale to more than one processor, we implement locks with memory-mapped test-and-set registers.

## 2.1 Fast Critical Sections via Thread Scheduling with Static Hazard Detection

While a multithreaded processor provides an excellent opportunity to tolerate the resulting synchronization latency, the simple round-robin thread-issue schemes used previously fall short for two reasons: (i) issuing instructions from a thread that is blocked on synchronization (e.g., spin-loop instructions or a synchronization instruction that repeatedly fails) wastes pipeline resources; and (ii) a thread that currently owns a lock and is hence in a critical section only issues once every $N-1$ cycles (assuming support for $N-1$ thread contexts), exacerbating the synchronization bottleneck for the whole system. Hence we identified a method for *scheduling* threads that is more sophisticated than round-robin but does not significantly increase the complexity nor area of our soft multithreaded processor.

In our approach we *de-schedule* any thread that is awaiting a lock. In particular, any such thread will no longer have instructions issued until any lock is released in the system—at which point the thread may spin once attempting to acquire the lock and if unsuccessful it is blocked again.[1] Otherwise, for simplicity we would like to issue instructions from the unblocked threads in round-robin order.

To implement this method of scheduling we must first overcome two challenges. The first is relatively minor: to eliminate the need to track long latency instructions, our processors *replay* instructions that miss in the cache rather than stalling [5]. With non-round-robin thread scheduling, it is possible to have multiple instructions from the same thread in the pipeline at once—hence to replay an instruction, all of the instructions for that thread following the replayed instruction must be squashed to preserve the program order of instructions execution.

The second challenge is greater: to support any thread schedule other than round-robin means that there is a possibility that two instructions from the same thread might issue with an unsafe distance between them in the pipeline, potentially violating a data or control hazard. We solve this problem by performing *static hazard detection*: we identify hazards between instructions at compile time and encode hazard information into spare bits in the MIPS instruction encoding, decoding it when instructions are fetched into the instruction cache, and storing it by capitalizing on spare bits in the width of FPGA block-RAMs.

# 3. MULTIPROCESSOR ARCHITECTURE

Our base processor is a single-issue, in-order, 5-stage, 4-way multithreaded processor, shown to be the most area-efficient compared to a 3- and 7-stage pipeline in earlier work [5]. We eliminate the hardware multipliers from our processors, which are not heavily used by our applications. The processor is big-endian which avoids the need to perform network-to-host byte ordering transformations. To take advantage of the space available in the FPGA, we replicate our base processor core and interconnect the replicas to provide them with a coherent common view of the shared data.

[1]Note that a more sophisticated approach that we leave for future work would only unblock threads that are waiting on the particular lock that was released.

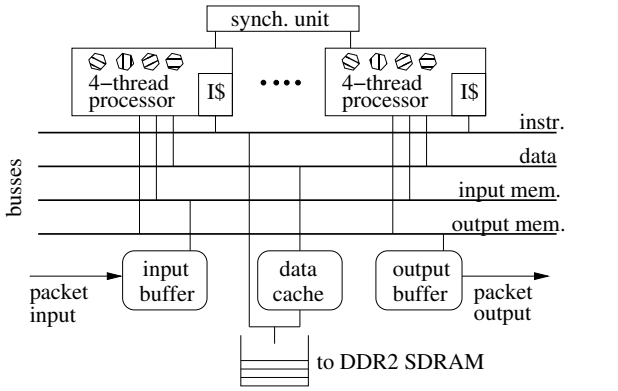



**Figure 1: The architecture of a 2-processor soft packet multiprocessor.**

As shown in Figure 1, the memory system is composed of a private instruction cache for each processor, and three data memories that are shared by all processors; this design is sensitive to the two-port limitation of block RAMs available on FPGAs. The first memory is an input buffer that receives packets on one port and services processor requests on the other port via a 32-bit bus, arbitrated across processors. The second is an output memory buffer that sends packets to the NetFPGA output-queues on one port, and is connected to the processors via a second 32-bit arbitrated bus on the second port. Both input and output memories are 16KB, allow single-cycle random access and are controlled through memory-mapped registers; the input memory is read-only and is logically divided into ten fixed-sized packet slots. The third is a shared memory managed as a cache, connected to the processors via a third arbitrated 32-bit bus on one port, and to a DDR2 SDRAM controller on the other port. For simplicity, the shared cache performs 32-bit line-sized data transfers with the DDR2 SDRAM controller (similar to previous work [8]), which is clocked at 200MHz. The SDRAM controller services a merged load/store queue of 16 entries in-order; since this queue is shared by all processors it serves as a single point of serialization and memory consistency, hence threads need only block on pending loads but not stores. Finally, each processor has a dedicated connection to a synchronization unit that implements 16 mutexes.

Soft processors are configurable and can be extended with accelerators as required, and those accelerators can be clocked at a separate frequency. To put the performance of the soft processors in perspective, handling a $10^9$ bps stream (with an inter-frame gap of 12 bytes) with 2 processors running at 125 MHz implies a maximum of 152 cycles per packet for minimally-sized 64B packets; and 3060 cycles per packet for maximally-sized 1518B packets. Since our multi-

processor architecture is bus-based, in its current form it will not easily scale to a large number of processors. However, as we demonstrate later in Section 6, our applications are mostly limited by synchronization and critical sections, and not contention on the shared buses; in other words, the synchronization inherent in the applications is the primary roadblock to scalability.

# 4. OUR NETFPGA PROGRAMMING ENVIRONMENT

This section describes our NetFPGA programming environment including how software is compiled, our NetFPGA configuration, and how we do timing, validation, and measurement.

**Compilation:** Our compiler infrastructure is based on modified versions of `gcc` 4.0.2, `Binutils` 2.16, and `Newlib` 1.14.0 that target variations of the 32-bit MIPS I [9] ISA. We modify MIPS to support 3-operand multiplies (rather than MIPS Hi/Lo registers [4,7]), and eliminate branch and load delay slots. Integer division and multiplication are both implemented in software. To minimize cache line conflicts in our direct-mapped data cache, we align the top of the stack of each software thread to map to equally-spaced blocks in the data cache.

**NetFPGA Configuration:** Our processor designs are inserted inside the NetFPGA 2.1 `Verilog` infrastructure [1], between a module arbitrating the input from the four 1GigE Media Access Controllers (MACs) and a CPU DMA port and a module managing output queues in off-chip SRAM. We added to this base framework a memory controller configured through the Xilinx Memory Interface Generator to access the 64 Mbytes of on-board DDR2 SDRAM. The system is synthesized, mapped, placed, and routed under high effort to meet timing constraints by Xilinx ISE 10.1.03 and targets a Virtex II Pro 50 (speed grade 7ns).

**Timing:** Our processors run at the clock frequency of the Ethernet MACs (125MHz) because there are no free PLLs (a.k.a. Xilinx DCMs) after merging-in the NetFPGA support components. Due to these stringent timing requirements, and despite some available area on the FPGA, (i) the private instruction caches and the shared data write-back cache are both limited to a maximum of 16KB, and (ii) we are also limited to a maximum of two processors. These limitations are not inherent in our architecture, and would be relaxed in a system with more PLLs and a more modern FPGA.

**Validation:** At runtime in debug mode and in RTL simulation (using `Modelsim` 6.3c [10]) the processors generate an execution trace that has been validated for correctness against the corresponding execution by a simulator built on MINT [11]. We also extended the simulator to model packet I/O and validated it for timing accuracy against the RTL simulation. The simulator is also able to process packets outgoing or incoming from network interfaces, virtual network (tap) devices and packet traces.

**API:** The memory mapped clock and packet I/O registers are accessible through a simple non-intrusive application programming interface (the API has less than twenty calls), that is easy to build upon. We have developed a number of test applications providing a wealth of routines such as bitmap operations, checksum routines, hashtable and read-write locks.

**Measurement:** We drive our design, for the packet echo experiment, with a generator that sends copies of the same preallocated packet through `Libnet` 1.4 and otherwise with a modified `Tcpreplay` 3.4.0 that sends packet traces from a Linux 2.6.18 Dell PowerEdge 2950 system, configured with two quad-core 2GHz Xeon processors and a Broadcom NetXtreme II GigE NIC connecting to a port of the NetFPGA used for input and a NetXtreme GigE NIC connecting to another NetFPGA port used for output. To simplify the analysis of throughput measurements, we allow packets to be processed out-of-order so long as the correctness of the application is preserved. We characterize the throughput of the system as being the maximum sustainable input packet rate. We derive this rate by finding, through a bisection search, the smallest fixed packet inter-arrival time where the system does not drop any packet when monitored for five seconds—a duration empirically found long enough to predict the absence of future packet drops at that input rate.

# 5. APPLICATIONS

In contrast with prior evaluations of packet-processing multiprocessor designs [12–14] we focus on *stateful* applications—i.e., applications in which shared, persistent data structures are modified during the processing of most packets. When the application is composed of parallel threads, accesses to such shared data structures must be synchronized. These dependences make it difficult to pipeline the code into balanced stages of execution to extract parallelism. Alternatively, we adopt the *run-to-completion/pool-of-threads* model, where each thread performs the processing of a packet from beginning-to-end, and where all threads essentially execute the same program code.

To take full advantage of the software programmability of our processors, our focus is on control-flow intensive applications performing deep packet inspection (i.e., deeper than the IP header). Network processing software is normally closely-integrated with operating system networking constructs; because our system does not have an operating system, we instead inline all low-level protocol-handling directly into our programs. To implement time-stamps and time-outs we require the hardware to implement a device that can act as the system clock. We have implemented the following packet processing applications, as detailed in Table 1 (Section 6.1), along with a precise traffic generator tool evaluated in another paper [15].

`UDHCP` is derived from the widely-used open-source DHCP server. The server processes a packet trace modeling the expected DHCP message distribution of a network of 20000 hosts [16]. As in the original code, leases are stored in a linearly traversed array and IP addresses are pinged before being leased, to ensure that they are unused.

`Classifier` performs a regular expression matching on TCP packets, collects statistics on the number of bytes transfered

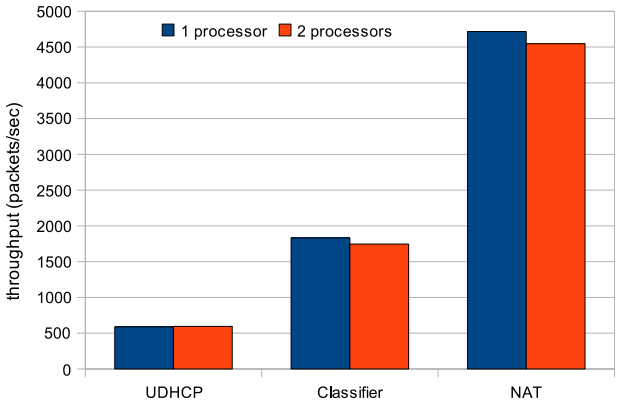



**Figure 2: Throughput (in packets per second) measured on the NetFPGA with either 1 or 2 CPUs.**

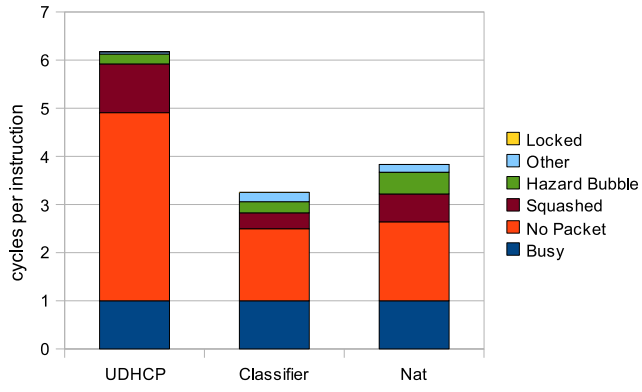



**Figure 3: Breakdown of how cycles are spent for each instruction (on average) in simulation.**

and monitors the packet rate for classified flows to exemplify network-based application recognition. In the absence of a match, the payloads of packets are reassembled and tested up to 500 bytes before a flow is marked as non-matching. As a use case, we configure the widely used PCRE matching library [17] with the HTTP regular expression from the "Linux layer 7 packet classifier" [18] and exercise our system with a publicly available packet trace [19] with HTTP server replies added to all packets presumably coming from an HTTP server to trigger the classification.

`NAT` exemplifies network address translation by rewriting packets from one network as if originating from one machine, and appropriately rewriting the packets flowing in the other direction. As an extension, `NAT` collects flow statistics and monitors packet rates. Packets originate from the same packet trace as `Classifier`, and like `Classifier`, flow records are kept in a synchronized hash table.

# 6. EXPERIMENTAL RESULTS

We begin by evaluating the raw performance that our system is capable of, when performing minimal packet processing for tasks that are completely independent (i.e., unsynchronized). We estimate this upper-bound by implementing a simple packet echo application that retransmits to a different network port each packet received. With minimum-sized packets of 64B, the echo program executes 300±10 dynamic instructions per packet (essentially to copy data from the input buffer to the output buffer as shown in Figure 1), and a single round-robin CPU can echo 124 thousand packets/sec (i.e., 0.07 Gbps). With 1518B packets, the maximum packet size allowable by Ethernet, each echo task requires 1300±10 dynamic instructions per packet. With two CPUs and 64B packets, or either one or two CPUs and 1518B packets, our PC-based packet generator cannot generate packets fast enough to saturate our system (i.e., cannot cause packets to be dropped). This amounts to more than 58 thousand packets/sec (>0.7 Gbps). Hence the scalability of our system will ultimately be limited by the amount of computation per packet/task and the amount of parallelism across tasks, rather than the packet input/output capabilities of our system.

Figure 2 shows the maximum packet throughput of our (real) hardware system with thread scheduling. We find that our applications do not benefit significantly from the addition of a second CPU due to increased lock and bus contention and cache conflicts: the second CPU either slightly improves or degrades performance, motivating us to determine the performance-limiting factors.

## 6.1 Identifying the Bottlenecks

To reduce the number of designs that we would pursue in real hardware, and to gain greater insight into the bottlenecks of our system, we developed a simulation infrastructure. While verified for timing accuracy, our simulator cannot reproduce the exact order of events that occurs in hardware, hence there is some discrepancy in the reported throughput. For example, `Classifier` has an abundance of control paths and events that are sensitive to ordering such as routines for allocating memory, hash table access, and assignment of mutexes to flow records. We depend on the simulator only for an approximation of the relative performance and behavior of applications on variations of our system.

To obtain a deeper understanding of the bottlenecks of our system, we use our simulator to obtain a breakdown of how cycles are spent for each instruction, as shown in Figure 3. In the breakdown, a given cycle can be spent executing an instruction (`busy`), awaiting a new packet to process (`no packet`), awaiting a lock owned by another thread (`locked`), squashed due to a mispredicted branch or a preceding instruction having a memory miss (`squashed`), awaiting a pipeline hazard (`hazard bubble`), or aborted for another reason (`other`, memory misses or bus contention). Figure 3 shows that our thread scheduling is effective at tolerating almost all cycles spent spinning for locks. The fraction of time spent waiting for packets (`no packet`) is significant and is a result of reducing the worst-case processing latency of a small fraction of packets. The fraction of cycles spent on squashed instructions (`squashed`) is significant with our thread scheduling scheme: if one instruction must replay, we must also squash and replay any instruction from that thread that has already issued. The fraction of cycles spent on bubbles (`hazard bubble`) is significant: this indicates that the CPU is frequently executing instructions from only

| Benchmark | Dyn. Instr. ×1000 /packet | Dyn. Sync. Instr. %/packet | Sync. Uniq. Addr. /packet | |
|---|---|---|---|---|
| | | | Reads | Writes |
| UDHCP | 34.9±36.4 | 90±105 | 5000±6300 | 150±60 |
| Classifier | 12.5±35.0 | 94±100 | 150±260 | 110±200 |
| NAT | 6.0±7.1 | 97±118 | 420±570 | 60±60 |

**Table 1: Application statistics (mean±standard-deviation): dynamic instructions per packet, dynamic synchronized instructions per packet (i.e., in a critical section) and number of unique synchronized memory read and write accesses.**

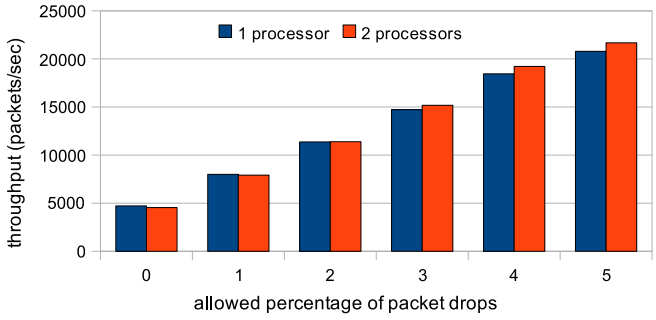



**Figure 4: Throughput in packets per second for NAT as we increase the tolerance for dropping packets from 0 to 5%, with either 1 or 2 CPUs.**

one thread, with the other threads blocked awaiting locks.

In Table 1, we measure several properties of the computation done per packet in our system. First, we observe that task size (measured in dynamic instructions per second) has an extremely large variance (the standard deviation is larger than the mean itself for all three applications). This high variance is partly due to our applications being best-effort unpipelined C code implementations, rather than finely hand-tuned in assembly code as packet processing applications often are. We also note that the applications spend over 90% of the packet processing time either awaiting synchronization or within critical sections (dynamic synchronized instructions), which limits the amount of parallelism and the overall scalability of any implementation, and in particular explains why our two CPU implementation provides little additional benefit over a single CPU. These results motivate future work to reduce the impact of synchronization, as discussed in Section 8.

Our results so far have focused on measuring throughput when zero packet drops are tolerated (over a five second measurement). However, we would expect performance to improve significantly for measurements when packet drops are tolerated. In Figure 4, we plot throughput for NAT as we increase the tolerance for dropping packets from 0 to 5%, and find that this results in dramatic performance improvements for both fixed round-robin and our more flexible thread scheduling—confirming our hypothesis that task-size variance is undermining performance.

## 6.2 FPGA resource utilization

Our two-CPU full system hardware implementation consumes 165 block RAMs (out of 232; i.e., 71% of the total capacity). The design occupies 15,671 slices (66% of the total capacity) and more specifically, 23158 4-input LUTs when optimized with high-effort for speed. Considering only a single CPU, the synthesis results give an upper bound frequency of 129MHz.

# 7. CONCLUSIONS

In most cases, network processing is inherently parallel between packet flows. We presented techniques to improve upon commercial off-the-shelf soft processors and take advantage of the parallelism in stateful parallel applications with shared data and synchronization. We implemented a multithreaded multiprocessor and presented a compilation and simulation framework that makes the system easy to use for an average programmer. We observed that synchronization was a bottleneck in our benchmark applications and plan to pursue work in that direction.

# 8. FUTURE WORK

In this section, we present two avenues to improve on our architecture implementation to alleviate some of its bottlenecks.

**Custom Accelerators** Because soft processors do not have the high operating frequency of ASIC processors, it is useful for some applications to summarize a block of instructions into a single custom instruction [20]. The processor interprets that new instruction as a call to a custom logic block (potentially written in a hardware description language or obtained through behavioral synthesis). We envision that this added hardware would be treated like another processor on chip, with access to the shared memory busses and able to synchronize with other processors. Because of the bit-level parallelism of FPGAs, custom instruction can provide significant speedup to some code sections [21, 22].

**Transactional Execution** When multiple threads/processors collaborate to perform the same application, synchronization must often be inserted to keep shared data coherent. With multiple packets serviced at the same time and multiple packet flows tracked inside a processor, the shared data accessed by all threads is not necessarily the same, and can sometimes be exclusively read by some threads. In those cases, critical sections may be overly conservative by preventively reducing the number of threads allowed in a critical section. Reader and writer locks may not be applicable, or useful, depending on the implementation. To alleviate the problem, a possibility is to allow a potentially unsafe number of threads in a critical section, detect coherence violations if any, abort violated threads and restart them in an earlier checkpointed state. If the number of violations is small, the parallelism, and the throughput, of the application can be greatly increased [23].

NetThreads is available online [24].

## 9. REFERENCES


[1] 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 *Proc. of MSE '07*, June 3-4 2007.

[2] B. Fort, D. Capalija, Z. G. Vranesic, and S. D. Brown, "A multithreaded soft processor for SoPC area reduction," in *Proc. of FCCM '06*, 2006, pp. 131–142.

[3] R. Dimond, O. Mencer, and W. Luk, "Application-specific customisation of multi-threaded soft processors," *IEE Proceedings—Computers and Digital Techniques*, vol. 153, no. 3, pp. 173– 180, May 2006.

[4] M. Labrecque and J. G. Steffan, "Improving pipelined soft processors with multithreading," in *Proc. of FPL '07*, August 2007, pp. 210–215.

[5] M. Labrecque, P. Yiannacouras, and J. G. Steffan, "Scaling soft processor systems," in *Proc. of FCCM '08*, April 2008, pp. 195–205.

[6] R. Moussali, N. Ghanem, and M. Saghir, "Microarchitectural enhancements for configurable multi-threaded soft processors," in *Proc. of FPL '07*, Aug. 2007, pp. 782–785.

[7] M. Labrecque, P. Yiannacouras, and J. G. Steffan, "Custom code generation for soft processors," in *Proc. of RAAW '06*, Florida, US, December 2006.

[8] R. Teodorescu and J. Torrellas, "Prototyping architectural support for program rollback using FPGAs," in *Proc. of FCCM '05*, April 2005, pp. 23–32.

[9] S. A. Przybylski, T. R. Gross, J. L. Hennessy, N. P. Jouppi, and C. Rowen, "Organization and VLSI implementation of MIPS," Stanford University, CA, USA, Tech. Rep., 1984.

[10] Mentor Graphics Corp., "Modelsim SE," http://www.model.com, Mentor Graphics, 2004.

[11] J. Veenstra and R. Fowler, "MINT: a front end for efficient simulation of shared-memory multiprocessors," in *Proc. of MASCOTS '94*, NC, USA, January 1994, pp. 201–207.

[12] T. Wolf and M. Franklin, "CommBench - a telecommunications benchmark for network processors," in *Proc. of ISPASS*, Austin, TX, April 2000, pp. 154–162.

[13] G. Memik, W. H. Mangione-Smith, and W. Hu, "NetBench: A benchmarking suite for network processors," in *Proc. of ICCAD '01*, November 2001.

[14] B. K. Lee and L. K. John, "NpBench: A benchmark suite for control plane and data plane applications for network processors," in *Proc. of ICCD '03*, October 2003.

[15] G. Salmon, M. Ghobadi, Y. Ganjali, M. Labrecque, and J. G. Steffan, "NetFPGA-based precise traffic generation," in *Proc. of NetFPGA Developers Workshop'09*, 2009.

[16] B. Bahlmann, "DHCP network traffic analysis," *Birds-Eye.Net*, June 2005.

[17] "PCRE - Perl compatible regular expressions," [Online]. Available: http://www.pcre.org.

[18] "Application layer packet classifier for linux," [Online]. Available: http://l7-filter.sourceforge.net.

[19] Cooperative Association for Internet Data Analysis, "A day in the life of the internet," WIDE-TRANSIT link, January 2007.

[20] H.-P. Rosinger, "Connecting customized IP to the MicroBlaze soft processor using the Fast Simplex Link (FSL) channel," XAPP529, 2004.

[21] R. Lysecky and F. Vahid, "A study of the speedups and competitiveness of FPGA soft processor cores using dynamic hardware/software partitioning," in *Proc. of DATE '05*, 2005, pp. 18–23.

[22] C. Kachris and S. Vassiliadis, "Analysis of a reconfigurable network processor," in *Proc. of IPDPS*. Los Alamitos, CA, USA: IEEE Computer Society, 2006, p. 173.

[23] C. Kachris and C. Kulkarni, "Configurable transactional memory," in *Proc. of FCCM '07*, April 2007, pp. 65–72.

[24] "NetThreads - project homepage," [Online]. Available: http://netfpga.org/netfpgawiki/index.php/Projects:NetThreads.

# NetFPGA-based Precise Traffic Generation

Geoffrey Salmon, Monia Ghobadi, Yashar Ganjali
Department of Computer Science
University of Toronto
{geoff, monia, yganjali}@cs.toronto.edu

Martin Labrecque, J. Gregory Steffan
Department of Electrical and Computer Engineering
University of Toronto
{martinl,steffan}@eecg.toronto.edu

## ABSTRACT

Generating realistic network traffic that reflects different network conditions and topologies is crucial for performing valid experiments in network testbeds. Towards this goal, this paper presents Precise Traffic Generator (PTG), a new tool for highly accurate packet injections using NetFPGA. PTG is implemented using the NetThreads platform, an environment familiar to a software developer where multithreaded C programs can be compiled and run on the NetFPGA. We have built the PTG to take packets generated on the host computer and transmit them onto a gigabit Ethernet network with very precise inter-transmission times. Our evaluations show that PTG is able to exactly reproduce packet inter-arrival times from a given, arbitrary distribution. We demonstrate that this ability addresses a real problem in existing software network emulators — which rely on generic Network Interface Cards for packet injections — and predict that the integration of PTG with these emulators would allow valid and convincing experiments which were previously difficult or impossible to perform in the context of network testbeds.

