# Design and Evaluation of a High-Performance Dynamically Extensible Router [*]

Fred Kuhns, John DeHart, Anshul Kantawala, Ralph Keller, John Lockwood, Prashanth Pappu,
David Richard†, David Taylor†, Jyoti Parwatikar, Ed Spitznagel, Jon Turner, and Ken Wong
Department of Computer Science, †Department of Electrical Engineering and
The Applied Research Laboratory, Washington University in St. Louis, USA
{fredk,jdd,anshul,keller,lockwood,prashant,wdr,det3,jp,ews1,jst,kenw}@arl.wustl.edu

## Abstract

*This paper describes the design, implementation and performance of an open, high performance, dynamically extensible router under development at Washington University in St. Louis. This router supports the dynamic installation of software and hardware plugins in the data path of application data flows. It provides an experimental platform for research on programmable networks, protocols, router software and hardware design, network management, quality of service and advanced applications. It is designed to be flexible without sacrificing performance. It supports gigabit links and uses a scalable architecture suitable for supporting hundreds or even thousands of links. The system's flexibility makes it an ideal platform for experimental research on dynamically extensible networks that implement higher level functions in direct support of individual application sessions.*

## 1. Introduction

In the last decade, the Internet has undergone a fundamental transformation, from a small-scale network serving academics and select technology companies to a global infrastructure serving people in all walks of life and all parts of the world. As the Internet has grown, it has become more complex, making it difficult for researchers and engineers to understand its behavior and that of its many interacting components. This increases the challenges faced by those seeking to create new protocols and technologies that can potentially improve the Internet's reliability, functionality and performance. At the same time, the growing importance of the Internet is dramatically raising the stakes. Even small improvements can have a big payoff.

In this context, experimental studies aimed at understanding how Internet routers perform in realistic network settings are essential to any serious research effort in Internet technology development. Currently, academic researchers have two main alternatives for experimental research in commercial routers and routing software. With commercial routers, researchers are using state-of-the-art technology but are generally limited to treating the router as a black box with the only access provided by highly constrained management interfaces. The internal design is largely hidden and not subject to experimental modification. The other alternative for academic researchers is to use routing software running on standard computers. Open source operating systems, such as Linux and NetBSD, have made this a popular choice. This alternative provides the researcher with direct access to all functionality and provides complete extensibility but lacks most of the advanced architectural features found in commercial routers, making such systems unrealistic.

The growing performance demands of the Internet have made the internal design of high performance routers far more complex. Routers now support large numbers of gigabit links and use dedicated hardware to implement many protocol processing functions. Functionality is distributed among the line cards that interface to the links, the control processors that provide high level management and the interconnection network that moves packets from inputs to outputs. The highest performance systems use multistage interconnection networks capable of supporting hundreds or even thousands of 10 Gb/s links. To understand how such systems perform, one must work with systems that have the same architectural characteristics. A single processor with a handful of relatively low speed interfaces uses an architecture which is both quantitatively and qualitatively very different. The kinds of issues one faces in systems of this sort are very different from the kinds of issues faced by designers of modern high performance routers. If academic research is to be relevant to the design of such systems, it needs to be supported by systems research using compara-

[*] This work was supported in part by NSF grants ANI-0096052, ANI-9714698, ANI-9813723, ANI-9616754 and DARPA grant N66001-98-C-8510.

ble experimental platforms.

This paper describes a highly flexible, dynamically extensible router that is under development at Washington University. It provides an ideal platform for advanced networking research in the increasingly complex environment facing researchers and technology developers. It is built around a switch fabric that can be scaled up to thousands of ports. While typical research systems have small port counts, they do use the same parallel architecture used by much larger systems, requiring researchers to address in a realistic way many of the issues that arise in larger systems. The system has embedded, programmable processors at every link interface, allowing packet processing at these interfaces to be completely flexible and allowing dynamic module installation that can add specialized processing to individual application data flows. An extension to the system architecture, which is now in progress, will enable all packet processing to be implemented in reconfigurable hardware, allowing wire-speed forwarding at gigabit rates. The design of all software and hardware used in the system is being placed in the public domain, allowing it to be studied, modified and reused by researchers and developers interested in advancing the development of open, extensible, high performance Internet routers.

This paper contains two parts. Part I (Sections 2 to 4) describes the architecture of Washington University's Dynamically Extensible Router (DER). Section 2 describes the overall system architecture and some novel hardware components. The principle configurations are distinguished by the type of hardware performing the port processing functions. In the simplest configuration, all port processors are Smart Port Cards (SPCs) which contain a general-purpose processor. This SPC-only system can be extended by off loading most of the traditional IP processing functions to a reconfigurable hardware module called the Field Programmable port eXtender (FPX). In this FPX/SPC system, the SPC handles special processing such as active packets and some IP options. Section 3 describes the design and implementation of system-level processing elements and some of the design issues related to its distributed architecture. Section 4 describes the processing done at the port processors in an SPC-only system.

Part II (Sections 5 to 9) describes the use and evaluation of the DER. Section 5 describes our experience with an active networking application that exploits the system's extensibility. Section 6 describes performance measurements of packet forwarding speed on an SPC-only prototype system. The measurements quantify the ability of an SPC-only system to forward packets and provide fair link access. Section 7 describes the evaluation of our combined packet queueing and scheduling algorithm called Queue State Deficit Round Robin (QSDRR). Section 8 describes the evaluation of our distributed queueing algorithm which is aimed at maximizing output link utilization with considerably lower cost than output queueing systems.

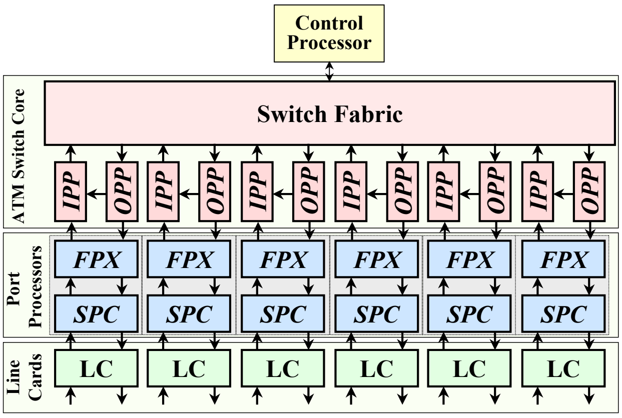


**Figure 1. DER Hardware Configuration**

Section 9 describes how an FPX/SPC system would handle active IP packets and presents preliminary measurements of its packet forwarding speed with and without active traffic. These measurements quantify the effect of extending an SPC-only system with field programmable modules. Finally, Section 10 closes with final remarks on the current status of the system and future extensions.

# Part I: Architecture

## 2. System Overview

The Washington University Dynamically Extensible Router (DER) is designed to be a scalable, high-performance, open platform for conducting network research. It employs highly reconfigurable technology (programmable hardware and dynamic software modules) to provide high-speed processing of IP packets and can be extended to provide application-specific processing for selected flows. Because it is built around an ATM core, it can also support ATM virtual circuits. Figure 1 shows the overall architecture of the DER and its main components: Control Processor (CP), ATM switch core, Field Programmable port eXtenders (FPXs), Smart PortCards (SPCs) and Line Cards (LCs). Figure 2 shows one DER unit with its cover off in the right photograph.

The main function of the router is to forward packets at a high speed from its input side to its output side. The system uses a multistage interconnection network with dynamic routing and a small internal speed advantage (i.e., the internal data paths can forward packets at a faster rate

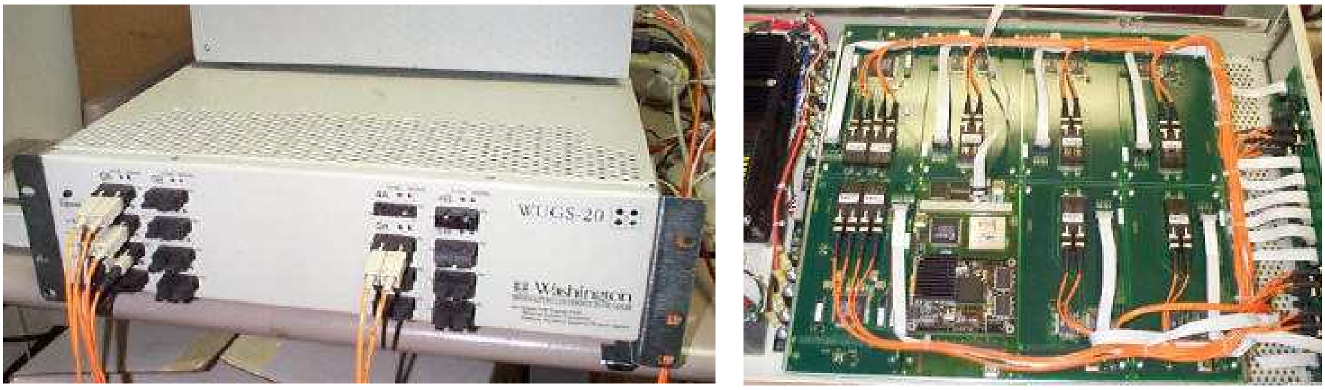


**Figure 2. Washington University Dynamically Extensible Router**

than the external links) to connect the input side Port Processors (PPs) to the output side PPs. Our use of an ATM switch core is a typical approach used by commercial systems where IP is often implemented over ATM in order to get the benefits of cell-switching. A PP can include either a Field Programmable port eXtender (FPX) or a Smart PortCard (SPC) or both. These PPs perform packet classification, route lookup and packet scheduling.

