Report Number: wucse-2009-78

2009

Supercharged PlanetLab Platform Architecture

Jonathan Turner, Patrick Crowley, John DeHart, Mart Haitjema, Fred Kuhns Kuhns, Ritun Patney, Michael Wilson, Charlie Wiseman, and David Zar

This report describes the Supercharged Planetlab Platform (SPP), a system designed as a prototype of an internet-scale overlay hosting platform. Overlay networks have become an important vehicle for delivering Internet applications. Overlay network nodes are typically implemented using general purpose servers or clusters. The SPP offers a more integrated architecture, combining general-purpose servers with high performance Network Processor (NP) subsystems. SPP nodes have recently been deployed as part of the Global Environment for Network Innovation (GENI) and are available for use by research users.
... Read complete abstract on page 2.

Supercharged PlanetLab Platform Architecture

Jonathan Turner, Patrick Crowley, John DeHart, Mart Haitjema, Fred Kuhns Kuhns, Ritun Patney, Michael Wilson, Charlie Wiseman, and David Zar

Complete Abstract:

This report describes the Supercharged Planetlab Platform (SPP), a system designed as a prototype of an internet-scale overlay hosting platform. Overlay networks have become an important vehicle for delivering Internet applications. Overlay network nodes are typically implemented using general purpose servers or clusters. The SPP offers a more integrated architecture, combining general-purpose servers with high performance Network Processor (NP) subsystems. SPP nodes have recently been deployed as part of the Global Environment for Network Innovation (GENI) and are available for use by research users.

2009-78

Supercharged PlanetLab Platform Architecture

Authors: Jon Turner, Patrick Crowley, John DeHart, Mart Haitjema,
Fred Kuhns, Ritun Patney, Michael Wilson, Charlie Wiseman, David Zar

Corresponding Author: jon.turner@wustl.edu

Web Page: http://wiki.arl.wustl.edu/index.php/Internet_Scale_Overlay_Hosting

Abstract: This report describes the Supercharged Planetlab Platform (SPP), a system designed as a prototype of an internet-scale overlay hosting platform. Overlay networks have become an important vehicle for delivering Internet applications. Overlay network nodes are typically implemented using general purpose servers or clusters. The SPP offers a more integrated architecture, combining general-purpose servers with high performance Network Processor (NP) subsystems. SPP nodes have recently been deployed as part of the Global Environment for Network Innovation (GENI) and are available for use by research users.

Type of Report: Other

Supercharged PlanetLab Platform Architecture

Jon Turner, Patrick Crowley, John DeHart, Mart Haitjema, Fred Kuhns,
Ritun Patney, Michael Wilson, Charlie Wiseman, David Zar
Washington University, St. Louis, MO

Abstract

This report describes the *Supercharged Planetlab Platform* (SPP), a system designed as a prototype of an internet-scale overlay hosting platform. Overlay networks have become an important vehicle for delivering Internet applications. Overlay network nodes are typically implemented using general purpose servers or clusters. The SPP offers a more integrated architecture, combining general-purpose servers with high performance Network Processor (NP) subsystems. SPP nodes have recently been deployed as part of the Global Environment for Network Innovation (GENI) and are available for use by research users.

Keywords. PlanetLab, overlay networks, network processors, Global Environment for Network Innovation (GENI)

1. INTRODUCTION

Network overlays have become a popular tool for implementing Internet applications. While content-delivery networks provide the most prominent example of the commercial application of overlays [DI02, KO04], systems researchers have developed a variety of experimental overlay applications, demonstrating that the overlay approach can be an effective method for deploying a broad range of innovative systems [BH06, FR04,RH05, ST02]. Rising traffic volumes in overlay networks make the performance of overlay nodes an issue of growing importance. Currently, overlays nodes are constructed using general purpose servers, often organized into a cluster with a load-balancing switch acting as a front end. This report describes the architecture of the *Supercharged Planetlab Platform* (SPP), a research system that explores an alternative approach that combines general purpose server blades and high performance *Network Processor* (NP) subsystems into an integrated architecture designed to support multiple overlay applications. The SPP is designed for scalability and high performance, with the objective of supporting internet-scale overlays, with router-like performance. We plan to deploy five SPPs as part of the National Science Foundation's GENI initiative. The first three systems have recently been deploye, and the remaining two will be deployed by the end of 2010.

The SPP has been designed to operate within the PlanetLab overlay network testbed [CH03, PE02]. Since its inception, PlanetLab has become a popular experimental platform and deployment vehicle for systems researchers in networking and distributed systems. PlanetLab nodes are implemented using conventional PCs, running a modified version of Linux. This provides a familiar implementation environment and is inexpensive and easy to deploy. At the same time, it does have significant performance limitations that have become increasingly apparent as the usage of PlanetLab has grown, and as researchers have sought to deploy long-running services that carry significant volumes of traffic. Because PlanetLab applications run as user-space processes, their packet forwarding rates are typically limited to less than 50K packets

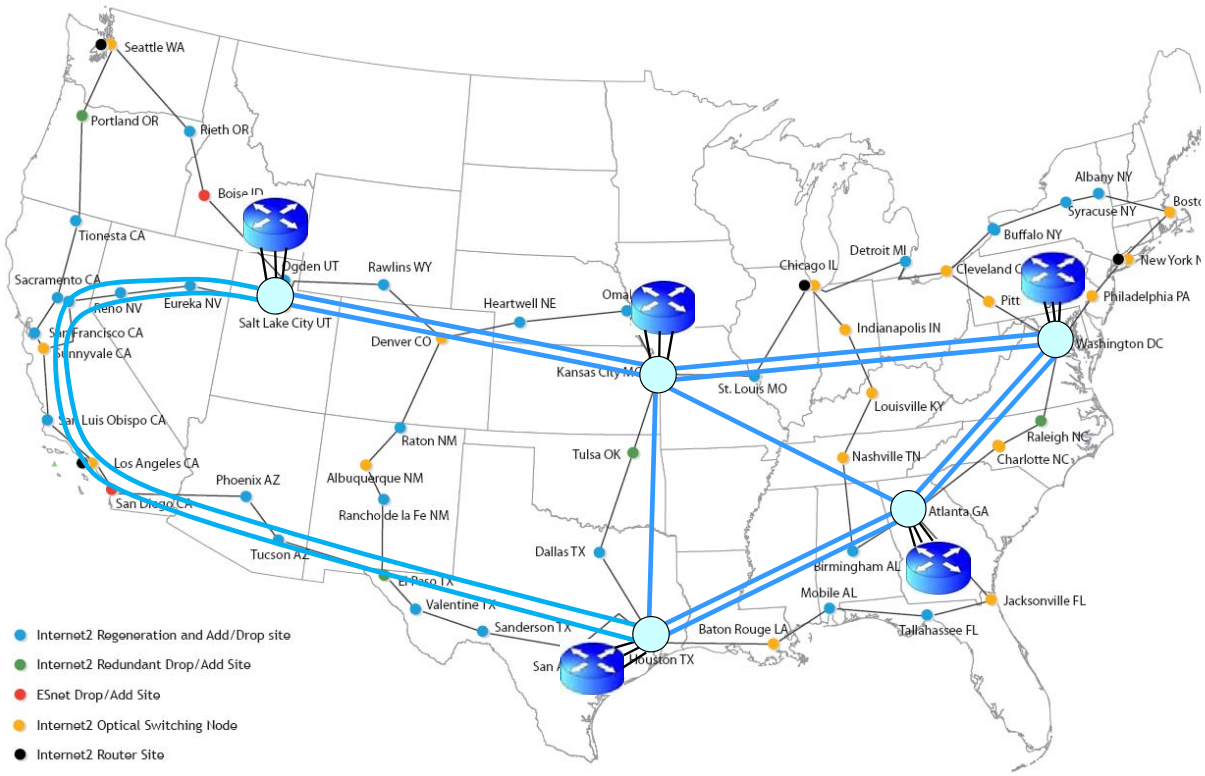

Figure 1. Planned Deployment of SPP Nodes in GENI Using Internet2

per second, which translates to less than 100 Mb/s for average packet lengths of 250 bytes. Applications that do significant processing of packets (rather than simply forwarding them) can have substantially smaller packet forwarding rates. In addition, applications running in PlanetLab are subject to high latencies (tens of milliseconds per hop), high delay jitter (tens to hundreds of milliseconds) and poor performance isolation. These characteristics are caused by the coarse-grained time-slicing provided by the operating system, and the failure to properly account for OS-level processing on behalf of different application processes. The SPP seeks to address these issues by integrating general purpose server blades with performance-optimized NP subsystems, into a platform that delivers the flexibility and ease-of-use of a conventional PlanetLab implementation, while delivering much higher levels of performance. By supporting a simple and familiar fast-path/slow-path application structure, we make it straight-forward for researchers to map the high volume part of their applications (which is typically fairly small) onto the NP resources, while enabling them to implement the more complex parts in the programmer-friendly environment offered by a general-purpose server. This report provides a detailed description of the SPP architecture, including all hardware and software components.