## 1. INTRODUCTION

Making any changes to the current Internet infrastructure is extremely difficult, if possible at all. Any new network component, protocol, or design implemented on a global scale requires extensive and accurate testing in sufficiently realistic settings. While network simulation tools can be very helpful in understanding the impact of a given change to a network, their predictions might not be accurate due to their simplified and restricted models and settings. Real network experiments are extremely difficult too: network operators usually do not like any modifications to their network, unless the proposed changes have been tested exhaustively in a large scale network. The only remaining option for testing the impact of a given change is using testbeds for network experiments.

To have meaningful experiments in a testbed, one must have realistic traffic. Generating high volumes of traffic is intrinsically difficult for several reasons. First, it is not always possible to use real network traces, as traces do not maintain the feedback loop between the network and traffic sources (for example the TCP closed-loop congestion feedback). Second, using a large number of machines to generate the required traffic is usually not an option, as it is extremely costly, and difficult to configure and maintain. Finally, depending on the purpose of the experiment, the generated traffic might have different sensitivity requirements. For example, in the context of testing routers with tiny buffers (e.g. 10-20 packets of buffering) even the slightest change in packet injection patterns can have major implications for the results of the experiment [1], whereas in the study of capacity planning techniques, the results are only sensitive to the aggregate bandwidth over relatively coarse timescales [2].

Commercial traffic generators are very useful for some experiments, but they have their own drawbacks. They are usually very expensive and their proprietary nature makes them very inflexible for research on new techniques and protocols. Also, it has been shown that their packet injection times are not accurate enough for time-sensitive experiments [3]. Additionally, commercial traffic generators do not always implement network protocols accurately: for example, Prasad *et al.* [4] describe differences observed between a TCP Reno packet sequence generated by a commercial traffic generator and the expected behavior of the standard TCP Reno protocol.

An alternative to a commercial traffic generator is open source packet generation software, where a small number of machines are used to generate high volumes of realistic traffic [2, 5]. These tools usually rely on generic hardware components that, depending on the vendor and model, can vary in their behavior; therefore, the output traffic might be inaccurate. For example, generic Network Interface Cards (NICs) usually provide little or no guarantees on the exact packet injection times. As a result, the actual traffic pattern depends on the NIC model and, depending on what model is used, significant differences in the generated traffic can be observed [1, 3].

Researchers at Stanford University have developed a packet generator that is capable of generating more precise traffic [6] (hereafter referred to as SPG), addressing the problems described above. The Stanford system is based on *NetFPGA*, a PCI-based programmable board containing an

FPGA, four gigabit Ethernet ports, and memory. The SPG system generates more accurate traffic by precisely replicating the transmission times recorded in a `pcap` trace file, similar to the operation of the `tcpreplay` software program; this method eliminates the dependence between the generated traffic and the NIC model.

While the traffic that SPG generates is more realistic than many prior approaches, it has several limitations. Because the trace files are based on past measurements, the closed-loop feedback for TCP sources (and any other protocol that depends on the feedback from the system) is not accurately captured. Furthermore, replaying a prerecorded trace on a link with different properties (such as capacity and buffer size) does not necessarily result in realistic traffic. Finally, SPG can only (i) replay the exact packet inter-arrival times provided by the trace file, or (ii) produce fixed inter-arrival times between packets (i.e., ignoring the variation of packet timings from the original trace).

In this paper, we introduce *Precise Traffic Generator* (PTG), a NetFPGA-based packet generator with highly-accurate packet injection times that can be easily integrated with various software-based traffic generation tools. PTG has the same accuracy level as SPG, but provides two key additional features that make it useful in a larger variety of network experiments: (i) packets in PTG are created dynamically and thus it can model the closed-loop behavior of TCP and other protocols; (ii) PTG provides the ability to follow a realistic distribution function of packet inter-arrival times such as the probability distributed functions presented by Katabi *et al.* [7][1].

PTG is built on *NetThreads* [8], a platform for developing packet processing applications on FPGA-based devices and the NetFPGA in particular. NetThreads is primarily composed of FPGA-based multithreaded processors, providing a familiar yet flexible environment for software developers: programs are written in `C`, and existing applications can be ported to the platform. In contrast with a PC or NIC-based solution, NetThreads is similar to a custom hardware solution because it allows the programmer to specify accurate timing requirements.

## 2. PRECISE TRAFFIC GENERATOR

In this section we present PTG, a tool which can precisely control the inter-transmission times of generated packets. To avoid implementing a packet generator in low-level hardware-description language (how FPGAs are normally programmed), we use NetThreads instead. We generate packets on the host computer and send them to the NetFPGA over the PCI bus. NetThreads provides eight threads that prepare and transmit packets. This configuration is particularly well-suited for packet generation: (i) the load of the threads' execution is isolated from the load on the host processor, (ii) the threads suffer no operating system overheads, (iii) they can receive and process packets in parallel, and (iv) they have access to a high-resolution system clock (much higher than that of the host clock).

[1]Here, we assume the distribution of different flows and the packet injection times are known a priori, and our goal is to generate traffic that is as close as possible to the given distribution.

In our traffic generator, packets are sent out of a single Ethernet port of the NetFPGA, and can have any specified sequence of inter-transmission times and valid Ethernet frame sizes (64-1518 bytes). PTG's main objective is to precisely control the transmission times of packets which are created in the host computer, continually streamed to the NetFPGA, and transmitted on the wire. Streaming packets is important because it implies the generator can immediately change the traffic in response to feedback. By not requiring separate load and send phases for packets, the PTG can support closed-loop traffic. PTG can easily be integrated with existing traffic generators to improve their accuracy at small time scales.

Let us start by going through the life cycle of a packet through the system, from creation to transmission. First a userspace process or kernel module on the host computer decides a packet should be sent at a particular time. A description of the packet, containing the transmission time and all the information necessary to assemble the packet is sent to the NetFPGA driver. In the driver, multiple packet descriptions are combined together and copied to the NetFPGA. Combining descriptions reduces the number of separate transfers required and is necessary for sending packets at the line rate of 1Gb/s. From there, the packet descriptions are each given to a single thread. Each thread assembles its packet in the NetThreads' output memory. Next, another thread sends all of the prepared packets in the correct order at the requested transmission times. Finally, the hardware pipeline of the NetFPGA transmits the packets onto the wire.

In the rest of this section we explain each stage of a packet's journey through the PTG in greater detail. We also describe the underlying limitations which influence the design.

**Packet Creation to Driver:** The reasons for and context of packet creation are application-specific. To produce realistic traffic, we envision a network simulator will decide when to send each packet. This simulation may be running in either a userspace process, like `ns-2` [9], or a Linux kernel module, as in ModelNet [10]. To easily allow either approach, we send packets to the NetFPGA driver using Linux NetLink sockets, which allow arbitrary messages to be sent and received from either userspace or the kernel. In our tests and evaluation, we create the packets in a userspace process.

At this stage, the messages sent to the NetFPGA driver do not contain the entire packet as it will appear on the wire. Instead, packets are represented by minimal descriptions which contain the size of the packet and enough information to build the packet headers. Optionally, the descriptions can also include a portion of the packet payload. The parts of the payload that are not set will be zeroes when the packet is eventually transmitted. In Section 4, we mention a work-around that may be useful when the contents of packet payloads are important to an experiment.

**Driver to NetThreads:** We modified the driver provided with NetFPGA to support the PTG. Its main task is to copy the packet descriptions to the NetFPGA card using DMA over the PCI bus. It also assembles the packet headers and computes checksums.

Sending packets to the NetFPGA over the PCI bus introduces some challenges. It is a 33MHz 32-bit bus with a top theoretical transfer rate of 1056 Mb/s, but there are significant overheads even in a computer where the bus is not shared. Most importantly, the number of DMA transfers between the driver and NetFPGA is limited such that the total throughput is only 260Mb/s when individually transferring 1518 byte packets. Limitations within the NetFPGA hardware pipeline mean we cannot increase the size of DMA transfers to the NetFPGA enough to reach 1 Gb/s. Instead we settle for sending less than 1 Gb/s across the PCI bus and rebuilding the packets inside the NetFPGA. Currently, the packet payloads are simply zeroed, which is sufficient both for our evaluation and for many of the tests we are interested in performing with the PTG.

To obtain the desired throughput, the driver combines the headers of multiple packets and copies them to the NetFPGA in a single DMA transfer. Next, the NetFPGA hardware pipeline stores them into one of the ten slots in the input memory of the NetThreads system. If there is no empty slot in the memory then the pipeline will stall, which would quickly lead to dropped packets in the input queue of the NetFPGA. To avoid this scenario, the software running on the NetThreads platform sends messages to the driver containing the number of packets that it has processed. This feedback allows the driver to throttle itself and to avoid overrunning the buffers in the NetFPGA.

**NetThreads to Wire:** The PTG runs as software on the NetThreads platform inside the NetFPGA. The driver sends its messages containing the headers of multiple packets and their corresponding transmission times, and the PTG needs to prepare these packets for transmission and send them at the appropriate times.

To achieve high throughput in NetThreads, it is important to maximize parallelism and use all eight threads provided by NetThreads. In the PTG, one thread is the sending thread. By sending all packets from a single thread we can ensure packets are not reordered and can easily control their transmission time. Each of the other seven threads continually pop a job from a work queue, performs the job and returns to the queue for another job. There are currently two types of jobs: 1) receive and parse a message from the driver and schedule further jobs to prepare each packet, and 2) prepare a packet by copying its header to the output memory and notifying the sending thread when complete.

When preparing outgoing packets, most of the work performed by the threads involves copying data from the input memory to the output memory. As described in [8], the buses to the input and output memories are arbitrated between both processors. For example, in a single clock cycle only one of the processors can read from the input memory. Similarly only one processor can write to the output memory in a given cycle. Fortunately, these details are hidden from the software, and the instructions squashed by an arbiter will be retried without impacting the other threads [11]. At first, it may appear that only one processor can effectively copy packet headers from the input memory to the output memory at any given time. However, the instructions that implement the `memcpy` function contain alternating loads and stores. Each 32-bit value must be loaded from the input memory into a register and then stored from the register into the output memory. Therefore, if two threads are copying packet headers, their load and store instructions will naturally interleave, allowing both threads to make forward progress.

# 3. EVALUATION

In this section we evaluate the performance of PTG by focussing on its accuracy and flexibility and also present measurements of an existing network emulator which clearly demonstrate the need that PTG fulfills. By contrast, previous works presenting traffic generators usually evaluate the realism of the resulting traffic [2,12]. While their evaluations present large test runs and often attempt to replicate the high-level properties seen in an existing network trace, our evaluation of PTG reflects its intended use; PTG is meant to complement existing traffic generators by allowing them to precisely control when packets are transmitted. Thus, we present relatively simple experiments where the most important metric is the accuracy of packet transmission times.

We perform our evaluations using Dell Power Edge 2950 servers running Debian GNU/Linux 5.0.1 (codename Lenny) each with an Intel Pro/1000 Dual-port Gigabit network card and a NetFPGA. In each test, there is a single server sending packets and a single server receiving packets and measuring their inter-arrival times. In the experiment described in Section 3.2, there is an additional server running a software network emulator which routes packets between the sender and receiver. The servers' network interfaces are directly connected – there are no intermediate switches or routers.

Since PTG's main goal is to transmit packets exactly when requested, the measurement accuracy is vital to the evaluation. As we discussed before, measuring arrival times in software using `tcpdump` or similar applications is imprecise; generic NICs combined with packet dumping software are intrinsically inaccurate at the level we are interested in. Therefore, we use a NetFPGA as the NIC to measure packet inter-arrival times at the receivers. Those receiving NetFPGAs are configured with the "event capturing module" of the NetFPGA router design [1] which provides timestamps of certain events, including when packets arrive at the router's output queues. To increase the accuracy of the timestamps, we removed two parts of the router pipeline that could add a variable delay to packets before they reach the output queues. This simplification is possible because we are only interested in measuring packets that arrive at a particular port and the routing logic is unnecessary. The timestamps are generated in hardware from the NetFPGA's clock and have a granularity of 8ns. We record these timestamps and subtract successive timestamps to obtain the packet inter-arrival times.

## 3.1 Sending Packets at Fixed Intervals

The simplest test case for the PTG is to generate packets with a fixed inter-transmission time. Comparing the requested inter-transmission time with the observed inter-arrival times demonstrates PTG's degree of precision.

As explained in Section 2, PTG is implemented as software running on what has previously been a hardware-only net-

work device, the NetFPGA. Even executing software, NetThreads should provide sufficient performance and control for precise packet generation. To evaluate this, we compare PTG's transmission times against those of SPG, which is implemented on the NetFPGA directly in hardware.

| Requested Inter-arrival (ns) | PTG mean error (ns) | SPG mean error (ns) |
|---|---|---|
| 1000000 | 8.57 | 8.46 |
| 500000 | 4.41 | 5.12 |
| 250000 | 3.89 | 3.27 |
| 100000 | 3.87 | 1.49 |
| 50000 | 1.87 | 0.77 |
| 25000 | 1.04 | 0.42 |
| 20000 | 0.82 | 0.35 |
| 15000 | 0.62 | 0.35 |
| 13000 | 0.54 | 0.27 |

**Table 1: Comparing the mean error in ns between the Precise Traffic Generator (PTG) and Stanford's packet generator (SPG)**

Table 1 shows the mean absolute error between the observed inter-arrival times and the requested inter-transmission times for various requested intervals. For each interval, we transmit 100000 packets of size 1518 bytes with both PTG and SPG. For inter-transmission times less than 50 $\mu$s, the average absolute error is less than 2ns for both packet generators. Note that clock period of 1000BASE-T gigabit Ethernet is 8ns, so an average error of 2ns implies most of the inter-transmission times are exactly as requested. This shows that even though NetThreads is executing software, it still allows precise control of when packets are transmitted.

Although both packet generators are of similar accuracy, SPG has a limitation that makes it unsuitable for the role we intend for the PTG. The packets sent by SPG must first be loaded onto the NetFPGA as a `pcap` file before they can be transmitted. This two-stage process means that SPG can only replay relatively short traces that have been previously captured[2]. Although it can optionally replay the same short trace multiple times to generate many packets, it can not continually be instructed to send packets by a software packet generator or network emulator using a series of departure times that are not known a priori. PTG, on the other hand, can be used to improve the precision of packet transmissions sent by any existing packet generation software.

## 3.2 Accuracy of Software Network Emulators

The goal of network emulators is to allow arbitrary networks to be emulated inside a single machine or using a small number of machines. Each packet's departure time is calculated based on the packet's path through the emulated network topology and on interactions with other packets. The result of this process is an ordered list of packets and corresponding departure times. How close the actual transmission times are to these ideal departure times is critical for the precision of the network emulator.

[2]The largest memory on the board is 64MB which is only about 0.5 seconds of traffic at the 1 Gb/s line rate.

Existing software network emulators have been built on Linux and FreeBSD [10, 13, 14]. To minimize overhead, all three process packets in the kernel and use a timer or interrupt firing at a fixed interval to schedule packet transmissions. They effectively divide time into fixed-size buckets, and all packets scheduled to depart in a particular bucket are collected and sent at the same time. Clearly, the bucket size controls the scheduling granularity; i.e., packets in the same bucket will essentially be sent back-to-back.

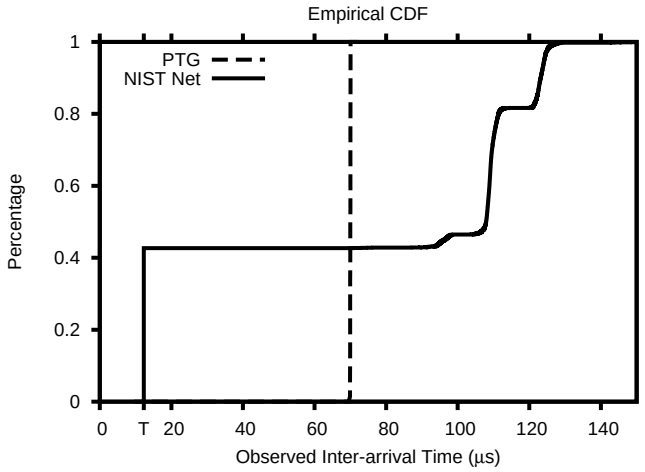



**Figure 1: Effect of NIST Net adding delay to packets sent 70 $\mu$s apart. $T = 12.304\mu s$ is the time it takes to transmit a single 1518-byte packet at 1 Gb/s: packets with that inter-arrival are effectively received back-to-back.**

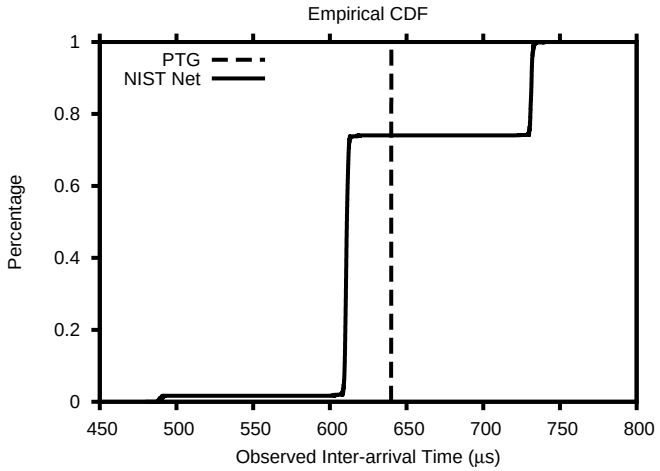



**Figure 2: Effect of NIST Net adding delay to packets sent 640 $\mu$s apart.**

To quantify the scheduling granularity problem, we focus on the transmission times generated by NIST Net [13], a representative network emulator. Here, we generate UDP packets with 1472 byte payloads at a fixed arrival rate using the PTG. The packets are received by a server run-

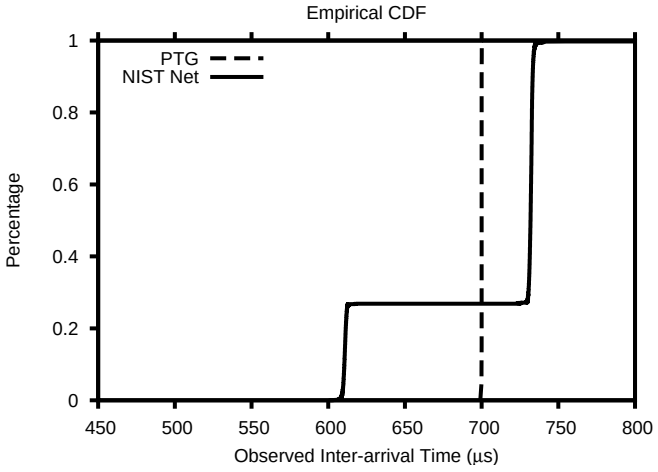



**Figure 3: Effect of NIST Net adding delay to packets sent 700 $\mu$s apart.**

ning NIST Net, pass through the emulated network, and are routed to a third server which measures the resulting packet inter-arrival times. NIST Net is configured to add 100ms of delay to each packet. Although adding a delay to every packet is a simple application of a network emulator, by varying the input packet inter-arrival times, NIST Net's scheduler inaccuracy is clearly visible.

Figure 1 is a CDF of the measured intervals between packet arrivals in NIST Net's input and output traffic. To measure the arrival times of the input traffic we temporarily connect the generating server directly to the measuring server. Here a packet is sent by the PTG to NIST Net, and thus should depart from NIST Net, every 70 $\mu$s. This interval is smaller than the fixed timer interval used by NIST Net, which has a period of 122 $\mu$s [13], so NIST Net will either send the packet immediately or in the next timer interval. Consequently, in Figure 1, 40% of the packets are received back-to-back if we consider that it takes just over 12 $\mu$s to transmit a packet of the given size on the wire (the transmission time of a single packet is marked with a "T" on the x-axis). Very few packets actually depart close to the correct 70 $\mu$s interval between them. Most of the remaining intervals are between 100 $\mu$s and 140 $\mu$s.

Even when the interval between arriving packets is larger than NIST Net's bucket size, the actual packet transmission times are still incorrect. Figures 2 and 3 show the measured arrival intervals for 640 $\mu$s and 700 $\mu$s arrivals, respectively. Note that in both figures, most of the intervals are actually either 610 $\mu$s or 732 $\mu$s, which are multiples of NIST Net's 122 $\mu$s bucket size. It is only possible for NIST Net to send packets either back-to-back or with intervals that are multiples of 122 $\mu$s. When we vary the inter-arrival time of the input traffic between 610 $\mu$s and 732 $\mu$s, it only varies the proportion of the output intervals that are either 610 $\mu$s or 732 $\mu$s.

The cause of the observed inaccuracies is not specific to NIST Net's implementation of a network emulator. Any software that uses a fixed-size time interval to schedule packet transmissions will suffer similar failures at small time scales, and the generated traffic will not be suitable for experiments that are sensitive to the exact inter-arrival times of packets. The exact numbers will differ, depending on the length of the fixed interval. To our knowledge, Modelnet [10] is the software network emulator providing the finest scheduling granularity of 100 $\mu$s with a 10KHz timer. Although higher resolution timers exist in Linux that can schedule a single packet transmission relatively accurately, the combined interrupt and CPU load of setting timers for every packet transmission would overload the system. Therefore, our conclusion is that an all-software network emulator executing on a general-purpose operating system requires additional hardware support (such as the one we propose) to produce realistic traffic at very small time scales.

## 3.3 Variable Packet Inter-arrival Times

Another advantage of PTG is its ability to generate packets with an arbitrary sequence of inter-arrival times and sizes. For example, Figure 4 shows the CDFs of both the requested and the measured transmission times for an experiment with 4000 packets with inter-arrival times following a Pareto distribution. Interestingly, only a single curve is visible in the figure since the two curves match entirely (for clarity we add crosses to the figure at intervals along the input distribution's curve). This property of PTG is exactly the component that the network emulators mentioned in Section 3.2 need. It can take a list of packets and transmission times and send the packets when requested. The crucial difference between PTG and SPG is that SPG has a separate load phase and could not be used by the network emulators.

As another example, Figure 5 shows the CDFs of the requested and the measured transmission times when the requested inter-arrival of packets follows the spike bump pattern probability density function observed in the study on packet inter-arrival times in the Internet by Katabi *et al.* [7]. Here 10000 packets are sent with packet sizes chosen from a simple distribution: 50% are 1518 bytes, 10% are 612 bytes, and 40% are 64 bytes. Note that, again, PTG generates the traffic exactly as expected and hence only one curve is visible.

# 4. DISCUSSION

In this section, we describe the limitations of PTG's current implementation. As PTG is intended to be integrated into existing traffic generators and network emulators, we also briefly describe a prototype we are developing that allows packets from the popular network simulator `ns-2` [9] to be sent on a real network using PTG.

**Limitations:** The limitations of the PTG stem from copying packets between the host computer and the NetFPGA over the 32 bit, 33 MHz PCI bus, which has a bandwidth of approximately 1Gb/s. As explained in Section 2, the payloads of packets sent by the PTG are usually all zeros, which requires sending only the packet headers over the PCI bus. This is sufficient for network experiments that do not involve packet payloads. A larger body of experiments ignore most of the packet payloads except for a minimal amount of application-layer signaling between sender and receiver. To support this, arbitrary custom data can be added to the start of any packet payload. This additional data is copied

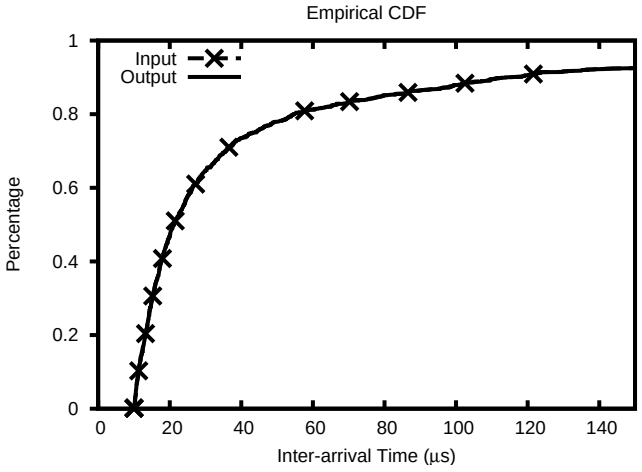



**Figure 4: CDF of measured inter-arrival times compared with an input pareto distribution.**

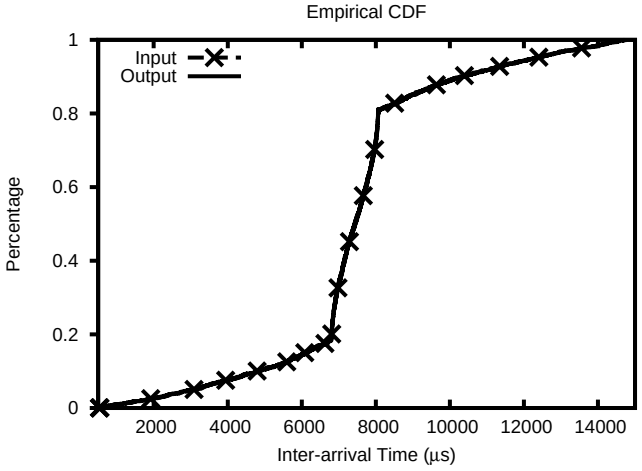



**Figure 5: CDF of measured inter-arrival times compared with an input distribution.**

to the NetFPGA card and is included in the packet. In the future, we plan to allow a number of predefined packet payloads to be copied to the NetFPGA in a preprocessing phase. These payloads could then be attached to outgoing packets without the need to repeatedly copy them over the PCI bus. We envision this feature would support many experiments where multiple flows send packets with the same or similar payloads.