The system employs a number of interesting techniques aimed at achieving high performance and flexibility. A *distributed queueing* algorithm is used to gain high throughput even under extreme overload. A *queue state deficit round robin* packet scheduling algorithm is used to improve average goodput (throughput excluding retransmissions) and goodput variability with less memory utilization. The PPs include an efficient dynamic plugin framework that can provide customized services that are integrated into the normal packet processing path. The PPs also use a packet classification algorithm that can run at wire speed when implemented in hardware. The CP runs open source route daemons that support standard protocols such as OSPF as well as the DER's own flow-specific routing protocol. Furthermore, the key router functions are efficiently distributed among its hardware components by exploiting the high bandwidth and connection-oriented circuits provided by the ATM switch core. The remainder of this section gives an overview of the DER hardware components.

### 2.1. Control Processor

The Control Processor (CP) runs software that directly or indirectly controls and monitors router functions. Some of these functions include:

- Configuration discovery and system initialization
- Resource usage and status monitoring
- Routing protocols
- Control of port-level active processing environments
- Forwarding table and classifier database maintenance
- Participation in higher level protocols for both local and global resource management

The processing associated with some of these functions is described in Sections 3 and 4. The CP is connected to one of the DER's ports and uses native ATM cells and frames (AAL0 and AAL5) to communicate with and control the individual port processors.

### 2.2. Switch Fabric and Line Cards

The DER's ATM switch core is a Washington University Gigabit ATM Switch (WUGS)[1, 2]. The current WUGS has eight ports with Line Cards (LCs) [3] capable of operating at rates up to 2.4 Gb/s and supports ATM multicasting using a novel cell recycling architecture.

Each LC provides conversion and encoding functions required for the target physical layer device. For example, an ATM switch link adapter provides parallel-to-serial, encoding, and optical-to-electrical conversions necessary for data transmission over fiber using one of the optical transmission standards, e.g., SONET. Current LCs include a dual 155 Mb/s OC-3 SONET [4] link adapter, a 622 Mb/s OC-12 SONET link adapter, and a 1.2 Gb/s Hewlett Packard (HP) G-Link [5] link adapter. A gigabit Ethernet LC is currently being designed.

### 2.3. Port Processors

Commercial switches and routers typically employ specialized integrated circuits to implement complex queueing and packet filtering mechanisms. Active processing environments that use commercial routers rely on a general-purpose processing environment with high-level language

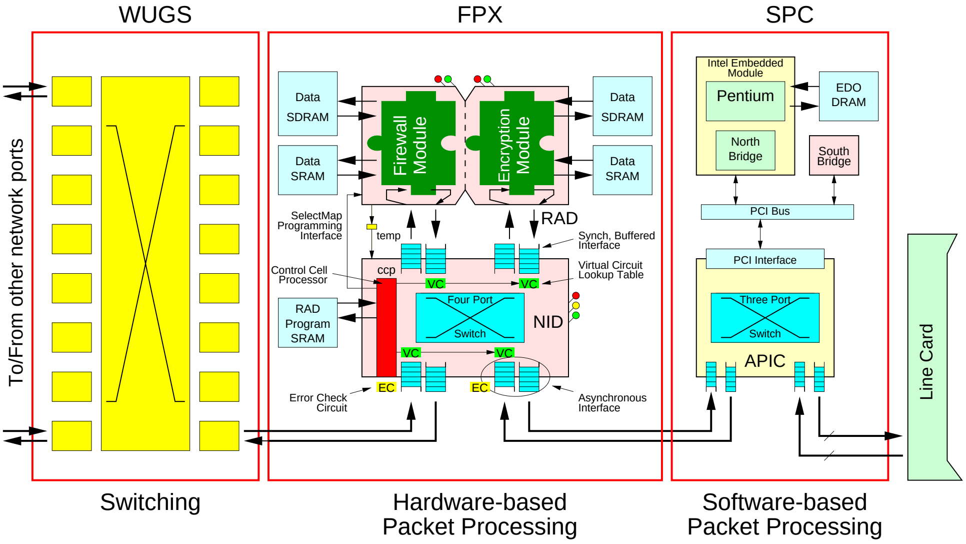


**Figure 3. An FPX/SPC Port Processor**

support such as Java and NodeOS [6, 7]. This approach results in an inflexible, high-performance standard packet forwarding path and a relatively low-performance active processing path.

The DER takes a different approach to supporting active processing and high-performance packet forwarding. The port processors are implemented using two separate devices: an embedded general-purpose computing platform called the Smart Port Card (SPC) and an embedded programmable hardware device called the Field Programmable port eXtender (FPX). Figure 3 illustrates how the SPC, FPX and WUGS are interconnected. In general, the FPX performs basic IP packet processing, forwarding and scheduling [8] and leaves non-standard processing to an SPC which acts as a network processor. This implementation approach takes advantage of the benefits of a cooperative hardware/software combination [9, 10].

While a high-performance configuration would contain both FPXs and SPCs, this is not required. The SPC can handle all IP forwarding functions (i.e., IP route lookup, flow classification, distributed queueing and packet scheduling) as well as active processing.

**Smart Port Card (SPC):** As shown in Figure 4, the Smart Port Card (SPC) consists of an embedded Intel processor module, 64 MBytes of DRAM, an FPGA (Field Programmable Gate Array) that provides south bridge functionality, and a Washington University APIC ATM host-network interface [11]. The SPC runs a version of the NetBSD operating system [12] that has been substantially modified to support fast packet forwarding, active network processing and network management.

The Intel embedded module contains a 166 MHz Pentium MMX processor, north bridge [13] and L2 cache. (We are currently developing an SPC 2, a faster version of this module, which will replace the current SPC.) The “System FPGA” provides the functionality of the south bridge chip [14] found in a normal Pentium system and is implemented using a Xilinx XC4020XLA-08 Field Programmable Gate Array (FPGA) [15]. It contains a small boot ROM, a Programmable Interval Timer (PIT), a Programmable Interrupt Controller (PIC), a dual UART interface, and a modified Real Time Clock (RTC). See [16] for additional details.

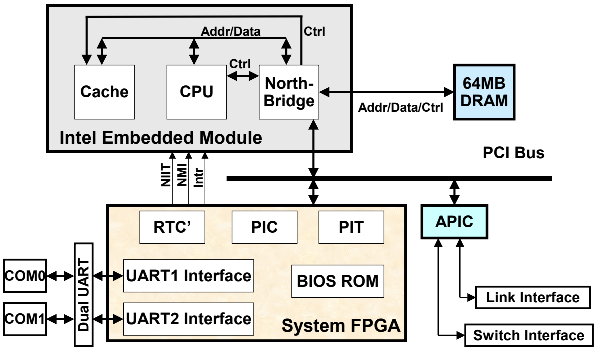


**Figure 4. Block Diagram of the Smart Port Card (SPC)**

On the SPC, ATM cells are handled by the APIC [17, 18]. Each of the ATM ports of the APIC can be independently operated at full duplex rates ranging from 155 Mb/s to 1.2 Gb/s. The APIC supports AAL-5 and is capable of performing segmentation and reassembly at maximum bus rate (1.05 Gb/s peak for PCI-32). The APIC directly transfers ATM frames to and from host memory and can be programmed so that cells of selected channels pass directly from one ATM port to another.

We have customized NetBSD to use a disk image stored in main memory, a serial console, a self configuring APIC device driver and a “fake” BIOS. The fake BIOS program acts like a boot loader: it performs some of the actions which are normally done by a Pentium BIOS and the NetBSD boot loader during power-up.

**Field Programmable Port Extender (FPX):** The FPX is a programmable hardware device that processes packets as they pass between the WUGS backplane and the line card (shown in the middle of Figure 3). All of the logic on the FPX is implemented with two FPGA devices: the Network Interface Device (NID) and the Reprogrammable Application Device (RAD) [8]. The FPX is implemented on a 20 cm × 10.5 cm printed circuit board that interconnects the FPGAs with multiple banks of memory.

The Network Interface Device (NID) controls how packets are routed to and from its modules. It also provides mechanisms to load hardware modules over the network. These two features allow the NID to dynamically load and unload modules on the RAD without affecting the switching of other traffic flows or the processing of packets by the other modules in the system [19].

As shown in the lower-center of Figure 3, the NID has several components, all of which are implemented on a Xilinx Virtex XCV-600E FPGA device. It contains: 1) A four-port switch to transfer data between ports; 2) Flow look-up tables on each port to selectively route flows; 3) An on-chip *Control Cell Processor* to process control cells that are transmitted and received over the network; 4) Logic to reprogram the FPGA hardware on the RAD; and 5) Synchronous and asynchronous interfaces to the four network ports that surround the NID.

A key feature of the FPX is that it allows the DER to perform packet processing functions in modular hardware components. As shown in the upper-center of Figure 3, these modules are implemented as regions of FPGA logic on the RAD. A standard interface has been developed that allows a module to process the streaming data in the packets as they flow through the module and to interface with off-chip memory [20]. Each module on the RAD connects to one Static Random Access Memory (SRAM) and to one wide Synchronous Dynamic RAM (SDRAM). In total, the modules implemented on the RAD have full control over four independent banks of memory. The SRAM is used for applications that need to implement table lookup operations such as the routing table for the Fast IP Lookup (FIPL) module. The other modules in the system can be programmed over the network to implement user-defined functionality [21].