2. GENI DEPLOYMENT PLAN

Five SPPs are being deployed within Internet 2 as part of NSF's GENI initiative, and will be made available for use by the networking research community. Systems will be deployed at five locations (Salt Lake City, Kansas City, Washington D.C., Atlanta and Houston) and connected by gigabit links. Some sites will have multiple links connecting them, as shown in Figure 1. In addition to the direct links to other SPPs, each node will have several gigabit links to an Internet 2 IP router. These interfaces will have IP addresses that are visible to any Internet 2 connected institution, allowing traffic from those institutions to reach the routers through the existing

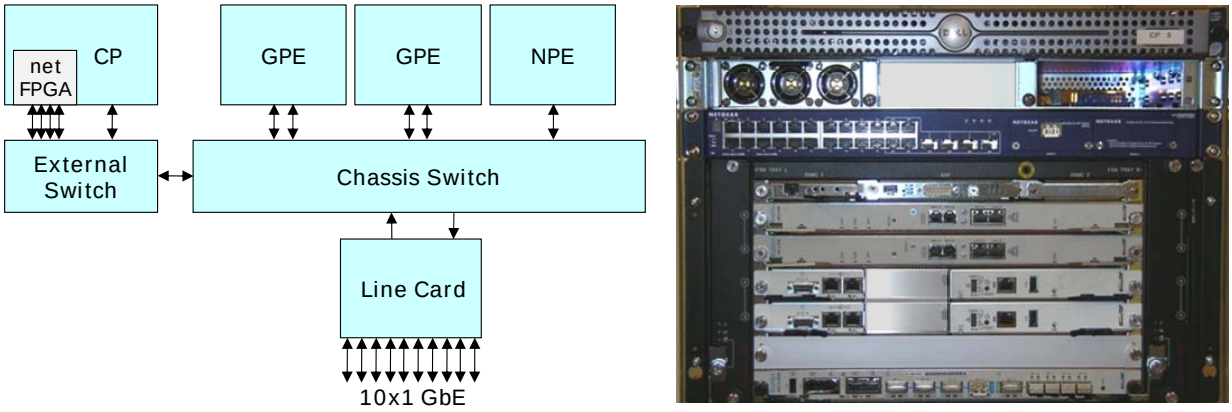

Figure 2. System organization showing Control Processor (CP), General Purpose Processing Engines (GPE) and Network Processing Engines (NPE), Line Card and Switches

Internet 2 infrastructure. While these IP connections support only best-effort services, because of the way Internet 2 is engineered, we expect congestion on the IP access paths through Internet 2 to be minimal.

To use the SPPs, users request slices through a controller called SPC-PLC at Washington University. SPP-PLC runs the Planet Lab Central (PLC) software, providing users with access to SPPs using the same web interface that they use to access PlanetLab. The SPPs will obtain slice configuration information from SPP-PLC and use it to setup accounts on individual SPPs that researchers can then use to login to SPPs so they can configure and run their experiments.

The deployment will take place in two stages. The first stage (which is now complete) includes Salt Lake City, Kansas City and Washington D.C. These will be connected in a ring, with a pair of gigabit links between each adjacent pair of nodes.

3. HARDWARE COMPONENTS

3.1. Overview

Figure 2 shows the main components of an SPP node. Most of the components are blades in an *Advanced Telecommunications Computing Architecture* (ATCA) blade server. ATCA is a standard for telecom-class blade servers, appropriate for a wide range of applications, including high capacity routers. All input and output occurs through the *Line Card* (LC), which is an NP-based subsystem with one or more physical interfaces. The LC forwards each arriving packet to the system component configured to process it, and queues outgoing packets for transmission, ensuring that each slice gets the appropriate share of the network interface bandwidth. The architecture can support multiple LCs, but the systems being deployed for GENI have one LC each. The *General Purpose Processing Engines* (GPE) are conventional dual processor server blades running the PlanetLab OS (currently Linux 2.6, with PlanetLab-specific extensions) and hosting vServers that serve application slices. The *Network Processing Engine* (NPE) is a server blade containing two NP subsystems, each comprising an Intel IXP 2850 NP, with 17 internal processor cores, 3 banks of SDRAM, 3 banks of QDR SRAM and a Ternary Content Addressable Memory (TCAM). The architecture supports multiple NPEs, but the deployed systems have a single NPE each. The NPE supports fast path processing for slices that elect to use this capability and provides up to 10 Gb/s of IO bandwidth. The *Control Processor* (CP) is a separate rack-mount server that hosts the software that coordinates the operation of the system as a whole. The CP also hosts a Net-FPGA card with four 1 GbE interfaces. The Net-FPGA will be

Figure 3. General Purpose Processing Engine (Radisys ATCA 4310)

made available for use by researchers. The switching substrate includes a chassis switch and a separate external switch, which provides additional 1 GbE ports. The chassis switch board actually includes two switches, a *Fabric Switch* with both 1 GbE and 10 GbE ports for data traffic and a *Base Switch* with 1 GbE ports for control traffic.

3.2. General Purpose Processing Engine (GPE)

The GPEs (see Figure 3) are dual processor blade servers, specifically Radisys ATCA 4310 blades with 2 GHZ Intel Xeon processors with 4 GB of memory and an on-board SAS disk (37 GB). They have two GbE network interfaces, one on the fabric switch and one on the base switch. The base switch interface is reserved for control traffic only and is not directly accessible to user applications running on the GPEs.

3.3. Network Processing Engine (NPE)

The Network Processor Engine is implemented using a Radisys ATCA 7010 blade, which contains two Intel IXP 2850 NP subsystems. The 7010 blade communicates with the chassis switch through the Fabric Interface Card, a small mezzanine card that allows the 7010 to be used with different types of chassis switches. In the case of the SPP, the FIC provides a 10 GbE interface to the chassis switch that passes through the ATCA backplane. The 7010 blade can be configured with an optional input/output card that is mounted on the rear-side of the chassis. Such rear-mounted cards are referred to as Rear Transition Modules (RTM). The NPE does not use the RTM, but the Line Card does. The various components on the blade communicate through a Serial Peripheral Interface (SPI) switch supporting data rates of just over 12 Gb/s. The SPI interface transfers data in fixed length cells of 64 bytes, so is subject to segmentation losses when transferring variable length packets. The two NP subsystems share a Ternary Content Addressable Memory (TCAM) which can store up to 18 Mb of data and can be configured to support word lengths ranging from 72 to 576 bits. The xScale processors share a separate network connection to the base switch, which is used for control communication.

