A Remotely Accessible Network Processor-Based Router for Network Experimentation

Charlie Wiseman, Jonathan Turner, Michela Becchi, Patrick Crowley, John DeHart, Mart Haitjema, Shakir James, Fred Kuhns, Jing Lu, Jyoti Parwatikar, Ritun Patney, Michael Wilson, Ken Wong, David Zar

Department of Computer Science and Engineering
Washington University in St. Louis
{cgw1,jst,mbecchi,crowley,jdd,mah5,scj1,fredk,jl1,jp,ritun,mlw2,kenw,dzar}@arl.wustl.edu

ABSTRACT

Over the last decade, programmable Network Processors (NPs) have become widely used in Internet routers and other network components. NPs enable rapid development of complex packet processing functions as well as rapid response to changing requirements. In the network research community, the use of NPs has been limited by the challenges associated with learning to program these devices and with using them for substantial research projects. This paper reports on an extension to the Open Network Laboratory testbed that seeks to reduce these "barriers to entry" by providing a complete and highly configurable NP-based router that users can access remotely and use for network experiments. The base router includes support for IP route lookup and general packet filtering, as well as a flexible queueing subsystem and extensive support for performance monitoring. In addition, it provides a *plugin* environment that can be used to extend the router's functionality, enabling users to carry out significant network experiments with a relatively modest investment of time and effort. This paper describes our NP router and explains how it can be used. We provide several examples of network experiments that have been implemented using the plugin environment, and provide some baseline performance data to characterize the overall system performance. We also report that these routers have already been used for ten non-trivial projects in an advanced architecture course where most of the students had no prior experience using NPs.

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*network communications*

General Terms

Design, Experimentation

ANCS'08, November 6–7, 2008, San Jose, CA, USA.

Keywords

Programmable routers, network testbeds, network processors

1. INTRODUCTION

Multi-core Network Processors (NPs) have emerged as a core technology for modern network components. This has been driven primarily by the industry's need for more flexible implementation technologies that are capable of supporting complex packet processing functions such as packet classification, deep packet inspection, virtual networking and traffic engineering. Equally important is the need for systems that can be extended during their deployment cycle to meet new service requirements. NPs are also becoming an important tool for networking researchers interested in creating innovative network architectures and services. They also make it possible for researchers to create experimental systems that can be evaluated with Internet-scale traffic volumes. It is likely that NP-based processing elements will play a significant role in NSF's planned GENI initiative [6][15], providing researchers with expanded opportunities to make use of NPs in their experimental networks.

Unfortunately, NPs pose significant challenges to research users. While network equipment vendors can afford to invest significant time and effort into developing NP software, it is more difficult for academic researchers to develop and maintain the necessary expertise. There are several reasons that using NPs is challenging. First, it can be difficult to obtain NP-based products, since manufacturers of those products sell primarily to equipment vendors and not to end users. Second, developing software for NPs is challenging because it requires programming for parallel execution, something most networking researchers have limited experience with, and because it requires taking advantage of hardware features that are unfamiliar and somewhat idiosyncratic. Third, there is no established base of existing software on which to build, forcing researchers to largely start from scratch.

To address these challenges we have designed, implemented, and deployed a gigabit programmable router based on Intel's IXP 2800 network processor that is easy to use and can be easily extended through the addition of software *plugins.* Our *Network Processor-based Router* (NPR) takes care of standard router tasks such as route lookup, packet classification, queue management and traffic monitoring which enables users to focus on new network architectures and ser-

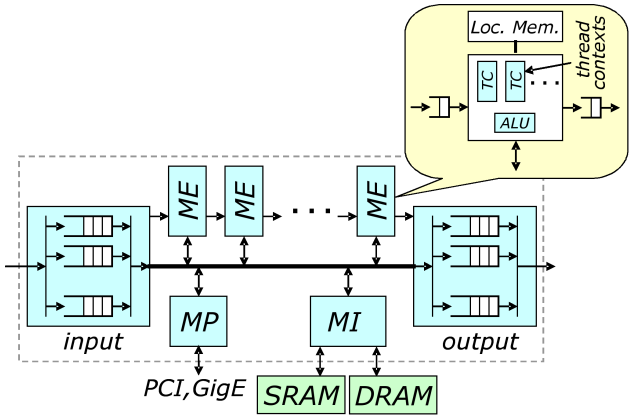

Figure 1: Basic block diagram representation of the IXP 2800 [16].

vices without the need to develop an entire router from the ground up. In addition, we provide a plugin environment with significant processing and memory resources, and an API that implements common functions needed by most users. This makes it possible for users to develop plugins for non-trivial network experiments without having to write a lot of new code. Fourteen of these routers are currently deployed in the Open Network Laboratory (ONL) testbed [5]. The ONL is an Internet-accessible testbed which makes it possible for anyone to use these resources without having to acquire and manage the systems themselves. All of the source code for the plugin API and example plugins, as well as source code for most of the main router pipeline[1] is available through the ONL website [14].

The rest of the paper is organized as follows. Section 2 gives a detailed description of the NPR, including how it fits into our larger network testbed. Section 3 discusses the programmable components in the router. Section 4 includes some examples of plugins that have already been developed for our routers. Section 5 provides a brief performance evaluation of the NPR. Section 6 contains related work, and Section 7 concludes the paper with a few closing remarks about our overall experience with the system.

2. NPR OVERVIEW

To understand the design of the NPR, it is important to know a few things about the architecture of the Intel IXP 2800 [8]. The IXP 2800, like most network processors, is designed specifically for rapid development of high performance networking applications. As such it has some unusual architectural features relative to general purpose processors which heavily influences how it is used.

Figure 1 is a block diagram showing the major components of the IXP. To enable high performance, while still providing application flexibility, there are 16 multi-threaded MicroEngine (ME) cores which are responsible for the majority of the packet processing in the system. Each ME has eight hardware thread contexts (i.e., eight distinct sets of registers). Only a single thread is active at any given time on one ME, but context switching takes a mere 2-3 clock cycles (about 1.5-2 ns). These threads are the mechanism by which IXP applications deal with the memory latency gap. Indeed, there are no caches in the IXP because caches are not particularly effective for networking applications that exhibit poor locality-of-reference. There is no thread preemption, so a cooperative programming model must be used. Typically, threads pass control from one to the next in a simple round-robin fashion. This is accomplished using hardware signals, with threads commonly yielding control of the processor whenever they need to access memory that is not local to the ME. This round-robin style of packet processing also provides a simple way to ensure that packets are forwarded in the same order they are received. Finally, each ME has a small hardware FIFO connecting it to one other ME, which enables a fast pipelined application structure. These FIFOs are known as next neighbor rings.

