Report Number: WUCSE-2003-22

2003-04-22

System-on-Chip Packet Processor for an Experimental Network Services Platform

David Taylor, Alex Chandra, Yuhua Chen, Sarang Dharmapurikar, John Lockwood, Wenjing Tang, and Jonathan Turner

As the focus of networking research shifts from raw performance to the delivery of advanced network services, there is a growing need for open-platform systems for extensible networking research. The Applied Research Laboratory at Washington University in Saint Louis has developed a flexible Network Services Platform (NSP) to meet this need. The NSP provides an extensible platform for prototyping next-generation network services and applications. This paper describes the design of a system-on-chip Packet Processor for the NSP which performs all core packet processing functions including segmentation and reassembly, packet classification, route lookup, and queue management. Targeted to a commercial... **Read complete abstract on page 2.**

System-on-Chip Packet Processor for an Experimental Network Services Platform

David Taylor, Alex Chandra, Yuhua Chen, Sarang Dharmapurikar, John Lockwood, Wenjing Tang, and Jonathan Turner

Complete Abstract:

As the focus of networking research shifts from raw performance to the delivery of advanced network services, there is a growing need for open-platform systems for extensible networking research. The Applied Research Laboratory at Washington University in Saint Louis has developed a flexible Network Services Platform (NSP) to meet this need. The NSP provides an extensible platform for prototyping next-generation network services and applications. This paper describes the design of a system-on-chip Packet Processor for the NSP which performs all core packet processing functions including segmentation and reassembly, packet classification, route lookup, and queue management. Targeted to a commercial configurable logic device, the system is designed to support gigabit links and switch fabrics with a 2:1 speed advantage. We provide resource consumption results for each component of the Packet Processor design.

2003-22

System-on-Chip Packet Processor for an Experimental Network Services Platform

Authors: Taylor, David; Chandra, Alex; Chen, Yuhua; Dharmapurikar, Sarang; Lockwood, John; Tang, Wenjing; Turner, Jonathan

April 22, 2003

Abstract: As the focus of networking research shifts from raw performance to the delivery of advanced network services, there is a growing need for open-platform systems for extensible networking research. The Applied Research Laboratory at Washington University in Saint Louis has developed a flexible Network Services Platform (NSP) to meet this need. The NSP provides an extensible platform for prototyping next-generation network services and applications. This paper describes the design of a system-on-chip Packet Processor for the NSP which performs all core packet processing functions including segmentation and reassembly, packet classification, route lookup, and queue management. Targeted to a commercial configurable logic device, the system is designed to support gigabit links and switch fabrics with a 2:1 speed advantage. We provide resource consumption results for each component of the Packet Processor design.

System-on-Chip Packet Processor for an Experimental Network Services Platform

David Taylor, Alex Chandra, Yuhua Chen, Sarang Dharmapurikar,
John Lockwood, Wenjing Tang, Jonathan Turner
Washington University in St. Louis

***Abstract*— As the focus of networking research shifts from raw performance to the delivery of advanced network services, there is a growing need for open-platform systems for extensible networking research. The Applied Research Laboratory at Washington University in Saint Louis has developed a flexible Network Services Platform (NSP) to meet this need. The NSP provides an extensible platform for prototyping next-generation network services and applications. This paper describes the design of a system-on-chip Packet Processor for the NSP which performs all core packet processing functions including segmentation and reassembly, packet classification, route lookup, and queue management. Targeted to a commercial configurable logic device, the system is designed to support gigabit links and switch fabrics with a 2:1 speed advantage. We provide resource consumption results for each component of the Packet Processor design.**

I. Motivation

PERFORMANCE has been a primary focus of networking research for the past twenty years. At one time, there was a large gap between raw network performance and both the potential of the underlying technology and the needs of emerging multimedia applications. Advances in the field have largely closed the raw performance gap leading to a fundamental shift in the focus of networking research to enabling advanced network services. The key to advanced network services is the development of *extensible network technologies* that can serve a wide range of application requirements. Developing structured, yet flexible extension mechanisms for advanced network services is a key challenge. Structured network extensions use policies specified by network administrators or user-initiated sessions to apply *plugins* to packet flows. We believe that this approach can make extensible networks viable by providing a supportive environment for application developers and providing high-performance via executable code and reconfigurable hardware. We argue that structured extension mechanisms provide a better foundation for advanced application delivery than the capsule-based active networking model that has received much attention in recent years [1].