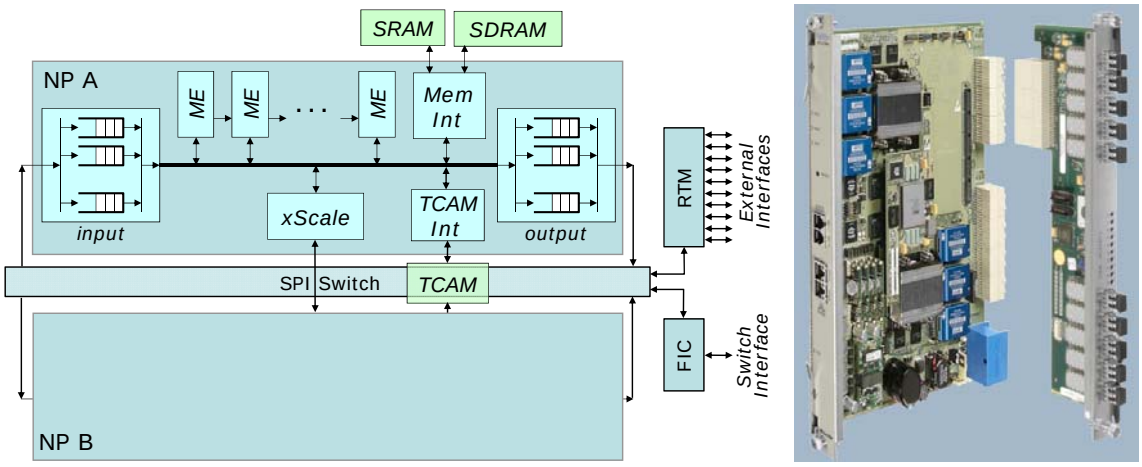

Figure 4. Radisys ATCA 7010 Network Processor Blade with 10x1 GbE IO card

Each of the IXP 2850s has an xScale management processor that runs an embedded version of Linux, plus 16 *MicroEngines* (ME), which are 32 bit RISC processors optimized for packet processing. Each ME has a small program store capable of storing 8K instructions, a register file and a small data memory. There is an on-chip SRAM that can be accessed by all of the MEs and multiple interfaces to off-chip memory. These include four SRAM and three DRAM interfaces. As with any modern processor, the primary challenge to achieving high performance is coping with the large processor/memory latency gap. Retrieving data from off-chip memory can take 50-100 ns (or more), meaning that in the time it takes to retrieve a piece of data from memory, a processor can potentially execute over 100 instructions. The challenge for processor designers is to try to ensure that the processor stays busy, in spite of this. Conventional processors cope with the memory latency gap primarily using caches. However for caches to be effective, applications must exhibit *locality of reference*, and unfortunately, networking applications typically exhibit very limited locality of reference, with respect to their data. Since caches are relatively ineffective for networking workloads, the IXP provides a different mechanism for coping with the memory latency gap, *hardware multithreading*. Each of the MEs has eight separate sets of processor registers (including Program Counter), which form the MEs hardware *thread contexts.* An ME can switch from one context to another in 2 clock cycles, allowing it to stay busy doing useful work, even when several of its hardware threads are suspended, waiting for data to be retrieved from external memory.

The MEs include small FIFOs (called *Next Neighbor Rings*) connecting to neighboring MEs, which can be used to support pipeline processing of packets. A pipelined program structure makes it easy to use the processing power of the MEs effectively, since the parallel components of the system are largely decoupled from one another. Pipelined processing also makes effective use of the limited ME program stores, since each ME need only store the instructions for its stage of the pipeline

3.4. Line Card (LC)

The Line Card is implemented using the same Radisys 7010 blade as the NPEs. The one difference is that the Line Cards are configured with the optional IO card. There are multiple IO cards available for the 7010. The Line Cards use an IO card with ten 1 GbE interfaces. The card supports swappable modules allowing it to accommodate either copper or fiber connections. Packets are transferred between the IO card and the 7010 boards NP subsystems through the on-board SPI switch. This requires that the Ethernet packets be fragmented into 64 byte cells for

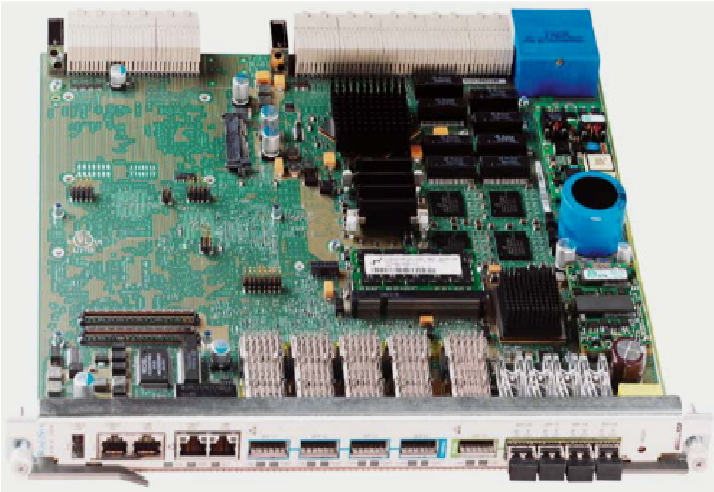

Figure 5. Switch blade (Radisys ATCA 2210)

transfer through the SPI switch, which can lead to fragmentation losses for packets that are just a little too long to fit into one SPI cell.

3.5. Switching Substrate

The chassis switch is a Radisys ATCA 2210 card which includes two switches, a fabric switch with 10 GbE ports (these can optionally be used for 1 GbE interfaces) and a base switch with 1 GbE ports. The fabric switch has four expansion ports on the front panel that can be used to connect directly to other components. The base switch also has four front panel ports which are used by the SPP for connections to the CP and the shelf manager. The external switch is a Netgear GSM 7228 with 24 1 GbE interfaces and the ability to host up to four 10 GbE interfaces. The SPP has one of these 10 GbE interfaces equipped and connected to the chassis switch through one of its front panel ports.

3.6. ATCA Chassis

The SPP uses a Shroff 5U six slot ATCA chassis (model name Zephyr) with an integrated Shelf Manager. The Shelf Manager has an on-board CPU that provides low level maintenance access to the chassis. This allows the various blades in the chassis to be remotely controlled. This is used primarily to force a reboot of a component that is not responding as expected.

3.7. Control Processor

The Control Processor (CP) is implemented by a Dell PowerEdge 860 with 2 GB of memory and 160 GB of disk. It has three 1 GbE network connections. One is connected to the base switch (for control communication), one is connected to the fabric switch (for data communication to and from the line card) and one serves as a "back door" for remote maintenance access to the CP. The CP is also equipped with several serial interfaces, which connect to the Shelf Manager, the chassis switch blade, the external switch and each of the two GPEs. These provide backup maintenance access, in the event that standard access mechanisms fail.

3.8. NetFPGA

The NetFPGA is a PCI card that hosts a Xilinx Virtex 2 Pro 50 FPGA, on-board SRAM and DRAM and four 1 GbE interfaces. Each SPP has one NetFPGA which is available as a resource for use by researchers. The NetFPGA is hosted by the CP, and its four network connections go to the external switch. From there, packets can be forwarded to any of the other components in the system. More details on the NetFPGA can be found at `http://www.netfpga.org/`.

Figure 6. NetFPGA card

4. DATAPATH SOFTWARE

In this section, we describe the software components that implement packet processing in the Line Card and NPE.

4.1. Line Card

The Line Card is part of the SPP substrate. That is, it implements packet processing functions that are common to all applications running on the SPP and performs no application-specific packet processing. However, elements of the SPP are configured for particular applications and we describe those elements and how they can be configured.

Figure 7 shows the software components that implement the Line Card's packet processing functionality. The software is organized into two pipelines, one that processes *ingress traffic* (that is traffic arriving on an external interface and passing through to the chassis switch) and one that processes egress traffic. Each of these pipelines is mapped onto a different NP subsystem as there is little interaction between the two. Each block in the diagram indicates the number of MicroEngines (ME) that are used to implement that component. In some cases, the multiple MEs just implement finer-grained pipeline stages. In other cases, they operate in parallel. Successive pipeline elements are separated by buffers, which are not shown in the diagram. Some of these are implemented using the Next Neighbor Rings. Others are implemented using the shared on-chip SRAM.

For most pipeline components, per packet processing overhead is the dominant performance concern, so the software was largely engineered around the case of minimum size packets. For the SPP, the minimum packet size is determined by the minimum Ethernet frame length (effectively 88 bytes when the VLAN tag, flag, preamble and inter-packet gap are all accounted for). If all ten physical interfaces are operating at full rate, this means that each of the pipelines must process a packet every 70 ns. This means that a pipeline stage that is implemented with a single ME has less than 100 processor cycles it can use to process each packet. Since an access to external memory has a latency of 150 cycles for SRAM and 300 cycles for DRAM, it can access external memory only 2-4 times, even if it uses all eight hardware thread contexts to mask the memory latency. This is the fundamental challenge that must be met to maintain high performance.