## 3. System-Level Processing

Figure 5 gives a global view of an SPC-only DER showing some of the functions along the control and data paths. Section 9 describes how the SPC works with the FPX in an FPX/SPC system. This section describes the activities that involve the Control Processor (CP) and its interaction with the PPs. The software framework running on the CP supports system configuration, status monitoring, PP (SPC and FPX) control, resource management, and routing. The managed resources include output link bandwidth, SPC buffers, active plugin environment, route tables and classifier rules.

### 3.1. Internal Communication

Communication among the DER’s distributed components is built on top of a set of ATM virtual circuits using VCI (Virtual Circuit Identifier) allocation rules that simplify their use. Also, the CP uses the concept of *virtual interfaces* to easily segregate traffic arriving from the PPs. It should be pointed out that using VCIs to segregate traffic has several practical consequences:

- Allows for early traffic demultiplexing in the SPC thereby reducing the per packet/cell processing overhead [22];
- IP packets and AAL5 control traffic may be sent simultaneously on different VCs without the possibility of interleaving ATM cells;
- Provides a simple abstraction and mechanism for implementing virtual interfaces (a physical interface may have multiple virtual interfaces, each associated with a different VCI).

The VCI space is partitioned into into three groups, one for each of the three traffic types:

1) *ATM Device Control:* The CP sends AAL0 cells to configure and monitor the various ATM specific devices in the DER (e.g., FPX, SPC ATM interface (APIC), ATM switch).

2) *IP Datagrams:* IP traffic travels from input PPs to output PPs using specific VCIs. The VCI allocation rules for inter-PP communication simplifies the identification of the sending and receiving ports by treating the VCI as a “tag”. For example, input PP $i$ sends to output PP $j$ on VCI ($j + 40$). The ATM switch translates the VCI so that PP $j$ receives the same frame on VCI ($i + 40$). Here, 40 is the base VCI for transiting IP traffic. Other traffic types use different base VCIs.

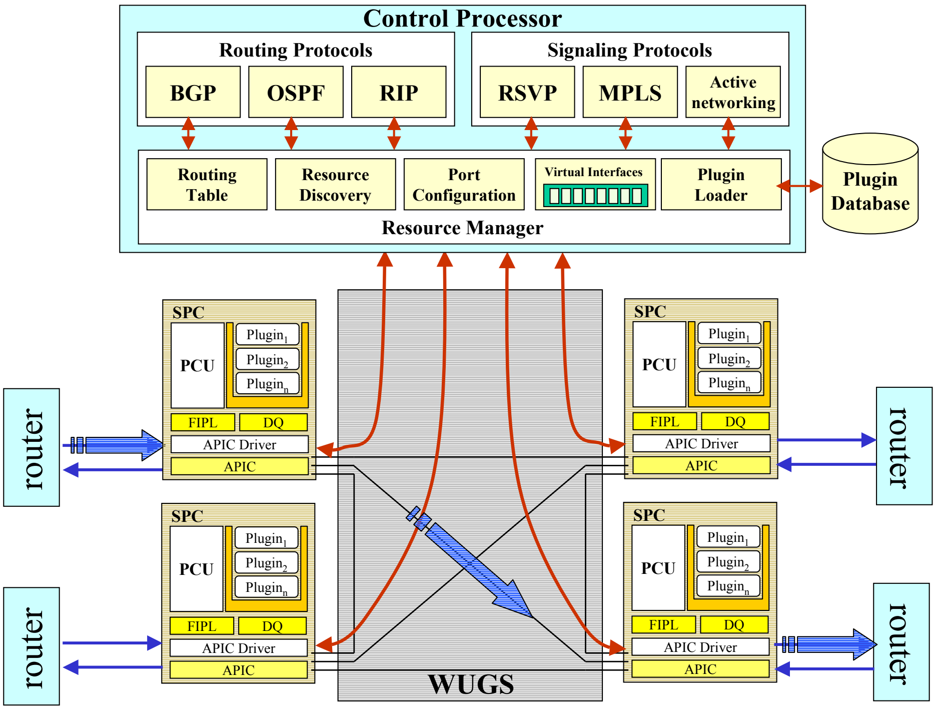


**Figure 5. DER Logical View**

3) *Command and Management Traffic:* A similar approach is used to send commands from the CP to the PPs. For example, the CP sends commands to SPC $j$ on VCI $(j + 84)$ while each SPC listens for these commands on VCI 62.

### 3.2. System Configuration

System initialization sets up communication paths among all processors (CP, PPs) and initializes the PPs with instructions and data. This boot process is multi-tiered. First, it performs a low-level initialization sequence so that the CP can communicate with the PPs. Next, it discovers the number, location, and types of computing resources and links at each port.

The following sequence is executed:

**Configuration:** The CP controls the operation of its PPs using ATM control cells. Therefore, communication must be established between the CP and its PPs even before the discovery step can be performed. The CP sets up predefined ATM virtual circuits (VCs). These include VCs for control cells, for program loading and for forwarding IP packets from input ports to output ports.

**Discovery:** The CP discovers the low-level configuration of a port by sending control cells in succession to the switch port processor and each of the adjoining cards. Each processor reports the characteristics of the adjacent card. The responses indicate the type of processors at each port and the link rate.

**SPC Initialization:** The CP downloads a NetBSD kernel and memory-resident filesystem to each SPC using a multicast VC and AAL5 frames and completes each SPC's identity by sending them their port number using a DER control message.

**FPX Initialization:** Initialization of an FPX follows a similar sequence. A program and configuration is loaded into the *RAD reprogram memory* under control of the NID using control cells. Once the last cell has been successfully loaded, the CP sends a control cell to the NID to initiate the reprogramming of the RAD using the contents of the reprogram memory.

The DER can also dynamically alter a large number of system parameters making it ideal for an experimental setting. These changes can be performed at a full range of system levels. Examples at the policy level include selecting a packet classifier or a particular distributed queueing algorithm. Examples at the lowest levels include the enabling of debug messages and the selection of the system message recipients. The DER's control cell framework makes it easy to add and control new dynamic configuration options.

### 3.3. Packet Classification

The DER architecture is based on a relatively simple architectural model which is similar to the integrated services routing architecture described in RFC1633 [23]. Packet classification is used to determine how a packet will be handled. The CP configures and maintains the classifiers at the PPs. The CP sends a set of rules to the PPs which together with the routing table define the classifier. A rule is composed of a filter (predicate) and an action perhaps involving some action specific data (e.g., resource requirements for a reserved flow).

A filter specifies a bit pattern that is compared against the received packet header. The DER supports both longest prefix match and general match filters. A general match filter permits partial matches which include the source and destination IP address fields, the corresponding netmasks, the source and destination port number fields and the protocol field. The longest prefix match filter compares up to 104 bits from the IP and transport headers and selects the filter with longest matching prefix.

The classification function is performed by the *general filters* and the *flow/route lookup* modules at each input PP. When a packet is received at an input PP, it is classified resulting in a set of associated actions and resource allocations. Additional information associated with a packet may include a representation of the next hop's address, QoS reservations, an indication of route pinning and whether a copy of the packet should be sent to a management module. Actions supported in the DER include:

- *Deny:* Drop the packet, an action typical of a firewall implementation where a set of rules are defined to specifically list application flaws or traffic classes that must not be forwarded by the router.

- *Permit:* Forward normally: this is the default action when no other actions are specified.

- *Reserved:* Forward based on reservation: the flow has an existing resource reservation.

- *Active:* The flow is associated with an active plugin. So, the packet must be sent to the active processing environment.

### 3.4. Route Management

The DER maintains information about other routers in the network by running Zebra [24], an open-source routing framework distributed under the GNU license. Zebra supports various interior (OSPF, RIP) and exterior (BGP) routing protocols. Each individual routing protocol contributes routes to a common routing table managed by the CP. Based on this routing table, the CP computes a forwarding table for each port that it keeps synchronized with the routing table. As routing protocols receive updates from neighboring routers that modify the routing table, the CP continuously recomputes the forwarding tables and propagates the changes to each port.

The forwarding tables stored in the FPXs and SPCs use a tree bitmap structure for fast packet classification with efficient memory usage [25]. The Fast IP Lookup (FIPL) algorithm employs multibit trie data structures that are ideally suited for fast hardware implementation [8]. Section 9.1 describes the algorithm and its performance in a prototype configuration.

When the CP receives a route update from another router to add or delete a path, it creates a new internal tree bitmap structure that reflects the modified forwarding table. Then, it sends ATM control cells to the Fast IP Lookup components in the SPC or FPX representing the modifications to the multibit trie structure.

### 3.5. Signaling and Resource Management

The CP supports various signaling protocols (e.g., RSVP, MPLS) and supports active networking. The signaling protocols allow applications to make bandwidth reservations required for QoS guarantees. When a bandwidth reservation request arrives at the CP, the CP first performs admission control by checking for sufficient resources. If admission control succeeds, the CP reserves the required bandwidth on both the input and output ports and returns a signaling message to grant the reservation. If the reservation is granted, then a new rule is added to the classifier to bind the reserved flow to its reservation. The route entry may also be pinned so that packets belonging to the same flow are routed consistently. This mechanism is useful for applications with user-specified processing needs that need to transit specific network nodes in a given order.

In addition, the DER provides flow-specific processing of data streams. An active flow is explicitly set up using signaling mechanisms which specify the set of functions which will be required for processing the data stream. In the context of the DER, *plugins* are code modules that provide a specific processing function and can be dynamically loaded and configured at each port.

If an active signaling request references a plugin that has not been deployed on the router, the CP retrieves the plugin code from a remote code server, checks its digital signature and then downloads it to a PP where it is configured using a command protocol between the CP and the target PP. Once the plugin has been successfully loaded and configured, the CP installs a filter in the port's classifier so that matching packets will be routed to the plugin.