II. System Overview

In order to facilitate research efforts in extensible networks we have developed the Network Services Platform (NSP), an open-platform extensible router capable of supporting next-generation applications. A logical view of the NSP port architecture is shown in Figure 1. Each port of the NSP comprises a Packet Processor and one or more Processing Elements. To support IP-over-ATM interfaces, the Packet Processor supports reassembly on the ingress path. Packets then undergo classification and route lookup. Based on the results of this step, packets are queued for transmission to either the output port associated with the next hop link or the application plugin associated with a flow identifier in the processing element. Packets destined for the switch fabric are queued in virtual output queues and scheduled using a *Distributed Queuing* (DQ) mechanism [2]. The switch fabric carries packets in fixed length cells, requiring segmentation of packets when sending them through the fabric and reassembly on the output side. For more details on application plugins and architectures for software and hardware processing elements, we refer the reader to our previous work [3][4][5].

Egress processing is very similar to ingress processing. Packets received from the switch must be reassembled, since their constituent cells may become interleaved with cells of other packets from different input ports. Egress packets do not require route lookup, but must be classified in order to support egress plugin processing and reserved bandwidth flows. Packets belonging to outgoing flows with reserved bandwidth are queued in *rate controlled* per flow queues. Best-effort packets are distributed across a set of datagram queues. This provides a degree of traffic isolation and yields better performance for backlogged TCP traffic. Packets may be processed by plugins at both ingress and egress ports.

In current high-performance routers, packet processors typically include separate devices for segmentation and reassembly (SAR) or framing, packet classification, route lookup, queuing and scheduling. Even systems employing network processors typically require additional devices for packet classification and route lookup. The remainder of this paper will focus on the design and implementation of a system-on-chip Packet Processor for the NSP. To our knowledge, this is the first open source Packet Processor design capable of supporting reserved flows and advanced application plugins in attached processing elements. The NSP leverages components from earlier research systems developed at Washington University in Saint Louis. All components of the implementation platform are described in Section IV.

III. Packet Processor Architecture

The Packet Processor is an efficient system-on-chip supporting all necessary packet processing functions for the NSP. As shown in Figure 2, the Packet Processor minimizes

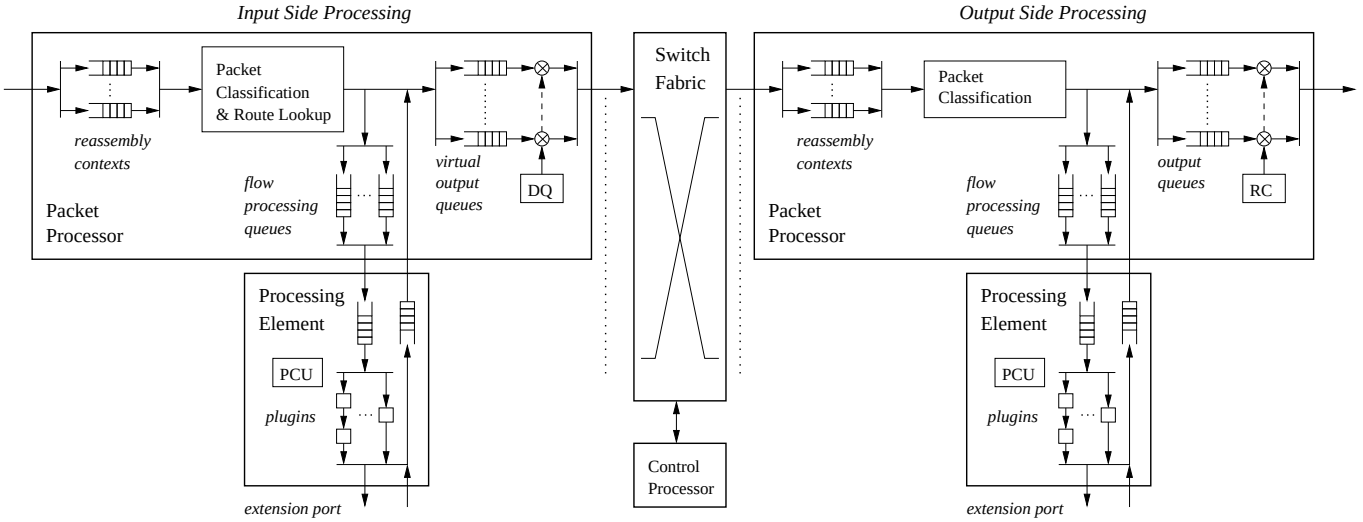

Fig. 1. Logical view of the Network Services Platform (NSP) port architecture.