We start by describing the ingress pipeline. The *RxIn* block transfers packets from the IO interface of the IXP chip into memory. This involves reassembling SPI cells into packets, which are placed in DRAM buffers of 2 KB each. The RxIn block also allocates a *buffer descriptor* which

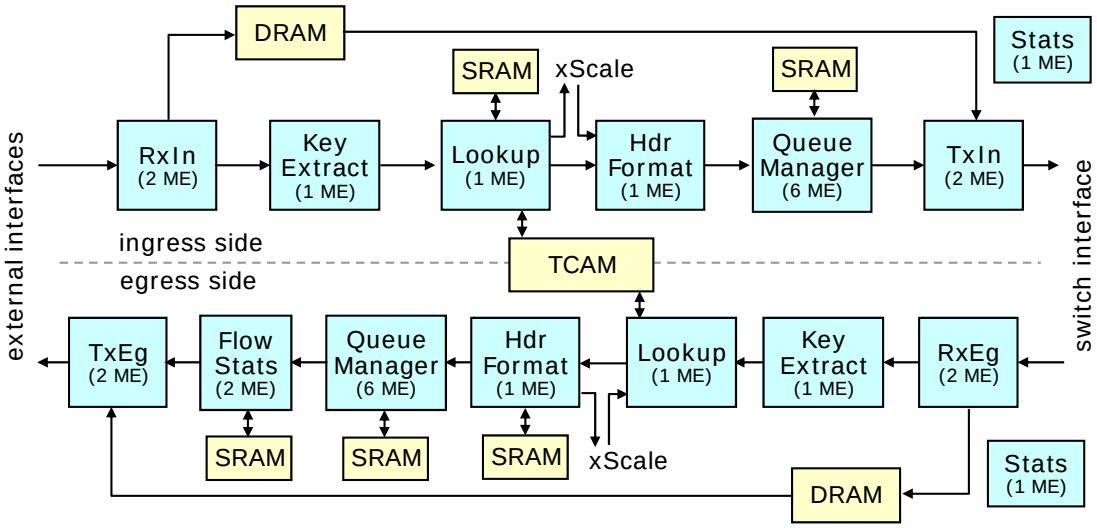

Figure 7. Line Card data path software components and hardware mapping

is initialized and stored in SRAM. RxIn passes several pieces of metadata to the next pipeline element. This includes the physical interface on which the packet was received, its Ethernet frame length, several status flags and a reference to its buffer.

The *Key Extract* block extracts selected header fields from the buffer. These include the IP packet length, the value of the IP protocol field and the destination IP address and port number. These are added to the metadata passed down the pipeline. The *Lookup* block performs a lookup in the shared TCAM. This is an exact match lookup, based on the interface number, protocol, destination IP address and port number. The result of the lookup specifies the queue the packet is to be placed in, a translated port number and a VLAN number, which is used within the chassis switch to determine where it goes next. It also includes a *Statistics Index* which is used to identify traffic counters that are to be updated for the given packet. The *Header Format* block makes any required changes to the packet headers in the DRAM buffer. This includes rewriting the TCP or UDP destination port number and rewriting MAC address to reflect the address of the component the packet is to be forwarded to next. The Lookup block can send selected packets to the xScale (as determined by the lookup results) and the xScale can insert packets into the pipeline through the Header Format block.

The *Queue Manager* (QM) is the most complex of the components in the datapath. It is implemented using six MEs. Four of these implement the actual queueing functions, while one distributes packets received from the Header Format block across the four queueing engines, while the other provides a similar interface function on output. Each of the four queueing engines manages a separate set of linked list packet queues that are stored in external DRAM. The IXP provides low level hardware support for managing such queues, making the basic list operations highly efficient. However, maintaining high throughput is still challenging, as successive accesses to a single queue inherently requires at least one access per packet and each such access takes 150 cycles (the SRAM memory latency). Each queueing engine also implements five separate packet schedulers, which can be individually rate controlled. Each of these schedulers has its own list of queues, and implements a Weighted Deficit Round Robin scheduling policy. In the Line Card, queues are assigned to schedulers based on their destinations. In particular, all queues assigned to a particular scheduler share a common "next hop" (the CP, GPE1, GPE2 or the NPE). Multiple schedulers can share a common destination

and this is used to enable higher overall throughput, as there are performance limits associated with both individual MEs and individual schedulers.

The TxIn block transfers packets from the DRAM buffers to the IO interface. This involves segmenting packets into SPI cells. The *Statistics* block is used by most of the other components to record traffic statistics. It has an input FIFO (not shown) which is implemented using the on-chip SRAM accessible to all MEs. The IXP hardware provides low level support to enable multiple MEs to write to such FIFOs without interference and without requiring explicit software concurrency control. This allows them to issue update requests for statistics counters without having to interrupt their main packet processing flow. The Stats block processes these requests and maintains the traffic counters in external memory. This delegates the memory access overheads associated with updating the statistics counters to a separate ME, in order to minimize the impact on those MEs processing packets. A typical statistics request updates both a packet counter and a byte counter, allowing effective monitoring of both packets processed and aggregate bandwidth. There is one other ME that is not shown in the diagrams. This ME maintains the free space list of packet buffers. It also is used by multiple MEs and accepts "buffer recycling requests" from other MEs using a similar input FIFO.

The *egress* pipeline is similar to the ingress pipeline, but there are a few differences. First, the egress pipeline includes a *Flow Statistics* module that maintains information about outgoing packet flows. This data is collected to allow for accountability of outgoing packets. Since the SPP is a shared platform used by researchers to carry out networking experiments, it is possible for users of the platform the use it to send packets to Internet destinations that don't want to receive those packets (this can happen inadvertently or maliciously). This can lead the users of the computers at those destinations to complain about the unwanted traffic. When this happens, it's important for SPP operators to have the ability to determine the individual user whose experiment is generating the unwanted traffic. The Flow Stats module provides the low level data collection needed to support this. The requirement for flow statistics originates with the PlanetLab testbed, and the data collected is compatible with the data collected for conventional PlanetLab nodes. The Flow Stats module maintains its data in an external SRAM which can be read by the xScale control processor. Software running on the xScale aggregates the data produced by the Flow Stats module and periodically transfers it to the Control Processor, which stores it on disk and makes it available to system administrators.

All the other components of the egress pipeline are similar to their counterparts in the ingress pipeline, although there are small differences. For example, the header fields used as the lookup key in the Lookup module are different, and include the VLAN on which the packet arrived and its source IP address and port number. The Queue Manager is the same as in the ingress pipeline but is configured differently. Each of the Line Card's outgoing interfaces is assigned a distinct packet scheduler and each experiment running on the SPP (more precisely each slice) that is using an interface has a queue on that interface with a weight that reflects its assigned share of the interface bandwidth. In the simplest case, each of the outgoing interfaces is a separate physical interface, but it is possible to define multiple virtual interfaces on a given physical interface and associate a separate scheduler with each virtual interface. This makes it possible to provision each virtual interface with a specific share of the physical interface bandwidth. The external interfaces each have an associated IP address and the GPEs associate these same IP addresses with their own virtual interfaces. So, a GPE sends a packet out of the SPP on a particular external interface by sending it with the source IP address associated with that interface. This allows the GPE software to be largely oblivious to the fact that it is operating within an SPP.

Because the SPP supports multiple processing engines, the TCP and UDP port numbers associated with the external interfaces must be shared by the CP, the GPEs and the NPE. Since the CP and GPE operating systems each control the port numbers that they associate with sockets, it's possible that port numbers selected by different components will conflict with one another. This requires a form of Network Address Translation (NAT) on the part of the Line Card. More specifically, the Line Card must translate the port numbers of the SPP's endpoint of TCP and UDP connections that originate with the SPP (i.e. client connections). That is, it must translate the source port number for packets leaving the SPP and the destination port number for packets arriving at the SPP. The port number translation is implemented by packet filters inserted in the Line Card's TCAM. All packets forwarded by the Line Card must match such a filter. If there is no filter for a given packet, that packet will be directed to the xScale, where a NAT daemon will determine if the packet belongs to a new outgoing connection, and if so will assign it a port number and install a TCAM filter to implement the translation of subsequent packets. For TCP connections, an additional filter is installed to send copies of SYN packets to the NAT daemon, so it can detect the closing of the connection, remove the associated filters and de-allocate the assigned port number. Traffic on UDP connections is monitored continuously, and the associated port numbers are freed after an extended period of inactivity. The NAT daemon also performs translations on outgoing ICMP echo packets, allowing applications on GPEs to send ping packet and receive the corresponding reply.