The current PTG software implementation does not yet handle received packets from the network. For experiments with a high traffic volume, it would not be possible to transfer all of the received packet payloads from the 4 Gigabit Ethernet ports of the NetFPGA to the host computer over the PCI bus. Only a fraction of the packets could be transfered or the packets could be summarized by the NetFPGA.

**Integration with `ns-2`:** Because many researchers are already familiar with `ns-2`, this is a useful tool to test real network devices together with simulated networks. Compared to previous attempts to connect `ns-2` to a real network [15], the integration of PTG with `ns-2` will enable generating real packets with transmission times that match the `ns-2` simulated times even on very small time scales. For example, a particular link in `ns-2`'s simulated network could be mapped to a link on a physical network and when simulated packets would arrive at this link, they would be given to the PTG to be transmitted based on the requested simulated time.

# 5. CONCLUSION

Generating realistic traffic in network testbeds is challenging yet crucial for performing valid experiments. Software network emulators schedule packet transmission times in software, hence incurring unavoidable inaccuracy for inter-transmission intervals in the sub-millisecond range. Thus, they are insufficient for experiments sensitive to the inter-arrival times of packets. In this paper we present NetFPGA-based Precise Traffic Generator (PTG) built on top of the NetThreads platform. NetThreads allows network devices to be quickly developed for the NetFPGA card in software while still taking advantage of the hardware's low-level timing guarantees. The PTG allows packets generated on the host computer to be sent with extremely accurate inter-transmission times and it is designed to integrate with existing software traffic generators and network emulators. A network emulator that uses PTG to transmit packets can generate traffic that is realistic at all time scales, allowing researchers to perform experiments that were previously infeasible.

## Acknowledgments

This work was supported by NSERC Discovery, NSERC RTI as well as a grant from Cisco Systems.

# 6. REFERENCES

[1] N. Beheshti, Y. Ganjali, M. Ghobadi, N. McKeown, and G. Salmon, "Experimental study of router buffer sizing," in *IMC'08: Proceedings of the 8th ACM SIGCOMM conference on Internet measurement*, 2008.

[2] K. V. Vishwanath and A. Vahdat, "Realistic and responsive network traffic generation," in *SIGCOMM'06: Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*, 2006.

[3] N. Beheshti, Y. Ganjali, M. Ghobadi, N. McKeown, J. Naous, and G. Salmon, "Performing time-sensitive network experiments," in *ANCS'08: Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, 2008.

[4] R. Prasad, C. Dovrolis, and M. Thottan., "Evaluation of Avalanche traffic generator," 2007.

[5] A. Rupp, H. Dreger, A. Feldmann, and R. Sommer, "Packet trace manipulation framework for test labs," in *IMC'04: Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, 2004.

[6] G. A. Covington, G. Gibb, J. Lockwood, and N. McKeown, "A packet generator on the NetFPGA platform," in *FCCM'09: Proceedings of the 17th annual IEEE symposium on field-programmable custom computing machines*, 2009.

[7] D. Katabi and C. Blake, "Inferring congestion sharing and path characteristics from packet interarrival times," Tech. Rep., 2001.

[8] M. Labrecque, J. G. Steffan, G. Salmon, M. Ghobadi, and Y. Ganjali, "NetThreads: Programming NetFPGA with threaded software," in *NetFPGA Developers Workshop'09, submitted*.

[9] The Network Simulator - ns-2. http://www.isi.edu/nsnam/ns/.

[10] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostić, J. Chase, and D. Becker, "Scalability and accuracy in a large-scale network emulator," *SIGOPS Operating Systems Review archive*, vol. 36, no. SI, pp. 271–284, 2002.

[11] M. Labrecque, P. Yiannacouras, and J. G. Steffan, "Scaling soft processor systems," in *FCCM'08: Proceedings of the 16th annual IEEE symposium on field-programmable custom computing machines*, April 2008.

[12] J. Sommers and P. Barford, "Self-configuring network traffic generation," in *IMC'04: Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, 2004.

[13] M. Carson and D. Santay, "NIST Net: a Linux-based network emulation tool," *SIGCOMM Computer Communication Review*, vol. 33, no. 3, pp. 111–126, 2003.

[14] Dummynet. http://info.iet.unipi.it/ luigi/ip_dummynet/.

[15] "Network emulation in the Vint/NS simulator," in *ISCC'99: Proceedings of the The 4th IEEE Symposium on Computers and Communications*, 1999, p. 244.

# AirFPGA: A Software Defined Radio platform based on NetFPGA

Hongyi Zeng, John W. Lockwood
G. Adam Covington
Computer Systems Laboratory
Stanford University
Stanford, CA, USA
{hyzeng, jwlockwd,
gcoving}@stanford.edu

Alexander Tudor
Agilent Labs
Santa Clara, CA, USA
alex_tudor@agilent.com

## ABSTRACT

This paper introduces AirFPGA, a scalable, high-speed, remote controlled software defined radio (SDR) platform implemented using a NetFPGA board, a digitizer and a Radio Frequency (RF) receiver.

The AirFPGA is a system built on the NetFPGA that allows baseband recording and playback of wireless signals, as well as distributed processing. It captures radio signals, processes them in reconfigurable hardware, and sends the data via high speed Gigabit Ethernet to PCs or other NetFPGAs for additional computations. This paper describes the system architecture, data path, testbed implementation and test results. The paper demonstrates the system's verification using a signal generator. It also describes an application consisting of monitoring a commercial AM station.

## 1. INTRODUCTION

Software Defined Radio (SDR), which replaces typical hardware components of a radio system with personal computer software or other embedded computing devices, becomes a critical technology for both wireless service providers and researchers. A basic SDR may consist of a computer equipped with an analog-to-digital converter, preceded by RF front end. Large part of signal processing are on the general purpose processor, rather than special purpose hardware. It produces a radio that can operate a different form of radio protocol by just running different software.

Traditionally, SDRs combine data acquisition (RF antenna with analog-to-digital converter) and signal processing (CPU, FPGA, DSP, etc.) on the same device. This design will *limit the scale* of data acquisition units and *strain computing capacity* due to space and power availability.

For example, a multiple antenna system exploits spacial diversity to allows signals received from multiple antennas to be collected. Existing MIMO systems process signals on multiple antennas that are attached to the host device (e.g. a wireless router) at the same place. These antennas may suffer similar channel fading. Separating antennas from the host device, and combining waveforms collected by antennas at different locations can make the system stabler.

Some "computing intensive" applications, e.g. HDTV, radio astronomy telescope, and satellite receivers, also require decoupling the location of the signal processing from the antenna element. These applications often needs more computing resources than a single processing unit (FPGA/CPU), could provide. The antenna location often does not have the room and the power for a super computing facility. System designers may want to make use of a remote computing cluster, while maintaining the positions of antenna.

In conclusion, it is desirable that data acquisition and processing be performed separately in some SDR systems. AirFPGA is a SDR system that is designed for remote processing. It consists the NetFPGA card as radio to network interface, an RF receiver, and a digitizer. The base AirFPGA system can capture radio signals from RF receiver, packetize the data into UDP packets, and transmit it over NetFPGA's 4 Gigabit Ethernet ports. Developers can further reconfigure circuits to add in local DSP units according to the targeting application. These blocks may, for example, be downconverter, resampler, and various filters. These units are lightweight and mostly for data reduction/compression to meet the network constraints. AirFPGA can then be connected to one or more "master" signal processing nodes as long as they support UDP/IP stack.

## 2. SYSTEM ARCHITECTURE

The AirFPGA enables the construction of a signal processing network distributed platform consisting of NetFPGAs, PCs and other computing devices. The architecture consists of four parts: receiver and digitizer (A/D converter) integrated into a single device, the NetFPGA card, the radio server, and radio clients. This is shown in Figure 1. The Radio Frequency (RF) signal received by the antenna is down-converted, digitized to IQ pairs and transferred to the NetFPGA card. The NetFPGA packetizes the received data and sends it over Gigabit Ethernet. On the other side of the network radio clients receive the packets for further processing.

### 2.1 NetFPGA

The NetFPGA card [1, 2] is the core of the AirFPGA system. It is a network hardware accelerator that augments the function of a standard computer. The card has four Gigabit Ethernet ports, Static RAM (SRAM) and Dynamic RAM (DRAM). The NetFPGA attaches to the Peripheral Communication Interconnect (PCI) bus on a PC. A Xilinx Virtex-II Pro FPGA on the NetFPGA performs all media access control (MAC) and network layer functions.

### 2.2 Radio Server

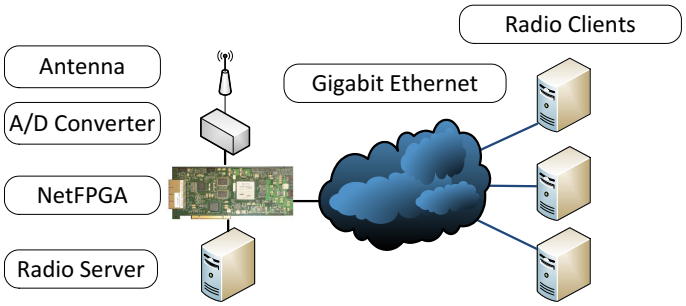



**Figure 1: AirFPGA System Architecture**

The radio server PC hosts the NetFPGA card. The packetization and forwarding are all accomplished on the FPGA.

A USB cable connects the radio [3] to the PC in which the NetFPGA card is installed. The radio server first reads data from the radio via USB cable, then loads it to NetFPGA's SRAM using the memory access tool provided by NetFPGA gateware. NetFPGA can then treat the data as though it was directly acquired from the radio, which is what we expect to accomplish in the next generation of this system. Further details will be given in Section 4.2.

## 2.3 Radio Client

The radio client is the receiver of packets generated by the NetFPGA card on the radio server PC. A radio client is a PC with or without a NetFPGA card. It can also be another NetFPGA card in a serial signal processing chain, or another device that has a large amount of Digital Signal Processing (DSP) resources, such as the ROACH [4].

# 3. FEATURES

The AirFPGA has the functionality that enables its novel use as a network distributed signal processing platform.

## 3.1 Scalability

The NetFPGA card has 4 Gigabit Ethernet ports. Several clients may participate in the computation needed for signal processing. Signals buffered by the NetFPGA card can be sent through Gigabit Ethernet to devices that have processing resources. In this case, which is the system's current implementation, NetFPGA is a data transport device. Other processing topologies of this "DSP Network" can be envisioned.

## 3.2 High speed

The AirFPGA's current design uses a single NetFPGA Gigabit Ethernet port. Up to 190kHz baseband bandwidth can be captured and streamed by the radio. Even though the onboard ADC samples 14bits at 66MS/s, the data is decimated according to the chosen narrow-band demodulation in order to fit the USB connection's 400Mb/s speed. Future work will use all four Gigabit Ethernet ports to enable wide-band modulations and MIMO.

Xilinx Virtex II Pro is suitable for processing narrow-band signals; its use for partial processing for wide-band is the subject of further investigation. For that purpose, Xilinx's DSP IP cores (e.g. numerically controlled oscillators (NCO), mixers, filters, FIFOs, etc.) will be considered.

## 3.3 Remote Controlled

AirFPGA separates data acquisition and signal processing. Radio clients and the radio server are logically and physically separated. You can listen to an AM radio far away from your town by deploying a radio server there. With more bandwidth you can watch HDTV received on your roof when you are abroad, or you can monitor the electromagnetic environment in a particular region.

# 4. IMPLEMENTATION

## 4.1 Reference Pipeline

The original reference pipeline of NetFPGA, as shown in Figure 2, is comprised of eight receive queues, eight transmit queues, and the user data path. The receive and transmit queues are divided into two types: MAC and CPU. The MAC queues are assigned to one of the four interfaces on the NetFPGA, and there is one CPU queue associated with each of the MAC queues.

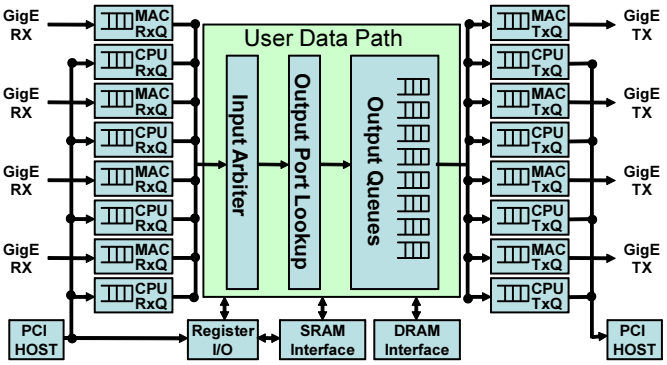



**Figure 2: NetFPGA Reference Pipeline**

Users add and connect their modules to the User Data Path. The Input Arbiter and the Output Queues modules are present in almost all NetFPGA designs. The source code for these modules are provided in the NetFPGA Verilog library [5]. The Input Arbiter services the eight input queues in a round robin fashion to feed a wide (64 bit) packet pipeline.

The register system allows software to read and write the contexts of key registers, counters and the contents of SRAM from the host PC. The architecture allows modules to be inserted into the pipeline with minimal effort. The register interface allows software programs running on the host system to send data to and receive data from the hardware modules.

## 4.2 AirFPGA Data Path

The User Data Path of the AirFPGA is shown in Figure 3. We remove all of 8 input queues, the input arbiter, and output queue lookup module, since NetFPGA is served as a transport an data acquisition device in the current architecture.

We are investigating additional NetFPGA functionality consisting of the radio control interface and possibly a PHY (physical layer decoder) and a PHY FEC (Forward Error Correction). The radio output is baseband IQ pairs or real

| CTRL | NetFPGA 64bit Data Path | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| - | **Bits 0-7** | **Bits 8-15** | **Bits 16-23** | **Bits 24-31** | **Bits 32-39** | **Bits 40-47** | **Bits 48-55** | **Bits 56-63** |
| 0xFF | port_dst 16 | | word_length 16 | | port_src 16 | | byte_length 16 | |
| 0x00 | mac_dst 48 | | | | | | mac_src_hi 16 | |
| 0x00 | mac_src_lo 32 | | | | mac_ethertype 16 | | ip_version 4 + ip_header_length 4 | ip_ToS 8 |
| 0x00 | ip_total_length 16 | | ip_id 16 | | ip_flags 3 + ip_flag_offset 13 | | ip_TTL 8 | ip_prot 8 |
| 0x00 | ip_header_checksum 16 | | ip_src 32 | | | | ip_dst_hi 16 | |
| 0x00 | ip_dst_lo 16 | | udp_src 16 | | udp_dst 16 | | udp_length 16 | |
| 0x00 | udp_checksum 16 | | airfpga_reserved 16 | | airfpga_seq_num 32 | | | |
| 0x00 | IQ 32 | | | | IQ 32 | | | |
| 0x00 | IQ 32 | | | | IQ 32 | | | |
| 0x00 | ... | | | | ... | | | |
| 0x01 | IQ 32 | | | | IQ 32 | | | |

**Table 1: Packet Format of AirFPGA**

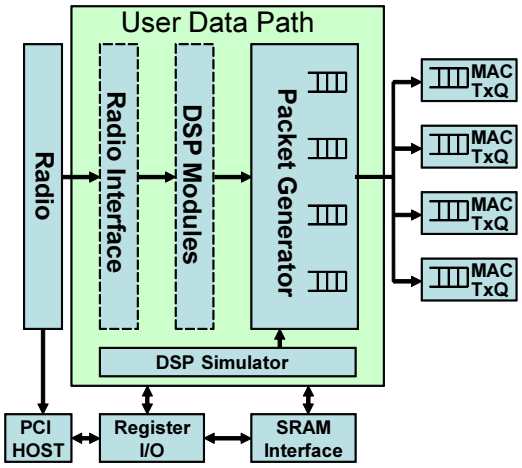



**Figure 3: AirFPGA data path. The envisioned path is shown in dotted lines, through the radio interface and DSP modules. The current path is through the DSP simulator that feeds the packet generator with "DSP results" directly from SRAM.**

numbers. Depending on how much processing takes place on the NetFPGA, IQ pairs, demodulated and decoded signals, or even MAC bits are packetized according to the packet format described in Section 4.3, and sent out the Gigabit Ethernet port(s). Only IQ pairs are now packetized.

The above describes a theoretical data path, still under investigation. The current AirFPGA implementation uses an alternative data path to circumvent the radio interface and DSP modules, while preserving the overall architecture. The radio server PC loads the radio signals to the NetFPGA's SRAM. The DSP simulator module reads the data from SRAM and feeds it to the packet generator as "DSP results". The remaining part of data path is the same as the previous path.

Radio clients receive packets using standard UDP/IP network sockets.

## 4.3 Packet Format

The NetFPGA data width is 64 bits, along with 8 bit control (CTRL) signals. Table 1 shows the packet format using for packetizing process. We choose UDP as transport layer protocol to achieve highest throughput and reduce the complexity of state machine design. Our applications are tolerant with potential packet loss with UDP flow. In UDP payload, we reserved first 16 bits for storing information used by the AirFPGA software, such as central frequency, power, antenna status, and data width. A 32 bit sequence number is embedded to maintain the order of packets and discover packet loss.

The headers from top down are: IOQ module headers defined in reference design for NetFPGA to route the packet correctly, IEEE 802.3 ethernet MAC header, IPv4 header, UDP header, AirFPGA header (reserved bits and sequence number). The AirFPGA payload consists a serial of 32 bit IQ data, where 16 bit I (in-phase) is preceded by 16 bit Q (quadrature).

## 4.4 Registers

The NetFPGA register interface exposes hardware's registers, counters and tables to the software and allows software to modify them. [6] This is achieved by memory-mapping the internal hardware registers. The memory-mapped registers appear as I/O registers to software. The software can then access the registers using ioctl calls.

The AirFPGA employs 11 main registers to realize runtime parameters modification and DSP simulator controlling. They are listed in Table 2.

# 5. TESTBED AND RESULTS

We have implemented AirFPGA on the NetFPGA board and built a testbed for verification. The architecture of the testbed is shown in Figure 5.

The AirFPGA testbed receives signals from an antenna or a signal generator. The receiver, an HF (30MHz) direct digital down-converter, can continuously stream up to 190kHz of spectrum. The product, called SDR-IQ, is manufactured by RFSpace Inc. The signal generator outputs an AM modulated RF signal. An antenna can also feed the SDR-IQ. The SDR-IQ down-converts the RF signal, digitizes it and converts it to IQ pairs. The samples are then sent to the radio server via USB. The radio server loads the IQ pairs to the NetFPGA's SRAM via the PCI interface. NetFPGA

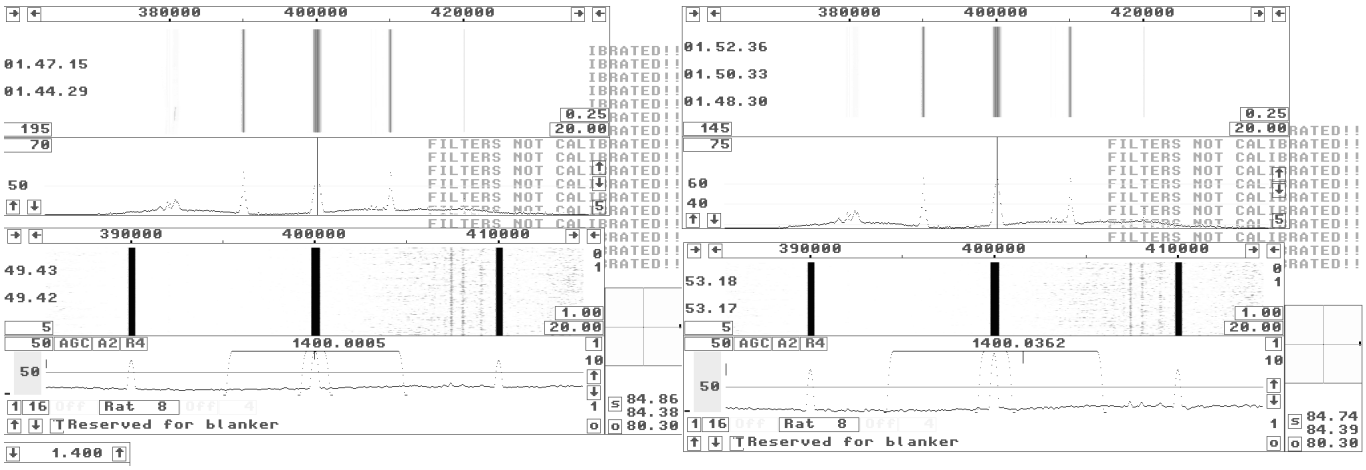



**Figure 4: 10kHz single tone AM at carrier frequency 1.4MHz. 3 main spectrum are shown on the waterfall: carrier at 1.4MHz, sidebands at 1.39MHz and 1.41MHz. The spectrum are shown on Linrad 1 (left) and Linrad 2 (right).**

| Register Name | Description |
|---|---|
| **Packet Parameters** | |
| AIRFPGA_MAC_SRC_HI | High 16 bits:source MAC |
| AIRFPGA_MAC_SRC_LO | Low 32 bits:source MAC |
| AIRFPGA_MAC_DST_HI | High 16 bits:dest. MAC |
| AIRFPGA_MAC_DST_LO | Low 32 bits:dest. MAC |
| AIRFPGA_IP_SRC | Source IP |
| AIRFPGA_IP_DST | Destination IP |
| AIRFPGA_UDP_SRC | Source UDP port |
| AIRFPGA_UDP_DST | Destination UDP port |
| **DSP Simulator Control** | |
| AIRFPGA_SIM_ADDR_LO | Start addr of SRAM |
| AIRFPGA_SIM_ADDR_HI | End addr of SRAM |
| AIRFPGA_SIM_ENABLE | Enable DSP Simulator |

**Table 2: Registers for AirFPGA**

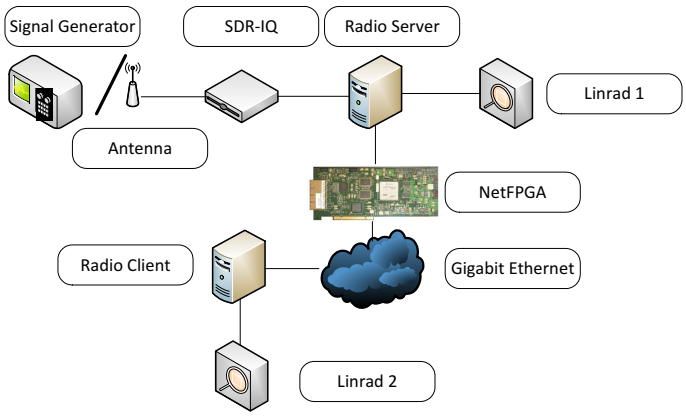



**Figure 5: AirFPGA Testbed**


**Figure 6: SDR-IQ**

packetizes and sends them over the Gigabit Ethernet.

## 5.1 SDR-IQ

SDR-IQ [3] (Figure 6) converts the RF signal into IQ pairs. It features a 14 bits analog to digital converter. The device is supported by several platforms, for either Windows or Linux, as a front end for spectrum analysis and dozens narrowband modulations, both analog and digital.

The hardware directly converts 30MHz to IQ pairs using a direct digital converter (DDC) chip from Analog Devices (AD6620) running at 66.6MHz.

The SDR-IQ comes with an HF amplified front-end with switched attenuators, switched filters and 1Hz tuning.

## 5.2 Linrad

The testbed has two Linrad's [7] running on the server and client side. Linrad is a software spectrum analyzer and demodulator running under Linux (as well as Microsoft Windows). For signal integrity verification we compare the displayed spectrum between two Linrad(s) as shown in Section 5.3.

Linrad operates with any sound card when audible modulations are used. Linrad supports the SDR-IQ and other hardware platforms.

The Linrad DSP software is independent of the hardware. It can process any bandwidth produced by the hardware subject to the computing resources of the PC on which Linrad is running. Linrad has a general purpose architecture and can be seen as a receiver design kit.

## 5.3 Signal Generator + SDR-IQ + AirFPGA

In the first scenario, we connect an Agilent E8267D PSG Vector Signal Generator to the SDR-IQ. It generates analog (AM, FM) and digital (ASK, FSK, MSK, PSK, QAM) modulated signals.

Figure 4 shows how a single tone AM signal looks like on Linrad's running at the radio server (left) and the radio client (right). The upper half of Linrad is the wideband waterfall and spectrum. The carrier is at 1.4MHz and two modulating sidebands are on 1.39MHz and 1.41MHz respectively. The baseband waterfall and spectrum is on the lower part.