data movement by storing packets in off-chip memory then forwarding copies of packet headers to the Classification and Route Lookup (CARL) block. All control cells are forwarded to the Control Cell Processor (CCP) which manages route and filter tables, the register set, and distributed queuing status and updates. Also note that both ingress and egress traffic share the same processing path making the design more efficient, especially for multicast flows.

All traffic arrives at the *Input Segmentation and Reassembly* (ISAR) block in the form of cells. The ISAR extracts the payloads of cells to form IP packets. Because the NSP supports several virutal interfaces for each link, cells from different packets can be interleaved on the link. To handle this, the ISAR maintains multiple reassembly contexts. For traffic arriving from the virtual interfaces of the link, the ISAR must insert an NSP shim, an internal packet header used to communicate information regarding packet handling throughout the NSP. The contents and functionality of the shim are explained in Section III-D. The ISAR buffers and writes fixed size chunks of arriving packets to the *Packet Storage Manager* (PSM). The PSM passes a packet pointer to the ISAR which uniquely identifies the packet in memory. A more detailed description of the PSM is provided in Section III-A.

Once an entire packet has been received by the ISAR, the packet pointer, shim fields, and packet header fields are forwarded to the *Classification and Route Lookup* (CARL) block. The search algorithms used by CARL are briefly discussed in Section III-B. Upon completion of a lookup, CARL updates the shim fields and, if necessary, makes copies of the packet pointer and shim fields. Multiple copies are required in the case of multicast flows or a non-exclusive filter match for network monitoring. Note that only one copy of the packet is stored in SDRAM, while multiple copies of the packet pointer and shim fields may by stored in the *Queue Manager* (QM). Based on the shim fields, the QM decides in which queue to place the packet. The QM participates in a *distributed queueing algorithm* [2] to determine appropriate rates for its virtual output queues. It implements rate-controlled per-flow

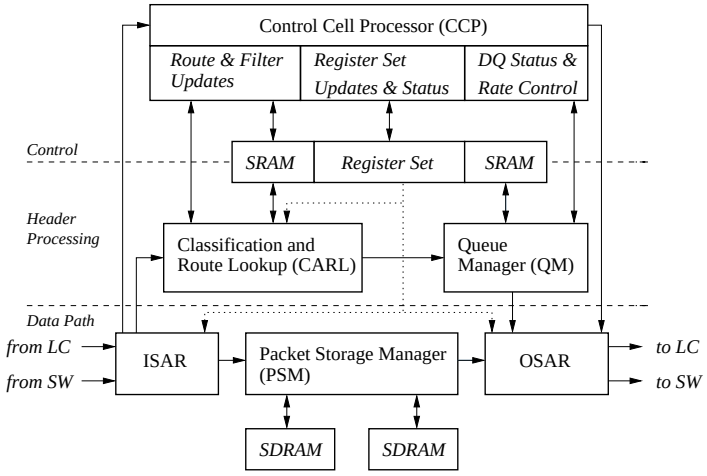

Fig. 2. Block diagram of Packet Processor.

queues for the outgoing link and dynamically regulates packet flows to the processing elements in order to keep them busy, without exceeding their processing capacity.

The packet pointer and shim fields of the next outgoing packet are sent to the OSAR. The OSAR retrieves the packet from the PSM, formats a frame, and transmits the cells of the frame to either the switch (SW) or line card (LC) interface. The OSAR also removes NSP shims from the headers of packets transmitted to the link. For packets with copy counts greater than one, the PSM keeps track of how many copies of the packet have been sent. The following sub-sections provide more detailed descriptions of the key components of the Packet Processor and discuss inter-module communication and multicast traffic support.