4.2. Network Processing Engine Software – version 1

The organization of the first version of the NPE software and its mapping onto MEs is shown in Figure 8. This software uses one of the two NP subsystems on the Radisys 7010 blades. Our original plan was to instantiate the same software on both blades, allowing them to be used as largely independent NPEs. Unfortunately, limitations of the 7010's input/output layer made this infeasible, so in the initial deployment of the SPP, only one of the two NP subsystems is available to users. The next section describes another version of the software that is under development and which will replace the first version, as soon it has been completed.

As in the Line Card, the software components that implement the NPE are organized as a pipeline. Packets received from the chassis switch are copied to DRAM buffers by the *Receive* (Rx) block on arrival, which also passes a reference to the packet buffer through the main packet processing pipeline. Information contained in the packet header can be retrieved from DRAM by subsequent blocks as needed, but no explicit copying of the packet takes place in the processing pipeline. At the end of the pipeline, the *Transmit* (Tx) block forwards the packet to the output. Buffer references (and other information) are passed along the pipeline primarily using FIFOs linking adjacent MEs. Pipeline elements typically process 8 packets concurrently using the hardware thread contexts. The *Substrate Decapsulation* block determines which slice the packet belongs to, by doing a lookup in a table stored in one of the SRAMs. It also effectively strips the outer header from the packet by adjusting a pointer to the packet's buffer before passing it along the pipeline.

The *Parse* block includes slice-specific program segments. More precisely, *Parse* includes program segments that define a preconfigured set of *Code Options*. Slices are configured to use one of the available code options and each slice has a block of memory in SRAM that it can use for slice-specific data. Currently, code options have been implemented for IPv4 forwarding and for the Internet Indirection Infrastructure (I3) [ST02]. New code options are fairly easy to add, but this does require familiarity with the NP programming environment and must be done with care to ensure that new code options do not interfere with the operation of the other

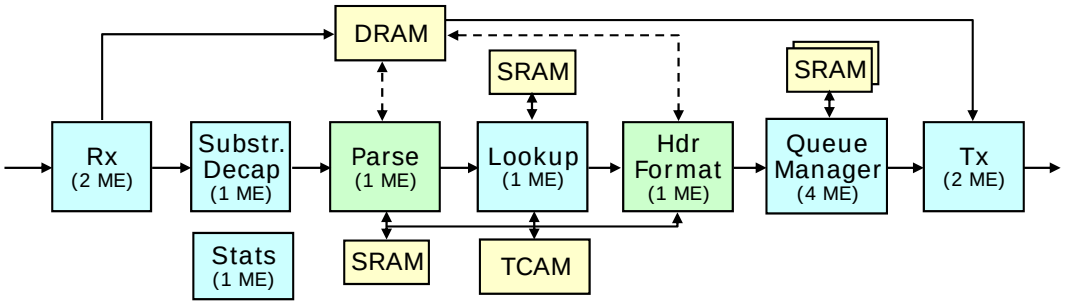

Figure 8. NPE software structure showing the use of memory by various software components, and the mapping of components onto Micro-Engines (ME)

components. The primary role of *Parse*, is to examine the slice-specific header and use it and other information to form a *lookup key*, which is passed to the *Lookup* block.

The *Lookup* block provides a generic lookup capability, using the TCAM. It treats the lookup key provided by *Parse* as an opaque bit string with 112 bits. It augments this bit string with a slice identifier before performing the TCAM lookup. The slice's control software can insert packet filters into the TCAM. These filters can include up to 112 bits for the lookup key and 112 bits of mask information. Software in the Management Processor augments the slice-defined filters with the appropriate slice id before inserting them into the TCAM. This gives each slice the illusion of a dedicated TCAM. The position of filter entries in the TCAM determines their lookup priority, so the data associated with the first filter in the TCAM matching a given lookup key is returned. The number of entries assigned to different slices is entirely flexible, but the total number of entries is 128K.

The *Header Formatter* which follows *Lookup* makes any necessary changes to the slice-specific packet header, based on the result of the lookup and the semantics of the slice. It also formats the required outer packet header used to forward the packet to either the next PlanetLab node, or to its ultimate destination.

The *Queue Manager* (QM) implements a configurable collection of queues. More specifically, it provides 20 distinct packet schedulers, each with a configurable output rate, and each with an associated set of queues. Separate schedulers are needed for each external interface supported by Line Cards. The number of distinct schedulers that can be supported by each ME is limited by the need to reserve some of the ME's local memory for each. Each scheduler implements the *weighted deficit round robin* scheduling policy, allowing different shares to be assigned to different queues. When a slice's control software inserts a new filter, it specifies a slice-specific queue id. The filter insertion software remaps this to a physical queue id, which is added, as a hidden field, to the filter result. Slices can configure which external interface its queues are associated with, the effective length of each queue and its share of the interface bandwidth.

The *Statistics* module maintains a variety of counts on behalf of slices. These can be accessed by slices through the xScale, to enable computation of performance statistics. The counting function is separated from the main processing pipeline to keep the associated memory accesses from slowing down the forwarding of packets, and to facilitate optimizations designed to overcome the effects of memory latency. The counts maintained by the Statistics module are kept in one of the external SRAMs and can be directly read by the xScale.

4.3. Network Processing Engine Software – version 2

Figure 9 shows the organization of version 2 of the NPE software. There are two primary objectives for this version. The first is to take advantage of both NP subsystems on the Radisys

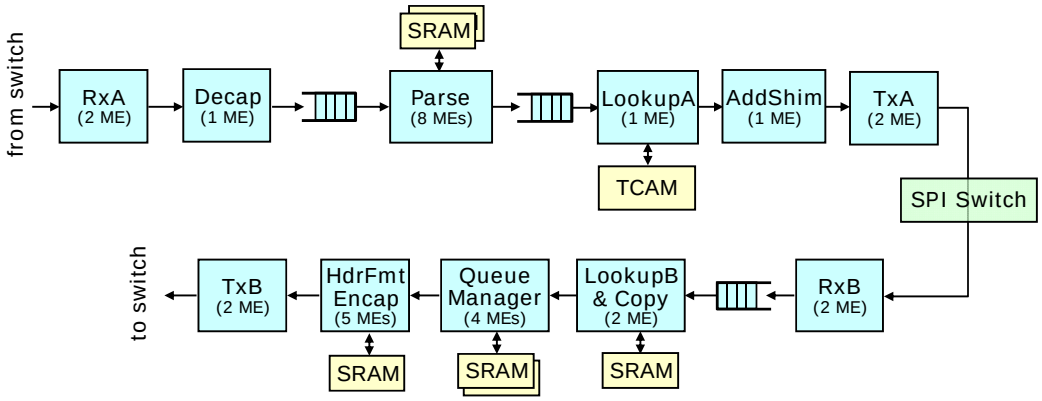

Figure 9. Version 2 of the NPE datapath software

7010 blade, to increase the amount of traffic that can be handled from 5 Gb/s to 10 Gb/s. The second is to provide support for packet replication, allowing applications to more easily implement services that require multicast.

In this version, the packet processing pipeline is distributed across both NPs. On NPU_A (the top one in the figure), eight MEs are assigned to the Parse block, allowing for more extensive processing of user packets. These will operate in parallel and will take advantage of all eight thread contexts, allowing up to 64 packets to be processed concurrently. This allows for up to 780 instructions to be executed per minimum size packet, and for up to 20 DRAM references or 41 SRAM references per packet.

The result returned by the Lookup block on NPU_A includes a *Result Index*, which is passed to NPU_B in a shim header which is added to the packet by the *AddShim* block. The *Lookup and Copy* block on NPU_B uses the result index to select an entry from an SRAM-resident table that specifies one or more queues in which the packet is to be placed. For multicast packets, a *Header Buffer* is created for each copy. The associated buffer descriptor includes a reference to the original packet buffer, which is now referred to as the *Payload Buffer*. For each copy, the IP packet header information needed to forward that copy to its next destination is placed in the header buffer and a reference to the header buffer is passed to the Queue Manager, along with the appropriate queue and packet scheduler information. A reference count is also placed in the descriptor of the payload buffer, so that it can be deallocated when the last header buffer has been processed.