### 3.6. Distributed Queueing

Under sustained overload, the internal links of the WUGS can become congested leading to substantially reduced throughput. Our Distributed Queueing (DQ) algorithm allows the DER to perform like an output queueing system (switch fabric and output queues operate at the aggregate input rate) but without the $N$ times speed-up required by a true output queueing switch [26]. It avoids switch fabric congestion while maintaining high output link utilization. We describe the basic algorithm below but leave the discussion of refinements until Section 8.

**Mechanism:** The DQ algorithm employs a *coarse scheduling* approach in which queue backlogs at the inputs and outputs are periodically broadcast to all PPs. The DER uses *Virtual Output Queueing* [27, 28, 29] to avoid head-of-line blocking. Each input maintains separate queues for each output allowing inputs to regulate the flow of traffic to each of the outputs so as to keep data moving to the output queues in a timely fashion while avoiding congestion within the switch core and output underflow.

| | Definition |
|---|---|
| $N$ | Number of output (or input) ports |
| $S$ | Switching fabric speed-up ($S \geq 1$) |
| $L$ | External link rate |
| $B_j$ | Backlog at output $j$ |
| $B_{i,j}$ | Input $i$'s backlog to output $j$ |
| $hi_{i,j}$ | Input $i$'s share of input backlog to output $j$ |
| $lo_{i,j}$ | Input $i$'s share of total backlog to output $j$ |
| $wt_{i,j}$ | Ratio of $lo_{i,j}$ to $lo_{i,1} + ... + lo_{i,N}$ |
| $r_{i,j}$ | Allocated sending rate from input $i$ to output $j$ |

**Table 1. Distributed Queueing Parameters**

**Algorithm:** At every update period (currently 500 $\mu$sec), each PP $i$ receives backlog information from all PPs and recalculates the rates $r_{i,j}$ at which it can send traffic to each output $j$. Roughly, the algorithm computes $r_{i,j}$, the allocated sending rate from input $i$ to output $j$ by apportioning the switch capacity $S \cdot L$ based on relative backlogs so as to avoid both switch congestion and output queue underflow. Table 1 lists the parameters which appear in the DQ algorithm.

Let

$$hi_{i,j} = \frac{B_{i,j}}{\sum_h B_{h,j}} \tag{1}$$

Note that if input $i$ is limited to a rate no more than $S \cdot L \cdot hi(i,j)$, we can avoid congestion at output $j$. Now, let

$$lo_{i,j} = \frac{B_{i,j}}{B_j + \sum_h B_{h,j}} \tag{2}$$

and note that if input $i$ always sends to output $j$ at a rate at least equal to $L \cdot lo(i,j)$, then the queue at output $j$ will never become empty while there is data for output $j$ in an input queue. These two observations are the basis for the rate allocation. The allocated rates $r_{i,j}$ are:

$$r_{i,j} = S \cdot L \cdot \min(hi_{i,j}, wt_{i,j}) \tag{3}$$

where

$$wt_{i,j} = \frac{lo_{i,j}}{\sum_h lo_{i,h}} \tag{4}$$

## 4. Port-Level Processing

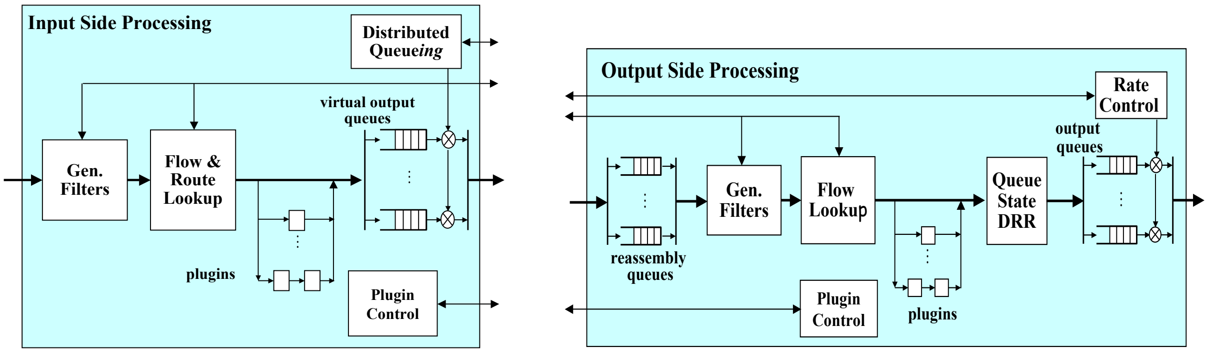
### 4.1. IP Processing

This section describes IP packet processing and the programmable network environment at the port level in an SPC-only system. When the SPC is the only PP, it must handle all input and output processing. Although ideally every port should have both an FPX to handle the typical case (e.g., no active processing or options) and an SPC to handle special cases (e.g., application-specific processing), it is desirable to have an SPC that has full port functionality for several reasons:

- *Rapid Prototyping*: A prototype DER testbed can be constructed even though the FPX is still under development.
- *Lower Cost*: A lower cost (but slower) DER can be constructed using only SPC PPs.
- *Measurement and Experience Base*: Experience with the SPC may be fruitful in the development of the FPX, and experimental features can be examined using the SPC as a preliminary step to committing to hardware. Furthermore, the acceleration benefits of using the FPX can be quantified.

The dark path in Figure 6 shows the main IP data path through the SPC kernel. In order to reduce overhead, the majority of the IP data path, active processing environment, resource management functions and command processing have been incorporated into the APIC device driver. Only functions that require periodic processing (e.g., packet scheduling and distributed queueing updates) are performed outside this path. These are performed within the clock interrupt handler which runs at a higher priority than the APIC driver.

**Code Path Selection:** As indicated in Section 3.1, VCIs are used as demultiplexing keys for incoming packets. The VCI of an incoming packet indicates to the kernel whether it is from a previous hop router, the CP or one of the connected DER input ports. A packet coming from a previous



(a) Input Side

(b) Output Side

**Figure 6. SPC IP Processing**

hop is sent to the input-side code: general-match lookup, route lookup and virtual output queue (VOQ). Route lookup is done by a software implementation of the FIPL (Fast IP Lookup) algorithm described later in Section 9.1. The Distributed Queueing (DQ) algorithm described earlier in Section 3.6 controls the VOQ drain rates.

A packet coming from an input port is sent to the output-side code and goes through a sequence similar to the input side except that it is scheduled onto an output queue by the Queue State DRR algorithm. An output queue corresponds to a *virtual interface* which may represent one or more connected hosts or routers. Active packets can be processed on either side of the DER.

**APIC Processing and Buffer Management:** The operation of an APIC reduces the load on the SPC by asynchronously performing sequences of read/write operations described by *descriptor chains*. An APIC descriptor is a 16-byte structure that describes the data buffer to be written to for receive, or read from on transmit. During initialization a contiguous chunk of memory is allocated for the descriptors (half for TX (transmit) and half for RX (receive)). The driver and APIC hardware then use a base address and index to access a particular descriptor.

During initialization, another contiguous region of memory is allocated for IP packet buffers. Each buffer is 2 KB, and there are an identical number of buffers and RX descriptors. Each buffer is bound to an RX descriptor such that their indexes are the same. Consequently, given a descriptor address or index, the corresponding RX buffer can be located simply and quickly. The reverse operation from buffer to descriptor is equally fast. This technique makes buffer management trivial, leaving only the management of the RX descriptor pool as a non-trivial task.

Since there are the same number of TX descriptors as RX descriptors, we are always guaranteed to be able to send a packet once it is received. Note that when sending a packet, the receive buffer is bound to the TX descriptor. The corresponding RX descriptor is not available for reuse until the send operation completes. This has the nice effect that the SPC will stop receiving during extreme overload avoiding unnecessary PCI and memory traffic, as well as receiver livelock.

**Queue State DRR Packet Scheduling:** The Queue State Deficit Round Robin (QSDRR) algorithm is used during output-side IP processing to provide fair packet scheduling and buffer congestion avoidance. In our prototype implementation, QSDRR can be configured so that multiple flows can share a single queue.

QSDRR adds a packet discard algorithm with hysteresis to DRR's approximately fair packet scheduling. Section 7 shows that QSDRR provides higher average goodput and lower goodput variation while using less memory than conventional algorithms such as Random Early Discard (RED) [30] and Blue [31]. Furthermore, QSDRR requires no tuning.

The basic idea behind QSDRR is that it continues to drop packets from the same queue when faced with congestion until that queue is the smallest amongst all active queues. Intuitively, this policy penalizes the least number of flows necessary to avoid link congestion. When there is short-term congestion, only a small number of flows will be affected, thus reducing the synchronization effect of the TCP flow control mechanism and reducing oscillation in the buffer occupancy.

### 4.2. Programmable Networks Environment

The SPC environment includes functionality to support both traditional IP forwarding as well as flow-specific processing. Each SPC runs a modified NetBSD kernel that provides a programmable environment for packet processing.

If a DER port is equipped with an FPX, packets are classified using the hardware implementation of the general-match lookup and Fast IP Lookup (FIPL) algorithm and sent to the SPC on a special VCI. This special VCI indicates to the SPC that the packet is already classified. If no FPX is present at a port, the packet arrives at the standard VCI and the SPC performs the lookup itself using a software implementation of the IP classifier.