The IXP 2800 also comes equipped with 3 DRAM channels and 4 SRAM channels. There is an additional small segment of data memory local to each ME along with a dedicated program store with a capacity of 8K instructions. A small, shared on-chip scratchpad memory is also available. Generally, DRAM is used only for packet buffers. SRAM contains packet meta-data, ring buffers for inter-block communication, and large system tables. In our systems, one of the SRAM channels also supports a TCAM which we use for IP route lookup and packet classification. The scratchpad memory is used for smaller ring buffers and tables.

Finally, there is an (ARM-based) XScale Management Processor, labeled MP in Figure 1, that is used for overall system control and any tasks which are not handled by the rest of the data path. The XScale can run a general-purpose operating system like Linux or a real-time operating system like VxWorks. Libraries exist which provide applications on the XScale direct access to the entire system, including all memory and the MEs. With this background, we now proceed to a detailed description of the NPR.

2.1 Data Plane

The software organization and data flow for the NPR are shown in Figure 2. Note the allocation of MEs to different software components. The main data flow proceeds in a pipelined fashion starting with the *Receive block* (Rx) and ending with the *Transmit block* (Tx). Packets received from the external links are transferred into DRAM packet buffers by Rx. Rx allocates a new buffer for each incoming packet and passes a pointer to that packet to the next block in the pipeline, along with some packet meta-data. In order to best overlap computation with high latency DRAM operations, Rx processing is broken up into two stages with each stage placed on a separate ME. Packets flow from the first stage to the second over the next neighbor ring between the MEs.

Note that, in general, the information passed between blocks in the diagram consists of packet references and selected pieces of header information, not packet data. Thus, packets are not copied as they move (logically) from block to block. Each block can access the packet in DRAM using its buffer pointer, but since DRAM accesses are relatively expensive, the system attempts to minimize such accesses. Also note that most blocks operate with all eight threads running in the standard round-robin fashion.

The *Multiplexer block* (Mux) serves two purposes. Each packet buffer in DRAM has an associated 32B entry in SRAM which stores some commonly needed information about the packet, such as the packet length. Mux initial-

[1]There is a very small amount of proprietary code from our NP vendor.

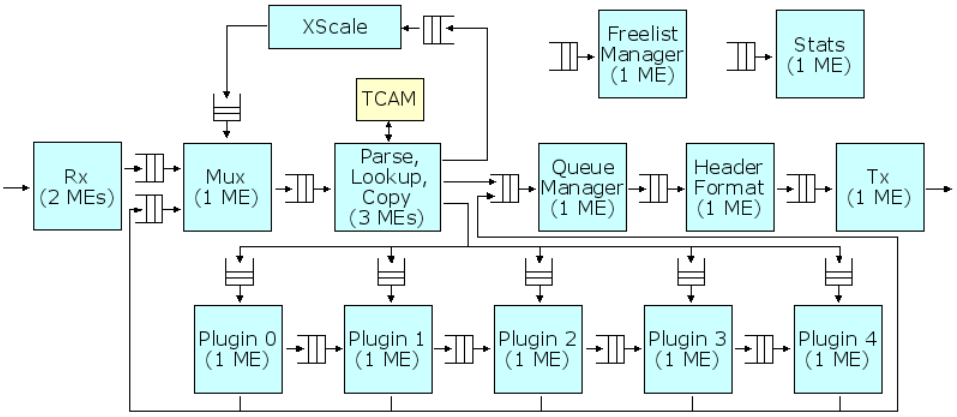

Figure 2: Data plane of the NPR.

izes this meta-data for packets coming from Rx. Its second function is to multiplex packets coming from blocks other than Rx back into the main pipeline. This includes packets coming from the XScale and packets coming from plugins. A simple user-configurable priority is used to determine how to process packets from the different sources.

The *Parse, Lookup, and Copy block* (PLC) is the heart of the router. Here, packet headers are inspected to form a lookup key which is used to find matching routes or filters in the TCAM. Routes and filters are added by the user, and will be discussed further in Section 3.2. Based on the result of the TCAM lookup, PLC takes one of five actions. First, the packet can be sent out towards the external links via the *Queue Manager block* (QM). Second, the packet can be sent to the XScale if it has some special processing needs not handled by the ME pipeline. Third, the packet can be sent to a plugin ME (hosting user code). Plugins will be discussed fully in Section 3.1, but, as can be seen in Figure 2, plugins will be able to forward packets to many other blocks including the QM, Mux, and other plugins. Fourth, the packet can be dropped. Finally, multiple references to the same packet can be generated and sent to distinct destinations. For example, one reference may be sent to a plugin and another directly to the QM. In fact, this mechanism allows the base router to support IP multicast (among other things). Reference counts are kept with the packet meta-data in SRAM to ensure that the packet resources are not reclaimed prematurely. Note that this does not allow different "copies" to have different packet data because there is never more than one actual copy of the packet in the system.

Before moving on, it is worth discussing the design choices made for PLC. Three MEs all run the entire PLC code block with all 24 threads operating in a round-robin fashion. One alternative would be to break up the processing such that Parse is implemented on one ME, Lookup on a second, and Copy on a third. Our experience shows that the integrated approach chosen here yields higher performance. This is primarily due to the nature of the operations in PLC. To form the lookup key, Parse alternates between computation and high latency DRAM reads of packet headers, and Lookup spends most of its time waiting on TCAM responses. On the other hand, Copy is computation-bound due to the potentially complex route and filter results which must be interpreted. Combining all three blocks together provides enough computation for each thread to adequately cover the many memory operations.

Continuing down the main router pipeline, the QM places incoming packets into one of 8K per-interface queues. A weighted deficit round robin scheduler (WDRR) is used to pull packets back out of these queues to send down the pipeline. In fact, there is one scheduler for each external interface, servicing all the queues associated with that interface. Each queue has a configurable WDRR quantum and a configurable discard threshold. When the number of bytes in the queue exceeds the discard threshold, newly arriving packets for that queue are dropped. The QM has been carefully designed to get the best possible performance using a single ME. The design uses six threads. One handles all enqueue operations, and each of the remaining five implements the dequeue operations for one outgoing interface. This decouples the two basic tasks and enables the QM to achieve high throughput.