The Header Format block includes slice-specific code that formats the header of the slice's outgoing packets, by writing the header information in the header buffer. Of course, the slice-specific code may also make changes to the payload of a packet by writing to the payload buffer. The Header Format block passes the reference to the header buffer to the TxB block, which formats the outgoing packet by reading the header from the header buffer, and the payload from the payload buffer.

5. CONTROL SOFTWARE

The major control software components are shown in Figure 10. The SPP-PLC is a separate system that runs the PlanetLab Central software, providing an interface through which users can request new slices and instantiate those slices on one or more SPP. The *System Resource Manager* is the top level controller and coordinates the use of various resources by the different components of the architecture. The *Resource Manager Proxy* provides an interface through which user slices can request and configure resources. The *Substrate Control Daemons* (SCD) in

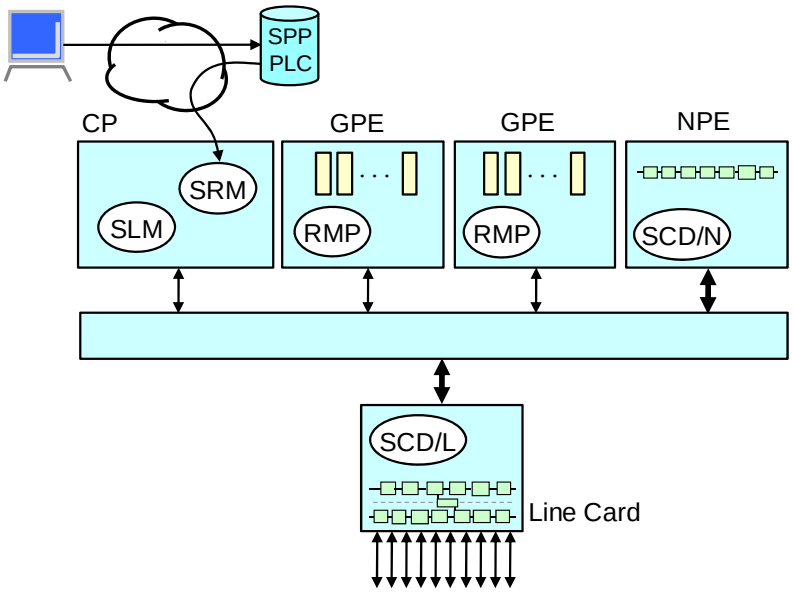

Figure 10. Major Control Software Modules

the Line Card and NPE provide an interface through which the datapath software running in the network processors is configured. The *SPP Login Manager* (SLM) provides a mechanism to enable users to login to the vServers for their individual slices, so they can install code, request and configure resources and run experiments. More details of the various components are provided below.

5.1. System Resource Manager (SRM)

The SRM is the top level controller for the SPP and provides several services. These include acquiring slice definitions from SPP-PLC, instantiating slice definitions, reserving and assigning resources to slices and coordinating the initialization of the whole system. The SRM implements functions provided by the Node Manager on a conventional PlanetLab node, but must provide this functionality in the context of a system with a more complex internal structure, and a richer set of resources.

The SRM polls SPP-PLC periodically to obtain new slice definitions. When a new slice is detected, the SRM selects one of the two GPEs on which to instantiate the slice. Slice instantiation involves creating a vServer on the selected slice, initializing it and configuring a login so that users can access their assigned vServer.

Once assigned to a vServer, a user can run programs that send and receive packets on the external interfaces. Outgoing connections are subjected to port number translation at the Line Cards, as described in Section 4. Users may also request the use of specific external port numbers in order to run servers that listen on specific ports. User requests are made through an interface provided by the RMP on the user's assigned GPE. The RMP forwards these requests to the SRM which manages all system level resources, including external port numbers, physical interface bandwidth and NPE resources.

5.2. Resource Manager Proxy (RMP)

The RMP provides an API used by applications running in vServers. The API allows users to reserve resources in advance (such as external port bandwidth and NPE fastpaths), to acquire those resources when a reservation period starts and configure the resources as needed. The RMP is implemented as a daemon that runs in the root context and is accessed through a set of

library routines. A command line interface is also provided so that users can reserve and configure resources interactively, or through a shell script. The command line interface converts the given commands to API calls.

The main API calls are listed below in topical sub-sections, along with a brief description of how each call is used. We use a representation that attempts to informally describe the interface semantics. More precise descriptions are given in the reference manual. We use an abstract interface syntax that has the form "$R \leftarrow F(A_1,\ldots,A_n)$" where F is the function name, A_i is the i-th argument, and R is the return value. Mnemonic names are used to convey usage while data type modifiers have been omitted. The following abbreviations and mnemonics are used in argument names and descriptions:

- FP FastPath
- EP EndPoint – a logical interface used by a slice and mapped to a physical interface
- LC LineCard
- BW BandWidth
- DB DataBase
- Xdescr X description where X is Q, EP or FP for Queue, EndPoint, or FastPath
- Xid X identifier where X is F, FP, MI, Q or S for Filter, FastPath, MetaInterface, Queue, or Slice

5.2.1. Interfaces

`ifList ← get_ifaces(ifList)`

Return a list of all physical interfaces of the SPP. Slices configure MIs using the information from this list. The returned list indicates for each physical interface the attributes of the interface; i.e., interface number, the interface type (Internet or peering), the IP address, the total bandwidth and the available bandwidth.

`ifNum ← get_ifn(EPaddr)`

Return the physical interface number of the EP.

`ifAttributes ← get_ifattrs(ifNum,ifAttributes)`

Return the attributes of the physical interface.

`IPaddr ← get_ifpeer(ifNum)`

Return the IP address of the physical interface.

5.2.2. GPE Interface Bandwidth

`rmpCode ← resrv_pl_ifbw(ifNum,BWkbps)`

Reserve bandwidth (Kbps) on the physical interface.

`rmpCode ← free_pl_ifbw(ifNum,BWkbps)`

Release bandwidth (Kbps) from the physical interface.

5.2.3. GPE Endpoints

`EPdescr ← alloc_endpoint(EPdescr)`

Given an EP description, allocate a new EP, and return a reference to the EP. A filter is installed in the LC to direct matching traffic to the GPE. For TCP or UDP, you can select the port number or have the system give you one.

`RMPcode ← free_endpoint(EPdescr)`

Free the endpoint, de-install the LC filter for the EP, and return the status.

5.2.4. FastPaths

`FPdescr ← alloc_fastpath(codeOpt,bwSpec,resSpec,memSpec,FPdescr)`

Given specifications for the aggregate bandwidth, other resource (filters, queues, buffers and stats) and memory, allocate a new FP for the code option, and return a reference to the FP description.

`free_fastpath(FPid)`

Free the resources of the FP.

5.2.5. FastPath Bandwidth

`RMPcode ← resrv_fpath_ifbw(FPid,ifNum,BWkbps)`

Reserve bandwidth (Kbps) on a physical interface for a FP.

`RMPcode ← free_fpath_ifbw(FPid,ifNum,BWkbps)`

Free the bandwidth (Kbps) of a FP from a physical interface, and return the status.

5.2.6. FastPath MetaInterfaces

`MIid ← alloc_udp_tunnel(FPid,EPdescr )`

Given a UDP tunnel EP description allocate the EP for the FP, and return the MI identifier.

`RMPcode ← free_udp_tunnel(FPid,MIid)`

Free the MI of a FP, and return the status.

`EPdescr ← get_endpoint(FPid,MIid,EPdescr)`

Return the UDP tunnel EP description for a given MI of a FP.

5.2.7. FastPath Queue Management

`RMPcode ← bind_queue(FPid,MIid,qidListType,qidList)`

Associate the listed queues to the MI of the FP, and return the status.

`Qdescr ← get_queue_params(FPid,Qid,Qdescr)`

Return the parameters (threshold, bandwidth) for the FP queue, and return a description of the queue.

`BWkbps ← set_queue_params(FPid,Qid,Qdescr)`

Set the queue parameters (threshold, bandwidth) for the FP queue, and return the bandwidth of the queue.