Regardless of how the packet is identified as requiring active processing, it is passed to the Plugin Control Unit (PCU) which, in turn, passes the packet to the target plugin or plugins. The PCU provides an environment for loading, configuring, instantiating and executing plugins. Plugins are dynamically loadable NetBSD kernel modules which reside in the kernel's address space. Since no context switching is required, the execution of plugins is highly efficient.

For the design of plugins, we follow an object-oriented approach. A *plugin class* specifies the general behavior of a plugin and defines how it is initialized, configured and how packets need to be processed. A *plugin instance* is a runtime configuration of a plugin class bound to a specific flow. It is desirable to have multiple configurations of a plugin, each processing its specific flow and having its own data segment that includes the internal state. Multiple plugin instances can be bound to one flow, and multiple flows can be bound to a single instance.

Through a virtual function table, each plugin class responds to a standardized set of methods to initialize, configure and process plugins. All code for initialization, configuration and processing is encapsulated in the plugin itself. Therefore, the PCU is not required to know anything about a plugin's internal details.

Once a packet is associated with a plugin or plugin chain, the plugin environment invokes the processing method of the corresponding plugin instance, passing it a reference to the packet to be processed. The processing might alter the packet payload as well as the header. If a packet's destination address has been modified, the packet needs to be reclassified since the output port might have changed before the packet is finally forwarded.

# Part II: Use and Evaluation

This part begins with a description of an active application that illustrates some of the features of the DER. Then, it describes the evaluation of several parts of the DER: 1) packet forwarding; 2) Queue State DRR; 3) distributed queueing; and 4) the reconfigurable hardware extension.

## 5. Wave Video (An Active Application)

### 5.1. The Application

The Wave Video application demonstrates an innovative active video scaling architecture that uses on-demand deployment of executable code in the form of plugins [32]. Video transmission triggers the retrieval of a plugin from a nearby code server and its installation in a router's kernel. Since the plugin code runs in kernel space, the scheme is highly efficient making it suitable for data path applications with high link rates.

The application is based on a wavelet-based encoding method. The high-performance video scaling algorithm executes as a plugin in the kernel space of a router. The plugin adapts the video stream to fit the current network load. If an output link is congested, the video stream is lowered to the bandwidth that the packet scheduler can guarantee. The video adaptation scheme ensures that low-frequency wavelet coefficients (which are crucial for the general definition of the image) are always forwarded but drops high-frequency parts (that describe image details) if bandwidth becomes scarce. The video adaptation algorithm can react within less than 50 ms to network load fluctuations. Thus, receivers see no disturbing artifacts, motion-blur, or wrongly coded colors even on networks with very bursty traffic patterns.

This approach yields better performance than layered multicast. Layered multicast reacts much more slowly to congestion (several seconds) than the wavelet-based method. And it needs a large number of multicast groups to provide similar granularity.

### 5.2. Experimental Evaluation

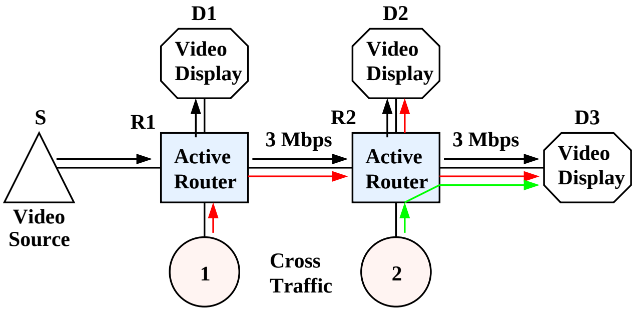


Figure 7. Wave Video Network

Figure 7 shows the network configuration used in our experimental evaluation of the Wave Video application. The video source (S) multicasts the video to three destinations (D1, D2, and D3). Receiver D1 is not affected by cross traffic at all, and therefore displays a disturbance-free video in all test cases. D2 and D3 are exposed to cross traffic: the link between R2 and D2 is shared by the video and one cross traffic stream, whereas the link to D3 is shared by the video and two cross traffic streams. Thus, D2 is moderately congested and D3 heavily congested. For all of our experiments we use two test videos: 1) *Akiyo*, which is a low-motion sequence, requires 1.3 Mb/s on average, and 2) *Foreman*, which has a higher degree of motion, requires 2.6 Mb/s. Both videos are lossless encoded with QCIF (Quarter Common Intermediate Format) resolution at 12 frames/s. To bring the routers' downstream links into a congested state by the cross traffic, we restrict the respective ATM link rate to 3 Mb/s using hardware pacing in the APIC. The routers run an implementation of the Deficit Round Robin (DRR) packet scheduler, which assigns an equal share of the available bandwidth to each flow. Thus, when the link is shared with one competing cross traffic flow, the video flow will get 1.5 Mbit/s. With two concurrently active cross traffic flows, the video gets 1 Mbit/s. If the bandwidth provided by the packet scheduler to the video flow is insufficient and no scaling is active, the packet scheduler drops the oldest packets (*plain-queueing*) from the video flow queue.

Reference [32] gives the details of four performance studies of the Wave Video system that were conducted. These studies evaluated:

- The cost of the video scaling algorithm;
- the time to download a plugin;
- the quality of the video in terms of PSNR (Picture Signal-to-Noise Ratio); and
- the reaction time of the wave video plugin to network congestion.

We describe below the results of the last study on reaction time to network congestion.

One major advantage of the video scaling architecture is that nodes have local knowledge of the load. Because the WaveVideo plugin can directly interact with the packet scheduler, it can immediately react to congestion. To demonstrate the node's ability to quickly respond to an overload situation, the video quality perceived at the receiver was measured during periods containing cross traffic bursts. When cross traffic is active, the DRR scheduler assigns the cross traffic an equal share of the link bandwidth, thus limiting the available bandwidth for the video flow to 1.5 Mb/s.

The quality of the plain-queued video stream suffers whenever the cross traffic is active, disturbing the video seriously. Further, the video quality does not recover until the burst is finished, making the video defective for the complete duration of the burst. On the other hand, active video scaling follows closely the video quality of the original video with only minor degradation after the cross traffic is turned on. As soon as the WaveVideo plugin discovers a decline in bandwidth (which is in the worst-case the 50 ms profile update period), it scales the video to the new available bandwidth. Doing so, the video stream quality recovers rapidly to a level very close to the original video stream showing no disturbing artifacts during the bursts.

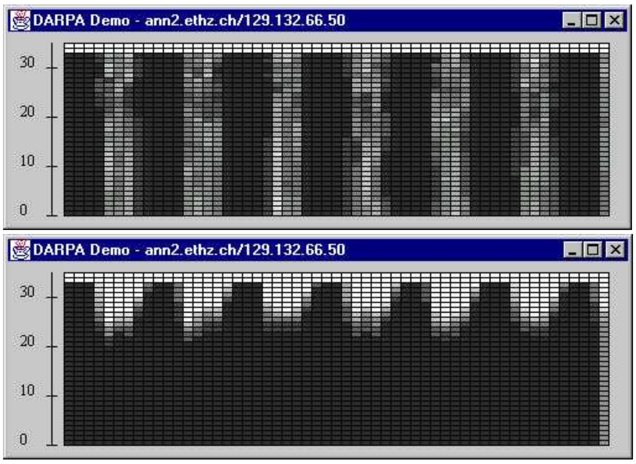

DARPA Demo - ann2.ethz.ch/129.132.66.50
30
20
10
0
DARPA Demo - ann2.ethz.ch/129.132.66.50
30
20
10
0

**Figure 8. Plain Queueing (Top) and Active Queueing (Bottom)**

To further demonstrate that quality is indeed gained by active dropping, we analyze the received frequency subbands at the receivers. Figure 8 shows a histogram of the subband distribution on receiver D2 when the routers are exposed to cross traffic bursts. Our test video sequence uses 33 subbands. The lowest frequency subband (containing the most crucial image information) is shown at the bottom of the graph and the highest frequency subband is displayed on top. The gray level of each cell indicates how many times a specific subband was received during a period of 8 frames: if a cell is white, the subband was never received at all, and if a cell is completely black, the subband was present in all the last 8 frames.

Active dropping now clearly shows its benefits: during burst activity, plain-dropping shows a random distribution of loss in all frequency subbands, leading to poor video quality. On the other hand, active dropping ensures that low-frequency subbands (which are crucial for the general image definition) are always forwarded to the receivers.

## 6. SPC-Only Packet Forwarding Experiments

This section focuses on the router throughput when using only SPCs as both input and output port processors. In particular, we measure the packet forwarding rate and data throughput for different IP packet sizes and examine the overhead timing components.

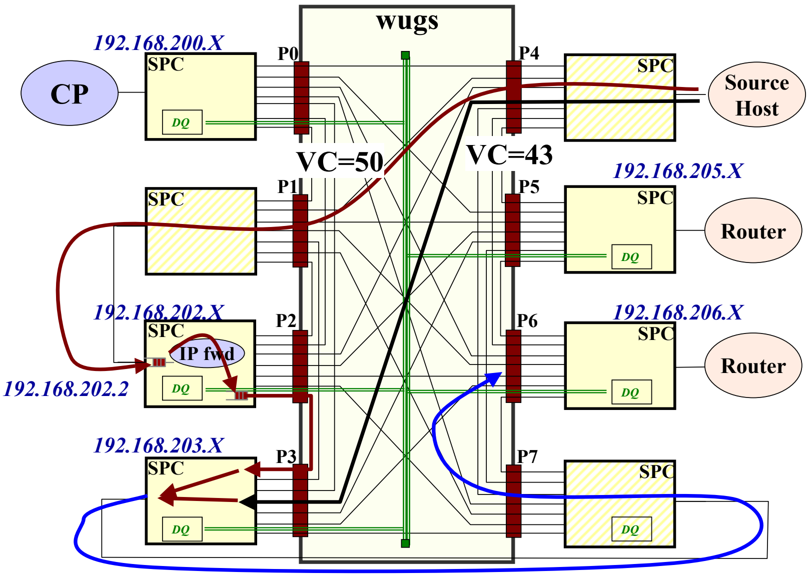


Figure 9. Experimental Setup

### 6.1. Experimental Setup

Figure 9 shows the experimental setup used for our tests. The configuration includes a DER with CP and one PC on port P4 acting as a traffic source. The ATM switch core is an eight-port WUGS configured with an SPC on each port. The CP and traffic source are both 600 MHz Pentium III PCs with APIC NICs.