Following the QM is the *Header Format block* (HF) which prepares the outgoing Ethernet header information for each packet. It is responsible for ensuring that multiple copies of a packet (that have potentially different Ethernet addresses) are handled correctly. Finally, the *Transmit block* (Tx) transfers each packet from the DRAM buffer, sends it to the proper external link, and deallocates the buffer.

There are two additional blocks which are used by all the other blocks in the router. First is the *Freelist Manager block* (FM). Whenever a packet anywhere in the system is dropped, or when Tx has transmitted a packet, the packet reference is sent to the FM. The FM then reclaims the resources associated with that packet (the DRAM buffer and the SRAM meta-data) and makes them available for reallocation. The *Statistics block* (Stats) keeps track of various counters through-out the system. There are a total of 64K counters that can be updated as packets progress through the router. Other blocks in the system issue counter updates by writing a single word of data to the Stats ring buffer, which includes the counter to be updated and increment to be added. For example, there are per-port receive and transmit counters which are updated whenever packets are successfully received or transmitted, respectively. There are also counters for each route or filter entry that are updated both before and after matching packets are queued in the QM. This provides a fine-grained view of packet flow

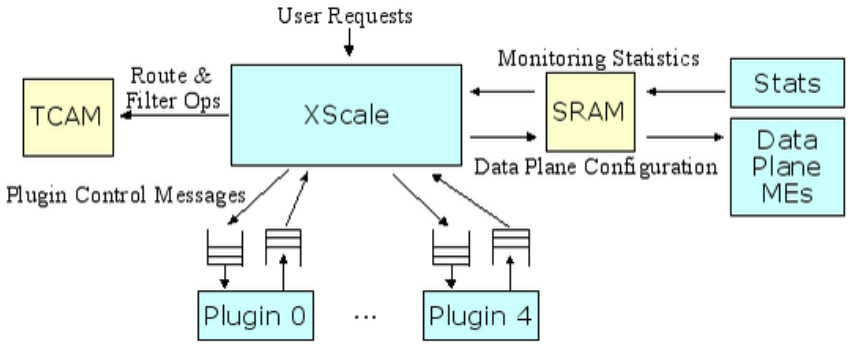

Figure 3: Control plane of the NPR.

in the router. The counters all ultimately reside in SRAM, but there are 192 counters which are also cached locally in the Stats ME. One thread periodically updates the SRAM counterparts to these counters while the other threads all process update requests from the ring buffer.

The final block in the diagram is the XScale. The primary purpose of the XScale is to control the operation of the data plane, as discussed in detail in the next section. However, the XScale also plays a small role in the data plane. Specifically, it handles any IP control packets and other exceptional packets. This includes all ICMP and ARP handling as defined by the standard router RFCs. All messages generated by the XScale are sent back through Mux to PLC, allowing users to add filters to redirect these packets to plugins for special processing, should they desire to do so.

2.2 Control Plane

The XScale's main function is to control the entire router and act as the intermediary between the user and the data plane. This is accomplished by a user-space control daemon running under Linux on the XScale. Libraries provided by Intel are used to manage the MEs and the memory in the router, and a library provided by the TCAM vendor contains an API for interacting with the TCAM. Figure 3 summarizes the most important roles of the XScale daemon. Messages come from the user to request certain changes to the system configuration. Typically the first such request is to start the router which involves loading all the base router code (i.e., everything except plugins) onto the MEs and enabling the threads to run. Once the data path has been loaded successfully, there are several types of control operations that can be invoked.

The first of these involves configuration of routes and filters, which are discussed in detail in Section 3.2. The XScale also supports run-time configuration of some data plane blocks, by writing to pre-defined control segments in SRAM. For example, queue thresholds and quanta used by the QM can be dynamically modified by the user in this way.

The XScale also provides mechanisms to monitor the state of the router. All of the counters in the system are kept in SRAM, which allows the XScale to simply read the memory where a certain counter is stored to obtain the current value. These values can be sampled periodically in order to drive real-time displays of packet rates, queue lengths, drop counters, etc. The system can support dozens of these ongoing requests concurrently.

All plugin operations are also handled by the XScale. In particular, this involves adding and removing user-developed plugin code on the plugin MEs and passing control messages from the user to the plugins after they are loaded. The control messages are forwarded by the XScale through per-plugin rings, and replies are returned in a similar way. The XScale is oblivious to the content of these control messages, allowing users to define whatever format and content is most appropriate for their applications. Plugins can also effect changes to the router by sending requests through these rings to the XScale. Once a plugin is loaded, users can add filters to direct specific packet flows to the plugin.

2.3 Testbed

The NPR has been deployed in the ONL testbed. Researchers can apply for accounts to gain Internet access to the testbed so that they can use the testbed resources to run their experiments (at no cost, of course). The general layout of the testbed is shown in Figure 4.

The testbed itself consists of standard Linux PCs along with the NPRs. All of the data interfaces on these components are connected directly to a gigabit configuration switch, and thus indirectly to all other components. The hosts all have separate control interfaces, and the XScales serve as the control interface for the NPRs as they have their own network interface which is separate from the data interfaces. Everything is managed by our testbed controller which runs on another Linux PC. Users configure network topologies with the Remote Laboratory Interface (RLI) on their local computer. The RLI also provides an intuitive way to control and monitor the individual routers and hosts in the topology. An example session is shown in the lower left of the figure.

Once connected to the testbed controller the RLI sends the topology to the controller so that hardware resources can be allocated to the user, and the chosen components can be connected together with VLANs in the configuration switch. Hardware resources are reserved ahead of time either through the RLI or on the ONL website so that when a user is ready to run their experiment enough resources are guaranteed to be available. All assigned resources are given entirely to the user so that there can be no interference from other concurrent experiments. Users are also granted SSH access to hosts in their topology so that they can log in to start any traffic sources and sinks.