A. Packet Storage Manager (PSM)

The *Packet Storage Manager* (PSM) buffers variable length IP packets in off-chip SDRAM. Memory is used efficiently by dynamically allocating and deallocating fixed sized chunks based on packet size. A list of pointers to free chunks is

maintained in a FIFO data structure. As shown in Figure 3, the PSM is partitioned into four sub-modules: *Chunk Writer*, *Packet Reader*, *Free List Manager*, and *SDRAM Controller*. *Chunk Writer* allocates memory to incoming packets by pulling chunks from the free list. It links chunks of the same packet into a linked list. When a complete packet is buffered, it returns a pointer to the first chunk as a packet pointer. When a packet is to be read out, the corresponding packet pointer is supplied to the *Packet Reader* which reads all the chunks from SDRAM, by following the linked list. Chunks allocated to unicast packets are released. Multicast packets are read multiple times; hence, the chunks allocated to multicast packets are not freed until all copies are read. Whenever chunks associated with a packet are released, the corresponding pointers are appended to the free chunk pointer list.

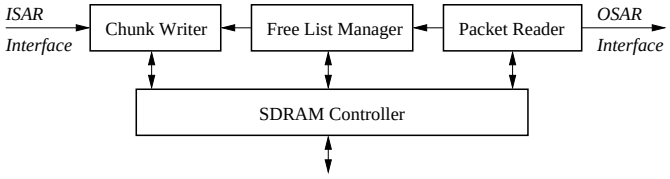

Fig. 3. Block diagram of the Packet Storage Manager (PSM)

The *Free List Manager* appends free chunk pointers released by the *Packet Reader* and supplies free chunk pointers to the *Chunk Writer* as needed. Since the free chunk pointer list is too big to be kept on-chip, it is maintained in SDRAM with the stored packets. Pulling and appending chunk pointers to the free list in SDRAM involves considerable latency and can become a potential bottleneck; therefore, a portion of this list is cached on-chip and used as a mini-list. When chunks are needed, they are taken from the mini-list and when the chunks are released, they are appended to the mini-list. Under balanced traffic conditions, operations on the list maintained in the SDRAM can be bypassed. When the cached mini-list grows beyond a threshold, some of the pointers in it are sent to the list maintained in the SDRAM. Similarly, when the mini-list depletes below a threshold, some pointers are fetched from the SDRAM list. It is important to note that these cases only will occur during unbalanced traffic patterns and the SDRAM bandwidth used to transfer the free list pointers is "free".

Since each of these modules interacts with a single SDRAM, the *SDRAM Controller* must arbitrate read/write transactions of three modules. Sequential accesses to the same SDRAM bank, referred to as a *bank conflict*, results in high access latency. The *SDRAM Controller* attempts to avoid bank conflicts by scheduling simultaneous requests from the three modules. While this does not preclude worst-case access patterns, it improves average case performance without adding significant complexity.

B. Classification and Route Lookup (CARL)

The *Classification and Route Lookup* block (CARL) determines the processing and queuing actions to be performed for each packet based on the packet header fields received from the ISAR. A Queue Identifier (QID) for a specific transmission or processing queue is passed to the Queue Manger (QM) along with the packet pointer and other shim fields upon completion of a lookup. As shown in Figure 4, CARL employs three distinct classification blocks. *Route Lookup* performs a Longest Prefix Match (LPM) on the IPv4 destination address of ingress packets and returns the output port and interface for the packet. An implementation of Eatherton and Dittia's Tree Bitmap algorithm, referred to as Fast IP Lookup (FIPL), is used to search the set of prefixes in the route table [6][7]. Tree Bitmap represents the set of stored prefixes as a compressed multi-bit trie and supports fast incremental updates. The data structure, next hops, and per-prefix packet counters are stored in an off-chip SRAM shared between *Route Lookup* and *Exact Filter Match*. Performance measurements of FIPL resulted in a storage requirement of 6.3 bytes per prefix and performance of over one million lookups per FIPL engine. The FIPL design supports up to eight parallel lookup engines sharing a single memory interface. Each FIPL engine performs one off-chip memory access every eight clock cycles. The memory controller in CARL interleaves the memory accesses from a pair of FIPL engines and a pair of exact filter match engines.

