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.

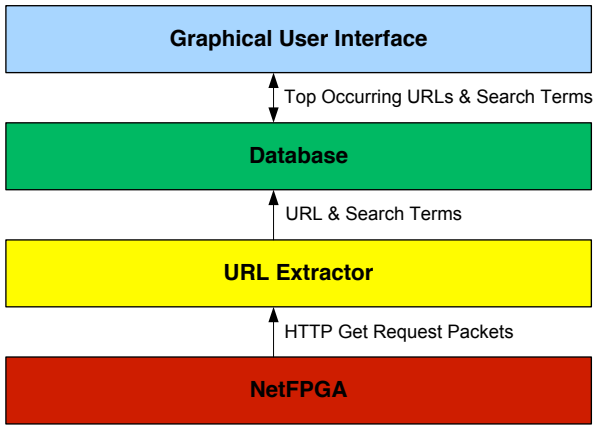

Fig. 1. System Diagram

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

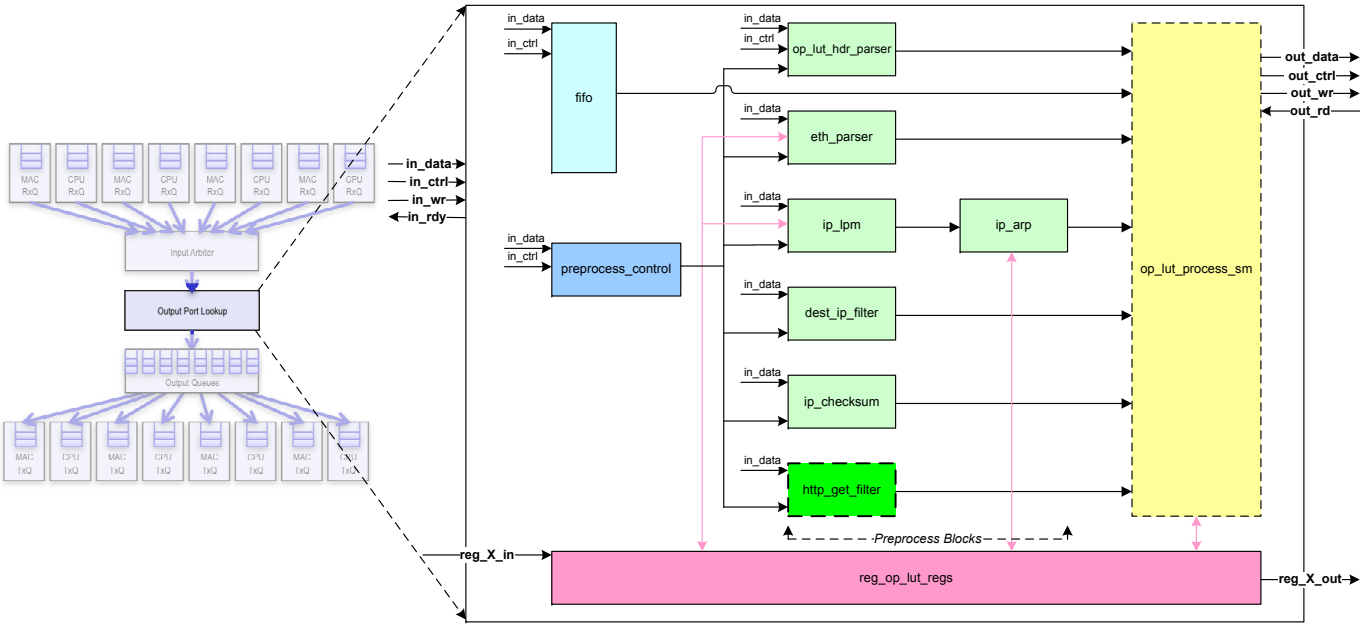

Fig. 2. Submodule layout of the modified Output Port Lookup. The *http_get_filter* is a new submodule and the *op_lut_process_sm* has been altered.

	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 implementations 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

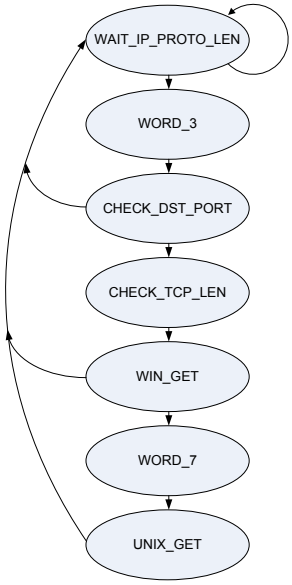

Fig. 4. Diagram of State Machine Used to Identify GET Packets

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.

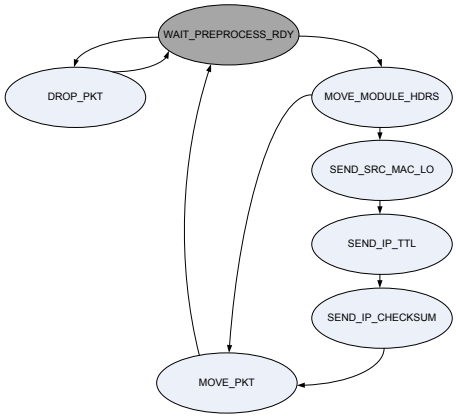

Fig. 5. State diagram of the *op_lut_process_sm* submodule. Code in the *WAIT_PREPROCESS_RDY* state has been altered.

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 are identical as the filtering level does not affect the 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 represents dropped packets at the output queue.

The input rate into the software router 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.

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. 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].