The ONL currently has 14 NPRs and over 100 hosts as well as some additional routers and other networking components. The routers themselves are built on ATCA tech-

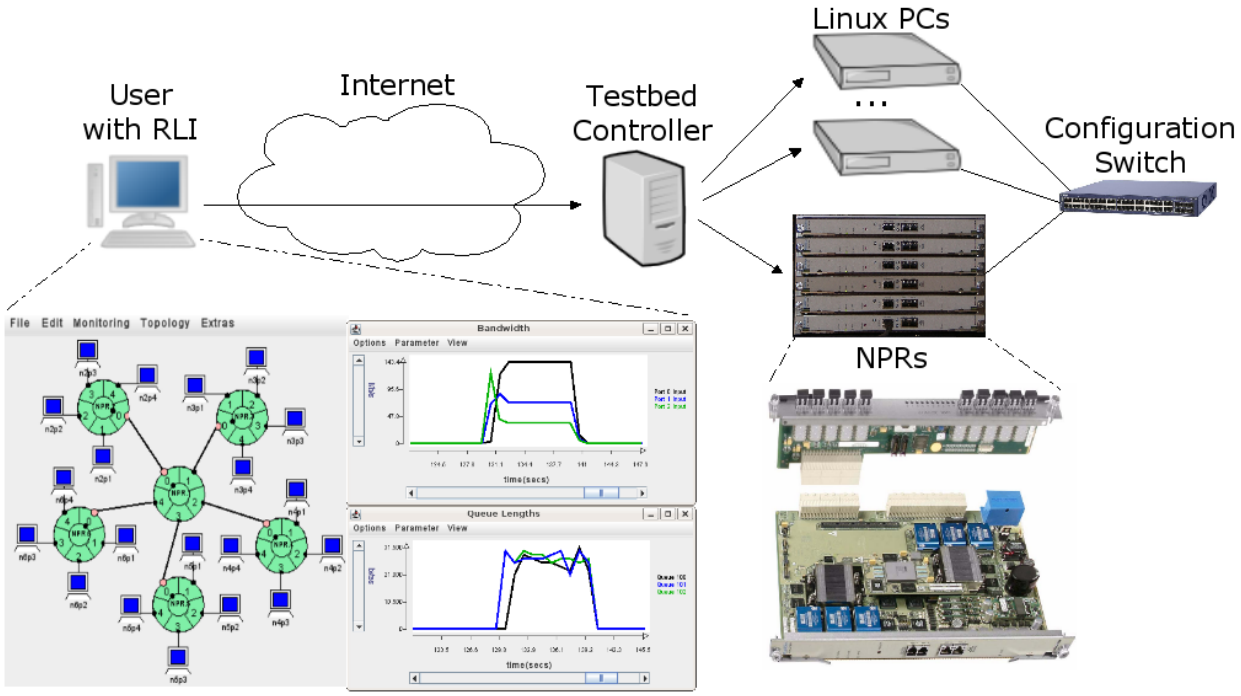

Figure 4: A high-level view of our Internet-accessible testbed.

nology, which is an industry standard for networking components that has broad industry support. ATCA technology is also proving to be a boon to networking research, as it enables the assembly of powerful, yet highly flexible experimental networking platforms. In particular, the NPR is built on Radisys ATCA-7010 boards [11], shown in the bottom right of Figure 4. Each board has two IXP 2800s, a shared TCAM, and ten 1 gigabit data interfaces. In our context, we use the two IXPs as separate five port routers, assigning five of the data interfaces to each IXP.

3. PROGRAMMABILITY

Now that the basics of the NPR have been covered, we turn our attention to the progammable components of the router. There are two primary facets of the overall programmability of the NPR: plugins and filters. Together they provide users a rich selection of options with which to customize processing and packet flow in the NPR.

3.1 Plugin Framework

Recall from Figure 2 that five MEs are used to host plugins. NPR users are free to load any combination of code blocks onto these MEs. In addition to the five MEs there are five ring buffers leading from PLC to the plugins which can be used in any combination with the MEs. For example, each plugin may pull packets from a separate ring (this is the default behavior) or any subset of plugins may pull from the same ring. The ring buffers, then, are simply a level of indirection that adds versatility to potential plugin architectures. The NPR also sets aside 4KB of the scratchpad memory and 5MB of SRAM exclusively for plugin use.

The actual processing done by any particular plugin is entirely up to the plugin developer. Plugins are written in MicroEngine C which is the standard C-like language provided by Intel for use on MEs. The most important differences between MicroEngine C and ANSI C are dictated by the IXP architecture. First, there is no dynamic memory allocation or use because there is no OS or other entity to manage the memory. Second, all program variables and tables must be explicitly declared to reside in a particular type of memory (registers, ME local memory, scratchpad, SRAM, DRAM) as there is no caching. Finally, there is no stack and hence no recursion. Also recall from the IXP review that the eight hardware contexts share control explicitly (no preemption).

To help users who are unfamiliar with this programming environment we have developed a framework that lowers the entry barrier for writing simple to moderately complex plugins. Our framework consists of a basic plugin structure that handles tasks common to most plugins and a plugin API that provides many functions that are useful for packet processing.

In the basic plugin structure there are three different types of tasks to which the eight threads are statically assigned at plugin compile time. The first of these tasks deals with packet handling. The framework takes care of pulling packets from the incoming ring buffer and then calls a user supplied function to do the actual plugin processing. When the function returns, the packet is pushed into the outgoing ring. As can be seen in Figure 2, the packet can be sent back to MUX which results in the packet being matched against routes and filters in the TCAM a second time. This is useful if something in the packet, such as the destination IP address, has changed and the packet might need to be re-routed. Alternatively, the plugin can send the packet directly to the QM so that it will be sent out to the external links. Packets can also be redirected to the next plugin ME via the next neighbor rings. In fact, although it is not shown in the figure to avoid confusion, plugins even have the ability

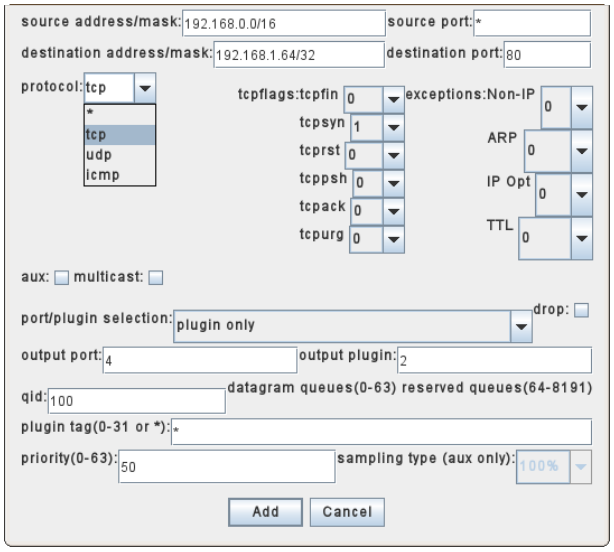

Figure 5: Adding a filter in the RLI.

```
api_pkt_ref_t packetRef; // packet reference
api_meta_data_t metaData; // local copy of meta-data
unsigned int ipHdrPtr; // pointer to IP header (DRAM)
api_ip_hdr_t ipHdr;    // local copy of IP header

api_get_pkt_ref(&packetRef);
api_read_meta_data(packetRef, &metaData);

ipHdrPtr = api_get_ip_hdr_ptr(packetRef,
                              metaData.offset);
api_read_ip_hdr(ipHdrPtr, &ipHdr);

switch(ipHdr.ip_proto)
{
  case PROTO_ICMP: api_increment_counter(0); break;
  case PROTO_TCP : api_increment_counter(1); break;
  case PROTO_UDP : api_increment_counter(2); break;
  default        : api_increment_counter(3); break;
}
```

Figure 6: Code for the Network Statistics Plugin.

to send packets to any other plugin ME by writing directly to the five ring buffers leading from PLC to the plugins.