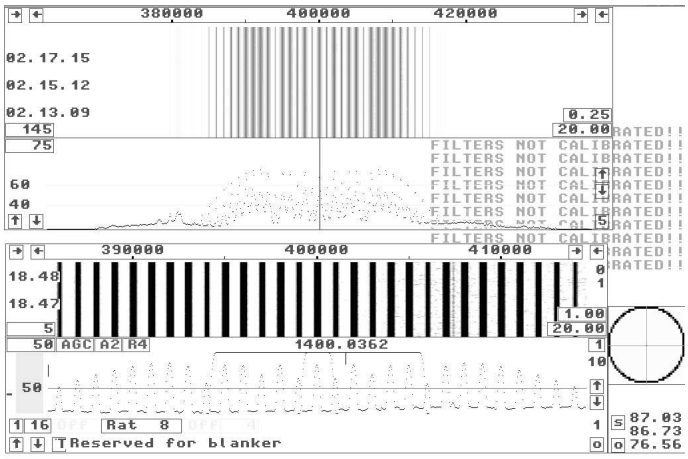



**Figure 7: 1kHz single tone ($\beta = 10$) Wideband FM signal at carrier frequency 1.4 MHz shown on client side Linrad. Several spectrum spaced by 1kHz.**

Figure 7 shows how an FM signal looks like on client side Linrad. It is a 1kHz single tone signal at carrier frequency 1.4MHz. We can see several spectrum spaced by 1kHz with magnitude of $n$-order modified Bessel function evaluated at $\beta = 10$.

## 5.4 Antenna + SDR-IQ + AirFPGA

In this scenario an AM antenna is connected to the SDR-IQ. Due to the limitation of SDR-IQ (30MHz), we are only able to receive commercial AM stations (520kHz-1,610kHz).

The AM antenna is a thin wire plugged into the RF input jack of the SDR-IQ.

Figure 8 shows the spectrum of a local AM station (KLIV 1590kHz) in Bay Area. Linrad is able to demodulate the received AM signal and send the waveform to the sound card. We can listen to this station either at the server side or the client side.

A demonstration video is available online[8].

# 6. DEVICE UTILIZATION

Table 3 describes the device utilization of AirFPGA. AirFPGA uses 38% of the available slices on the Xilinx Virtex II Pro 50 FPGA. The largest use of the slices are from the packet generator that packetizes DSP results into IP packets. 19% of the block RAMs available are used. The main use of block RAMs occurs in the FIFOs used between the modules and the main input and output queues of the system.

| Resources | XC2VP50 Utilization | Utilization Percentage |
|---|---|---|
| Slices | 9,210 out of 23,616 | 38% |
| 4-input LUTS | 10,204 out of 47,232 | 23% |
| Flip Flops | 9,148 out of 47,232 | 19% |
| Block RAMs | 46 out of 232 | 19% |
| External IOBs | 356 out of 692 | 51% |

**Table 3: Device utilization for AirFPGA**

# 7. RELATED WORK

The AirFPGA is not the first system designed for SDR. There are a number of popular platforms for the amateur radio community, such as SDR1000 from FlexRadio Systems [9], Softrock40 from American QRP [10]. These commercial platforms have no open source design, thus are difficult to modify by users.

The HPSDR [11] is an open source hardware and software SDR project for use by Radio Amateurs ("hams") and Short Wave Listeners (SWLs). It is being designed and developed by a group of SDR enthusiasts with representation from interested experimenters worldwide. The discussion list membership currently stands at around 750 and includes such SDR enthusiasts. However, HPSDR is designed for radio-amateur analog and digital communications.

GNU Radio [12] is a free software development toolkit that provides the signal processing runtime and processing blocks to implement software radios using readily-available, low-cost external RF hardware and commodity processors. It is widely used in hobbyist, academic and commercial environments to support wireless communications research as well as to implement real-world radio systems.

Rice University's WARP [13, 14, 15] is a scalable, extensible and programmable wireless platform to prototype wireless networks. The open-access WARP repository allows exchange and sharing of new physical and network layer architectures, building a true community platform. Xilinx FPGAs are used to enable programmability of both physical and network layer protocols on a single platform, which is both deployable and observable at all layers.

These two systems are designed for future digital wireless research, and popular in the academic world. GNU Radio and WARP both rely on centralized digital signal processing in one chip, thus lacking scalability.

Some platforms, although not designed for SDR, provide superior computing ability that can be used in SDR processing. UC Berkeley's BEE2 and ROACH are two examples. These boards can serve as radio clients in the AirFPGA architecture.

The BEE2 board [16] was originally designed for high-end reconfigurable computing applications such as ASIC design. It has 500 Gops/sec of computational power provided by 5 Xilinx XC2VP70 Virtex-II Pro FPGAs. Each FPGA connects to 4GB of DDR2-SDRAM, and all FPGAs share a 100Mbps Ethernet port.

The ROACH board [4] is intended as a replacement for BEE2 boards. A single Xilinx Virtex-5 XC5VSX95T FPGA provides 400 Gops/sec of processing power and is connected to a separate PowerPC 440EPx processor with a 1 Gigabit Ethernet connection. The board contains 4GB of DDR2 DRAM and two 36Mbit QDR SRAMs.

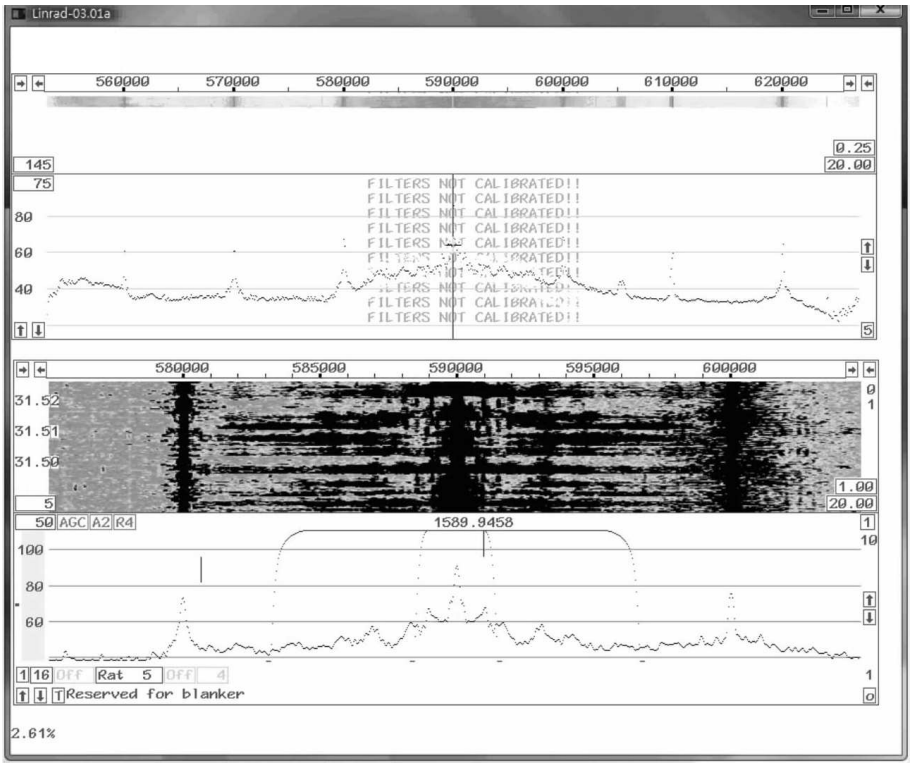



**Figure 8: KLIV (1590kHz) spectrum on AirFPGA**

# 8. FUTURE WORK

## 8.1 Multicast

The current implementation supports multicast IP address [17]. The destination IP address can be specified as a 224.x.y.z multicast IP address. Another solution is to implement 4 unicast UDP connection at the same time. The NetFPGA card has 4 Gigabit Ethernet ports. The additional ports could allow packets to be delivered to more than one port if there are multiple recipients of the data, each at a unique destination IP address. Four copies of each packet can be sent. They are different only in the destination IP address and the corresponding next-hop MAC address. The MAC address can be determined by ARP protocol, which has been implemented in the NetFPGA reference router.

## 8.2 Radio Daughterboard

We are developing a radio daughterboard and interface module that connects the radio directly to the NetFPGA. The daughterboard is the physical interface between the radio (consisting of RF front-end and analog to digital converter) and NetFPGA. Data is transmitted from the radio to the NetFPGA card through the daughterboard, and NetFPGA controls the radio through the same interface. The NetFPGA card provides several General Purpose Input-Output (GPIO) pins, which are suitable for interfacing with the daughterboard.

## 8.3 DSP Modules

We have not yet implemented any DSP modules on the NetFPGA. For the next stage we are investigating partial PHY processing of wide-band signals on the NetFPGA in order to reduce the transmitted data rate.

# 9. CONCLUSION

The AirFPGA is a network based SDR platform that performs distributed signal processing over high speed Ethernet. It is a novel server-client architecture designed for complex digital signal processing in wireless communication. The radio on the server side can be remotely controlled and the data it captures can be sent through Gigabit Ethernet to remote machines. The prototype of AirFPGA has been implemented on NetFPGA along with SDR-IQ and Linrad for capturing and displaying of analog modulations. Future versions of AirFPGA will include multicast, a radio daughterboard, and local digital signal processing, etc. Further information and demonstration on the AirFPGA can be found on AirFPGA's website. [8]

# 10. REFERENCES

[1] G. Gibb, J. W. Lockwood, J. Naous, P. Hartke, and N. McKeown, "NetFPGA: an open platform for teaching how to build gigabit-rate network switches and routers," *IEEE Transactions on Education*, vol. 51, no. 3, pp. 364–369, Aug. 2008.

[2] 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 *Microelectronic Systems Education, 2007. MSE '07. IEEE International Conference on*, San Diego, CA, June 2007, pp. 160–161.

[3] Rfspace Inc., "SDR-IQ Receiver," http://www.rfspace.com/SDR-IQ.html.
[4] ROACH Group, "ROACH Homepage," http://casper.berkeley.edu/.
[5] NetFPGA Team, "NetFPGA website," http://netfpga.org/.
[6] G. A. Covington, G. Gibb, J. Naous, J. Lockwood, and N. McKeown, "Methodology to contribute NetFPGA modules," in *International Conference on Microelectronic Systems Education (submitted to)*, 2009.
[7] Leif Asbrink, SM5BSZ, "Linrad Website," http://www.sm5bsz.com/linuxdsp/linrad.htm.
[8] AirFPGA Group, "AirFPGA Homepage," http://www.netfpga.org/airfpga.
[9] FlexRadio Systems, "FlexRadio Website," http://www.flex-radio.com/.
[10] "SoftRock-40," http://www.amqrp.org/kits/softrock40/.
[11] HPSDR, "High Performance Software Defined Radio," http://hpsdr.org/.
[12] GNU Radio Group, "GNU Radio Website," http://www.gnu.org/software/gnuradio/.
[13] WARP Group, "WARP Website," http://warp.rice.edu.
[14] P. Murphy, A. Sabharwal, and B. Aazhang, "Design of WARP: a wireless open-access research platform," in *EURASIP XIV European Signal Processing Conference*, September 2006.
[15] K. Amiri, Y. Sun, P. Murphy, C. Hunter, J. R. Cavallaro, and A. Sabharwal, "WARP, a unified wireless network testbed for education and research," in *Microelectronic Systems Education, 2007. MSE '07. IEEE International Conference on*, San Diego, CA, June 2007, pp. 53–54.
[16] C. Chang, J. Wawrzynek, and R. W. Brodersen, "BEE2: a high-end reconfigurable computing system," *IEEE Design & Test of Computers*, vol. 22, no. 2, pp. 114–125, Mar./Apr. 2005.
[17] S. Deering, "Host extensions for IP multicasting," Internet Engineering Task Force, RFC 1112, Aug. 1989. [Online]. Available: http://www.rfc-editor.org/rfc/rfc1112.txt

# Fast Reroute and Multipath Routing Extensions to the NetFPGA Reference Router

Hongyi Zeng, Mario Flajslik, Nikhil Handigol
Department of Electrical Engineering and Department of Computer Science
Stanford University
Stanford, CA, USA
{hyzeng, mariof, nikhilh}@stanford.edu

## ABSTRACT

This paper describes the design and implementation of two feature extensions to the NetFPGA reference router - fast reroute and multipath routing. We also share our insight into the inherent similarities of these two seemingly disparate features that enable us to run them simultaneously. Both features are designed to work at line-rate. With minimum modification of both hardware and software, the advanced features are tested and will be demonstrated on the NetFPGA card.

## 1. INTRODUCTION

One of the many systems built using NetFPGA is an IPv4 reference router [1]. The router runs the Pee-Wee OSPF [2] routing protocol, and does address lookup and packet forwarding at line-rate.

In this paper, we present two feature extensions to the NetFPGA reference router:

- *Fast reroute* - Detection of link failure or topology change in the reference router is generally based on the OSPF messages timing out. However, this causes packets to be dropped in the interval between the actual failure and failure detection. These intervals are as large as 90 seconds in PW-OSPF. Fast reroute [3] is a technique that detects link failures at the hardware level and routes packets over alternative routes to minimize packet drops. These alternative routes are pre-computed by the router software.

- *Multipath routing* - Multipath routing [4] is a routing strategy where next-hop packet forwarding to a single destination can occur over multiple "best paths". This enables load-balancing and better utilization of available network capacity. Our implementation of multipath routing is similar to ECMP; packets are forwarded over only those paths that tie for top place in routing metric calculations. This has the two-fold advantage of keeping the routing protocol simple and robust as well as minimizing packet reordering.

This work was originally intended as an advanced feature project for CS344 "Building an Internet Router" class, in the year 2009 at Stanford University.

## 2. DESIGN

The main goal of CS344 class is to design an output port lookup module for the NetFPGA. This module takes incoming packets, parses header information, queries the routing table and ARP cache, labels the packet with output port information, and finally puts it in output queues. Along with other modules in NetFPGA gateware, a functional Internet router can be built.

### 2.1 Architecture

The overall architecture of output port lookup module is shown in Figure 1.

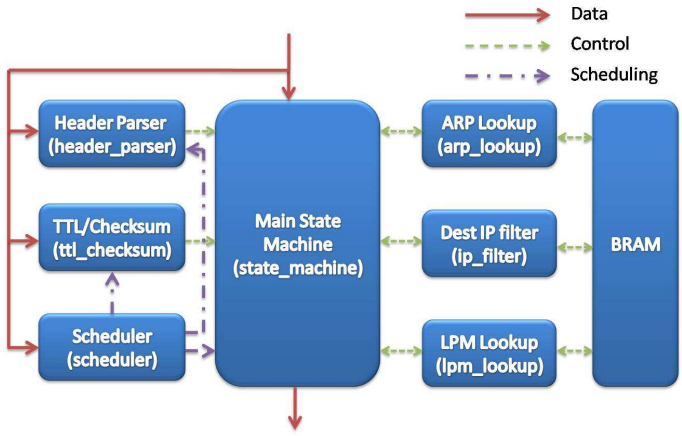



**Figure 1: Block Diagram of Output Port Lookup**

The scheduler provides "position" information to other modules. This simplifies the design of header parser and TTL/Checksum. The header parser parses the header of packets, and TTL/Checksum module manipulates TTL/Checksum information of IP packets.

There are three table lookup modules for ARP table, IP filter table, and routing table. The first two have similar lookup mechanism, while routing table lookup should be Longest Prefix Matching (LPM). In general, these modules accept a search key and a REQ signal, feed back an ACK signal with the results. On the other side, these modules connect to the Block RAM (BRAM) interface provided by Xilinx. Table entries are stored in BRAM.

The main state machine reads in the entire header to a FIFO. At the same time header_parser and ttl_checksum prepares the necessary information to the state machine. If the packet is a regular IP packet, the state machine issues a IP filter search request. If the address is found in the IP filter table, the packet will be kicked up to software. Otherwise,

the state machine does a routing table search, and a ARP search. In the last stage, the state machine modifies the MAC header, TTL, and checksum, and sends the packet to the destination port.

The extension code to support fast reroute and multipath routing is mainly in the routing table and lpm_lookup module. We will describe the two new features in the following subsections. Before that, the routing table structure and LPM lookup process will be presented.

## 2.2 Routing Table and LPM Lookup

### 2.2.1 Routing Table

Each entry of the routing table consists four parts: IP address as search key, the mask, next-hop IP, and port. The port information is stored as a one-hot-encoded number. This number has a one for every port the packet should go out on where bit 0 is MAC0, bit 1 is CPU0, bit 2 is MAC1, etc. The structure of the entry is depicted in Table 1.

| Search IP | Mask | Next-hop IP | Port |
|---|---|---|---|
| 192.168.100.0 | 255.255.255.0 | 192.168.101.2 | 0000 0001 |

**Table 1: Entry Structure of the Routing Table**

### 2.2.2 LPM Lookup

Due to the course requirement, we did not use the Xilinx Ternary Content Addressable Memory (TCAM) cores[5, 6]. Instead, we implement the routing table with BRAM on the NetFPGA card. Linear search is employed in LPM lookup as the size of the routing table is relatively small (32 entries required by the class). The entries with longer prefix are stored in front of those with shorter prefix. By doing this, entries with longest prefix will naturally come out first in a linear search.

## 2.3 Fast Reroute

In order to realize fast reroute feature, the router software needs to store a backup path for those "fast reroute protected" entry. In our router, the backup path information is in the form of duplicate entries only with *different port information*. In the normal case, the lpm_lookup module will return the *first* matched entry to the main state machine, making the port in this entry having the highest priority. When the port in the primary entry fails, the second entry with backup port information will be used and the flow will be rerouted. Table 2 is an example.

| | Search IP | Mask | Next-hop IP | Port |
|---|---|---|---|---|
| 1 | 192.168.100.0 | 255.255.255.0 | 192.168.101.2 | 0000 0001 |
| 2 | 192.168.100.0 | 255.255.255.0 | 192.168.101.2 | 0000 0100 |

**Table 2: Fast Reroute entries. The primary port is MAC0. The backup port is MAC1**

The reroute procedure is very fast because it is purely based on hardware. We make use of in-band link status information from Broadcom PHY chips as feedback. Once a link is down, lpm_lookup module will notice this immediately. The next coming packet will not follow the entry with invalid output port. Further details on in-band status information of NetFPGA's PHY chip can be found in Broadcom's BCM5464SR data sheet[7].

Besides link status feedback, the router hardware needs no modification under a linear search scheme. However, the duplicate entries will take up extra space in the routing table. At the same time, it is not applicable to TCAM based lookup mechanism, in which entries are not stored in order. Our solution is to extend port information section in the entry from 8bit to 16bit. The first 8bit is the primary port while the following 8bit is the backup. The primary port will be used first unless the associated link is down.

## 2.4 Multipath Routing

In the NetFPGA reference router, a routing table entry with multiple 1's in port section indicates itself as a multicast entry. Packets match this entry are sent to those ports at the same time. Based on the fact that in the current OSPF routing protocol, a packet is never sent to more than one port, we decided to take advantage of this section to implement multipath routing.

The goal of multipath routing is to allow packets destined to the same end-host making use of more than one route. In our multipath routing implementation, each entry in the routing table may have more than one output port, with multiple 1's in port section. Packets matching this entry could go to any port indicated in the entry. Note that for multipath entry, each output port will have its correspondent next-hop IP (gateway). We created another gateway table to store the gateway address. In routing table, we store a 8bit pointer (index) for each output port that can be used. Currently we use a simple round-robin fashion to choose the actual output port. A register keeps track of which port was last used and instructs lpm_lookup module to find the next available port. A multipath entry and gateway table example is shown in Table 3.

| Search IP | Mask | Next-hop IP index | Port |
|---|---|---|---|
| 192.168.100.0 | 255.255.255.0 | 02 00 00 01 | 0100 0001 |

| Index | Next-hop IP |
|---|---|
| 1 | 192.168.101.2 |
| 2 | 192.168.102.3 |

**Table 3: Multipath entry. Packets use MAC0, MAC3 in turns.**

We do not specify the priority of ports in the same entry. Each port, if available, will be used with equal probability. However, priority can still be realized by ordered duplicate entries described in the last section. One may optimize bandwidth, delay, quality of service, etc. by choosing the output port cleverly.

It is worth to point out that, unlike fast reroute, the multipath routing implementation is independent of how entries are stored. The same code applies to TCAM based router.

## 2.5 Limitation

We understand that there are a number of limitation in the design.

First, for fast reroute feature, the only feedback information is the link status. However, when the neighbor router goes down or freezes, sometimes the link status may remain active. In this case, it will not trigger the fast rerouting mechanism, and the application is subject to interruption. By design, our implementation is a hardware based improvement to the current OSPF protocol. With the software, the

topology is still recalculated regularly to overcome the router failures not resulting an inactive link state.

Another limitation of the design is packet reordering. We split a single flow into multiple paths without packet reordering protection. Packets could arrive at the destination in different order as they are sent. As the hardware router providing an interface to handle multipath routing, the software (multipath routing protocol, transport layer protocol such as TCP, or applications) may develop some methods to ensure the quality of service.

# 3. IMPLEMENTATION

## 3.1 Hardware

Fast reroute and multipath routing features have already been implemented in the hardware with linear search based implementation. The corresponding Verilog code is less than 100 lines.

In general, the two advanced features consume little logics in FPGA. However, duplicate entries for fast reroute may need more BRAMs to store. Table 4 describes the device utilization of out project. It uses 31% of the available flip-flops on the Xilinx Virtex II Pro 50 FPGA, which is almost equal to the reference router. 50% of the BRAMs available are used. The main use of BRAMs occurs in the three tables.

| Resources | XC2VP50 Utilization | Utilization Percentage |
|---|---|---|
| Occupied Slices | 14,781 out of 23,616 | 62% |
| 4-input LUTS | 17,469 out of 47,232 | 36% |
| Flip Flops | 14,918 out of 47,232 | 31% |
| Block RAMs | 118 out of 232 | 50% |
| External IOBs | 356 out of 692 | 51% |

**Table 4: Device utilization for Fast Reroute and Multipath Routing enabled Router**

## 3.2 Software

Software is responsible for providing correct tables to the hardware. Fast reroute and multipath routing features require changes only to the routing table. In the basic router implementation the Routing Table is generated using Dijkstra's algorithm to find shortest path to all known destinations. This calculation is done whenever the topology changes, as perceived by the router in question. To support fast reroute and multipath routing, it is not sufficient to find shortest paths to all destinations, but also second shortest (and possibly third shortest and more) paths are also necessary. This is calculated by running Dijkstra's algorithm four times (because NetFPGA has four interfaces). For each run of the algorithm, all interfaces on the router are disabled, except for one (a different interface is enabled in each run). Resulting hop count distances (i.e. the distance vectors) for each of the algorithm runs are then compared to provide the routing table.

### 3.2.1 Fast Reroute

When fast reroute feature is enabled, two entries for each destination will be added to the routing table if the destination can be reached over at least two interfaces. The first entry corresponds to the shortest path route and is preferred, and the second entry is the backup path if the primary path is disabled. The router will be in the mode where it uses a backup path only for the short time that it will take OSPF to update all routers. After that, the backup path will become the primary path, and a new backup path will be calculated (if available). Because of this, adding a second backup path, while possible, is deemed unnecessary. Also, this mechanism allows fast reroute to be enabled for some chosen routes, and disabled for others, which can potentially save space in the routing table.

### 3.2.2 Multipath Routing

*Equal cost* multipath has been chosen for its simplicity in implementation and limited packet reordering. To implement this we search if each of the destinations can be reached over multiple interfaces (as calculated by different algorithm runs) in the same minimum hop count. If this is so, all such interfaces are added to the routing table entry, if not, only the shortest path interface is added to the routing table.

In order to measure performance and demonstrate how fast reroute and multipath routing work, a demo application is being developed. This application consists of a GUI and a backend. The backend communicates with all routers and collects statistical information, such as packet count for each interface of each router. It is also aware of the network topology, which it then feeds to the GUI for visual presentation, together with the statistical data. Results will be available soon, as the demo application is completed.

# 4. CONCLUSION

In this paper we described the design and implementation of the fast reroute and multipath routing extensions to the NetFPGA reference router. Implemented with very little modification to the hardware pipeline, these features enhance the robustness and efficiency of the network. In addition, the GUI frontend can be used to visualize and validate the performance of the system.

This work is based on a beta version of the NetFPGA gateware, which lacks TCAM cores and SCONE (Software Component Of NetFPGA). In the future, we will port the code to NetFPGA beta-plus version, in order to achieve higher performance and reliability.

# 5. REFERENCES


[1] NetFPGA Group, "NetFPGA reference router," http://netfpga.org/wordpress/netfpga-ipv4-reference-router/.

[2] Stanford University CS344 Class, "Pee-Wee OSPF Protocol Details," http://yuba.stanford.edu/cs344/pwospf/.