`Qlen ← get_queue_len(FPid,Qid,Qlen)`

Return the length of the FP queue.

5.2.8. Fastpath Filter Management

`rmpCode ← write_fltr(FPid,Fid,Fltr)`

Install a FP filter, and return the status.

`rmpCode ← update_result(FPid,Fid,Fltr)`

Modify the FP filter, and return the status.

`Fltr ← get_fltr_byfid(FPid,Fid,Fltr)`

Return the FP filter given the filter ID.

`Fltr ← get_fltr_bykey(FPid,key,Fltr)`

Return the FP filter that matches the key.

`fltrResult ← lookup_fltr(FPid,key,Fltr)`

Return the result part of the FP filter that matches the key.

`rmpCode ← rem_fltr_byfid(FPid,Fid)`

Remove the FP filter given the filter ID, and return the status.

`rmpCode ← rem_fltr_bykey(FPid,key)`

Remove the highest priority FP filter that matches the key, and return the status.

5.2.9. FastPath Stats Management

`statsRecord ← read_stats(FPid,statsId,flags,statsRecord)`

Return the FP stats record (counter group) for the stats ID. The flags argument selects which counters to return. You can select the byte or packet counter and whether the preQ or postQ counter

`rmpCode ← clear_stats(FPid,statsId,flags)`

Reset the FP stats counters for the stats ID. The flags argument selects which counters to return.

```
statsHandle ← create_periodic(FPid,statsId,period,
                                    historySize,flags)
```

Create a periodic stats read event for the stats ID with the given period and history size, and return a handle for the operation. The flags argument indicates the retrieval method: either push the stats data to a registered port, or have the VM pull the data using the get_periodic command.

`rmpCode ← delete_periodic(FPid,statsHandle)`

Remove the periodic event, remove the callback state, and return the status.

`rmpCode ← set_callback( FPid,statsHandle,ipPortNum)`

Setup the callback for a periodic stats push model that sends stats records to the IP port number, and return the status.

`statsRecord ← get_periodic(FPid,statsHandle,statsRecord)`

Return the stats record associated with the stats handle.

5.2.10. FastPath Memory

Each code option is provided with a block of SRAM. A slice can read/write to any location in this block. A code option may elect to provide library functions to manipulate control structures within this block. The valBuf argument to the read/write functions is a structure that includes the number of bytes in the buffer and the buffer itself.

`rmpCode ← mem_write(FPid,offset,valBuf)`

Write data to the SRAM starting at offset within the FP block, and return the status. The valBuf argument is a structure that includes the number of bytes and the data.

`valBuf ← mem_read(FPid,offset,nbytes,valBuf)`

Read bytes into the value buffer, and return a reference to the value buffer.

5.2.11. Reservation Management

`rmpCode ← make_reservation(rsvRecord)`

Make a reservation, and return the status.

`rmpCode ← update_reservation(rsvRecord)`

Update a reservation.

`rmpCode ← cancel_reservation(date)`

Cancel the reservation that includes the specified date and time.

5.3. Substrate Control Daemons (SCD)

The SCDs run on the xScale processors of the Line Card and NPE. They provide a messaging interface, through which other control software components can exercise control. These include messages to access traffic counters, add/remove TCAM packet filters, configure queue parameters (including WDRR weights and discard thresholds), read/write specific memory locations used for control and status registers, etc. These are described in more detail below. All functions have a context ID (*contextID*) as an argument. A context ID of 0 indicates a privileged operation performed by the substrate. Any other context ID indicates a user context and is either a fastpath ID or internal slice ID.

Many of the functions (e.g. *write_fltr*) appear to be similar to ones in the RMP. This is expected because the evaluation of an RMP operation must often be relayed to an SCD for evaluation but with one important difference. The SCD has a substrate view of objects whereas the RMP provides a higher-level of abstraction.

The Line Card SCD allows the SRM to control various elements of the Line Card data path. This includes the TCAM-resident packet filters (on both input and output), interface addressing and bandwidth, NAT filter table configuration and queueing parameters.

The NPE SCD allows the SRM and the RMP to control various elements of the NPE data path. This includes fast path configuration data, per-slice packet filters resident in the TCAM and queueing parameters.

5.3.1. Control Table Initialization

There are several tables and control blocks used by the control software.

`set_sched_params(contextId,Sid,ifNum,BWkbpsMax,BWkbpsMin,valBuf)`
: Set the interface number and bandwidth characteristics for a Scheduler in the Per Scheduler Parameters table.

`set_encap_cb(contextId,Sid,srcIPaddr,dstMACaddr,valBuf)`
: On the NPE, set the source IP Address and destination MAC Address associated with the specified scheduler.

`set_sched_mac(contextId,Sid,dstMACaddr,srcMACaddr,valBuf)`
: On the LC, set the destination and source MAC Addresses for the specified scheduler.

`set_encap_gpe(contextId,FPid,GPEipAddr,NPEipAddr,valBuf)`
: On the NPE, for a fast path, set the GPE IP Address and NPE IP Address to be used for communication between the GPE and NPE for local delivery and exceptions.

`set_fpmi_bw(contextId,FPid,Sid,MIid,BWkbps,valBuf)`
: On the NPE, for a particular fast path, set the bandwidth for a MI using a particular scheduler.

`SCDcode ← set_src_hwaddr(contextId,MACaddr)`
: On the NPE, set the NPE's source MAC Address.

`SCDcode ← set_iface_table(contextId,ifTable)`
: On the NPE, initialize the RX Interface ID table. This table translates the receive destination address on a packet to a 4 bit index which will be used in the Lookup key.

5.3.2. FastPath (NPE SCD Only)

```
set_fast_path(contextId,FPid,codeOpt,vlanID,
            num_queues,num_filters,num_buffers,num_stats,
```

`SRAM_offset,SRAM_size,DRAM_offset,DRAM_size,valBuf)`

On the NPE, create a new fast path.

`rem_fast_path(contextId,FPid,valBuf)`

On the NPE, remove a fast path.

`SCDcode ← set_gpe_info(contextId,EXport,LDport,EXqid,LDqid)`

On the NPE, for a particular fast path, set the Local Delivery and Exception traffic port numbers and QIDs.

5.3.3. Memory

`write_sram(contextId,offset,valBuf)`

On the NPE, write to the SRAM block for a particular fast path.

`read_sram(contextId,offset,valBuf,count)`

On the NPE, read from the SRAM block for a particular fast path.

5.3.4. Queue Management

`SCDcode ← bind_queue(contextId,MIid,qidListType,qidVector)`

Associate the listed queues to the context's MI, and return the status.

`BWkbps ← set_queue_params(contextId,Qid,threshhold,BWkbps)`

Set the context's queue parameters (threshold, bandwidth) for the queue, and return the bandwidth of the queue.

`get_queue_params(contextId,Qid,threshhold,BWkbps)`

Return the context's parameters (threshold, bandwidth) for the queue through the *threshold* and *BWkbps* parameters, and return a description of the queue.

`get_queue_len(contextId,Qid,pktCnt,byteCnt)`

Return the length of the context's queue through the *pktCnt* and *byteCnt* parameters.

`set_queue_sched(contextId,Qid,Sid,valBuf)`

Associate a specified queue with the specified scheduler.

5.3.5. NPE Filter Management

`SCDcode ← npe_write_fltr(contextId,Fid,substrateFltr)`

Install a context's substrate (generic) filter with filter ID.

`SCDcode ← npe_update_result(contextId,Fid,result)`

Modify the result part of a context's substrate (generic) filter with filter ID.

`substrateFltr ← npe_get_fltr_by_key(contextId,key,substrateFltr)`

Return the context's substrate (generic) filter that matches the key.

`substrateFltr ← npe_get_fltr_by_fid(contextId,Fid,substrateFltr)`

Return the context's substrate filter given the filter ID.

`substrateResult ← npe_lookup_fltr(contextId,key,substrateResult)`

Return the result part of the context's substrate (generic) filter that matches the key.

`SCDcode ← npe_rem_fltr_by_key(contextId,substrateKey)`

Remove the context's highest priority substrate filter that matches the key, and return the status.

`SCDcode ← npe_rem_fltr_by_fid(contextId,Fid)`

Remove the context's substrate filter given the filter ID, and return the status.