The second type of task is the periodic task. Some plugins may need to do processing that is not dictated purely by packet arrivals. In such cases, plugin developers can assign threads to the periodic task which has the thread sleep for a configurable time and then call another user provided function to do the periodic processing. The last type of task is the control task, first mentioned in Section 2.2. This provides a mechanism for plugins to receive and respond to control messages from the RLI. As an example, we have a plugin which delays packets by N ms, where N can be set via a control message. These messages and their responses go through per-plugin control ring buffers as shown in Figure 3. These rings are also used for plugins to request modifications to the system outside of the plugin MEs. Such requests are processed by the XScale and allow plugins to modify queue parameters, add or remove routes and filters, and even add or remove plugin code from other plugin MEs.

To support plugin developers, we provide a plugin API. The API consists of helper functions for common packet processing steps as well as functions that hide some of the complexity of interacting with packets and packet meta-data.

3.2 Filters

In order to actually get packets to plugins, filters are installed to direct packet flows to specific destinations in the router. More generally, filters are used to modify default behavior of the router by superseding standard IP routing. As mentioned in Section 2.1, filters and routes are stored in the TCAM. Routes are simpler and used only for standard IP routing. That is, the packet's destination IP address is compared to the route database and the longest matching address prefix is returned. The result also contains the external interface that the packet should be forwarded on. Each NPR supports 16K routes.

Filters are more general than routes and include more fields in the lookup key and more options for the action to be taken. Figure 5 shows a screenshot of the dialog box provided by the RLI for adding filters. In general, the fields in the top half of the window constitute the lookup key and those in the bottom half describe what should happen to any matching packets. Each key field can be specified as a particular value or as "*" which means that any value for that field should match. In addition, IP address values can include ranges of addresses in the CIDR notation. An expanded version of the standard IP 5-tuple forms the core of the lookup key, including source and destination IP address ranges, source and destination transport protocol ports, and the IP protocol. For TCP packets, the TCP state flags are also part of the key allowing filters to match particular parts of TCP flows. The *plugin tag* is a 5 bit field that plugins can modify to force different matches to occur on any subsequent passes the packet takes through PLC. Finally, there are four *exception* bits which allow exceptional traffic, such as packets with a Time-to-Live of 0, to be matched and potentially handled in some way other than the default. In this particular example, the beginning of any HTTP flow from hosts in the 192.168.0.0/16 subnet going to 192.168.1.64 will be matched. Note that the TCP flags indicate only TCP SYN packets (and not SYN-ACKs) will match.

The rest of the fields determine exactly what happens to packets that match the filter. The most important fields are *port/plugin selection*, *output port*, and *output plugin* because they determine whether or not the matching packets will go directly to the QM or to a plugin. In the figure, the filter is configured to send the packets to plugin 2. The *output port* is part of the data that is passed to the plugin as well, so that the plugin knows where to send the packet next (if it decides to forward the packet). Note that the *output plugin* actually refers to the ring buffer leading by default to that plugin ME, but any plugin is capable of reading from any of the five rings, so plugin developers are free to configure plugins to process packets from the ring buffers in other ways as well. The *multicast* field can be set if multiple copies of the packet are desired. In that case, any combination of ports and plugins can be set in the *output port* and *output plugin* fields, and copies will be sent by PLC to each of the specified destinations. The *qid* determines which of the 8K per-interface queues the packet enters when it reaches the QM. In the event of multiple copies, each copy will go into the same numbered queue for whichever interface it is

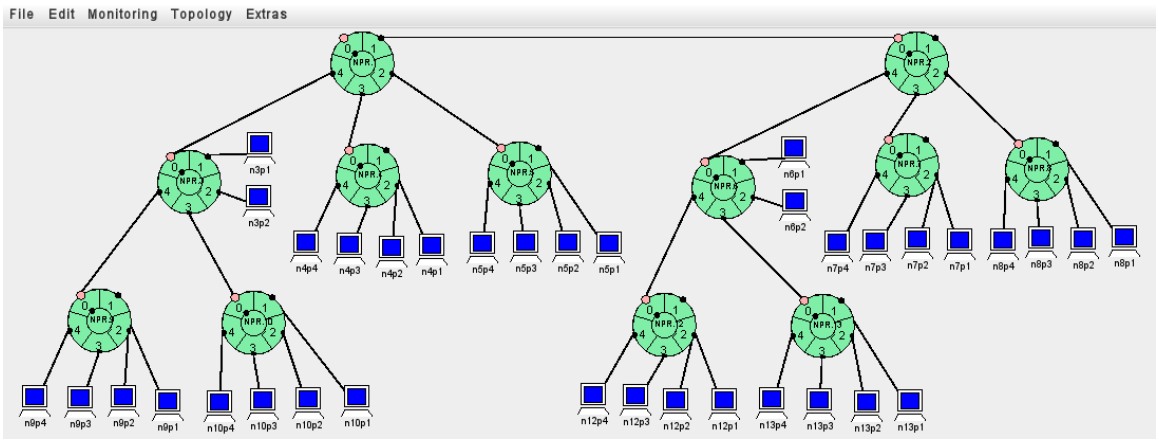

Figure 7: Example topology for a distributed game application.

destined. Users can also specify that all matching packets should be dropped by selecting the *drop* field.