The experiments use four key features of the WUGS: input port cell counters, a calibrated internal switch clock and ATM multicast and cell recycling. The CP reads the cell counter from the switch input ports and uses the switch cell clock to calculate a cell rate. The packet rate can easily be derived from the cell rate since the number of cells per packet is a constant for an individual experiment.

The multicast and recycling features of the WUGS were used to amplify the traffic volume for single-cell IP packets. Cell traffic can be amplified by $2^n$ by copying and recycling cells through $n$ VCIs before directing the cells to a target port. However, this feature can not be used for multi-cell IP packets since the ATM switch core does not prevent the interleaving of cells from two packets.

The SPCs on ports P2 and P3 were configured to operate as IP packet forwarders. Port P2 is used as the input port and port P3 as the output port. All other SPCs are disabled so that traffic will pass through them unaffected.

Typically, hosts or other routers would be connected to each port of a DER. However, to facilitate data collection we have directly connected the output of port P1 to the input of port P2 and the output of port P3 to the input of P7. Our data source is connected to port P4. Thus we can use:

- The cell counters at port P4 to measure the sending rate;
- The cell counters at port P2 to measure the traffic forwarded by the input side PP at port P2; and
- The cell counters at port P7 to measure the traffic forwarded by the output side PP at port P3.

IP traffic is generated by using a program that sends specific packet sizes at a prescribed rate. Packet sending rates are controlled using two mechanisms: 1) logic within the traffic generator program, and 2) for high rates, the APIC's pacing facility. These two mechanisms produced both high and consistent sending rates.

### 6.2. Small-Packet Forwarding Rate

In order to determine the per packet processing overhead, we measured the forwarding rate of 40-byte IP packets (one ATM cell each) at the input and output ports. Single-cell packet rates as high as 907 KPps (KiloPackets per second) were generated by using the ATM multicast and cell recycling features of the switch to multiply the incoming traffic by a factor of eight. The measurements were repeated by successively disabling three major IP processing overhead components:

1) Distributed queueing (no DQ processing occurs).

2) Fast IP Lookup (FIPL) (a simple IP lookup algorithm using a small table is used).

3) Word swapping.

Disabling word swapping is related to a bug in the APIC hardware that requires the SPC to swap pair-wise memory words of a packet (e.g., word[0]⇔word[1], word[2]⇔word[3]). When word swapping is disabled, the IP packet body words are not swapped, but the packet header is swapped.

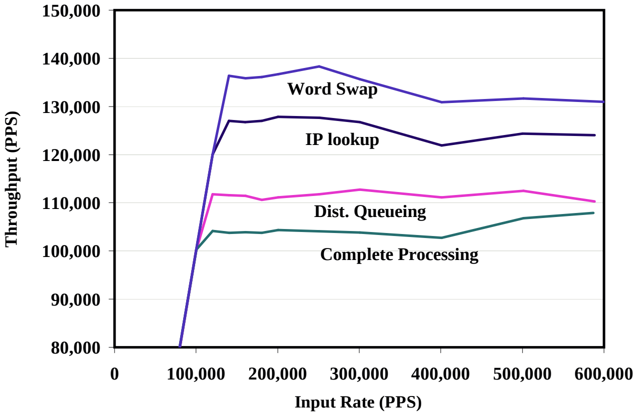


**Figure 10. Packet Forwarding Rate for 40-Byte IP Packets**

Figure 10 shows typical bottleneck curves where the throughput (forwarding rate in packets per second (Pps)) is equal to the input rate until the bottleneck is reached. The maximum throughput when all processing components are included (”Complete Processing”) is about 105,000 packets per second. The other curves show the cumulative effects of disabling each overhead component. Note that the throughput starts at 80,000 Pps. Thus, DQ, FIPL, and word swapping account for an additional overhead of approximately 6% (7 KPps), 15% (16 KPps), and 8% (9 KPps) respectively. We expect that the DQ algorithm can be accelerated since it is a first implementation.

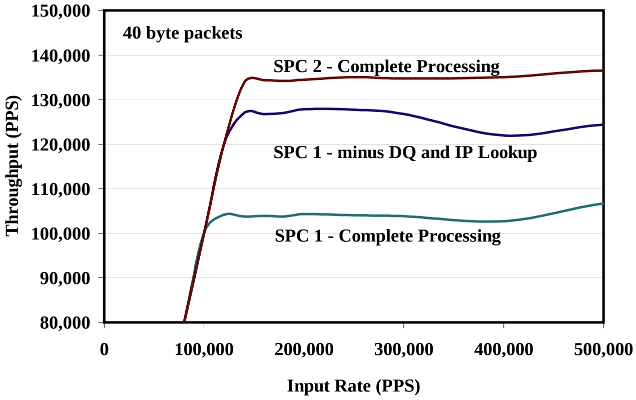


**Figure 11. Comparison With SPC2**

The maximum forwarding rate at an input port PP assuming an APIC hardware fix and infinite acceleration of DQ and FIPL is about 140 KPps (KiloPackets per second). But we have already been testing the *SPC 2*, a newer version of the SPC, which includes a 500 MHz PIII with a larger (256 MB) and higher bandwidth (1.6X) memory. Figure 11 shows that this processor can almost make up for the overheads and attain 140 KPps. Experiments with larger packet sizes show that a speed-up of 1.6 is possible. This is not surprising since the two principal resources used in packet forwarding are the PCI bus and the memory system.

The output port PP has a higher forwarding rate since it does not perform IP destination address lookup. Furthermore, the maximum throughput can be sustained even for a source rate as high as 900 KPps. This rate stability at high loads is a consequence of our receiver livelock avoidance scheme.

### 6.3. Throughput Effects of Packet Size

We next measured the effect of the packet size on the packet forwarding rate. Because IP packets larger than 40 bytes require more than one cell (there are 8 bytes of overhead), we no longer used ATM multicast with cell recycling to amplify the traffic. Instead, we used a single host to generate traffic using packet sizes ranging from 40 bytes to 1912 bytes.

Figure 12 shows the throughput (Mb/s) as a function of the packet size when the input packet rate is 40 KPps. For packets less than 400 bytes, the throughput is almost equal to the product of the input rate and the packet size. This region corresponds to the case when there are no bottlenecks along the path through the router. Then, a knee in the curve develops until around a packet size of 500 bytes when CPU, PCI bus and memory system bottlenecks begin to seriously limit the throughput. When the packet size is 1,500 bytes, the throughput is about 220 Mb/s.

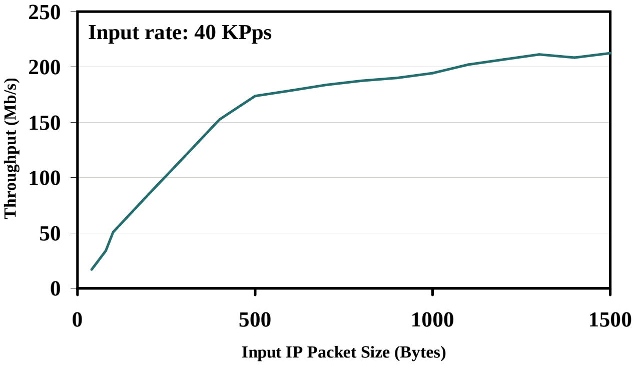


**Figure 12. Router Throughput for Various IP Packet Sizes**

Figure 13 shows the effect of limited memory bandwidth on the throughput. Word swapping, in effect, forces the SPC

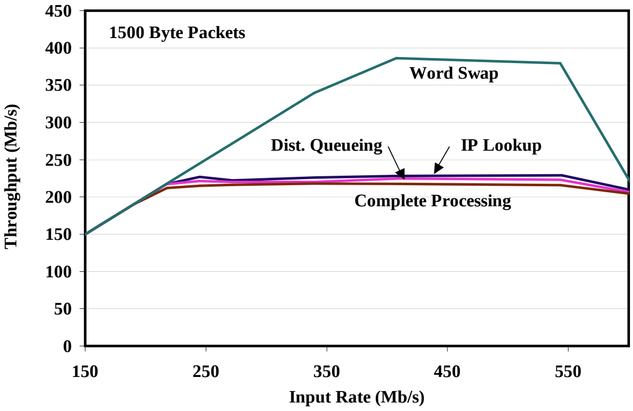


**Figure 13. SPC Throughput (Mb/s)**

to consume twice as much memory bandwidth as it should. When we remove the word swapping overhead, the maximum throughput is increased from 220 Mbps to 380 Mbps.

### 6.4. Analysis of Results

In analyzing our performance results, we considered three potential bottlenecks:

1) PCI Bus (33 MHz, 32 bit).

2) SPC Memory Bus (66 MHz EDO DRAM).