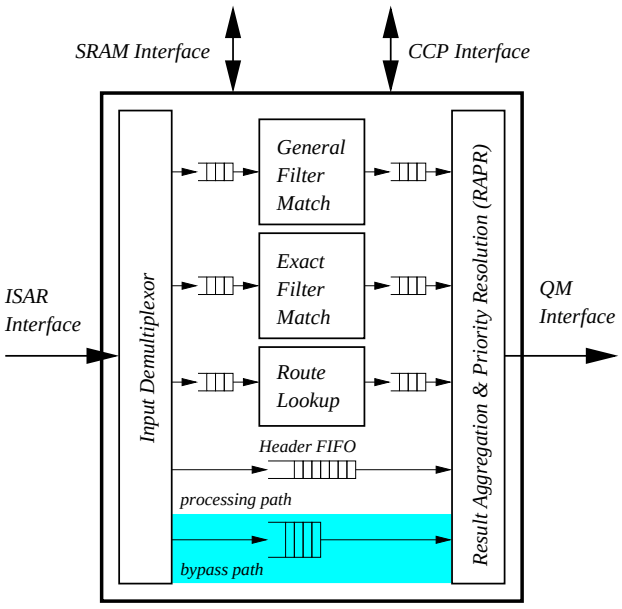

Fig. 4. Block diagram of CARL.

Exact Filter Match lookup block performs an exact match on the packet 5-tuple[1] of ingress and egress packets. Exact filters are used for reserved bandwidth flows and multicast flows. The search technique employs a hash lookup with chaining to resolve collisions. A hash key based on low-order bits of the source and destination address is used to probe an on-chip hash table containing "ingress valid" and "egress valid" bits. If the appropriate bit for the packet being processed is set, the hash key is used to index a table in off-chip SRAM. Off-chip table entries can be chained together in linked lists if multiple filters hash to the same key value. A pair of exact

[1]The packet 5-tuple refers to the IP source and destination address, the transport protocol, and the transport source and destination ports.

match search engines operate in parallel and each exact match search engine performs three memory accesses every eight clock cycles.

The *General Filter Match* block performs a five-dimensional filter match over the packet 5-tuple. This type of match consists of a LPM on the source and destination IP address, range match on the source and destination transport ports, and exact match on the protocol field; however, any field may be unspecified or "Don't Care". General filter matches may be performed on ingress only, egress only, or both ingress and egress packets. General filters are typically used for network management and security; hence, our research system is designed to handle a modest set of 32 filters. A set of parallel search engines linearly probes a wide on-chip memory containing the filter fields, priority, and QID. To support transparent network monitoring, filters may be *exclusive* or *non-exclusive*, and a search returns the highest priority matching filter of each type.

The *Input Demultiplexor* feeds the appropriate packet fields to the appropriate input FIFOs of the search engines, while a copy of the header fields and packet pointer are placed in the header FIFO. Packets not requiring classification or lookup are placed in the bypass FIFO. The *Result Aggregation and Priority Resolution (RAPR)* module retrieves results from the three engines, performs priority resolution, makes copies of packet headers if necessary, and forwards them to the QM.

C. Queue Manager

The *Queue Manager* (QM) manages separates sets of linked list queues for packets going to the switch fabric, the outgoing links and the Processing Elements. Packets going to the outgoing links can be placed either in a per-flow queue (for flows with explicit bandwidth reservations) or in one of a set of best-effort queues. The per flow queues are scheduled using Self-Clocked Fair Queueing [8], a relatively simple but effective virtual-based packet scheduler. To enable efficient handling of large numbers of per flow queues, the implementation uses *approximate radix sorting*, which trades off a bounded amount of inter-packet jitter to obain processing times which are independent of the number of flows. Packets that do not belong to reserved flows are distributed among one of 64 best effort queues. The use of multiple queues provides traffic isolation, preventing a few greedy flows from consuming all the link bandwidth, and allows high throughput for TCP flows, without the large amounts of buffering (and delay) typically required by internet routers. The improved TCP performance is obtained using *Queue State Deficit Round Robin* (QS-DRR) packet scheduler [9], which effectively desynchronizes backlogged TCP flows by adding hysteresis to the packet discard policy. This dramatically reduces queue fluctuations and greatly improves fairness among flows with different round trip times.

Packets going to the Processing Elements are dynamically regulated to limit the number of packets queued for processing within the processing elements. Packets are forwarded to the Processing Elements according to a schedule determined by the processing capacities reserved for different flows.