[3] P. Pan, G. Swallow, and A. Atlas, "Fast Reroute Extensions to RSVP-TE for LSP Tunnels," RFC 4090 (Proposed Standard), Internet Engineering Task Force, May 2005. [Online]. Available: http://www.ietf.org/rfc/rfc4090.txt

[4] D. Thaler and C. Hopps, "Multipath Issues in Unicast and Multicast Next-Hop Selection," RFC 2991 (Informational), Internet Engineering Task Force, Nov. 2000. [Online]. Available: http://www.ietf.org/rfc/rfc2991.txt

[5] Xilinx, Inc, "An overview of multiple CAM designs in

Virtex family devices," http://www.xilinx.com/support/documentation/application_notes/xapp201.pdf.

[6] ——, "Designing flexible, fast CAMs with Virtex family FPGAs," http://www.xilinx.com/support/documentation/application_notes/xapp203.pdf.

[7] Broadcom Corporation, "BCM5464SR Quad-Port 10/100/1000BASE-T Gb Transceiver with Copper/Fiber Media Interface," http://www.broadcom.com/products/Enterprise-Networking/Gigabit-Ethernet-Transceivers/BCM5464SR.

# Using the NetFPGA in the Open Network Laboratory

Charlie Wiseman, Jonathan Turner, John DeHart, Jyoti Parwatikar,
Ken Wong, David Zar

Department of Computer Science and Engineering
Washington University in St. Louis
{cgw1,jst,jdd,jp,kenw,dzar}@arl.wustl.edu

## ABSTRACT

The Open Network Laboratory is an Internet-accessible network testbed that provides access to a large set of heterogeneous networking resources for research and educational pursuits. Those resources now include the NetFPGA. ONL makes it easy for NetFPGA users to integrate multiple NetFPGAs into heterogeneous experimental networks, using a simple graphical user interface. The testbed software infrastructure automatically manages all of the details, including mapping the user's topology to actual hardware and time-sharing of resources via a standard reservation mechanism. The inclusion of NetFPGAs into the testbed allows users just getting started with NetFPGAs to conduct interesting research quickly without the need to set up and manage the NetFPGAs themselves. For more experienced users, the testbed provides an easy path to larger and more diverse experimental configurations.

## 1. INTRODUCTION

Networking and systems researchers have come to rely on a wide range of tools and platforms to conduct their experiments. This includes specific types of technology as well as simulation and testbed environments. One such technology that has recently seen wide adoption is the NetFPGA [11][9][15], which is a relatively inexpensive reprogrammable hardware platform. NetFPGAs are now also available as part of the Open Network Laboratory (ONL) testbed [6].

ONL is an Internet-accessible network testbed which features resources based on a variety of networking technologies. Users configure arbitrary network topologies with a simple GUI tool and then run interactive experiments using that topology. The user is granted sole control of the physical components that make up their topology for the duration of their experiment. Sharing is accomplished via a standard reservation mechanism. ONL is open to all researchers and educators (at no cost, of course).

There are advantages to using NetFPGAs in ONL for the entire range of NetFPGA users. Clearly, it opens the way for NetFPGA use by those who do not have the means to acquire, set up, and manage their own NetFPGA installations. This barrier is already low for many users due to the low cost of the platform and substantial available documentation. However, the ongoing overheads to manage many NetFPGAs can be high in certain contexts. One example is a course where many students have to coordinate to share a small number of NetFPGAs. ONL enables this sharing naturally with minimal additional overhead to both the educator and the students. More generally, ONL removes the management overhead when there is substantial sharing of NetFPGA resources.

For existing NetFPGA users, ONL provides an easy means to experiment with NetFPGAs in a variety of elaborate configurations. ONL currently has nearly 20 routers and over 100 PCs in addition to 6 NetFPGAs. This allows researchers to build experimental topologies that reflect many real world scenarios. Moreover, it only takes a few minutes with the GUI to configure an entirely different topology. It would require a substantial investment of money and time to reproduce this capability in a local environment. The ability to quickly run experiments over diverse network configurations is particularly useful in conjunction with the NetFPGA because of its highly flexible nature.

The rest of the paper is organized as follows. Section 2 provides background information about ONL including brief descriptions of the other resources currently available in the testbed. Section 3 describes how to use NetFPGAs as part of ONL along with three examples. Related work is covered in Section 4, and some future work is discussed in Section 5. Finally, Section 6 contains closing thoughts.

## 2. ONL

The Open Network Laboratory testbed provides a diverse set of user-configurable resources for network experimentation. This includes both research and educational contexts [23]. The Remote Laboratory Interface (RLI) is a simple Java GUI run on the user's computer to configure experimental topologies and monitor network traffic within the topology in real time. An example experiment consisting of a small dumbbell topology is shown in Figure 1. The top of the figure is the topology configuration window and the bottom is a real time chart with three data sets showing bandwidth at various points in the topology.

Users first configure their network topology with the RLI by selecting various components from the menus and linking them together. Once the topology is ready, a reservation is made to ensure that the required resources will be available for the duration of the user's experiment. Specifically, the reservation request consists of a length and a range of times that the user chooses during which they could run their experiment. The ONL software will either grant the reservation for a specific interval during the given range of times, or reject the reservation if there are not enough resources available during that time. The resource availability for the near future can be found on the ONL website [16] to aid in finding open time slots. Also note that each user may have many outstanding reservations. Users are guaran-

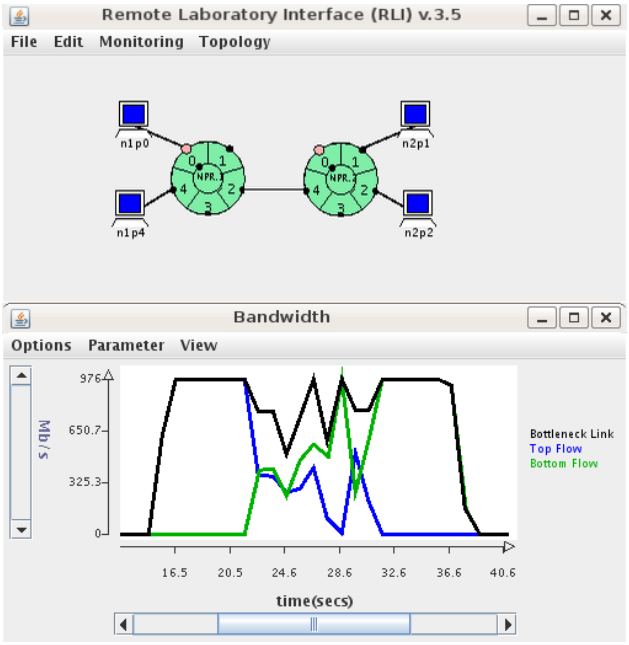



**Figure 1: Example ONL configuration with real time charts.**

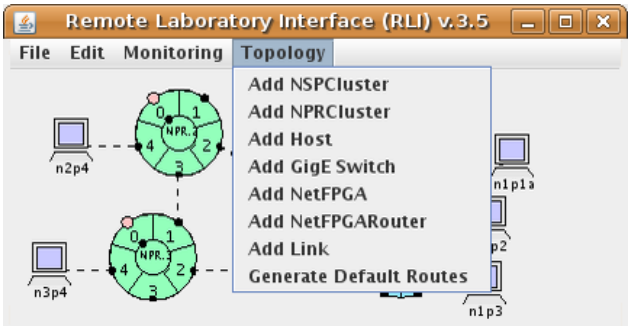



**Figure 2: RLI Topology Menu.**

teed to be able to run their experiment without interference from any other experiments for their reservation duration. When the time for the reservation arrives, the RLI is used to start the experiment as well as to monitor and configure the resources in the experiment topology.

Before moving on the NetFPGA, a brief description is given of the other ONL resources.

The Network Services Platform (NSP) [4] is a custom-built IP router designed in the style of larger, scalable router platforms. Each of the eight ports is an independent entity and consists of a 1 Gb/s interface, a Field Programmable Port Extender (FPX) [12], and a Smart Port Card (SPC) [7]. A 2 Gb/s cell switch connects the ports together. The FPX handles all the common packet processing tasks such as classification, queueing, and forwarding. The SPC contains a general purpose processor which is used to run *plugins*. Plugins are user developed code modules which can be dynamically loaded on to the router ports to support new or specialized packet processing. There are currently 4 NSPs in ONL.

Next is the Network Processor-based Router (NPR) [22], which is an IP router built on Intel IXP 2800s [1]. The NPR has five 1 Gb/s ports and a Ternary Content Addressable Memory (TCAM) which is used to store routes and packet filters. User written plugins are supported as in the NSP. In the case of the NPR, five of the sixteen MicroEngine cores on the IXP are dedicated to running plugin code and the user has the ability to dynamically map different plugins to different MicroEngines. Packet filters are used to direct specific packet flows to the plugins, and the plugin can forward the packet to outbound queues, to another plugin, back to the classification engine, or the packet can be dropped. There are currently 14 NPRs in ONL.

Standard Linux machines are used as traffic sources and sinks. Every host has two network interfaces: a 1 Gb/s data interface which is used in the experimental topology, and a separate management interface. Once a user's experiment is active, they are granted SSH access through the management interface to each host in their topology. The hosts are all installed with a range of utilities and traffic generators, and users are welcome to install their own software in their user directory (which is shared across all the hosts). The only limitation in the environment is that users are not given a root shell, so any specific actions requiring root privileges are granted through normal sudo mechanisms. There are currently over 100 hosts in ONL.

All of the ONL resources are indirectly connected through a set of configuration switches. Virtual Local Area Networks (VLANs) are used extensively in the switches to enforce isolation among different experiments. All of the configuration of these switches takes place automatically and invisibly from a user's perspective. The reservation system is responsible for ensuring that the configuration switches have sufficient capacity to guarantee that no experiment could ever interfere with any other experiment. VLANs also provide a way for ONL to support standard Ethernet switches in experimental topologies.

## 3. USING NETFPGAS IN ONL

The are currently 6 NetFPGAs available in ONL, each installed in a separate Linux host. The hosts have the base NetFPGA software distribution installed, as well as many of the other NetFPGA project distributions. Similar to the standard ONL hosts, users are granted SSH access to the NetFPGA host for each NetFPGA in their experimental topology so that they can run the utilities to load and configure the hardware. There are a few of these utilities that require root privileges to run, so sudo is employed as on the other ONL hosts. After each experiment ends, the hosts are rebooted automatically and the NetFPGAs are prepared for user programming.

Most of the other ONL resources directly support monitoring and configuration in the RLI via software daemons that are tightly coupled with the individual resources. No such daemon exists for the NetFPGA because the interfaces for monitoring and configuration change depending on what is currently running on the hardware. As such, all configuration is done via the utilities and scripts that are already provided with the existing projects. Monitoring can be done in the RLI indirectly by means of a simple mechanism supported by the ONL software. Specifically, the RLI can monitor and plot data that is put into a file on the host (the file

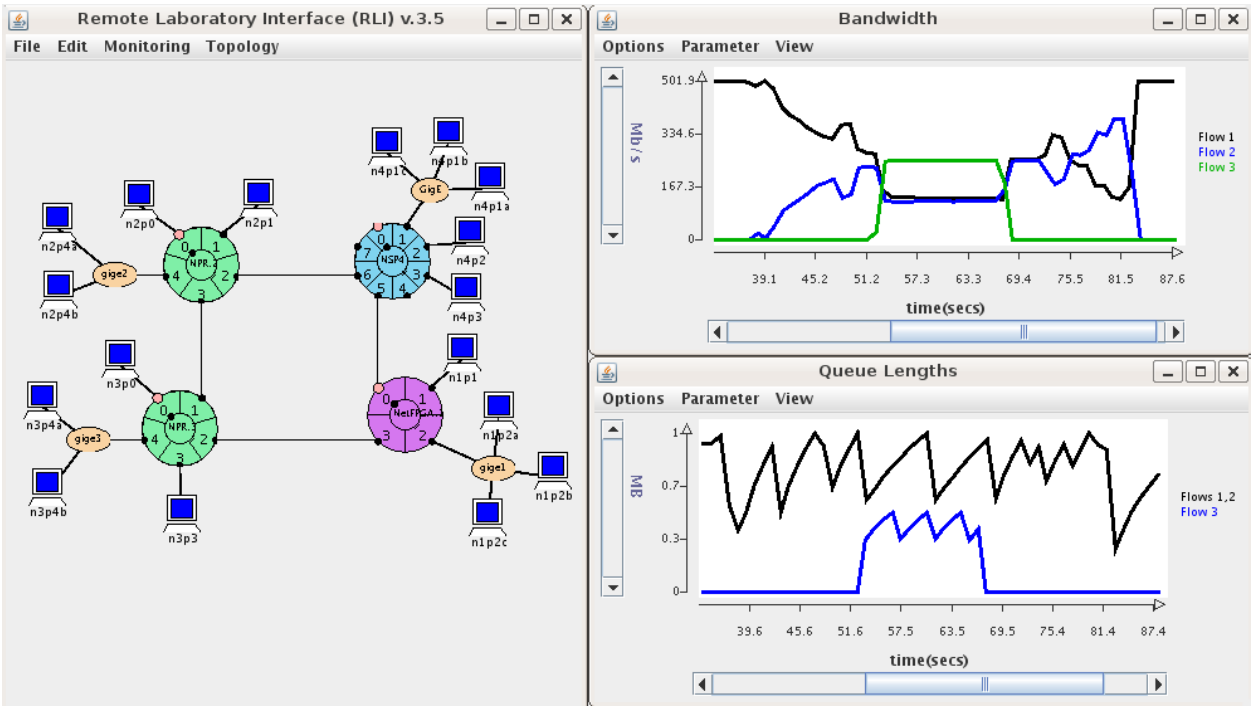



**Figure 3: Configuration using many different types of resources with bandwidth and queue length charts.**

location is specified in the RLI). Users write scripts to populate these files with whatever data they choose. For the NetFPGA, that will typically mean reading registers that contain packet or byte counts in order to display packet rates, bandwidths, or queue lengths in the RLI.

NetFPGAs are added to a configuration in the RLI similarly to other types of resources, with one exception. The core function of other resources is fixed in nature, i.e., a resource is either a router or something else such as a switch, or a traffic source or sink. This is important when the ONL software configures each resource, as hosts need to know their default gateway, and routers need to know if they are connected to other routers in order to build network routes correctly. NetFPGAs, however, can fill either role and so each NetFPGA is added either as a router node or as a non-router node. A screenshot of the actual RLI menu is shown in Figure 2.

Three examples are given next to illustrate how NetFPGAs are typically used in ONL.

## 3.1 IP Router

The first example is something that might be seen in an introductory networking course. The topology configuration is shown on the left of Figure 3. The 4 port device at the bottom right of the central "square" is a NetFPGA acting as an IP router. The 8 port device above it is an NSP and the 5 port devices to the left are NPRs. The small ovals connected to each of the routers are Ethernet switches. The others symbols are hosts.

The user configures the NetFPGA by logging in to the PC that hosts the NetFPGA assigned to this experiment, which can be determined by clicking on the NetFPGA in the RLI or by referencing shell variables that are automatically added to the user's ONL shell. For this experiment, the reference IP router bit file is loaded from the command line via the nf2_download utility. Then the software component of the reference router, SCONE, is started. In order to configure the router appropriately, SCONE must be given the Ethernet and IP information for each of the NetFPGA ports as well as the static routes for the router. Normally this information would be sent directly from the RLI to a software daemon that understands how to manage each type of resource. As mentioned above, the NetFPGA interface changes depending on the situation, so the user must build this information manually. We are currently working on ways to ease this burden.

Once the router hardware and software are running, the NetFPGA is acting as a standard IP router and is treated as such by the user. In this example, the user is studying TCP dynamics over a shared bottleneck link. Three TCP flows are sent from hosts in the bottom left of the topology to hosts in the bottom right across the link from the NPR to the NetFPGA. Two of the flows share an outgoing queue at the bottleneck link and the third flow has its own queue. The link capacity is set at 500 Mb/s, the shared queue is 1 MB in size, and the non-shared queue is 500 KB in size. The first flow is a long-lived flow, the second is a medium length flow, and the third is a short flow.

The top right of Figure 3 shows the throughput seen by each flow and the bottom right shows the queue lengths at the bottleneck. The first flow consumes the entire bottleneck capacity in the absence of other traffic. Once the second flow begins, the two attempt to converge to a fair share of the link, but the third flow begins before they reach it. The NPR is using a typical Weighted Deficient Round Robin scheduler. The user has configured the two queues to receive an equal share of the capacity by setting their scheduling quanta to be the same. The result is that the third flow

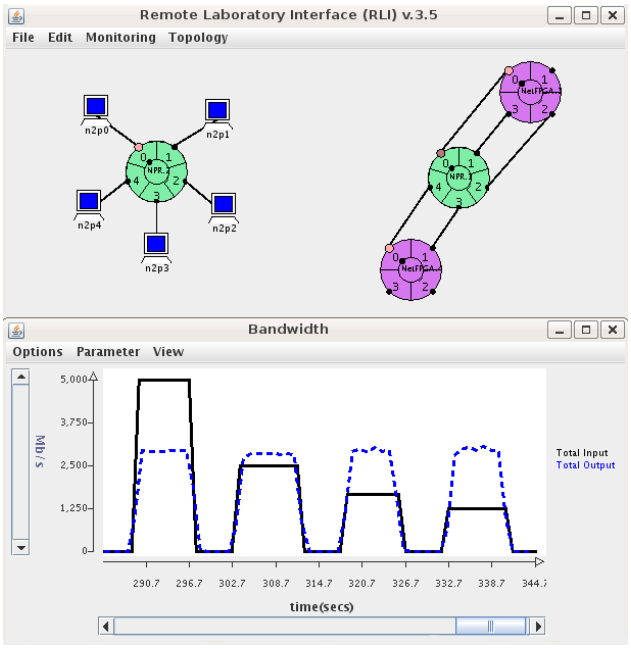



**Figure 4: Using NetFPGAs as packet generators to stress test NPR multicast.**

gets 250 Mb/s of the 500 Mb/s link because it is the only flow in its queue, and the first two share the remaining 250 Mb/s because they are sharing a queue.

## 3.2 Traffic Generation

The second example utilizes NetFPGAs as packet generators [5]. In general, it is quite difficult to produce line rate traffic for small packets from general purpose PCs. The ONL solution in the past was to either use many hosts to aggregate traffic on a single link or to use special purpose plugins in the routers that made many copies of each input packet and thus multiply the input traffic up to the line rate. Each option required the use of many resources and produced results that were difficult to control precisely. The NetFPGA packet generator is a much cleaner solution as it can produce line rate traffic on all four ports of any size packet. The packets are specified in the standard PCAP file format.

For this example, NetFPGA packet generators are used to stress test the NPR capabilities. The configuration used is shown in the top of Figure 4. There are actually two separate networks in this configuration. The one on the left is used to generate the packet traces needed for the packet generator, and the one on the right is used to perform the actual stress test. To generate the traces, the hosts send packets via normal socket programs and the user runs tcpdump on the nodes to record the packet streams.

The bottom of Figure 4 shows the results from one particular stress test. The NPR natively supports IP multicast (details in [22]), and this test is meant determine if the packet *fanout* has any effect on peak output rate for minimum size packets. The fanout is simply the replication factor for each incoming packet. Four sets of traffic are sent to the NPR in succession. The first set has a fanout of one, meaning that each input packet is sent out one other port.

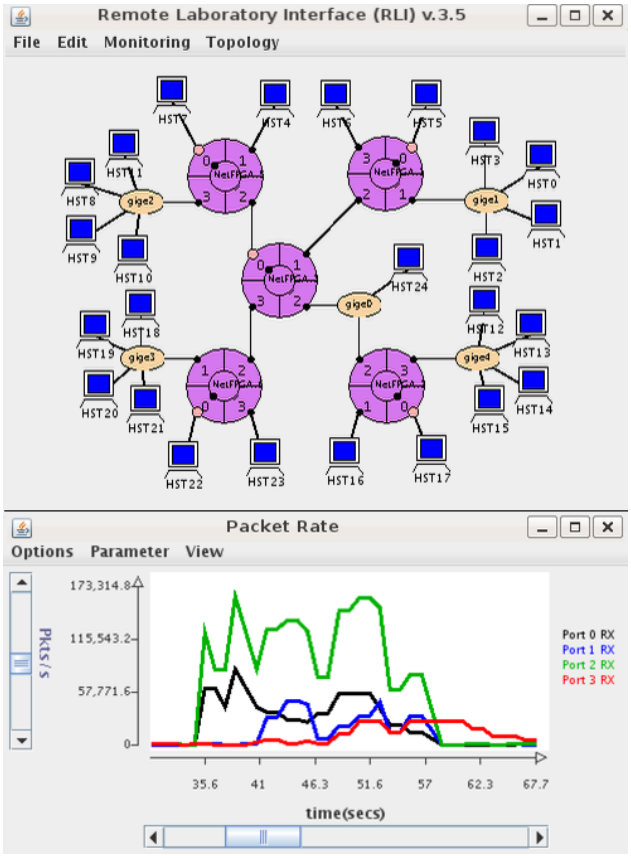



**Figure 5: An OpenFlow network with real time charts generated from counters on one OpenFlow switch.**

The second set has a fanout of two, the third has a fanout of three, and the last has a fanout of four. In each case the total input rate is set so that the total output rate would be 5 Gb/s if the router dropped no packets, and the input and output rates are each shared equally by all 5 ports. The chart shows the aggregate input rate (solid line) and output rate (dashed line) at the router. In each test, the output rate is around 3 Gb/s, showing that the fanout has little effect on the eventual output rate. Other tests confirm that the peak forwarding rate for minimum size packets is around 3 Gb/s for unicast traffic as well.

## 3.3 OpenFlow

The last example is of an OpenFlow [13] network. There is an existing NetFPGA OpenFlow switch implementation which supports the Type 0 specification [14]. A description of OpenFlow is out of scope for this paper, but more details can be found on the OpenFlow website [17].

The top of Figure 5 shows an OpenFlow network topology with 5 NetFPGAs acting as OpenFlow switches, 5 Ethernet switches, and 25 hosts. This example demonstrates OpenFlow switches inter-operating with non-OpenFlow (i.e, standard Ethernet) switches. Every OpenFlow network needs a controller that is responsible for interacting with and configuring the OpenFlow switches. The OpenFlow software distribution comes with a basic controller that allows each OpenFlow switch to operate as a normal learning Ether-

net switch. Nox [10] is another OpenFlow controller which is substantially more complex and configurable. Both controllers are available on all ONL hosts so that any ONL host can act as an OpenFlow controller.

The bottom of Figure 5 shows a chart monitoring packet rates through the OpenFlow switch in the middle of the topology when there are many flows traversing the network. The chart is generated as described above by parsing the output of an OpenFlow utility which reads per-flow counters in the switch. The results are placed in a file which allows them to be displayed in the RLI.

# 4. RELATED WORK

There are a few network testbeds that are generally similar to ONL. Of these, ONL is closest in nature to Emulab [21]. The Emulab testbed has been widely used for research and can often be found as part of the experimental evaluation in papers at top networking conferences. Emulab provides access to a large number of hosts which can be used to emulate many different types of networks. As in ONL, every host has at least two network interfaces. One is used for control and configuration, and the others are used as part of the experimental topology. Also as in ONL, Emulab uses a small number of switches and routers to indirectly connect all the hosts in the testbed. Emulab has many useful features which make it an attractive choice for conducting research. For example, users can configure individual link properties such as delay and drop rate, and the Emulab software automatically adds an additional host which is used as a traffic shaper for that link. According to the Emulab website [8], they have also added six NetFPGAs as testbed resources.

The Emulab software has also been made available so that other groups can use it to run their own testbeds. Many of these Emulab-powered testbeds are currently operating, although most of them are not open to the public. One exception is the DETER testbed [3] that is focused on security research. The control infrastructure is identical to Emulab, but additional software components were added to ensure that users researching security holes and other dangerous exploits remain both contained in the testbed and isolated from other experiments. The Wisconsin Advanced Internet Laboratory (WAIL) [20] is another such testbed that utilizes the Emulab software base. WAIL is unique among the Emulab testbeds in that they have extended the software to support commercial routers as fundamental resources available to researchers.

Another widely used testbed is PlanetLab [18]. At its core, PlanetLab simply consists of a large number of hosts with Internet connections scattered around the globe. At the time of this writing there are over 1000 PlanetLab nodes at over 480 sites. Each PlanetLab node can support multiple concurrent experiments by instantiating one virtual machine on the node for each active experiment. The PlanetLab control software takes user requests to run an experiment on a set of nodes and contacts the individual nodes to add the virtual machines for the user. Researchers use PlanetLab to debug, refine, and deploy their new services and applications in the context of the actual Internet. Unfortunately, PlanetLab's success has resulted in large numbers of active experiments, particularly before major conference deadlines. This often leads to poor performance for any individual experiment because each active experiment is competing for both processor cycles and the limited network bandwidth available on the host.

There have been a few efforts that aim to enhance PlanetLab through the design of new nodes that enable better and more consistent performance while still operating in the PlanetLab context. VINI [2] is at the front of these efforts. VINI nodes are currently deployed at sites in the Internet2, National LambdaRail, and CESNET backbones [19], and have dedicated bandwidth between them. The VINI infrastructure gives researches the ability to deploy protocols and services in a realistic environment in terms of network conditions, routing, and traffic.