3) Processor (166 MHz Pentium MMX).

Each of these comes into play at different points in the packet forwarding operation.

Our studies of the PCI bus operations that take place to forward a packet indicate that there are 3 PCI read operations and 5 PCI write operations which together consume 60 bus cycles. Additionally, under heavy load, we can expect 64 wait cycles to be introduced. Thus a total of 124 bus cycles (30.3 ns per cycle on a 33 MHz bus) or 3.72 $\mu$sec are consumed by PCI bus operations in the forwarding of a packet.

In an early measurement of a simplified test case, the average software processing time was shown to be 3.76 $\mu$sec. This, combined with the PCI bus time calculated above gives us a per packet forwarding time of 7.48 $\mu$sec. This is very close to the time of 7.5 $\mu$sec for the 40-byte packet forwarding rate of 128 KPps shown in Figure 10.

As indicated earlier, the bug in the APIC chip causes the received word order on an Intel platform to be incorrect. In order to work around this, the APIC driver must perform a word swap on all received data. Thus, each received packet may cross the memory bus four times:

1) APIC writes packet to memory.

2) CPU reads packet during word swapping.

3) CPU writes packet during word swapping.

4) APIC reads packet from memory

Figures 10 and 13 quantify this overhead.

## 7. Queue State DRR Experiments

### 7.1. Study Objectives

The Queue State DRR (QSDRR) algorithm has been evaluated using an `ns-2` simulation and compared with other competing algorithms. An early version of QSDRR has been implemented in an SPC-only prototype and is currently being evaluated. The goal of the simulation experiments is to find schedulers that satisfy the following properties:

- *High throughput with small buffers* to avoid long queueing delays.
- *Insensitivity to operating conditions and traffic* to reduce the need to tune parameters.
- *Fairness among flows with different characteristics* to reduce the need for discriminating among flows.

Our results show that QSDRR outperforms other common queueing algorithms such as RED and Blue [30, 31].

Backbone Internet routers are typically configured with large buffers that can store packets arriving at link speed for a duration equal to several coast-to-coast round-trip delays. Such buffers can delay packets for as much as half a second during congestion periods. When such large queues carry heavy TCP traffic loads and are serviced with the Tail Drop policy, the large queues remain close to full resulting in high end-to-end delays for long periods [33, 34].

RED maintains an exponentially-weighted moving average of the queue length and uses this as an indicator of congestion. When the average crosses a minimum threshold ($min_{th}$), packets are randomly dropped or marked with an explicit congestion notification (ECN) bit. When the queue length exceeds the maximum threshold ($max_{th}$), all packets are dropped or marked. RED parameters must be tuned with respect to link bandwidth, buffer size and traffic mix.

Blue improves on RED by adjusting its parameters automatically in response to queue overflow and underflow events. Although Blue is an improvement in some scenarios, it too is sensitive to different congestion conditions and network topologies. In practice, tuning these algorithms is very difficult since the input traffic mix is continuously varying.

QSDRR combines a queueing algorithm with fair scheduling over multiple queues. Although most previous work on fair scheduling used per-flow queues, we have shown that comparable performance can be obtained when queues are shared by multiple flows. While algorithms using multiple queues have historically been considered too complex, continuing advances in technology have made the incremental cost negligible and well worth the investment if these methods can reduce the required buffer sizes and resulting packet delays.

We have compared the performance of QSDRR with RED, Blue, tail drop FCFS (Tail Drop), longest queue drop DRR (DRR) and Throughput DRR (TDRR) and shown major improvements with respect to the three objectives listed above [35]. The class of DRR algorithms approximates fair-queueing and requires only $O(1)$ work to process a packet. Thus, it is simple enough to be implemented in hardware [36]. DRR with longest queue dropping is a well-known algorithm that drops packets from the longest active queue. For the rest of the paper, we refer to it as plain DRR or DRR since longest queue dropping was part of the original algorithm first proposed by McKenney [37]. TDRR drops packets from the highest throughput active queue. Some of our results for QSDRR and RED are presented below to illustrate the effectiveness of QSDRR.

### 7.2. Simulation Environment

We present some of the performance results below. Table 2 shows the algorithm independent parameters. See [35] for the parameters and network configurations used in the complete study.

| | Value | Definition |
|---|---|---|
| $N$ | 100 | Number of TCP sources |
| $M$ | 1500 Bytes | Packet size |
| $L_R$ | 500 Mb/s | Inter-router link rate |
| $L_A$ | 10 Mb/s | Access link rate |
| $T$ | 100 Sec | Observation period |
| $P$ | Reno | TCP Protocol |
| $Q$ | 100-20,000 Pkts | Bottleneck queue capacity |

**Table 2. Algorithm Independent Parameters**

For each of the configurations, we varied the bottleneck queue capacity from a 100 packets to 20,000 packets. A bottleneck queue capacity of 20,000 packets represents a half-second bandwidth-delay product buffer, a common buffer size deployed in current commercial routers. We ran several simulations to evaluate $max_p$ and $w_q$ for RED that worked best for our simulation environment to ensure a fair comparison against our multi-queue based algorithms. The RED parameters we used in our simulations are in Table 3.

| RED | | |
|---|---|---|
| $max_p$ | Max. drop probability | 0.01 |
| $w_q$ | Queue weight | 0.001 |
| $min_{th}$ | Min. threshold | $20\%$ of buffer size |
| $max_{th}$ | Max. threshold | Buffer size |

**Table 3. RED Parameters**

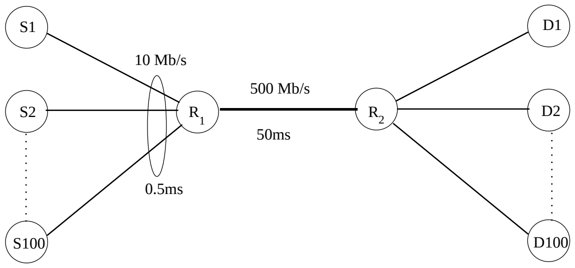


**Figure 14. Single Bottleneck Link Network Configuration**

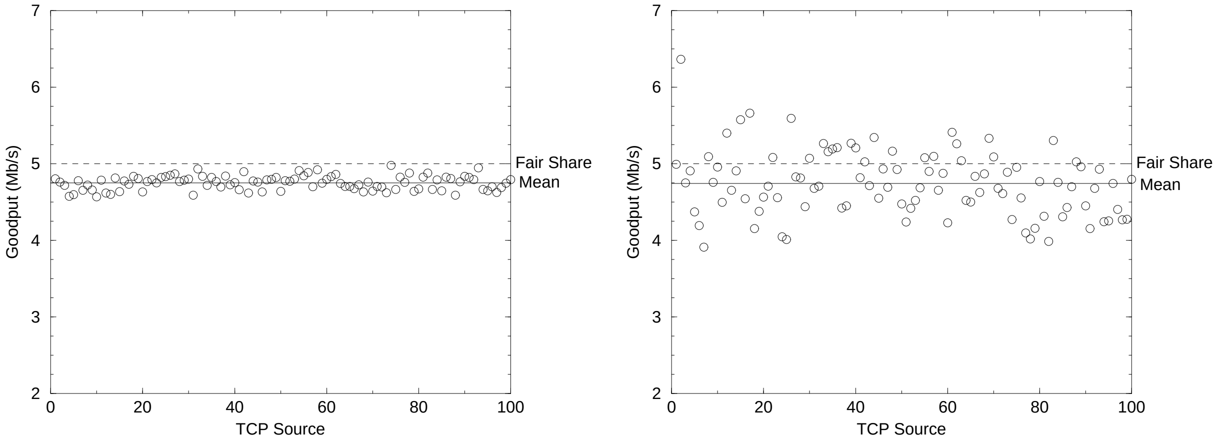
The network configuration of experiments is shown in Figure 14. $\{S_1, S_2, ...S_{100}\}$ are the TCP sources, each connected by 10Mb/s links to the bottleneck link. Since the bottleneck link capacity is 500 Mb/s, if all TCP sources send at the maximum rate, the overload ratio is 2:1. The destinations $\{D_1, D_2, ...D_{100}\}$ are directly connected to the router $R_2$. All 100 TCP sources are started simultaneously to simulate a worst-case scenario whereby TCP sources are synchronized in the network.

### 7.3. Results

We compared the queue management policies with respect to three metrics: 1) The average goodput of all TCP flows as a percentage of its fair-share; 2) the goodput distribution of all TCP sources over a single-bottleneck link; and 3) the variance in goodput. The *variance* in goodputs is a metric of the fairness of the algorithm: Lower variance implies better fairness.

For all our graphs, we concentrate on the goodputs obtained while varying the buffer size from 100 packets to 5000 packets. For multi-queue algorithms, the stated buffer size is shared over all the queues while for single queue algorithms, the stated buffer size is for that single queue. Since our bottleneck link speed is 500 Mb/s, this translates to a variation of buffer *time* from 2.4 msec to 120 msec. Only packet buffer sizes up to 5,000 packets are shown since the behavior with larger buffer sizes is the same for larger sizes.