There are actually two different filter types in the NPR, differentiated by the selection of the *aux* field. If the field is not set we call the filter a *primary* filter, and if it is, an *auxiliary* filter. The lookup key fields and actions all have the same meaning for either type, but auxiliary filters cause an additional reference to the matching packet to be created by PLC and sent to the destination contained in the auxiliary filter. This means that auxiliary filters represent a separate set of potential matches. On the other hand, primary filters are logically in the same set of potential matches as routes. This is where the *priority* field in the filter comes into play. All routes are assigned the same priority while each filter has its own priority. When a packet matches multiple primary filters, the highest priority filter is considered to be the matching one, unless the route priority is higher. In that case, the matching route is used to determine how the packet is forwarded. For auxiliary filters, the highest priority auxiliary filter is considered to be a match. Although matching packets against filters with priorities can potentially be fairly complex, the TCAM allows us to lay out all routes and filters in such a way that higher priority entries (and longer prefixes for routes) come first in the TCAM tables and are thus the first match returned when the TCAM is queried. The end result is that a single packet can match one primary filter or route, and one auxiliary filter. This can be quite useful if the user wishes to have passive plugins that monitor certain packet streams without disturbing the normal packet flow. Each NPR supports 32K primary filters and 16K auxiliary filters.

4. EXAMPLE PLUGINS

To provide a more concrete view of the capabilities of the NPR plugin environment, we now describe three example plugins that have been written and tested in the router.

4.1 Network Statistics

The first example is a simple plugin which keeps a count of how many packets have arrived with different IP protocols. The code written by the developer is shown in Figure 6 and consists mostly of API calls to read the IPv4 header from the packet in DRAM. First, the packet reference is filled in from the input ring data by calling `api_get_pkt_ref()`. The packet itself resides in a 2KB DRAM buffer with the beginning of the packet header at some offset into that buffer (to accommodate packets that may increase in size). The offset is part of the meta-data for that packet, so the second step is to read the meta-data using `api_read_meta_data()`. Once we have the offset, `api_get_ip_hdr_ptr()` is called to calculate the address of the IP header. `api_read_ip_hdr()` reads the header into a local struct which grants easy access to the header fields. Finally, based on the IP protocol one of four different plugin-specific counters is incremented with `api_increment_counter()`. These counters can be monitored easily in the RLI so that the user can see, in real time, how many packets of each type are passing through the plugin. The plugin does not explicitly decide where packets should go next and so by default all packets will be sent to the QM after leaving the plugin. This plugin has no need for periodic tasks or for control messages from the RLI so all eight threads are devoted to handling packets as they arrive.

4.2 Network Support for Distributed Games

Our next example is a system that provides network services in support of highly interactive distributed games. The network provides support for a distributed collection of game servers that must share state information describing the current status of various objects in the game world (e.g., user avatars, missiles, health packs, etc). State update packets are distributed using a form of *overlay multicast*. These updates are labeled with the game world *region* where the associated object is located and each region is associated with a separate multicast data stream. Servers can *subscribe* to different regions that are of interest to them, allowing them to control which state updates they receive.

Figure 7 shows an example ONL session for the distributed game application. This configuration uses 12 routers and 36 end systems acting as game servers, each supporting up to 10 players. The routers in this system host two distinct plugins that implement the region-based multicast. While it would have been possible to implement this application using the built-in IP multicast support, the use of application-level multicast frees the system from constraints on the availability of IP addresses, and enables a very lightweight protocol for updating subscriptions, making the system very responsive to player activity.

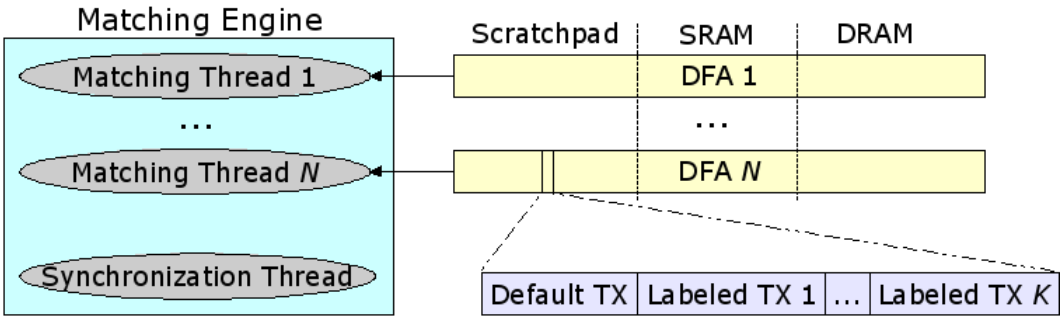

Figure 8: Design of the Regular Expression Matching Engine.

The first of our two plugins is a *Multicast Forwarder* that forwards the state update packets sent by the game servers, and the second is a *Subscription Processor*. The subscription state for each region in the game world is a bit vector specifying the subset of the five outgoing links that packets with a given region label should be forwarded to. These bit vectors are stored as a one-dimensional array in SRAM, which is shared by the two plugins. One megabyte of SRAM has been allocated to these multicast bit vectors, allowing up to one million regions in the game world.

The subscription processor runs on one ME and handles subscription messages from all router interfaces. A TCAM filter is configured for each of the router's input ports, directing all subscription packets to the subscription processor's input ring. Each of these messages contains one or more records with each record containing a join or a leave request for one region. The subscription processor reads each of these records from the packet in DRAM and updates the corresponding state in the subscription bit vector. All eight threads are used to process subscription packets.

Four of the MEs are used to host multicast forwarders. These MEs share a single input ring and process different packets in parallel. Altogether, 32 distinct packets can be processed concurrently (using the eight hardware thread contexts in each ME). A TCAM filter is configured for every port, to direct all state update traffic to the shared input ring. To process a packet, the multicast forwarder reads the packet header to determine which multicast stream the packet belongs to and then reads the subscription bit vector for that region to determine which ports the packet should be forwarded to. It then replicates the packet reference information as needed, and forwards these packet references to the appropriate queues in the QM. Note that the packets themselves are never copied.

4.3 Regular Expression Matching

The last and most complex example is a set of plugins that are used for high speed regular expression matching of packet data using Deterministic Finite Automata (DFAs). These could be used for any application based on deep packet inspection, such as network intrusion detection and content-based routing. The basic operation involves following state transitions in a DFA for every input character received, where one DFA encompasses many regular expressions. This topic has been studied extensively and we take advantage of many state of the art techniques in our plugins.

Regular expression matching is a memory intensive application both in terms of memory and bandwidth. To reduce the space requirements we utilize both default transition compression and alphabet reduction as in [1]. This allows us to fit more regular expressions into the memory available to plugins. Another technique involves pattern partitioning of the set of regular expressions to form multiple DFAs [17]. This can reduce the overall space needs, but increases the memory bandwidth since all of the DFAs have to be traversed for each input character. Fortunately, the hardware threads available on the plugin MEs provide the necessary parallelism to support this requirement.