5.3.6. Line Card Filter Management

There are two Line Card filter databases: ingress and egress. Ingress filters are used to determine which SPP component (e.g., NPE, GPE) should handle incoming packets. Egress filters are used to determine which output interface to send outgoing packets. The database ID (DBid) indicates the database to be used.

`write_fltr( contextId,DBid,Fid,key,mask,result,valBuf)`

Install a context's LC filter (key, mask, result) in the given database.

`update_result(contextId,DBid,Fid,result)`

Update a context's LC filter result in the specified database.

`get_fltr_by_key(contextId,DBid,key,mask,result,keyLen,resultLen)`

Given the key, retrieve a filter from the specified database.

```
get_fltr_by_fid(contextId,DBid,Fid,key,mask,result,keyLen,
                resultLen)
```

Given the filter id, retrieve a filter from the specified database.

`lookup_fltr(contextId,DBid,key,result,resultLen)`

Given the key, retrieve the filter result from the specified database.

`rem_fltr_by_key(contextId,DBid,key,valBuf)`

Given the key, remove the filter from the specified database.

`rem_fltr_by_fid(contextId,DBid,Fid,valBuf)`

Given the filter id, remove the filter from a specified database.

5.3.7. Statistics Management

`statsRecord ← read_stats(contextId,statsId,flags,statsRecord)`

Return the context's stats record (counter group) for the stats ID. The flags argument selects which counters to return. You can select the byte or packet counter and whether the preQ or postQ counter.

`SCDcode ← clear_stats(contextId,statsId,flags)`

Reset the context's stats counters for the stats ID, and return the status. The flags argument selects which counters to return.

```
statsHandle ←  create_periodic(contextId,statsId,period,count,
                                flags)
```

Create a periodic stats read event for the stats ID of the context with the given period and history size, and return a handle for the operation. The flags argument indicates the retrieval method: either push the stats data to a registered port, or have the VM pull the data using the get_periodic command.

`SCDcode ← del_periodic(contextId,statsHandle)`

Remove the context's periodic event, remove the callback state, and return the status.

`SCDcode ← set_callback(contextId,statsHandle,UDPport)`

Setup the context's callback for a periodic stats push model that sends stats records to the UDP port number, and return the status.

```
statsRecordVector ← get_periodic(contextId,statsHandle,
                                 statsRecordVector)
```

Return the context's stats record associated with the stats handle.

5.3.8. MicroEngine Management

`start_mes(contextId,valBuf)`

Start the MicroEngines on an NPU.

`stop_mes(contextId,valBuf)`

Stop the MicroEngines on an NPU.

5.3.9. NAT

```
nat_filters(contextId,ingressStartFid,ingressEndFid,
            egressStartFid, egressEndFid)
```

On the LC, initialize the NAT filter tables. This sets aside a block of the TCAM for the Ingress NAT filters and a block of the TCAM for the Egress NAT filters.

5.3.10. MetaInterface Management

`SCDcode ← create_mi(contextId,FPid,MIid,Sid)`

On the NPE,cCreate a new meta-interface for a fast path.

`SCDcode ← delete_mi(contextId,FPid,MIid)`

On the NPE, delete the specified meta-interface for the specified fast path.

`SCDcode ← set_mi_bw(contextId,FPid,MIid,BWkbps)`

On the NPE, for the specified fast path, set the bandwidth for a meta-interface.

`SCDcode ← bind_queue_sched(contextId,Qid,Sid)`

On the NPE, bind a queue to a scheduler.

`SCDcode ← unbind_queue_sched(contextId,Qid)`

On the NPE, unbind a queue from a scheduler and release its bandwidth on that scheduler.

`SCDcode ← unbind_queue(contextId,Qid)`

On the NPE, unbind a queue from a meta-interface and release its bandwidth on that meta-interface.

6. SUMMARY

The SPPs were designed and have been deployed to provide an experimental resource for the use of the networking research community. This document provides a detailed description of the SPP architecture to assist prospective users, interested in running experimental networks on the SPPs. We are in the process of transferring much of the information in this document in a wiki, along with additional user-level documentation. Comments and feedback on this document can be directed to the first author (jon.turner@wustl.edu).

REFERENCES

[BA06] Bavier, A., N. Feamster, M. Huang, L. Peterson, J. Rexford. "In VINI Veritas: Realistic and Controlled Network Experimentation," *Proc. of ACM SIGCOMM*, 2006.

[BH06] Bharambe, A., J. Pang, S. Seshan. "Colyseus: A Distributed Architecture for Online Multiplayer Games," In *Proc. Symposium on Networked Systems Design and Implementation* (NSDI), 3/06.

[CH02] Choi, S., J. Dehart, 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*, 5/02.

[CH03] Chun, B., D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. "PlanetLab: An Overlay Testbed for Broad-Coverage Services," *ACM Computer Communications Review*, vol. 33, no. 3, 7/03.

[CI06] Cisco Carrier Routing System. At www.cisco.com/en/ US/products/ps5763/, 2006

[DI02] Dilley, J., B. Maggs, J. Parikh, H. Prokop, R. Sitaraman, and B. Weihl. "Globally Distributed Content Delivery," *IEEE Internet Computing*, September/October 2002, pp. 50-58.

[FO07] Force 10 Networks. "S2410 Data Center Switch," http:// www.force10networks.com/products/s2410.asp, 2007

[FR04] Freedman, M., E. Freudenthal and D. Mazières. "Democratizing Content Publication with Coral," In *Proc. 1st USENIX/ACM Symposium on Networked Systems Design and Implementation*, 3/04.

[GE06] Global Environment for Network Innovations. http://www.geni.net/, 2006.

[HI98] Mike Hicks_ Pankaj Kakkar_ Jonathan T_ Moore_ Carl A_ Gunter_ and Scott Nettles. "PLAN, A packet language for active networks," In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming Languages*, 1998.

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

[KA02] Karlin, Scott and Larry Peterson. "VERA: An Extensible Router Architecture," In *Computer Networks*, 2002.

[KO00] Kohler, Eddie, Robert Morris, Benjie Chen, John Jannotti and M. Frans Kaashoek. "The Click modular router," *ACM Transactions on Computer Systems*, 8/2000.

[KO04] Kontothanassis, L. R. Sitaraman, J. Wein, D. Hong, R. Kleinberg, B. Mancuso, D. Shaw and D. Stodolsky. "A Transport Layer for Live Streaming in a Content Delivery Network," *Proc. of the IEEE, Special Issue on Evolution of Internet Technologies*, 9/04.

[PA03] Pappu, P., J. Parwatikar, J. Turner and K. Wong. "Distributed Queueing in Scalable High Performance Routers." *Proceeding of IEEE Infocom*, 4/03.

[PE02] Peterson, L., T. Anderson, D. Culler and T. Roscoe. "A Blueprint for Introducing Disruptive Technology into the Internet," *Proceedings of ACM HotNets-I Workshop*, 10/02.

[RA05] Radisys Corporation. "Promentum™ ATCA-7010 Data Sheet," product brief, available at http://www. radisys.com/files/ATCA-7010_07-1283-01_0505_datasheet.pdf.

[RH05] Rhea, S., B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica and H. Yu. "OpenDHT: A Public DHT Service and Its Uses," *Proceedings of ACM SIGCOMM*, 9/2005.

[SP01] Spalink, T., S. Karlin, L. Peterson and Y. Gottlieb. "Building a Robust Software-Based Router Using Network Processors," In *ACM Symposium on Operating System Principles* (SOSP), 2001.

[ST01] Stoica, I., R. Morris, D. Karger, F. Kaashoek and H. Balakrishnan. "Chord: A scalable peer-to-peer lookup service for internet applications." In *Proceedings of ACM SIGCOMM*, 2001.

[ST02] Stoica, I., D. Adkins, S. Zhuang, S. Shenker, S. Surana, "Internet Indirection Infrastructure," *Proc. of ACM SIGCOMM*, 8/02.

[TU06] Turner, J. "A Proposed Architecture for the GENI Backbone Platform," In *Proceedings of ACM- IEEE Symposium on Architectures for Networking and Communications Systems* (ANCS), 12/2006.

[VS06] Linux vServer. `http://linux-vserver.org`