With the exception of WAIL, all of these testbeds provide direct access only to hosts. WAIL does provide access to commercial routers, although the website indicates that it is read-only access, i.e., users do not have the ability to configure the routers themselves. In contrast, ONL does provide user-driven control of many different types of networking technology, and we hope to continue increasing the resource diversity in the future.

# 5. FUTURE WORK

As discussed earlier, most ONL resources are configured primarily through the RLI. Configuration updates are sent to the daemon that manages the resource and knows how to enact the updates. This is not possible with the NetFPGA because the types of configuration change depending on how it is being used. Moreover, two different implementations of the same functionality may export different configuration interfaces. For example, there are already two IP routers available that have different interfaces for adding routes. The result is that all configuration of NetFPGAs is done manually. Simple scripts help to reduce the time spent doing manual configuration, but they can not remove the need for it completely.

An alternative that is being considered is to have different NetFPGA options available in the RLI that correspond to specific NetFPGA implementations. This would allow a user to add, for example, a NetFPGA IP router using the NetFPGA reference router specifically. The RLI would then load the correct bit file, automatically compute and add routes, set up the NetFPGA port addresses, and be able to monitor values in the router directly. In fact, being able to monitor packet counters directly via the RLI would make such a feature valuable even to a project like the reference Ethernet switch which requires no user configuration. The largest obstacle is that there has to be an ONL software daemon for each NetFPGA project which can receive these configuration and monitoring requests. While some of the most common projects could be supported relatively easily by ONL staff, a more scalable approach would allow any user to dynamically provide the portions of code required to actually carry out the requests. In this way, users could leverage the ONL infrastructure even for their own private NetFPGA projects.

# 6. CONCLUSION

NetFPGAs are a particularly versatile networking platform which can be used as routers, switches, traffic generators, and anything else that NetFPGA developers can build. Deploying such a platform in a network testbed like the Open Network Laboratory benefits both existing testbed

users and NetFPGA users. Six NetFPGAs have been added to the ONL resource set and more could be added if there is sufficient interest and use. This allows those interested in using NetFPGAs to get started immediately without the need to set up and manage the NetFPGAs themselves. Educators in particular are likely to benefit from this. It also provides a means for existing NetFPGA users to test and experiment with their own projects in much broader and more diverse configurations than are likely available in a local lab. In general, we believe that the inclusion of NetFPGAs in ONL will help both projects to reach broader audiences. The ONL source code is available on request.

## 7. REFERENCES


[1] M. Adiletta, M. Rosenbluth, D. Bernstein, G. Wolrich, and H. Wilkinson. The next generation of intel ixp network processors. *Intel Technology Journal*, 6, 2002.

[2] A. Bavier, N. Feamster, M. Huang, L. Peterson, and J. Rexford. In vini veritas: Realistic and controlled network experimentation. In *SIGCOMM '06: Proceedings of the 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 3–14, New York, NY, USA, 2006. ACM.

[3] T. Benzel, R. Braden, D. Kim, C. Neuman, A. Joseph, K. Sklower, R. Ostrenga, and S. Schwab. Design, deployment, and use of the deter testbed. In *DETER: Proceedings of the DETER Community Workshop on Cyber Security Experimentation and Test on DETER Community Workshop on Cyber Security Experimentation and Test 2007*, pages 1–1, Berkeley, CA, USA, 2007. USENIX Association.

[4] S. Choi, J. Dehart, A. Kantawala, R. Keller, F. Kuhns, J. Lockwood, P. Pappu, J. Parwatikar, W. D. Richard, E. Spitznagel, D. Taylor, J. Turner, and K. Wong. Design of a high performance dynamically extensible router. In *Proceedings of the DARPA Active Networks Conference and Exposition*, May 2002.

[5] G. A. Covington, G. Gibb, J. Lockwood, and N. McKeown. A packet generator on the netfpga platform. In *The 17th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 5–7 April 2009.

[6] J. DeHart, F. Kuhns, J. Parwatikar, J. Turner, C. Wiseman, and K. Wong. The open network laboratory. In *SIGCSE '06: Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education*, pages 107–111, New York, NY, USA, 2006. ACM.

[7] J. D. DeHart, W. D. Richard, E. W. Spitznagel, and D. E. Taylor. The smart port card: An embedded unix processor architecture for network management and active networking. Technical report, Washington University, August 2001.

[8] Emulab. Emulab website. http://www.emulab.net.

[9] G. Gibb, J. W. Lockwood, J. Naous, P. Hartke, and N. McKeown. Netfpga: Open platform for teaching how to build gigabit-rate network switches and routers. 51(3):364–369, Aug. 2008.

[10] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. Nox: towards an operating system for networks. *SIGCOMM Comput. Commun. Rev.*, 38(3):105–110, 2008.

[11] 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 *Proc. IEEE International Conference on Microelectronic Systems Education MSE '07*, pages 160–161, 3–4 June 2007.

[12] J. W. Lockwood, N. Naufel, J. S. Turner, and D. E. Taylor. Reprogrammable network packet processing on the field programmable port extender (fpx). In *FPGA '01: Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays*, pages 87–93, New York, NY, USA, 2001. ACM.

[13] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. volume 38, pages 69–74, New York, NY, USA, 2008. ACM.

[14] J. Naous, D. Erickson, G. A. Covington, G. Appenzeller, and N. McKeown. Implementing an openflow switch on the netfpga platform. In *ANCS '08: Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, pages 1–9, New York, NY, USA, 2008. ACM.

[15] J. Naous, G. Gibb, S. Bolouki, and N. McKeown. Netfpga: reusable router architecture for experimental research. In *PRESTO '08: Proceedings of the ACM workshop on Programmable routers for extensible services of tomorrow*, pages 1–7, New York, NY, USA, 2008. ACM.

[16] ONL. Open network laboratory website. http://onl.wustl.edu.

[17] OpenFlow. Openflow website. http://openflowswitch.org.

[18] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A blueprint for introducing disruptive technology into the internet. In *Proceedings of HotNets–I*, Princeton, New Jersey, October 2002.

[19] VINI. Vini website. http://www.vini-veritas.net.

[20] WAIL. Wisconsin advanced internet laboratory website. http://www.schooner.wail.wisc.edu/.

[21] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *OSDI '02: Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 255–270, New York, NY, USA, 2002. ACM.

[22] C. Wiseman, J. Turner, M. Becchi, P. Crowley, J. DeHart, M. Haitjema, S. James, F. Kuhns, J. Lu, J. Parwatikar, R. Patney, M. Wilson, K. Wong, and D. Zar. A remotely accessible network processor-based router for network experimentation. In *ANCS '08: Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, pages 20–29, New York, NY, USA, 2008. ACM.

[23] K. Wong, T. Wolf, S. Gorinsky, and J. Turner. Teaching experiences with a virtual network laboratory. In *SIGCSE '07: Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education*, pages 481–485, New York, NY, USA, 2007.

# Implementation of a Future Internet Testbed on KOREN based on NetFPGA/OpenFlow Switches

Man Kyu Park, Jae Yong Lee, Byung Chul Kim, Dae Young Kim
Dept. of Infocomm Eng., Chungnam National University, Daejeon, 305-764, Korea
mkpark@ngn.cnu.ac.kr, {jyl, byckim, dykim}@cnu.ac.kr

*Abstract*—**A large-scale testbed implementation for Future Internet is very important to evaluate new protocols and functions designed by clean-slate approach. In Korea, a new project for deploying Future Internet testbed called FIRST just has been started. This project consists of two sub-projects. One is the 'FIRST@ATCA' for implementing large platform based on ATCA chassis and the other is the 'FIRST@PC' for implementing the PC-based platform utilizing NetFPGA/OpenFlow switches. Of them, the scope of work on the FIRST@PC project is to develop PC-based platform using NetFPGA/OpenFlow switch and to design service operation and control framework on the dynamic virtualized slices. In this paper, we first introduce some activities related to the Korea Future Internet testbed and interim results of ongoing FIRST@PC projects, especially about NetFPGA module extension and performance test on the KOREN network. We are now implementing a network emulator function on the NetFPGA platform which can control bandwidth, delay and loss for network experiments, and we plan to evaluate the performance of various high-speed TCP protocols using this emulator.**

***Keywords- Future Internet, testbed, NetFPGA, OpenFlow, GENI, Network Emulator***

## I. INTRODUCTION

Current Internet has many serious limitations in the aspects of scalability, security, QoS, virtualization, and so on. So, recently some major projects to design a new Internet architecture by using clean-slate approach have been started and large-scale testbeds for the evaluation of new protocols have just begun to be deployed. Future Internet testbed requires some advanced concept, such as programmability, virtualization, end-to-end slice, federation, and network resource management. Some of well known projects are GENI (Global Environment for Network Innovation) [1], FIRE (Future Internet Research and Experimentation) [2], and NwGN (New Generation Network) [3].

In Korea, we also started a new project to design and deploy Future Internet testbed, called 'FIRST' (Future Internet Research for Sustainable Testbed)[4], from Mar. 2009. This project consists of two sub-projects. One is the 'FIRST@ATCA' for implementing a large platform based on ATCA chassis and the other is the 'FIRST@PC' for implementing a PC-based platform utilizing NetFPGA and OpenFlow switches. The latter one is to implement a virtualized hardware-accelerated PC-node by extending the functions of NetFPGA card and build a Future Internet testbed on the KOREN and KREONET for evaluating newly designed protocols and some interesting applications.

In this paper, we first introduce some activities related to the Korea Future Internet testbed and interim results of ongoing FIRST@PC projects, especially about NetFPGA module extension and performance test on the KOREN/KREONET network. We have first implemented a 'Capsulator' user-space program using raw-socket in Linux to interconnect OpenFlow enabled switch sites on the KOREN and KREONET and tested throughput performance for varying packet sizes. Also, we are implementing a network emulator function on the NetFPGA which can control bandwidth, delay and loss for network experiments. and we plan to evaluate various high-speed TCP throughputs using this emulator. As a next step, we'll implement packet schedulers and buffer controllers such as WRED by extending NetFPGA basic module for Future Internet testbed functions.

## II. FIRST PROJECT

The new testbed project in Korea is named FIRST, in which the ETRI and 5 universities have participated since Mar. 2009. The scope of work on the ETRI's project, 'FIRST@ATCA', is to implement a virtualized programmable Future Internet platform. It consists of software for control/virtualization and ATCA-based COTS (Commercial Off The Shelf) hardware platform. Another approach to deploy Future Internet testbed is progressed by 5 universities (GIST, KAIST, POTECH, Kyung-Hee Univ., Chungnam Nat'l Univ.). The scope of this 'FIRST@PC' project is to develop a PC-based platform using NetFPGA/OpenFlow switches and to design a service operation and control framework on dynamic virtualized slices. These two types of platforms will be utilized as infra-equipments in the core and access networks of Korea Future Internet testbed.

The PC-based platform is to be built using a VINI-style [5] or hardware-accelerated NetFPGA/Openflow switches. The platform block diagram for supporting virtualization and programmable networking capability is shown in Figure 1. We call this platform as Programmable Computing/Networking Node (PCN).

Figure 2 shows the overall FIRST testbed model. This framework should support dynamic interconnection of all the PCNs according to user's request. The basic agent-based software stack should be implemented for configuring slices and controlling distributed services by using available

resources (processing power, memory, network bandwidth, etc.) efficiently. On this testbed, multimedia-oriented services will be runned for showing efficiency of control operation.

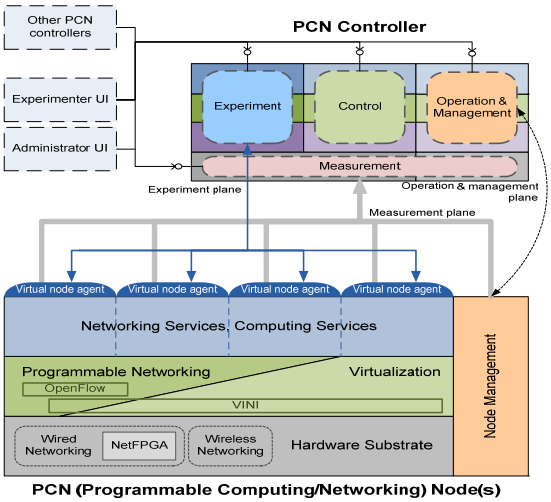



Figure 1. PC-based PCN platform architecture [6]

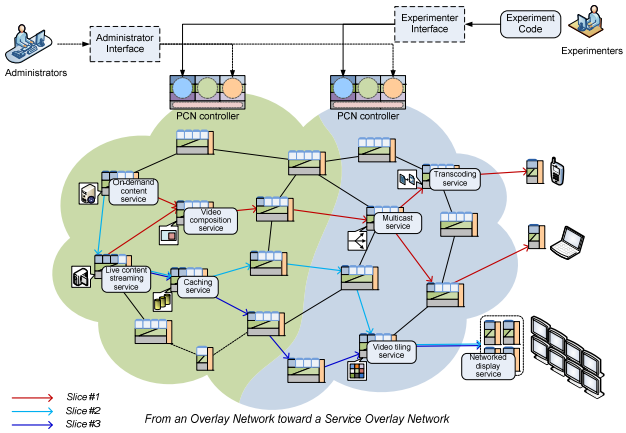



Figure 2. Service operation and control framework [6]

## III. KOREN TESTBED USING NETFPGA/OPENFLOW SWITCH

In this section, we explain the KOREN and an implementation of NetFPGA/Openflow switch testbed on the KOREN.

### A. *Introduction to the KOREN*

The KOREN (Korea Advanced Research Network) [7] is a non-profit testbed network infrastructure for field testing of new protocols and international joint research cooperation. The backbone network connects major cities in Korea at the speed of 10 ~ 20Gbps as shown in Fig. 3. Some previous usage examples on KOREN are as follows;

- Korea-China IPTV testbed configuration and overlay multicast research
- High-quality tele-lecture and video conference based on IPv6
- Research on mesh-based access network for Korea-US Future Internet
- Telemedicine demonstration using high definition video

This year, NIA (National Information Society Agency) calls for a new project about Future Internet testbed deployment using NetFPGA/OpenFlow switches on the KOREN. This project will be launched soon and about 10 universities will participate to evaluate the performance of NetFPGA-installed OpenFlow switches under various topology and environments on the KOREN. Using this testbed, we plan to test network functionality and various application services. It will be a good chance to encourage lots of Korean researchers to contribute to the NetFPGA community as a developer.

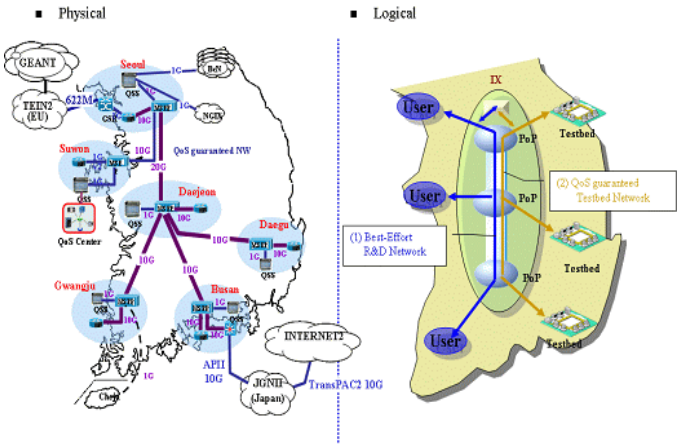



Figure 3. The KOREN topology

### B. *Implementing Openflow switch testbed on the KOREN*

We have been implementing the NetFPGA-based Openflow switch testbed on the KOREN in order to be used for Future Internet research activity. Since the Openflow switches forward their frames according to the flow tables inserted by an Openflow controller such as Nox controller [8], we can consider that the routing function is performed at the controller and the Openflow Switches are operated at Layer 2 frame level. Thus, we need to tunnel Openflow frames intact from one openflow site to another Openflow site through the KOREN, because there are no Openflow aware nodes now in the KOREN. So, we have implemented a Capsulator program first introduced in [9] which performs tunneling Openflow frames through the KOREN as shown in Figure 4. Using this Capsulator, L2 bridge is emulated on the KOREN. At this time there is one Nox controller for Openflow testbed on the KOREN located in CNU (Chungnam National University).

Basically, the Capsulator performs “MAC frame in IP” tunneling in which Openflow frames are encapsulated into IP packet for tunneling between openflow sites through the KOREN. We have implemented a Capsulator as an user-space application in Linux by using ‘libpcap’ socket [10] and UDP socket (or IP raw socket) as shown in Figure 5.

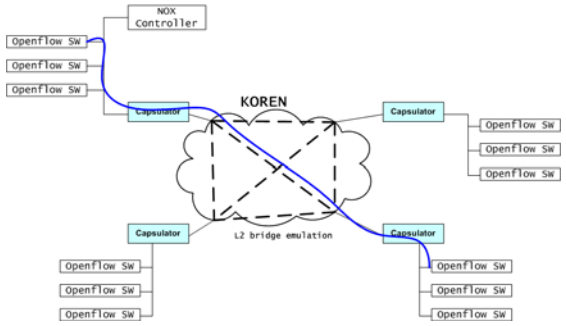



Figure 4. Service operation and control framework

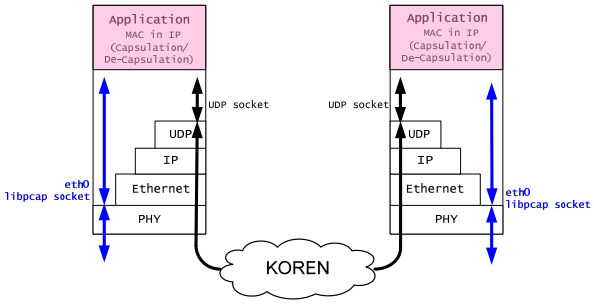



Figure 5. Capsulator model implemented as an user application

When an Openflow frame is captured, a flow ID is attached to the head of the frame after flow classification and then it is encapsulated into an IP packet as shown in Figure 5. Then, virtual bridge table is consulted to find the destination 'bridge ports' for forwarding. The flow ID can be utilized for various virtual network functions such as traffic isolation scheduler.

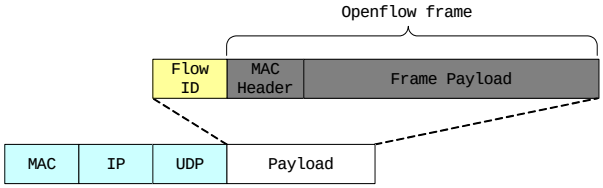



Figure 6. MAC frame encapsulation in a capsulator

We have measured the throughput performance of the Openflow testbed between two sites, CNU and GIST, by using 'iperf' program [11] with TCP Reno and UDP for various packet sizes. The host in CNU is attached to the KREONET and the host in GIST is attached to the KOREN. Both hosts have line-rate 1 Gbps.

Table 1 shows the measurement results for the two transport protocols. The throughput of TCP Reno is about 140 Mbps and that of UDP is about 180 Mbps. One can see that these throughput values are rather limited compared to the line rate 1 Gbps. The main reason for this is that the implemented Capsulator is a user space program. In order to enhance the performance, it is necessary to implement the Capsulator in the kernel level, or to use the NetFPGA platform for Capsulator implementation to utilize its hardware acceleration. We plan to implement a Capsulator on the NetFPGA platform.

TABLE I. THROUGHPUT PERFORMANCE OF OPENFLOW TESTBED BETWEEN THE KOREN AND KREONET.

| **Packet sizes (bytes)** | ***64*** | ***128*** | ***256*** | ***512*** | ***1024*** | ***1400*** |
|---|---|---|---|---|---|---|
| TCP Reno (Mbps) | ***3.21*** | ***3.34*** | ***10.8*** | ***27.1*** | ***57.3*** | ***139*** |
| UDP (Mbps) | ***3.4*** | ***6.85*** | ***13.7*** | ***139*** | ***179*** | ***179*** |

## IV. IMPLEMENTING A NETWORK EMULATOR ON THE NETFPGA PLATFORM

When network researchers want to evaluate the performance of newly designed protocols or network mechanisms, they need to deploy them in real networks and do experiments they want to perform. However, it is usually very difficult to adjust network environments for experiments as they want, because it is hard to manage the real network characteristics appropriate to their experiments. In this case, network emulators can do appropriate role by emulating a real network situation. The network emulators can control the bottleneck bandwidth, passing delay, and packet loss probability in order to emulate real network conditions.

The Dummynet [12] and the NISTnet [13] are widely used network emulators that can be installed and operated in ordinary PCs. However, the main drawback of these emulators is their performance limitations caused by software processing of control actions in rather low-performance PC platforms. They reveal marginal performance with almost 100% CPU utilization for two NICs with line rate 1 Gbps. Furthermore, the bottleneck of PCI bus performance is another main reason for performance degradation of network emulators.

The NetFPGA platform is a network hardware accelerator that can handle packet processing at line rate without CPU participation. Thus, if we use the NetFPGA for network emulator implementation, we can get an excellent network emulation tool and replace the software tools.

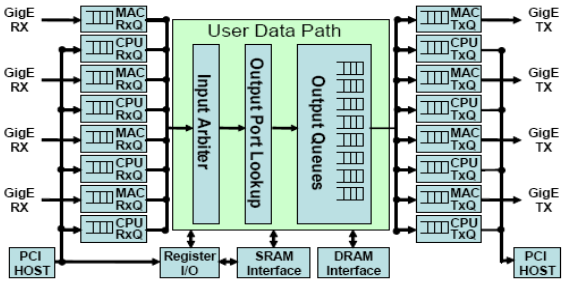



Figure 7. NetFPGA Reference Pipeline [14]

The gateware of the NetFPGA is designed in a modular fashion to allow users to modify or reconfigure modules to implement other useful devices. Basically, a network emulator needs to have 3 traffic control functions, i.e., the control of bottleneck bandwidth, the control of passing delay, and the control of packet loss probability. We can implement an efficient network emulator by using the design of the reference pipeline of the NetFPGA as shown in Figure 7 [14], which is comprised of eight receive queues, eight transmit queues, and user data path which includes input arbiter, output port lookup and output queues.

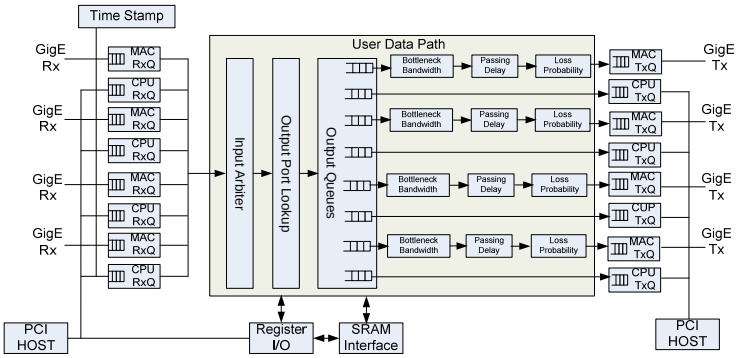



Figure 8. Nerwork Emaulator Pipeline Architecture

We can develop and add our modules to the user data path appropriately. The register interface allows software programs running on the host to exchange data with hardware module. We can provide parameters for network emulation through these register interfaces.

We can also utilize existing useful modules such as the NetFPGA packet generator application [14]. The 'bandwidth limiter' module can be used to implement the control of bottleneck bandwidth. The 'delay' and 'timestamp' modules can be used to implement the control of emulator passing delay, and the 'Drop_Nth_packet' module [15] can be modified to implement the control of packet loss probability of network emulators. We use the reference router pipeline for implementing the architecture of the network emulator as shown in Figure 8. In this Figure, we added three traffic control modules, i.e., the bottleneck bandwidth control module, the delay control module, and the loss probability control module to the end of each MAC output queue module. At the time of this writing, we are implementing a network emulator. We plan to do performance test of various high speed TCP protocols such as HSTCP[16], CUBIC TCP [17] and etc. Since the performance of these protocols is heavily dependent on bottleneck bandwidth and end-to-end delay, the network emulator can play very important role in the experiments. We will show the effectiveness of our network emulator compared to Dummynet or NISTnet.

## V. CONCLUSION

In this paper, we have introduced a new project in Korea for implementing platforms and deploying Future Internet testbed using them, called 'FIRST'. This project consists of two sub-projects. One is the 'FIRST@ATCA' for implementing large platform based on ATCA chassis, and the other is the 'FIRST@PC' for implementing PC-based platforms utilizing NetFPGA/OpenFlow switches. The scope of the FIRST@PC project is to develop a PC-based platform using NetFPGA/OpenFlow switches and to deploy Future Internet testbed on the KOREN. We have explained the ongoing status of the project, especially about NetFPGA module extension and performance test on the KOREN network. We have also showed the architecture of a network emulator which is now in implementing by using the NetFPGA platform. It will be useful in various network performance evaluations. We are now developing useful modules and components by using the NetFPGA platform and Openflow switches which will be the basis of Korean Future Internet testbed and global internetworking activities.

## ACKNOWLEDGMENT

This paper is one of results from the project (2009-F-050-01), "Development of the core technology and virtualized programmable platform for Future Internet" that is sponsored by MKE and KCC. I'd like to express my gratitude for the concerns to spare no support for the research and development of the project.