As in the previous example, there are two types of plugins. The *Packet Loader* prepares packets for the *Matching Engines* which actually run the DFAs. An auxiliary filter directs a copy of every packet to the Packet Loader. This ensures that even if our plugins fall behind in packet processing and potentially drop packets the actual packet stream will remain unaffected. The Packet Loader reads the packet header and passes the packet payload to the first Matching Engine through the next neighbor ring between them.

Up to four Matching Engines can be added to run on the remaining MEs depending on how many threads are needed. Figure 8 shows the design of the Matching Engine. As discussed above, the regular expressions are partitioned to produce a set of DFAs and each DFA is then run exclusively by one thread on one ME. One thread on each Matching Engine is used for synchronization, so up to seven threads can be processing DFAs. There is a one-to-one mapping of matching threads and DFAs, meaning that each matching thread always processes packet data against one particular DFA. Each packet is processed against all DFAs in order to determine if there are any matches.

The synchronization thread has two tasks. The first is to read data from the incoming next neighbor ring so that the local matching threads can access it, as well as passing the data to the next Matching Engine if there is one. The thread's main task is to handle synchronization of packet data across the matching threads. Synchronization is implemented through the use of a circular buffer stored in shared local memory. The synchronization thread writes data to the buffer so that the matching threads can read it. Each matching thread can then proceed through the buffer at its own pace, and the synchronization thread ensures that data that has been processed by all the threads is flushed.

Each matching thread invokes its DFA on the input data one character at a time. Alphabet translation is first performed on the input character (the translation table is stored in local memory). The next state is then found by traversing the DFA, and processing continues with the next character.

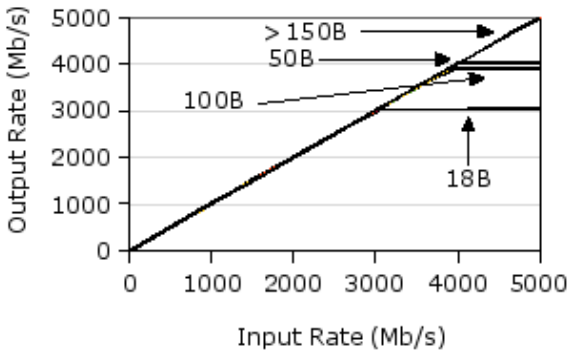

Figure 9: Throughput results for the NPR.

As is shown in Figure 8, the DFAs are stored hierarchically in memory with some state information in the scratchpad memory, some in SRAM, and some in DRAM. The memory layout is generated ahead of time so that states more likely to be reached are stored in faster memory. The figure also shows an example of the actual state information. Recall that we use the default transition method noted above. Each state contains a small number of labeled transitions and a default transition. If none of the labeled transitions are for the current input character, the default transition is taken. The operational details are given in [1].

5. PERFORMANCE

We include here a brief performance evaluation of the NPR as a baseline reference for plugin developers. UDP flows consisting of equally sized packets were directed to the router for aggregate input rates of up to 5 Gb/s, with input and output bandwidth split equally among the five interfaces. Figure 9 shows the forwarding rate when packets proceed directly through the main pipeline with no plugins installed. The different curves represent the output rate for flows of different packet sizes, where the labels indicate the UDP payload size of the packets. Note that the input and output rates are reported relative to the external interfaces. As such, an input rate of 5000 Mb/s means that all five input links are completely saturated.

For packets with UDP payloads larger than 150 bytes the NPR is able to keep up with any input rate. Streams of smaller packets cause the router to drop packets at high input rates. For 100 and 50 byte packets the output rate levels off near 4 Gb/s. It is interesting to note that the output rate is higher in the 50 byte case than in the 100 byte case. This apparent anomaly is a product of the way the Rx and Tx blocks interact with the external interfaces. Packets actually arrive in fixed-size cells which Rx reassembles into full packets. A UDP payload size of 100 bytes forces incoming packets to take two of these cells while 50 byte UDP payloads can fit in a single cell. This means that Rx has significantly more work to do in the former case which results in a lower overall output rate. As packets continue to decrease in size, the peak output rate continues to decrease. For minimum size Ethernet frames (UDP payload of 18 bytes) the output rate is around 3 Gb/s.

We performed the same evaluation with the entire input stream directed through a single *null* plugin. The null plugin uses the basic plugin structure described in Section 3.1 with an empty function to handle packets. The packets are then forwarded to the QM. The results were identical which means that a single plugin is capable of handling every packet that the router can forward.

The NPR was also evaluated with the distributed gaming plugins installed. There is not sufficient space for a full discussion of the results here, so a single benchmark result is given. Recall from Section 4.2 that the multicast forwarder plugins are responsible for generating multiple references to incoming packets based on the current subscriptions for that packet's multicast group. In the NPR, the packet replication factor, or *fan-out*, can be between 0 and 4 as there are 5 interfaces and packets are never sent out on the interface on which they arrived. To test the forwarding capability of the multicast plugins, UDP streams with UDP payloads of 150 bytes (a typical size for incremental game state updates) were directed through the plugins. As with the baseline evaluation, the input and output bandwidth was split equally among the five interfaces. Note that as the fan-out increases the input packet rate needed to produce the same output rate decreases. The peak output rates for fan-outs of 1, 2, 3, and 4 are, respectively, 4 Gb/s, 4.5 Gb/s, 4 Gb/s, and 3.6 Gb/s. To understand this result, consider how the workload changes as the fan-out changes. For low fan-outs the plugins have to process more packets per second, but the per-packet work is lower. On the other hand, when the packet rate is lower the per-packet work is significantly higher. This leads to optimal performance when the fan-out is 2, given the characteristics of the router.

To explore the bottlenecks in our system, we used Intel's cycle-accurate simulation environment. Unsurprisingly, no single block is responsible for not keeping up with line rate for small packets. As mentioned above, Rx has limitations for some size packets. For minimum size packets, both the QM and PLC blocks peak between 3 and 3.5 Gb/s. We could enable the main router path to perform better by using more MEs in those cases, but we believe that their use as plugin MEs is ultimately more beneficial. The traffic conditions which cause the router to perform sub-optimally are also unlikely to occur for extended periods. Under realistic traffic patterns the router is able to keep up with line rate.