Packet rates between the switch input ports and output ports are regulated using a distributed queuing algorithm [2]. In order to support distributed queueing, the QM maintains a separate set of queues for each outgoing link of the NSP. Each set includes per-flow queues for flows with reserved bandwidth and a separate queue for best-effort traffic. The distributed queueing algorithm periodically distributes information about total reserved bandwidth and backlogs to all ports. The QMs at each port use this information to adjust the rates at which they forward packets to each port. The algorithm seeks to move packets through the switch fabric as quickly as possible while avoiding congestion in the fabric and ensuring that bandwidth reservations are respected.

D. Inter-module Communication

To pass packet state information between functional blocks of the NSP, a custom header field, called a shim, is used. The shim is added to packets arriving from the links and deleted from packets prior to transmitting them on the link. Shims are used to carry information between ports as well as between components of the Packet Processor, hence a shim may assume one of two formats. InterPort shims carry information between ports of the NSP, while IntraPort shims carry information between components of the Packet Processor and the processing element(s). InterPort shims contain fields for the input port number and interface, output port number and interface, and the *Multicast Tree Position* (MTP) which will be described in the following sub-section. IntraPort shims carry those fields as well as the packet pointer, Queue Identifier (QID), queue length, and flags. Flags specify internal actions such as "drop the packet" as well as special cases such as multicast flow, no matching route or filter, and reserved bandwidth flow.

E. Support for Multicast Traffic

The NSP implements multicast using a variant of the binary tree multicast algorithm described in [10]. In this approach a multicast flow is broken down into binary copy steps and processed in multiple passes. To integrate multicast traffic with the virtual output queueing used for unicast traffic, the binary replication steps are done in the Packet Processors. The Multicast Tree Position (MTP) field allows one Packet Processor to store multiple filters for a single multicast session; this allows one port to participate in multiple binary copy steps. The MTP field is included in every shim, but is only used for multicast packets. This 8-bit field denotes the position of a packet or filter in the binary tree. The initial bits of the MTP field identify the left/right branches in the path from the root of the tree to a given tree node. Because the number of bits needed to identify such a path is variable, we add "padding" bits to complete the 8 bit field. The padding bits consist of a single 0 followed by zero or more 1's. With this encoding, the bits that specify the path are those bits that precede the last 0. Upon receiving a packet from the link, the ISAR initializes the MTP of the packet to 01111111. The CARL checks the packet's MTP field against the MTP field in the multicast flow filters in order to determine a match. When the CARL passes

a packet to the QM, it includes a Multicast Branch (MB) bit indicating if it is the 0 or 1 branch of a multicast tree. The QM stores the MB bit received from the CARL in its per-queue data structure. Prior to transmitting the packet, the OSAR modifies the MTP field in the shim by inserting the MB bit between the currently significant bits of the MTP field and the padding bits (which are are shifted to the right by one position).

IV. Implementation

The Network Services Platform leverages components from earlier open-platform research systems developed at Washington University in Saint Louis. The Washington University Gigabit Switch (WUGS), an eight port ATM switch based on a multi-stage Benes topology, provides a high-performance switch fabric [10]. Each port of the WUGS can be fitted with a Field-programmable Port eXtender (FPX), a flexible platform for various packet processing applications [11]. Among other functionality, the FPX provides an in-datapath FPGA with access to two SRAM and two SDRAM devices. A second FPX can be added to the port to implement hardware plugins [5]. Software plugins are hosted on the Smart Port Card (SPC), an add-on card with an embedded microprocessor and network interface ASIC [12].

The target implementation device for the Packet Processor is the Xilinx Virtex 2000E-6 FPGA of the FPX. While the flexibility of this device is ideal for studying multiple queuing and scheduling algorithms, its available resources and performance posed significant challenges for the Packet Processor design. Based on performance measurements and experience with the target device, the target clock frequency for the design was set at 75MHz. Given the target device performance and worst-case traffic analysis, the Packet Processor was designed to support a 1 Gb/s link, a 2:1 switch speedup, and a 200 Mb/s processing element interface for a total throughput of 6.4 Gb/s.

System specification required approximately 40,000 lines of VHDL code. The per-component normalized resource utilization is shown in Figure 5. Total resource usage is 20,302 flip-flops, 22,033 4-LUTs (4-input lookup tables), and 125 BlockRAMs (4096 bit dual-port embedded memories). Mapping these resources to the target device resulted in 99% cell usage with 18% of the cells containing unrelated logic.