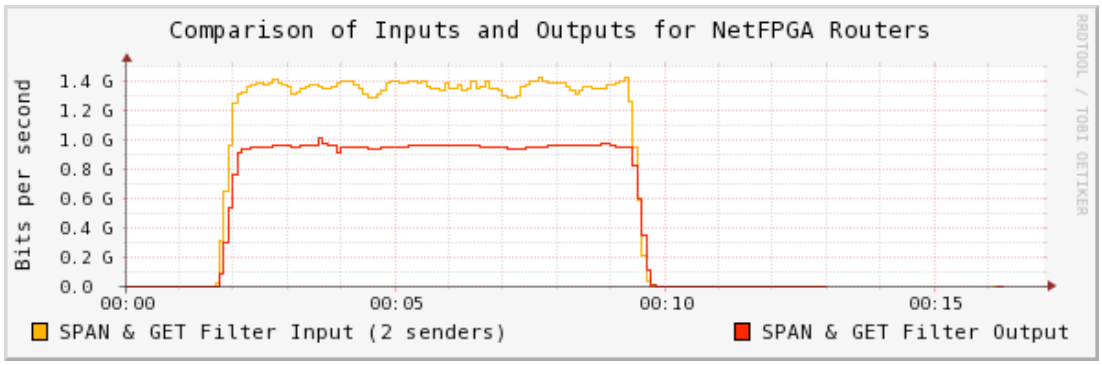

Fig. 7. Input and Output Throughputs for NetFPGA Routers

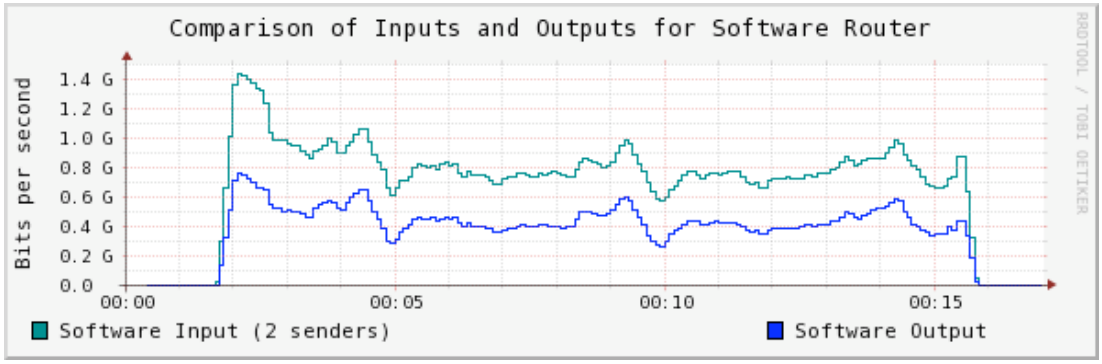

Fig. 8. Input and Output Throughput for Software Router

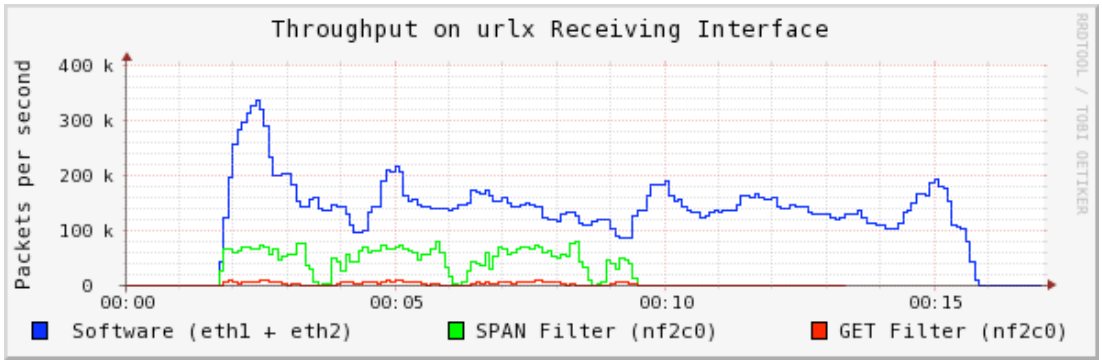

Fig. 9. Throughput on urlx Receiving Interface(s)

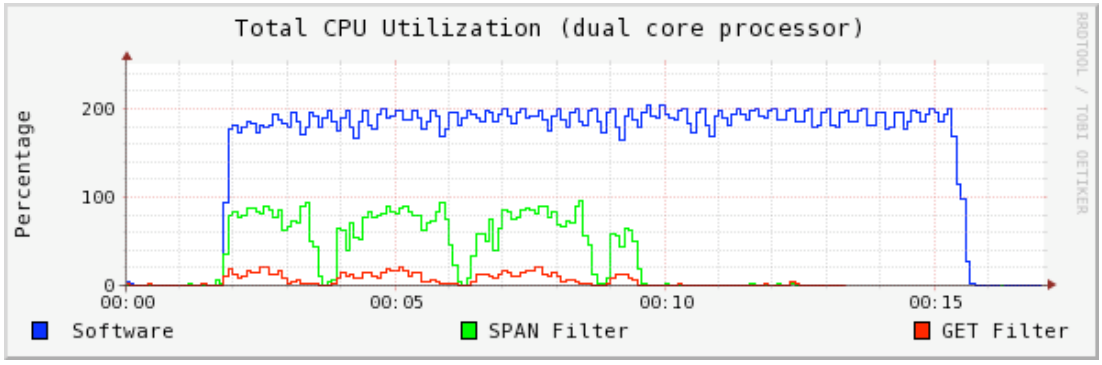

Fig. 10. Total CPU Utilization for Dual Core Processor

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.