We also monitored packet latency during the above evaluation by periodically sending ping packets through the router and recording the round-trip time. When the router is otherwise idle, the average RTT is around 100 μs with a standard deviation of around 2 μs. Note that much of that time is accounted for at the end hosts. The one-way, edge-to-edge latency of the router is no more than 20 μs. As expected due to the IXP architecture, these values do not change significantly with a change in router load. Indeed, even under the heaviest load for minimum-sized packets the average RTT doesn't change and the standard deviation only increases to around 4 μs for packets that are not dropped.

6. RELATED WORK

Router plugin architectures are certainly not a new idea [4]. Indeed, [3] describes a router already in ONL that has a general purpose processor for running plugins on each port of a hardware router. There have also been other extensible routers based on network processors. For example, [13] describes a router that uses the IXP 1200 network processor for fast-path processing in a standard PC, and [16] describes an ATCA-based platform that allows PlanetLab

[10] applications to achieve significantly better performance by incorporating IXP 2850 network processors as fast-path elements and line cards. In the traditional software router space, Click [9] takes a modular approach to extensibility by providing a framework for connecting various forwarding and routing modules together in an arbitrary fashion. XORP [7] is similar in nature to Click, but it is focused on modular routing stack components. Our work is based in part on all of these previous efforts, but aims to fill a different role. The NPR provides users direct access to a significant portion of the router resources, a software base on which to build, and the ability to have router code that performs substantially better than on a standard PC. Moreover, our routers are already deployed in the Open Network Laboratory which allows anyone to experiment with them in a sandboxed environment.

There is another body of related work that aims specifically to ease software development for NPs. One example is NP-Click [12] which provides a programming model that abstracts away many of the architectural aspects that make programming difficult. Shangri-La [2] is another example where a compiler and thin run-time environment are used to support higher-level programming languages. These systems employ many complex optimizations to bring application performance to similar levels achieved by hand-written assembly code. All of the work in this space is complementary to our work. Indeed, the NPR provides direct access to the plugin MEs and so these approaches could be used to produce NPR plugin code.

7. CONCLUSIONS

The NPR has already been used locally in an advanced architecture class focused on multi-core processors. Students were asked to use the NPR for their final projects, and all ten of the projects were completed successfully. The projects ranged from TCP stream reassembly to complex network monitoring to network address translation. We have already incorporated feedback from the class into the NPR and particularly into the plugin API. We are planning on further extending this work in a few ways.

Plugin developers can currently only write code for the MEs which can be somewhat limiting if they wish to support protocols that have complex control processing. For example, the main router pipeline sends ICMP and ARP packets to the XScale because they can require substantial processing and there is no particular reason to handle them in an ME. To support similar fast-path/slow-path structure in plugins, we are adding a mechanism to allow plugin developers to supply XScale counterparts to their ME plugins. More generally, our basic design could be used to build similar plugin-based routers on other architectures. We hope to explore this avenue with other network processors and potentially with many-core general purpose processors.

Based on our experience so far, we believe that the NPR is an excellent platform for network study and experimentation. Our plugin framework provides a flexible structure and API that helps users unfamiliar with similar systems to build expertise without hindering experts. The ONL provides an isolated and safe environment to try out ideas before potentially deploying them in the wild, in systems such as [16] or in GENI. We are currently accepting ONL accounts, and we strongly encourage any researchers interested in developing expertise with these types of systems to apply.

8. REFERENCES

[1] Becchi, M. and P. Crowley. "An Improved Algorithm to Accelerate Regular Expression Evaluation", *Proc. of ACM-IEEE Symposium on Architectures for Networking and Communications Systems*, 12/2007.

[2] Chen, M. K., et. al. "Shangri-La: Achieving High Performance from Compiled Network Applications while Enabling Ease of Programming", *Proc. of ACM SIGPLAN conference on Programming Language Design and Implementation*, 6/2005.

[3] Choi, S., et. al. "Design of a High Performance Dynamically Extensible Router", *Proc. of the DARPA Active Networks Conference and Exposition*, 5/2002.

[4] Decasper, D., Z. Dittia, G. Parulkar, and B. Plattner. "Router Plugins: A Software Architecture for Next Generation Routers", *Proc. of ACM SIGCOMM*, 9/1998.

[5] DeHart, J., F. Kuhns, J. Parwatikar, J. Turner, C. Wiseman, and K. Wong. "The Open Network Laboratory", *Proc. of ACM SIGCSE*, 3/2006.

[6] Global Environment for Network Innovations. http://www.geni.net.

[7] Handley, M., E. Kohler, A. Ghosh, O. Hodson, and P. Radoslavov. "Designing Extensible IP Router Software", *Proc. of the Second Symposium on Networked Systems Design and Implementation*, 5/2005.

[8] Intel IXP 2xxx Product Line of Network Processors. http://www.intel.com/ design/network/products/npfamily/ixp2xxx.htm.

[9] Kohler, E., R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. "The Click Modular Router", *ACM Transactions on Computer Systems*, 8/2000.

[10] Peterson, L., T. Anderson, D. Culler and T. Roscoe. "A Blueprint for Introducing Disruptive Technology into the Internet", *Proc. of ACM HotNets-I Workshop*, 10/2002.

[11] Radisys Corporation. "Promentum^{TM} ATCA-7010 Data Sheet", product brief, available at http://www.radisys.com/files/ATCA-7010_07-1283-01_0505_datasheet.pdf.

[12] Shah, N., W. Plishker, and K. Keutzer. "NP-Click: A Programming Model for the Intel IXP 1200", *Proc. of Second Workshop on Network Processors*, 2/2003.

[13] Spalink, T., S. Karlin, L. Peterson, and Y. Gottlieb. "Building a Robust Software-Based Router Using Network Processors", *Proc. of ACM Symposium on Operating System Principles*, 10/2001.

[14] The Open Network Laboratory. http://onl.wustl.edu.

[15] Turner, J. "A Proposed Architecture for the GENI Backbone Platform", *Proc. of ACM-IEEE Symposium on Architectures for Networking and Communications Systems*, 12/2006.

[16] Turner, J., et. al. "Supercharging PlanetLab a High Performance, Multi-Application, Overlay Network Platform", *Proc. of ACM SIGCOMM*, 8/2007.

[17] Yu, F., Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz. "Fast and Memory-Efficient Regular Expression Matching for Deep Packet Inspection", *Proc. of ACM-IEEE Symposium on Architectures for Networking and Communications Systems*, 12/2006.