V. Conclusions

The Packet Processor provides segmentation and reassembly, packet classification, route lookup, and queue management support for an open-source, experimental Network Services Platform. Combined ingress and egress processing, split header and payload paths, minimal data movement, and efficient lookup and scheduling algorithms result in an efficient system-on-chip design. The design is implemented using a commodity configurable logic device for use with open-platform research systems developed at Washington University in Saint Louis. In conjunction with hardware and software processing elements, the Packet Processor enables research on next-generation network services and applications using the NSP.

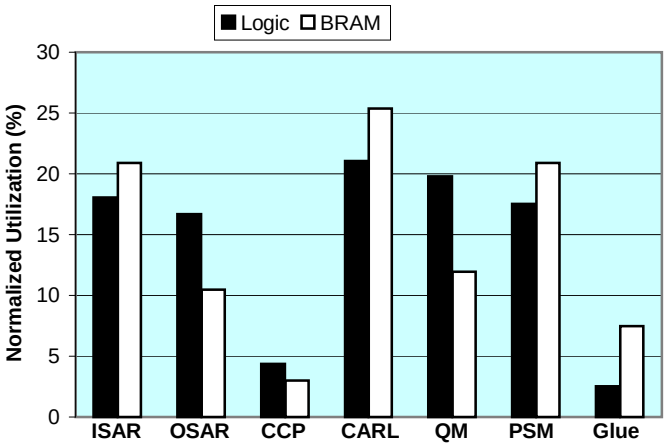

Fig. 5. Normalized per-component resource usage of the Packet Processor.

Acknowledgments

We would like to thank John DeHart for his invaluable assistance with system verification. We also would like to thank Fred Kuhns for his assistance with system design and integration.

References

[1] D. L. Tennenhouse et al., "A Survey of Active Network Research," in *IEEE Communications Magazine*, pp. 80–86, Jan. 1997.

[2] P. Pappu, J. Parwatikar, J. Turner, and K. Wong, "Distributed queueing in scalable high performance routers," in *Proceedings of IEEE Infocom*, April 2003.

[3] S. Choi, J. Dehart, R. Keller, J. W. Lockwood, J. Turner, and T. Wolf, "Design of a flexible open platform for high performance active networks," in *Allerton Conference*, (Champaign, IL), 1999.

[4] T. Wolf and J. Turner, "Design Issues for High Performance Active Routers," *IEEE Journal on Selected Areas of Communications*, vol. 19, pp. 404–409, March 2001.

[5] D. E. Taylor, J. S. Turner, J. W. Lockwood, and E. L. Horta, "Dynamic Hardware Plugins (DHP): Exploiting Reconfigurable Hardware for High-Performance Programmable Routers," *Computer Networks*, vol. 38, pp. 295–310, February 2002. Elsevier Science.

[6] W. N. Eatherton, "Hardware-Based Internet Protocol Prefix Lookups." thesis, Washington University in St. Louis, 1998. Available at `http://www.arl.wustl.edu/`.

[7] D. E. Taylor, J. S. Turner, J. W. Lockwood, T. S. Sproull, and D. B. Parlour, "Scalable IP Lookup for Internet Routers," *IEEE Journal on Selected Areas in Communications*, 2003.

[8] S. Golestani, "A self-clocked fair queuing scheme for high speed applications," in *Proceedings of IEEE INFOCOM*, 1994.

[9] A. Kantawala and J. Turner, "Queue Management for Short-Lived TCP Flows in Backbone Routers," in *Proceedings of High-Speed Symposium*, Globecom 2002.

[10] J. S. Turner, T. Chaney, A. Fingerhut, and M. Flucke, "Design of a Gigabit ATM switch," in *INFOCOM'97*, 1997.

[11] J. W. Lockwood, J. S. Turner, and D. E. Taylor, "Field programmable port extender (FPX) for distributed routing and queuing," in *ACM International Symposium on Field Programmable Gate Arrays (FPGA'2000)*, (Monterey, CA, USA), pp. 137–144, Feb. 2000.

[12] J. D. DeHart, W. D. Richard, E. W. Spitznagel, and D. E. Taylor, "The smart port card: An embedded Unix processor architecture for network management and active networking," Tech. Rep. WUCS-01-18, Applied Research Laboratory, Department of Computer Science, Washington University in Saint Louis, August 2001.