## REFERENCES

[1] GENI: Global Environment for Network Innovations, http://www.geni.net/

[2] FIRE: Future Internet Research and Experimentation, http://cordis.europa.eu/fp7/ict/fire/

[3] Shuji Esaki, Akira Kurokawa, and Kimihide Matsumoto, "Overview of the Next Generation Network," NTT Technical Review, Vol.5, No.6, June 2007.

[4] Jinho Hahm, Bongtae Kim, and Kyungpyo Jeon, "The study of Future Internet platfom in ETRI", The Magazine of the IEEK, Vol.36, No.3, March, 2009.

[5] Sapan Bhatia, Murtaza Motiwala, Wolfgang Muhlbauer, Vytautas Valancius, Andy Bavier, Nick Feamster, Larry Peterson, and Jennifer Rexford, "Hosting virtual networks on commodity hardware," Georgia Tech Computer Science Technical Report GT-CS-07-10, January 2008.

[6] FIRST@PC Project, http://trac.netmedia.gist.ac.kr/first/

[7] KOREN: Korea Advanced Research Network, http://koren2.kr/koren/eng/

[8] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martin Casado, Nick McKeown, and Scott Shenker, "NOX: towards an operating system for networks", ACM SIGCOMM Computer Communication Review, Vol.38, No.3, 2008.

[9] Capsulator, http://www.openflowswitch.org/wk/index.php/Capsulator

[10] Libpcap, http://www.tcpdump.org/pcap.htm

[11] iperf, http://sourceforge.net/projects/iperf/

[12] Dummynet, http://www.dummynet.com/

[13] NISTnet, http://www-x.antd.nist.gov/nistnet/

[14] G. Adam Covington, Glen Gibb, John Lockwood, and Nick McKeown, "A Packet Generator on the NetFPGA Platform", IEEE Symposium on. Field-Programmable Custom Computing Machines (FCCM), April 2009.

[15] Tutorial Proceeding : NetFPGA Tutorial in Seoul, http://fif.kr/netfpga/NetFPGA_KOREA_2009_02_19.pdf

[16] Sally Floyd, "HighSpeed TCP for large congestion windows", RFC3649, Dec.2003

[17] Injong Rhee, Lisong Xu, "CUBIC: A New TCP-Friendly High-Speed TCP Variant", PFLDnet 2005.

# A Fast, Virtualized Data Plane for the NetFPGA

Muhammad Bilal Anwer and Nick Feamster
School of Computer Science, Georgia Tech

## ABSTRACT

Network virtualization allows many networks to share the same underlying physical topology; this technology has offered promise both for experimentation and for hosting multiple networks on a single shared physical infrastructure. Much attention has focused on virtualizing the network control plane, but, ultimately, a limiting factor in the deployment of these virtual networks is data-plane performance: Virtual networks must ultimately forward packets at rates that are comparable to native, hardware-based approaches. Aside from proprietary solutions from vendors, hardware support for virtualized data planes is limited. The advent of open, programmable network hardware promises flexibility, speed, and resource isolation, but, unfortunately, hardware does not naturally lend itself to virtualization. We leverage emerging trends in programmable hardware to design a flexible, hardware-based data plane for virtual networks. We present the design, implementation, and preliminary evaluation of this hardware-based data plane and show how the proposed design can support many virtual networks without compromising performance or isolation.

## 1. Introduction

Network virtualization enables many logical networks to operate on the same, shared physical infrastructure. Virtual networks comprise virtual nodes and virtual links. Creating virtual nodes typically involves augmenting the node with a virtual environment (*i.e.*, either a virtual machine like Xen or VMWare, or virtual containers like OpenVZ). Creating virtual links involves creating tunnels between these virtual nodes (*e.g.*, with Ethernet-based GRE tunneling [4]). This technology potentially enables multiple service providers to share the cost of physical infrastructure. Major router vendors have begun to embrace router virtualization [5,7,9], and the research community has followed suit in building support for both virtual network infrastructures [2–4, 11] and services that could run on top of this infrastructure [6].

Virtual networks should offer good *performance*: The infrastructure should forward packets at rates that are comparable to a native hardware environment, especially as the number of users and virtual networks increases. The infrastructure should also provide strong *isolation*: Co-existing virtual networks should not interfere with one another. A logical approach for achieving both good performance and strong isolation is to implement the data plane in hardware. To date, however, most virtual networks provide only software support for packet forwarding; these approaches provide flexibility, ease of deployment, low cost and fast deployment, but poor packet forwarding rates and little to no isolation guarantees.

This paper explores how programmable network hardware can help us build virtual networks that offer both flexibility and programmability while still achieving good performance and strong isolation. The advent of programmable network hardware (*e.g.*, NetFPGA [8, 12]), suggests that, indeed, it may be possible to have the best of both worlds. Of course, even programmable network hardware does not inherently lend itself to virtualization, since it is fundamentally difficult to virtualize gates and physical memory. This paper represents a first step towards tackling these challenges. Specifically, we explore how programmable network hardware—specifically NetFPGA—might be programmed to support fast packet forwarding for multiple virtual networks running on the same physical infrastructure. Although hardware-based forwarding promises fast packet forwarding rates, the hardware itself must be shared across many virtual nodes on the same machine. Doing so in a way that supports a large number of virtual nodes on the same machine requires clever resource sharing. Our approach virtualizes the host using a host-based virtualized operating system (*e.g.*, OpenVZ [10], Xen [1]); we virtualize the data plane by multiplexing the resources on the hardware itself.

One of the major challenges in designing a hardware-based platform for a virtualized data plane is that hardware resources are fixed and limited. The programmable hardware can support only a finite (and limited) amount of logic, To make the most efficient use of the available physical resources, we must design a platform that *shares* common functions that are common between virtual networks while still isolating aspects that are specific to each virtual network (*e.g.*, the forwarding tables themselves). Thus, one of the main contributions of this paper is a design for hardware-based network virtualization that efficiently shares the limited hardware resources without compromising packet forwarding performance or isolation.

We present the design, implementation, and preliminary evaluation of a *hardware-based, fast, customizable virtualized data plane*. Our evaluation shows that our design provides the same level of forwarding performance as native hardware forwarding. Importantly for virtual networking, our design also shares common hardware elements between multiple virtual routers on the same physical node, which achieves up to 75% savings in the overall amount of logic that is required to implement independent physical routers. Additionally, our design achieves this sharing without compromising isolation: the virtual router's packet drop behavior under congestion is identical to the behavior of a single physical router.

The rest of this paper is organized as follows. Section 2 presents the basic design of a virtualized data plane based on

programmable hardware; this design is agnostic to any specific programmable hardware platform. Section 3 presents an implementation of our design using the NetFPGA platform. Section 4 concludes with a summary and discussion of future work.

## 2. Design Goals

This section outlines our design goals for a hardware-based virtual data plane, as well as the challenges with achieving each of these design goals.

1. **Virtualization at layer two.** Experimenters and service providers may wish to build virtual networks that run other protocols besides IP at layer three. Therefore, we aim to facilitate virtual networks that provide the appearance of layer-two connectivity between each virtual node. This function provides the illusion of point-to-point connectivity between pairs of virtual nodes. Alternatives, for achieving this design goal, are tunneling/encapsulation, rewriting packet headers, or redirecting packets based on virtual MAC addresses. In Section 3, we justify our design decision to use redirection.
2. **Fast forwarding.** The infrastructure should forward packets as quickly as possible. To achieve this goal, we push each virtual node's forwarding tables to hardware, so that the interface card itself can forward packets on behalf of the virtual node. Forwarding packets directly in hardware, rather than passing each packet up to a software routing table in the virtual context, results in significantly faster forwarding rates, less latency and higher throughput. The alternative—copying packets from the card to the host operating system—requires copying packets to memory, servicing interrupts, and processing the packet in software, which is significantly slower than performing the same set of operations in hardware.
3. **Resource guarantees per virtual network.** The virtualization infrastructure should be able to allocate specific resources (bandwidth, memory) to specific virtual networks. Providing such guarantees in software can be difficult; in contrast, providing hard resource guarantees in hardware is easier, since each virtual network can simply receive a fixed number of clock cycles. Given that the hardware forwarding infrastructure has a fixed number of physical interfaces, however, the infrastructure must also determine how to divide resources across the virtual interfaces that are dedicated to a single physical interface.

The next section describes the hardware architecture that allows us to achieve these goals.

## 3. Design and Implementation

This section describes our design and implementation of a hardware-based virtual data plane. The system associates each incoming packet with a virtual environment and forwarding table. In contrast to previous work, the hardware itself makes forwarding decisions based on the packet's association to a virtual forwarding environment; this design provides fast, hardware-based forwarding for up to eight virtual routers running in parallel on shared physical hardware. By separating the control plane for each virtual node (*i.e.*, the routing protocols that compute paths) from the data plane (*i.e.*, the infrastructure responsible for forwarding packets) each virtual node can have a separate control plane, independent of the data plane implementation.

**Overview** In our current implementation, each virtual environment can have up to four virtual ports; this is a characteristic of our current NetFPGA-based implementation, not a fundamental limitation of the design itself. The physical router has four output ports and, hence, four output queues. Each virtual MAC is associated with one output queue at any time, this association is not fixed and changes with each incoming packet. Increasing the number of output queues, allows us to increase the number of virtual ports per virtual router. The maximum number of virtual ports then depends on how much resources we have to allocate for the output queues. In addition to more output queues we would also need to increase the size of VMAC-VE (Virtual MAC to Virtual Environment) mapping table and the number of context registers associated with a particular instance of virtual router. There are four context registers for each virtual router and they are used to add source MAC addresses for each outgoing packet, depending upon the outgoing port of packet.

Each virtual port on the NetFPGA redirects the packet to the appropriate virtual environment or forward the packet to the next node, depending on the destination address and the packet's association to a particular virtual environment. We achieve this association by establishing a table that maps virtual MAC addresses to virtual environment. These virtual MAC addresses are the addresses assigned by the virtual environment owner and can be changed any time. By doing so, the system can map traffic from virtual links to the appropriate virtual environments without any tunneling.

In the remainder of this section, we describe the system architecture. First, we describe the control plane, which allows router users to install forwarding table entries into the hardware, and how the system controls each virtual environment's access to the hardware. Next, we describe the software interface between processes in each virtual environment and the hardware. Finally, we describe the system's data path, which multiplexes each packet into the appropriate virtual environment based on its MAC address.

### 3.1 Control Plane

The virtual environment contains two contexts: the virtual environment context (the "router user") and the root context (the "super user"). The router user has access to the container that runs on the host machine. The super user can control all of the virtual routers that are hosted on the FPGA, while router users can only use the resources which are allocated to them by the super user. Our virtual router implementation has a set of registers in FPGA, that provides access to the super user and to the router users. This separation of privilege corresponds to that which exists in a typical OpenVZ setup, where multiple containers co-exist on a sin-

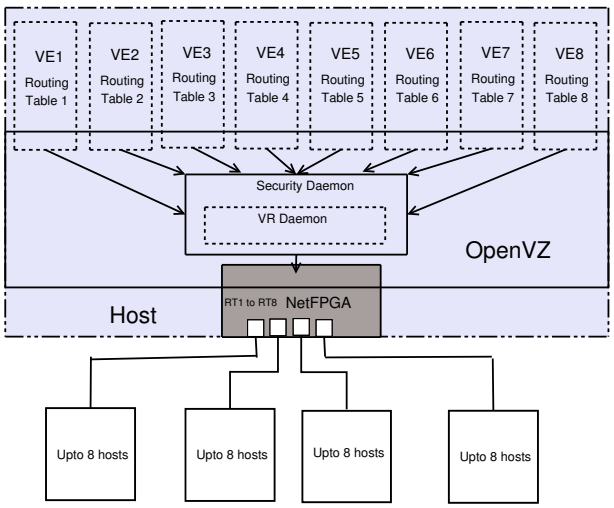



**Figure 1: OpenVZ and virtual router.**

gle physical machine, but only the user in the root context has access to super user privileges.

**Virtual environments** As in previous work (*e.g.*, Trellis [4]), we virtualize the control plane by running multiple virtual environments on the host machine. The number of OpenVZ environments is independent of the virtual routers sitting on FPGA, but the hardware can support at most eight virtual containers. Each container has a router user, which is the root user for the container; the host operating system's root user has super user access to the virtual router. Router users can use a command-line based tool to interact with their instance of virtual router. These users can read and write the routing table entries and specify their own context register values.

Each virtual router user can run any routing protocol, but all the routing table entries update/read requests must pass through the security daemon and the virtual router daemon (as shown in Figure 1). The MAC addresses stored in the context registers must be the same addresses that the virtual router container uses to reply for the ARP requests. Once a virtual router user specifies the virtual port MAC addresses, the super user enters these addresses in the VMAC-VE table; this mechanism prevents a user from changing MAC addresses arbitrarily.

**Hardware access control** This *VMAC-VE table* stores all of the virtual environment ID numbers and their corresponding MAC addresses. Initially, this table is empty to provide access of a virtual router to a virtual environment user. The system provides a mechanism for mediating router users' access to the hardware resources. The super user can modify the VMAC-VE (Virtual MAC and Virtual Environment mapping) table. Super user grants the router user access to the fast path forwarding provided by the hardware virtual router by adding the virtual environment ID and the corresponding MAC addresses to the VMAC-VE table. If the super user wants to destroy a virtual router or deny some users access to the forwarding table, it simply removes the virtual environment ID of the user and its corresponding MAC addresses. Access to this VMAC-VE table is provided by a register file which is only accessible to super user.

**Control register** As shown in Figure 1, each virtual environment copies the routing table from its virtual environment to shared hardware. A 32-bit control register stores the virtual environment ID that is currently being controlled. Whenever a virtual environment needs to update its routing tables, it sends its request to *virtual router daemon*. After verifying the virtual environment's permissions, this daemon uses the control register to select routing tables that belong to the requesting virtual environment and updates the IP lookup and ARP table entries for that particular virtual environment. After updating the table values, daemon resets the control register value to zero.

## 3.2 Software Interface

As shown in Figure 1, the *security daemon* prevents unauthorized changes to the routing tables by controlling access to the virtual router control register. The virtual router control register is used to select the virtual router for forwarding table updates.The security daemon exposes an API that router users can use to interact with their respective routers, including reading or writing the routing table entries. Apart from providing secure access to all virtual router users, the security daemon logs user requests to enable auditing.

The *software interface* provides a mechanism for processing packets using software exceptions. The hardware-based fast path cannot process packets with the IP options or ARP packets, for example. These packets are sent to virtual router daemon without any modifications, virtual router daemon,also maintains a copy of VMAC-VE table. It looks at the packet's destination MAC and sends the packet to the corresponding virtual environment running on the host environment. Similarly, when the packet is sent from any of the containers, it first received by virtual router daemon through the security daemon, which sends the packet to the respective virtual router in hardware for forwarding.

The super user can interact with all virtual routers via a command-line interface. Apart from controlling router user accesses by changing the VMAC-VE table, the super user can examine and modify any router user's routing table entries using control register.

## 3.3 Data Plane

To provide virtualization in a single physical router, the router must associate each packet with its respective virtual environment. To determine a packet's association with a particular virtual environment, the router uses virtual environment's MAC address; this MAC address as described earlier, apart from allowing/denying access to the virtual router users, the VMAC-VE table determines how to forward packets to the appropriate virtual environment, as well as whether to forward or drop the packet.

**Mapping virtual MACs to destination VEs** Once the table is populated and a new packet arrives at the virtual router, its destination MAC is looked up in VMAC-VE table, which provides mapping between the virtual MAC addresses and

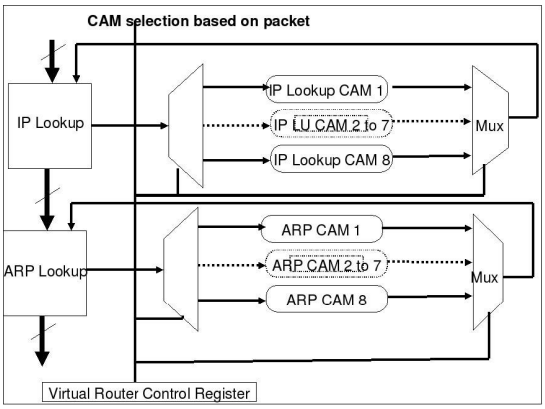



**Figure 2: Virtual router table mappings.**

virtual environment IDs. Virtual MAC addresses in VMAC-VE table correspond to the MAC addresses of the virtual ethernet interfaces used by virtual environment. A user has access to four registers, which can be used to update the MAC address of user's choice. These MAC addresses must be the same as the MAC addresses of virtual environment. Since there are four ports on NetFPGA card each virtual environment has a maximum of four MAC addresses inside this table; this is only limitation of our current implementation. As explained earlier, increasing the number of output queues and context registers will permit each virtual environment to have more than four MAC addresses. The system uses a CAM-based lookup mechanism to implement the VMAC-VE table. This design choice makes the implementation independent of any particular vendor's proprietary technology. For example, the proprietary TEMAC core from Xilinx provides a MAC address filtering mechanism, but it can only support 4 to 5 MAC addresses per TEMAC core, and most importantly it can't provide demuxing of the incoming packets to the respective VE.

**Packet demultiplexing and forwarding** In the current implementation, all four physical ethernet ports of the router are set into promiscuous mode, which allows the interface receive any packet for any destination. After receiving the packet, its destination MAC address is extracted inside the virtual router lookup module, as shown in Figure 2. If there is a match in the table, the packet processed and forwarded; otherwise, it is dropped.

This table lookup also provides the virtual environment ID (VE-ID) that is used to switch router context for the packet which has just been received. In a context switch, all four MAC addresses of the router are changed to the MAC addresses of the virtual environment's MAC addresses. As shown in Figure 3, the VE-ID indicates the respective IP lookup module. In the case of IP lookup hit, the MAC address of next hop's IP is looked up in ARP lookup table. Once the MAC address is found for the next hop IP, the router needs to provide the source MAC address for the outgoing packet. Then, context registers are used to append the corresponding source MAC address and send the packet.

Based on the packet's association with a VE, the context register values are changed that correspond to the four MAC addresses for virtual router in use. The router's context remains active for the duration of a packet's traversal through FPGA and changes when the next incoming packet arrives. Each virtual port appears one physical port with its own MAC address. Once the forwarding engine decides a packet's fate, it is directed to the appropriate output port. The outgoing packet must have the source MAC address that corresponds to the virtual port that sends the packet. To provide each packet with its correct source MAC address, the router uses context registers. The number of context registers is equal to the number of virtual ports associated with the particular router. The current implementation uses four registers, but this number can be increased if the virtual router can support more virtual ports.

**Shared functions** Our design maximizes available resources to share different resources with other routers on the same FPGA. It only replicates those resources which are really necessary to implement fast path forwarding. To understand virtual router context and its switching with every new packet, we first describe the modules that can be shared in an actual router and modules that cannot be shared. Router modules that involve decoding of packets, calculating checksums, decrementing TTLs, etc. can be shared between different routers, as they do not maintain any state that is specific to a virtual environment. Similarly, the input queues and input arbiter is shared between the eight virtual routers. Packets belonging to any virtual router can come into any of the input queues and they are picked up by arbiter to be fed into virtual router lookup module. Output queues are shared between different virtual routers, and packets from different virtual routers can be placed in any output queue.

**VE-specific functions** Some resources can not be shared between the routers. The most obvious among them is the forwarding information base. In our current virtual router implementation we have used, NetFPGA's reference router implementation as our base implementation. In this implementation a packet that needs to be forwarded, needs at least three information resources namely IP lookup table, MAC address resolution table and router's MAC addresses. These three resources are unique to every router instance and they can not be removed and populated back again with every new packet. Therefore, the architecture maintains a copy of each of these resources for every virtual router. The current implementation maintains a separate copy of all these resources for every virtual router instantiated inside the FPGA.

## 4. Discussion and Future Work

In this section, we describe several possible extensions to the current implementation. Some of these additions come from making these virtual routers more router like; some stem from the requirements of extending the current implementation to better support virtual networks. We believe that the current implementation must have a minimum set of these additions to completely function as a virtual router.

The virtualized data plane we have presented could be extended to support collecting statistics about network traffic in each virtual network. Today, administrators of physical networks can obtain the traffic statistics for their respective

networks; we aim to provide similar function for virtual networks. Extending this function to the NetFPGA requires adding new function to the current logic on the NetFPGA; it also entails addressing challenges regarding the efficient use of the relatively limited memory available on the physical card itself.

Each physical router user is able to update her forwarding tables with the frequency of their like. Our current implementation lacks this feature as once one user tries to update his/her respective table others are blocked. Allowing concurrent packet forwarding and forwarding-table updates requires a completely different register set interface for each virtual router to update its respective forwarding table.

Virtual routers must provide speed, scalability and isolation. We have been able to meet the fast forwarding path and scalability requirements. The hardware-based implementation provides isolation to the CPU running on the host machine of the NetFPGA card. However, the current implementation does not isolate the traffic between different virtual networks: in the current implementation, all virtual networks share the same physical queue for a particular physical interface, so traffic in one virtual network can interfere with the performance observed by a different virtual network. In an ideal case, no traffic in a different virtual network should affect other virtual router's bandwidth. In our ongoing work, we are examining how various queuing and scheduling discimplines might be able to provide this type of isolation, while still making efficient use of the relatively limited available resources.

## 5. Conclusion

Sharing the same physical substrate among a number of different virtual networks amortizes the cost of the physical network; as such, virtualization is promising for many networked applications and services. To date, however, virtual networks typically provide only software-based support for packet forwarding, which results in both poor performance and isolation. The advent of programmable network hardware has made it possible to achieve improved isolation and packet forwarding rates for virtual networks; the challenge, however, is designing a hardware platform that permits sharing of common hardware functions across virtual routers without compromising performance or isolation.

As a first step towards this goal, this paper has presented a design for a fast, virtualized data plane based on programmable network hardware. Our current implementation achieves the isolation and performance of native hardware forwarding and implements shares hardware modules that are common across virtual routers. Although many more functions can ultimately be added to such a hardware substrate (*e.g.*, enforcing per-virtual router resource constraints), we believe our design represents an important first step towards the ultimate goal of supporting a fast, programmable, and scalable hardware-based data plane for virtual networks.

## REFERENCES


[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proc. 19th ACM Symposium on Operating Systems Principles (SOSP)*, Lake George, NY, Oct. 2003.

[2] A. Bavier, M. Bowman, D. Culler, B. Chun, S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink, and M. Wawrzoniak. Operating System Support for Planetary-Scale Network Services. In *Proc. First Symposium on Networked Systems Design and Implementation (NSDI)*, San Francisco, CA, Mar. 2004.

[3] A. Bavier, N. Feamster, M. Huang, L. Peterson, and J. Rexford. In VINI Veritas: Realistic and Controlled Network Experimentation. In *Proc. ACM SIGCOMM*, Pisa, Italy, Aug. 2006.

[4] S. Bhatia, M. Motiwala, W. Muhlbauer, Y. Mundada, V. Valancius, A. Bavior, N. Feamster, L. Peterson, and J. Rexford. Trellis: A Platform for Building Flexible, Fast Virtual Networks on Commodity Hardware. In *3rd International Workshop on Real Overlays & Distributed Systems*, Oct. 2008.

[5] Cisco Multi-Topology Routing. `http://www.cisco.com/en/US/products/ps6922/products_feature_guide09186a00807c64b8.html`.

[6] N. Feamster, L. Gao, and J. Rexford. How to lease the Internet in your spare time. *ACM Computer Communications Review*, 37(1):61–64, 2007.

[7] JunOS Manual: Configuring Virtual Routers. `http://www.juniper.net/techpubs/software/erx/junose72/swconfig-system-basics/html/virtual-router-config5.html`.

[8] J. Lockwood, N. McKeown, G. Watson, G. Gibb, P. Hartke, J. Naous, R. Raghuraman, and J. Luo. NetFPGA–An Open Platform for Gigabit-Rate Network Switching and Routing. In *IEEE International Conference on Microelectronic Systems Education*, pages 160–161. IEEE Computer Society Washington, DC, USA, 2007.

[9] Juniper Networks: Intelligent Logical Router Service. `http://www.juniper.net/solutions/literature/white_papers/200097.pdf`.

[10] OpenVZ: Server Virtualization Open Source Project. `http://www.openvz.org`.

[11] J. Turner, P. Crowley, J. DeHart, A. Freestone, B. Heller, F. Kuhns, S. Kumar, J. Lockwood, J. Lu, M. Wilson, et al. Supercharging planetlab: a high performance, multi-application, overlay network platform. In *Proc. ACM SIGCOMM*, Kyoto, Japan, Aug. 2007.

[12] G. Watson, N. McKeown, and M. Casado. NetFPGA: A tool for network research and education. In *2nd workshop on Architectural Research using FPGA Platforms (WARFP)*, 2006.

# A Windows Support Framework for the NetFPGA 2 Platform

Chen Tian[1,2], Danfeng Zhang[1], Guohan Lu[1], Yunfeng Shi[1], Chuanxiong Guo[1], Yongguang Zhang[1]
[1]Mircosoft Research Asia
[2]Huazhong University of Science and Technology
{v-tic, v-daz, lguohan, v-yush, chguo, ygz}@microsoft.com

## ABSTRACT

The NetFPGA 2 platform is widely used by the networking research and education communities. But the current software package supports only Linux. This paper describes the development of a Windows Support Framework for the NetFPGA 2 platform. We present the Windows Support Framework design after we briefly introduce the Windows Network Driver Interface Specification (NDIS). We then describe the implementation details such as drivers structure, the packet send/receive procedures, and the user-mode tool kit. Experiments show that our implementation achieves 160Mb/s sending rate and 230Mb/s receiving rate, respectively. We hope the Windows Support Framework brings NetFPGA to those researchers and students who are familiar with the Windows operating system.

## 1. INTRODUCTION

The NetFPGA 2 platform enables researchers and students to prototype high-performance networking systems using field-programmable gate array (FPGA) hardware [5]. As a line-rate, flexible, and open platform, it is widely accepted by the networking community: over 1,000 NetFPGA systems have been shipped and many innovative networking designs have been implemented [1, 3]. But currently the platform only supports Linux, and the developments of NetFPGA projects are limited to the Linux environments. Given the dominant market share of the Windows operating system, adding support for Windows will help those researchers and students who are familiar with the Windows system.

In this paper, we describe the design and implementation of our Windows Support Framework(WSF) for NetFPGA 2. The WSF is driven by both the requirements of our Windows-based testbed in Microsoft Research Asia and the community benefits. Windows Support Framework has two components:

- *Kernel Drivers.* A suit of kernel mode device drivers that enable the deployment of NetFPGA 2 platform in the Windows operating system.
- *User-Mode Tool Kit.* Common development functions such as registers reading/writing, FPGA bitfile downloading, and packet injecting/intercepting, are implemented as user mode tools.

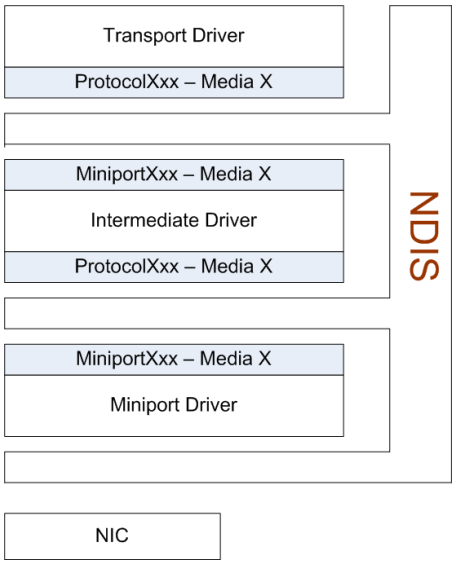



**Figure 1: NDIS architecture**

The rest of the paper is organized as follows. In Section 2, we first briefly introduce the Windows network driver architecture, and then present the support framework design. We describe the implementation details such as important routines of kernel drivers and the packet send/receive procedures in Section 3. We present experimental results in Section 4. Section 5 concludes the paper and discusses future work.

## 2. DESIGN

### 2.1 Windows Network Driver Architecture

The Windows operating system use Network Driver Interface Specification (NDIS) architecture to support network devices and protocols. Based on the OSI seven-

layer networking model, the NDIS library abstracts the network hardware from network drivers. NDIS also specifies the standard interfaces between the layered network drivers, thereby abstracting lower-level drivers that manage hardware for upper-level drivers. To support the majority of Windows users, we design WSF drivers to conform with NDIS version 5.1, which is Windows 2000 backward compatible. Most drivers are written in Kernel-Mode Driver Framework(KMDF) style, which provides object-based interfaces for Windows drivers [4].

As we show in Fig. 1, there are three primary network driver types [4]:

- *Miniport Drivers.* A Network Interface Card(NIC) is normally supported by a miniport driver. An NDIS miniport driver has two basic functions: managing the NIC hardware, including transmiting and receiving data; interfacing with higher-level drivers, such as protocol drivers through the NDIS library. The NDIS library encapsulates all operating system routines, that a miniport driver must call, to a set of functions (*NdisMXxx()* and *NdisXxx()* functions). The miniport driver, in turn, exports a set of entry points (*MiniportXxx()* routines) that NDIS calls for its own purposes or on behalf of higher-level drivers to send down packets.

- *Protocol Drivers.* Transport protocols, e.g. TCP-/IP stack, are implemented as protocol drivers. At its upper edge, a protocol driver usually exports a private interface to its higher-level drivers in the protocol stack. At its lower edge, a protocol driver interfaces with miniport drivers or intermediate network drivers. A protocol driver initializes packets, copies sending data from the application into the packets, and sends the packets to its lower-level drivers by calling *NdisXxx()* functions. It must also exports a set of entry points (*ProtocolXxx()* routines) that NDIS calls for its own purposes or on behalf of lower-level drivers to indicate up received packets.

- *Intermediate Drivers.* Intermediate drivers are layered between miniport drivers and transport protocol drivers. They are used to translate between different network media or map virtual miniports to physical NICs. An intermediate driver exports one or more virtual miniports at its upper edge. To a protocol driver, a virtual miniport that was exported by an intermediate driver appears to be a real NIC; when a protocol driver sends packets to a virtual miniport, the intermediate driver propagates these packets to an underlying miniport driver. At its lower edge, the intermediate driver appears to be a protocol driver to an underlying miniport driver; when the underlying miniport driver indicates received packets, the intermediate driver propagates the packets up to the protocol drivers that are bound to its virtual miniport.

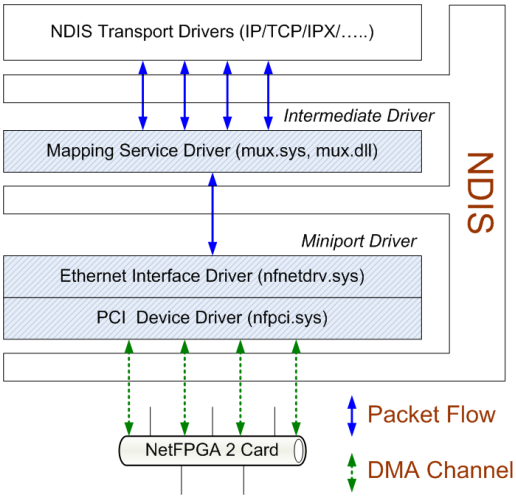



**Figure 2: The structure of the kernel mode drivers.**

## 2.2 The NDIS Kernel Drivers

As a PCI board, NetFPGA card has a memory space for PCI configuration information. The information describes the hardware parameters of the devices on the board. The configuration of the NetFPGA reference design contains only one PCI device. All four Ethernet ports share the same interrupt number, and tranmit/receive over their own DMA channels.

Supporting four NetFPGA Ethernet Ports in Linux is relatively simple. During the initialization phase, the Linux driver calls system routine *register_netdev()* four times to register NetFPGA ports as four distinct logical Ethernet devices.

Network driver support in NDIS 5.1 context is more sophisticated, mainly due to Plug-and-Play (PnP) and power management. The initialization process of a network device is managed by the Plug-and-Play(PnP) manager, hence one NetFPGA card can register only one logical Ethernet device. Apparently, one miniport driver alone is not enough to support four ports of NetFPGA in NDIS 5.1 context. We need an additional intermediate driver to map one single NetFPGA PCI card to four virtual Ethernet miniports.

As shown in Fig 2, the NDIS Kernel Driver of NetFPGA has three driver modules. From bottom-up, the PCI Device Driver (PDD) directly interfaces with the hardware and provides I/O services, such as DMA operations and access of registers; the Ethernet Interface Driver (EID) registers the NetFPGA card as a logical

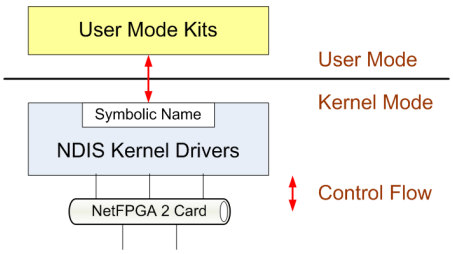



**Figure 3: User mode/kernel mode Communication.**

Ethernet device in response to PnP manager; these two drivers can be regarded as two submodules of a single NDIS miniport driver. The Mapping Service Driver (MSD) is an intermediate driver, which maps the underlying NetFPGA device to four virtual miniports and exports them to upper layer protocols.

The physical ports and virtual miniports maintain an exact match relationship. Every packet received by PDD is associated with its network port number; EID indicates the packet up together with the port; the MSD then indicates the packet to the corresponding virtual miniports. For the packet sending procedure, the order is reversed.

### 2.3 User-Mode Tool Kit

Besides kernel mode drivers, developers need to download FPGA bitfiles, or read/write registers for their own purpose. Basically, the main operations of bitfile download are also registers reading and writing. Communications between user mode tools and kernel mode drivers are needed to pass values up and down. As shown in Fig 3, each driver exports an associated symbolic device name: applications can open a driver handle by calling a *CreateFile()* routine; the communications can then be performed by calling *DeviceIoControl()* function and passing the I/O control commands down and reading the reported data back to applications.

To facilitate packet analysis of user derived protocols, packet injecting/intercepting functions are also implemented. All these implementation details will be given in the next section.

## 3. IMPLEMENTATION

This section gives implementation details. First the important routines of the three drivers are presented; then the complete packet send/receive procedures are illustrated to help understanding the asynchronous interactions among drivers; finally the implementation details of the development tool kit are also presented.

### 3.1 PCI Device Driver

The PCI Device Driver takes KMDF *PCI9x5x* example of Windows Driver Kit [4] as its template. PDD provides asynchronous hardware access interfaces to its upper layer EID module. The important routines are listed below:

- During device initialization, callback routine *PCIEEvtDeviceAdd()* is called in response to Windows' PnP manager. The routine registers all the callbacks and allocates software resources required by the device; the most important resources are *WriteQueue* and *PendingReadQueue*, which will serve write/read requests later.
- After the PnP manager has assigned hardware resources to the device and after the device has entered its uninitialized working (D0) state, callback routine *PCIEEvtDevicePrepareHardware()* is called to set up the DMA channels and map memory resources, make the device accessible to the driver.
- Each time the device enters its D0 state, callback routine *PCIEEvtDeviceD0Entry()* is called just before the enable of hardware interrupt; the NetFPGA card registers are initialized here.
- When NetFPGA generates a hardware interrupt, the driver's Interrupt Service Routine (ISR) *PCIEEvtInterruptIsr()* quickly save interrupt information, such as the interrupt status register's content, and schedules a Deferred Procedure Call (DPC) to process the saved information later at a lower Interrupt Request Level(IRQL).
- DPC routine *PCIEEvtInterruptDpc()* is scheduled by ISR. This routine finishes the servicing of an I/O operation.
- In response to a write request, callback routine *PCIEEvtProgramWriteDma()* programs the NetFPGA device to perform a DMA transmit transfer operation.

### 3.2 Ethernet Interface Driver

The Ethernet Interface Driver takes KMDF *ndisedge* example as its template. To its lower edge, it interfaces with PDD by I/O request packets (IRPs) operations; to its upper edge, it acts as a standard Ethernet device. The important routines are listed below:

- The driver's entry routine *DriverEntry()* calls NDIS function *NdisMRegisterMiniport()* to register the miniport driver's entry points with NDIS.
- Callback routine *MPInitialize()* is the entry point of Initialize Handler. Called as part of a system PnP operation, it sets up a NIC for network I/O operations, and allocates resources the driver needs to carry out network I/O operations.

- Callback routine *MPSendPackets()* is the entry point of Send Packets Handler. An upper layer protocol sends packets by calling NDIS function *NdisSendPackets()*. NDIS then calls this routine on behalf of the higher-level driver. This routine prepares write resources and initiates a write request. The port number is associated with the request by *WdfRequestSetInformation()* operations.
- *NICReadRequestCompletion()* is the completion routine for the read request. This routine calls NDIS function *NdisMIndicateReceivePacket()* to indicate the received packet to NDIS. The receive port number is associated with the packet by saving it in the *MiniportReserved* field of the NDIS_PACKET structure.

## 3.3 Mapping Service Driver

The Mapping Service Driver takes the famous *MUX* intermediate driver as its template. A *MUX* intermediate driver can expose virtual miniports in a one-to-$n$, $n$-to-one, or even an $m$-to-$n$ relationship with underlying physical devices. One challenge is how to configure the protocol binding relationships: only a NetFPGA card is legal to be bound to this intermediate driver and to export four virtual miniports. We achieve this goal by modifying the accompanying installation DLL component.

The important routines are listed below:

- Callback routine *MPInitialize()* is the entry point of virtual miniport Initialize Handler. The MAC addresses of virtual miniports are read from the corresponding registers during the initialization phase.
- Similar to its counterpart of EID, callback routine *MPSendPackets()* is the entry point of Send Packets Handler. The send port number is associated with the packet by saving it in the *MiniportReserved* field of NDIS_PACKET structure.
- Callback routine *PtReceivePacket()* is the entry point of Receive Packet Handler. This routine associates each packet with its corresponding virtual miniport and call NDIS function *NdisMIndicateReceivePacket()* to indicate it up.

## 3.4 Packet Sending Procedure

Fig 4 gives the life cycle of a sending packet.

1. When an application wants to send data, the upper layer transport protocol prepares packets and calls NDIS function *NdisSendPackets()*; a packet is passed by NDIS to a corresponding virtual miniport interface of MSD.
2. NDIS calls MSD's callback routine *MPSendPackets()* with the packet; this routine associates the corresponding send port with the packet, then call *NdisSendPackets()* to pass the packet down to EID.
3. The NIC write request can be initiated asynchronous when a packet is ready. NDIS calls MSD's callback routine *MPSendPackets()* with the packet. This routine prepares write resources and initiates an asynchronous write request to PDD.
4. Upon receiving a write request, the PDD callback routine *PCIEEvtProgramWriteDma()* acquires a transmit spinlock to obtain DMA control; after that, a new DMA transmit transfer can be started.
5. After the completion of the DMA transmit transfer, the INT_DMA_TX_COMPLETE bit is set in a physical interrupt; the ISR reads the status and schedules a DPC; the DPC routine informs EID of the write request completion and releases the spinlock.
6. The EID's completion routine for the write request *NICWriteRequestCompletion()* is called; it frees write resources and calls NDIS function *NdisMSendComplete()* to inform the upper layer MSD.
7. Consequently the MSD's send completion callback routine *PtSendComplete()* is triggered by NDIS, and it also calls *NdisMSendComplete()* to inform its upper layer transport protocol.
8. On behalf of MSD, NDIS calls the upper layer protocol's callback routine *ProtocolSendComplete()*, the packet send process is finally completed.

## 3.5 Packet Receiving Procedure

For packet receiving, Fig 5 gives the life cycle of a packet.

1. After initialization, EID posts a sequence of NIC read requests to PDD in advance.
2. When a packet arrives, the INT_PKT_AVAIL bit is set in a physical interrupt; the ISR reads the status and schedules a DPC; the PDD DPC routine dequeues a read request, acquires a receive spinlock to obtain DMA control; after that, a new DMA receive transfer can be started.
3. After the packet is received, the INT_DMA_RX _COMPLETE bit is set in a physical interrupt; the ISR reads the status and schedules a DPC; the PDD DPC routine informs EID of the read request completion and releases the spinlock.
4. The EID's completion routine for the read request *NICReadRequestCompletion()* is called; the routine associates the corresponding receive port with the packet, then calls NDIS function *NdisMIndicateReceivePacket()* to inform its upper layer driver MSD.

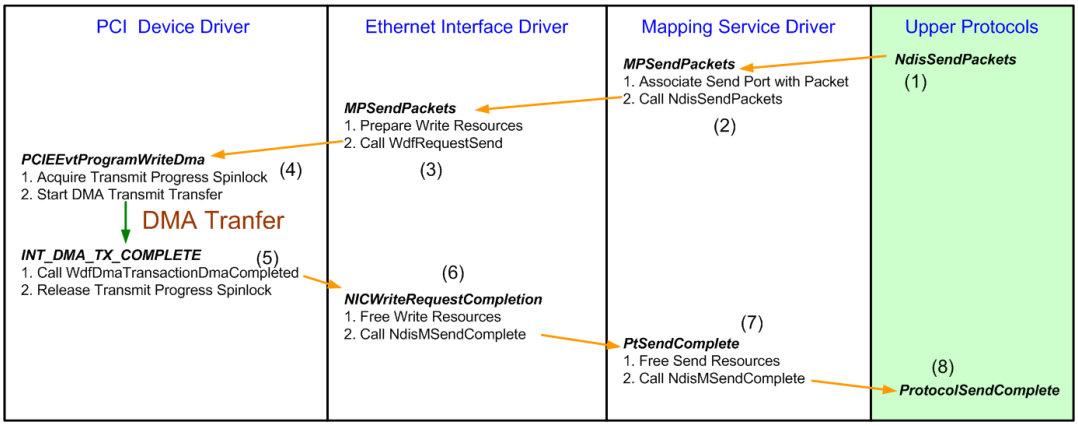



**Figure 4: Packet sending procedure.**

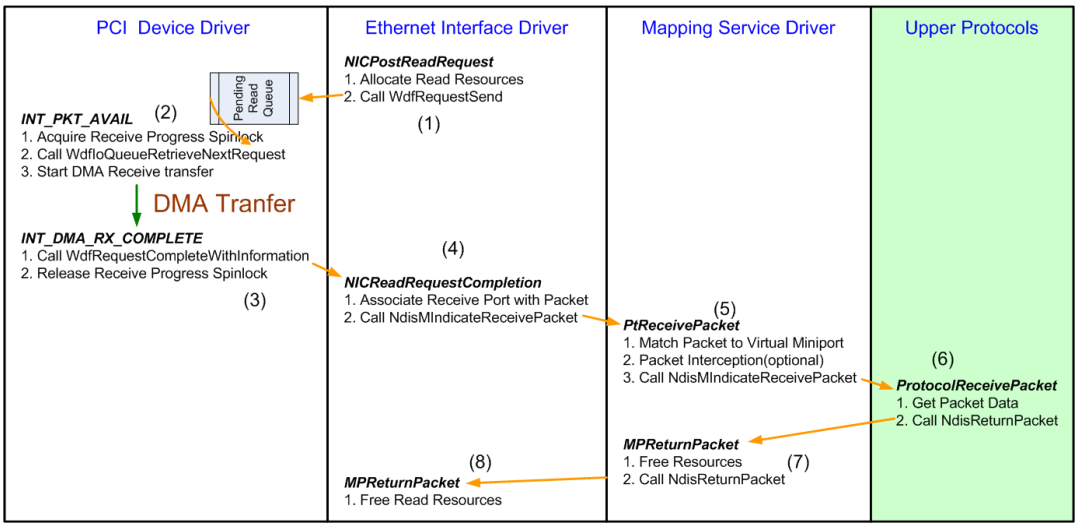



**Figure 5: Packet receiving procedure.**

5. The MSD's callback routine *PtReceivePacket()* is called by NDIS; the routine matches the packet to its corresponding virtual miniport and also indicates the packet up by NDIS function *NdisMIndicateReceivePacket()*;

6. After the packet data is received, the upper layer protocol calls NDIS function *NdisReturnPacket()*.

7. Consequently the MSD's receive completion callback routine *MPReturnPacket()* is triggered by NDIS, and it also calls *NdisReturnPacket()* to inform its lower layer driver EID.

8. The EID's callback routine *MPReturnPacket()* is called by NDIS, and the packet receive process is finally completed.

## 3.6 Implementation Details of the Tool Kit

The registers read and write operations are shown in Fig 6(a): an I/O control command is issued to EID first; the EID then builds an internal IRP and recursively calls the lower PDD to complete the request.

The packet injecting/intercepting functions are implemented in the MSD, as shown in Fig 6(b). After reception of a packet from the application level tool, the routine *PtSend()* routine injects the packet into outgoing packet flow path. A boolean value is associated with each packet send request to distinguish the injected packets from normal packets; in step (7) of Fig 4 when callback routine *PtSendComplete()* is called, only the completion of normal packets are required to be reported to upper layer protocols.

The packet interception logic is embedded in step (5)

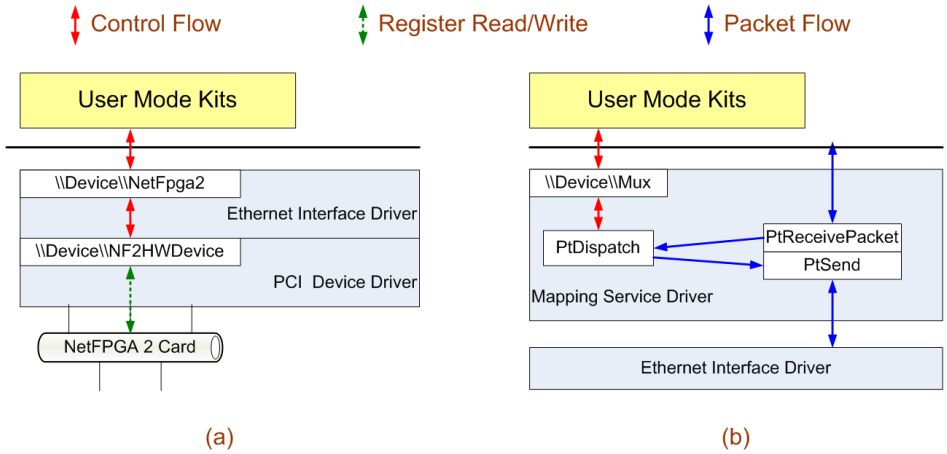



**Figure 6: (a) Register Read/Write. (b) Packet Intercept/Inject.**

of Fig 5. The interception kit posts a sequence of read IRPs to MSD in advance. In routine *PtReceivePacket()*, every incoming packet is checked first; the decision can be drop, modify, copy up to user mode kit, or continue without intervention.

# 4. FOR USERS OF WSF

## 4.1 Package

The package of WSF can be downloaded from our group web site [2]. The source code of both NDIS kernel drivers and user mode tool kits are included. Users who want to write a Windows program that interfaces with the NetFPGA can take the source code as their template.

An installation manual is included [6] in the package to guide the preparation of users and illustrate the use of the WSF from a users perspective. Some usage examples are also given for tool kits in the manual .

## 4.2 Performance

We build the source code using Microsoft Windows Device Driver Kit (DDK) Release 6001.18001 version. The experiment servers are DELL optiplex 755 with Windows Server 2003 Enterprize Edition with service pack 2. Each server has a 2.33G E6550 Intel Core 2 Duo CPU and 2 GB memory. The *iperf* software [7] is selected to perform the test with the 1500 bytes packets.

The forwarding performance is not affected by operating systems. So we only test the send/receive throughput of NetFPGA equipped host. In our experiment, the two servers connect to a gigabit switch using one NetFPGA port. Both 32-bit/64-bit performance are evaluated. The results are shown in Table. 1.

The host throughput achievable by Windows drivers are lower than that in Linux environment. The reason is that we implement the NetFPGA roles(PCI device/Network device/Port Mapping) to three separate drivers: each send/receive packet must pass through three drivers. A future version(Discussed in Section 5) may compact all functions to a single driver to improve host throughput. However the NetFPGA community is focused on hardware forwarding engine, which may make this throughput performance still acceptable.

| Throughput | Send (Mb/s) | Receive (Mb/s) |
|---|---|---|
| Win2003 x86 | 158 | 234 |
| Win2003 x64 | 156 | 233 |
| Linux | 186 | 353 |

**Table 1: Throughput of experiments.**

# 5. CONCLUSION

We have presented the design and implementation of a Windows Support Framework for the NetFPGA 2 Platform. We hope the Windows Support Framework brings NetFPGA to those researchers and students who are familiar with the Windows operating system.

The Windows Support Framework for NetFPGA 2 Platform is still in active development. Our future works may include but not limit to:

- *Migrate to NDIS 6.0.* Many new features are provided in NDIS 6.0 version, and we have a chance to compact all functions to a single driver and at the same time improve performance.
- *Support new NetFPGA hardware.* We plan to port our implementation to support the upcoming 10G NetFPGA once it becomes available.

# 6. REFERENCES

[1] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. In *Proc. SIGCOMM*, 2008.

[2] Data Center Networking at MSRA. http://research.microsoft.com/en-us/projects/msradcn/default.aspx.

[3] Dilip Antony Joseph, Arsalan Tavakoli, Ion Stoica, Dilip Joseph, Arsalan Tavakoli, and Ion Stoica. A policy-aware switching layer for data centers, 2008.

[4] MSDN. Microsoft windows driver kit (wdk) documentation, 2008.

[5] J. Naous, G. Gibb, S. Bolouki, and N. McKeown. NetFPGA: Reusable Router Architecture for Experimental Research. In *PRESTO*, 2008.

[6] Chen Tian. Netfpga windows driver suite installation.

[7] Ajay Tirumala, Les Cottrell, and Tom Dunigan. Measuring end-to-end bandwidth with iperf using web100. In *Proc. of Passive and Active Measurement Workshop*, page 2003, 2003.