Figure 15 compares the distribution of goodputs for all 100 TCP Reno flows for QSDRR and RED when the buffer



(a) QSDRR

(b) RED

**Figure 15. Goodput Distribution (Single-Bottleneck Link, 200 Pkt Buffer)**

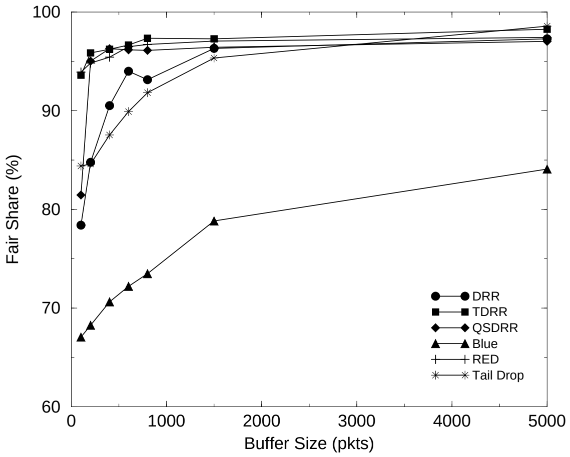


**Figure 16. Fair Share Performance**

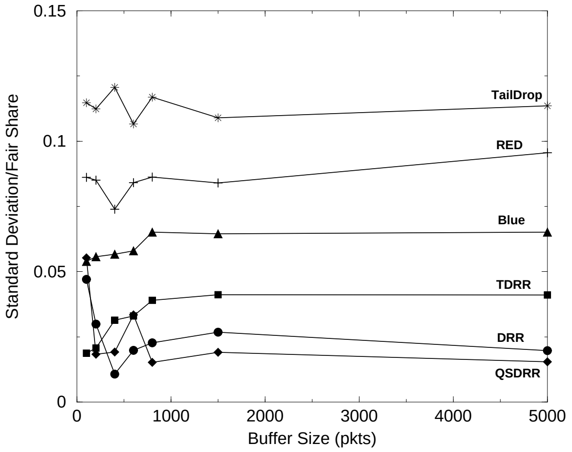


**Figure 17. Standard Deviation Over Fair-Share**

size is small (200 packets). For QSDRR, all TCP flows had goodputs very close to the mean, and the mean goodput is very near the fair-share threshold. This low variability of the goodputs between different TCP flows indicates good fairness. The RED plot shows considerably more variability which implies less fairness.

Figure 16 shows the average fair-share bandwidth percentage received by the TCP Reno flows using different buffer sizes. For small buffer sizes (i.e., under 500 packets), TDRR and QSDRR outperform RED significantly, and DRR is comparable to RED. It is interesting to note that even for large a buffer size (5000 packets), all policies significantly outperform Blue including Tail Drop.

Figure 17 shows the ratio of the goodput standard deviation of the TCP Reno flows to the fair share bandwidth while varying the buffer size. Even at higher buffer sizes, the standard deviations for DRR and QSDRR are very small, and the ratio to the fair share bandwidth is less than 0.025. TDRR exhibits a higher goodput standard deviation, but it is still significantly below Blue, RED and Tail Drop. RED exhibits about 10 times the variance compared to QSDRR and DRR while Blue exhibits about 5 times the variance. Overall, the goodput standard deviation is between $2\% - 4\%$ of the fair share bandwidth for the multi-queue policies compared to $6\%$ for Blue, $10\%$ for RED and $12\%$ for Tail Drop. Thus, even for a single-bottleneck link, the multi-queue policies offer much better fairness between TCP flows.

Experiments with multi-hop network configurations and a mix of TCP flow characteristics showed the superiority of QSDRR over the other algorithms [35]. QSDRR significantly outperforms RED and Blue for various configurations and traffic mixes: QSDRR results in higher average goodput for each flow and lower variance in goodputs. QSDRR also performs well even when memory is limited and when multiple sources are aggregated into one queue.

## 8. Distributed Queueing Experiments

### 8.1. The Stress Test

The Distributed Queueing (DQ) algorithm described in Section 3.6 was implemented in a discrete-event simulator and in an SPC-only prototype. The simulator allowed us to quickly evaluate alternative algorithms in a controlled environment before implementing the algorithm in the more restrictive and less controllable environment of an SPC-only prototype. This section describes the results of an experiment that was designed to assess the performance and stability of the DQ algorithm under extreme loading conditions.

This *stress traffic load* goes through $R-1$ stages where each of $K$ inputs builds backlogs for each of $R-1$ outputs. Arriving data at one of the inputs then switches to an $R$th output, creating a situation that can lead to underflow at one or more of the outputs. The experiment parameters are shown in Table 4 and are the same parameters used in experiments with an SPC-only prototype.

$K$ sources overload the switch fabric by each sending at the link rate $L$. Since the speed-up is less than the number of sources ($S < K$), the switch will be overloaded and input backlogs will develop. All traffic is initially aimed at output 1 for a duration $T$. Then, the traffic pattern is changed in stages stepping through the outputs in ascending order until reaching output $R$ at which time all sources turn off except source 1. The beginning of stage $j$ is chosen such that there will be input backlogs at the $K$ inputs in stage $R$.

Figure 18 shows the stages of the stress load for the case $K = 4$ and $R = 5$. Let $T_j$ be the ending time of stage $j$. At time $T_1 = T$, all sources switch their traffic to output 2. In stage 2, input backlogs for output 1 will drain while the input backlogs for output 2 climb. All sources switch to output 3 at time $T_2$ when the input backlog $B_{i,2}$ equals $B_{i,1}$, and stage 3 begins. This traffic switching pattern continues until stage 5 when all sources except source 1 shut off.

| | Value | Definition |
|---|---|---|
| $N$ | 8 | Number of output (or input) ports |
| $S$ | 2 | Switch speed-up ($S \geq 1$) |
| $L$ | 70 Mb/s | External link rate (source rates) |
| $D$ | 500 $\mu$sec | DQ update period |
| $M$ | 520 Bytes | Packet size |
| $K$ | 4 | Number of traffic sources |
| $R$ | 5 | Number of traffic stages |
| $T$ | 400 DQs<br>(=200 msec) | Time stage 1 ends (=$T_1$) |

**Table 4. Parameters of the DQ Experiment**

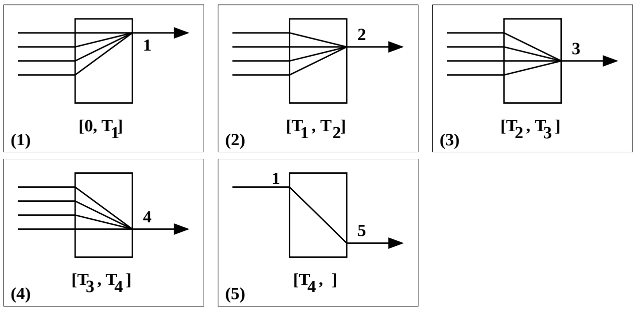


**Figure 18. Stress Traffic Load**

### 8.2. Simulation Experiments

The simulation results presented in this section are for a variant of the algorithm presented in Section 3.6. Small modifications to Equations 1 and 2 improve system responsiveness by reserving some bandwidth for queues that frequently have very small or no backlog. The new equations are:

$$hi_{i,j} = \frac{B_{min} + B_{i,j}}{NB_{min} + \sum_h B_{h,j}} \tag{5}$$

$$lo_{i,j} = lo_{min} + \frac{B_{i,j}}{B_j + \sum_h B_{h,j}} \tag{6}$$

where $B_{min}$ is a small, artificial input backlog, and $lo_{min} = 0.5/N$ is a small, artificial minimum rate.

Figure 19 shows the output and input backlogs $B_1$ and $B_{1,j}$. Figure 20 shows the allocated rates $r_{1,j}$. Since the duration of stage 1 is 400 DQ periods (200 msec), we expect the other beginning stage times to be 600, 700, and 750 DQ periods.

Consider the backlog plot shown in Figure 19. The curves labeled *Output 1, 2, 3* and *4* are the output backlogs $B_j$ for $j = 1..4$. The other curves labeled *1=>j*, $j = 1..3$, are the input backlogs $B_{1,j}$ for $j = 1..3$ ($B_{1,4}$ is unlabeled). Clearly, the system is overloaded since the switch capacity is $S \cdot L$ = 140 Mb/s while the aggregate source rate is

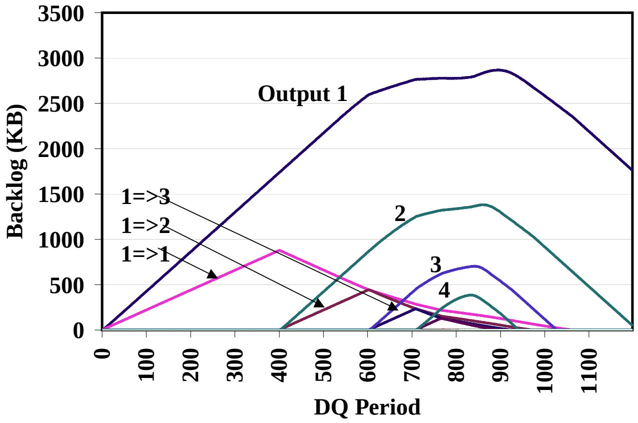


**Figure 19. Output Backlogs and Input 0 Backlogs**

$K \cdot L$ = 280 Mb/s. So, the output backlogs build up at a rate of approximately 70 Mb/s (= $S \cdot L - L$) or about 1.76 MB in 400 DQ periods. This build-up continues until back pressure from an overloaded output reduces the sending rate below 35 Mb/s. The input backlogs build up at a rate of approximately 35 Mb/s (= $S \cdot L/K$) or about 880 KB in 400 DQ periods.

Output 1's backlog rate begins to decrease around 600 DQ periods, just when the sources are transitioning from output 2 to output 3. Meanwhile the input backlog $B_{1,1}$ that had been draining at a rate of 35 Mb/s, starts draining slower around 600 DQ periods because of the back pressure from output 1.

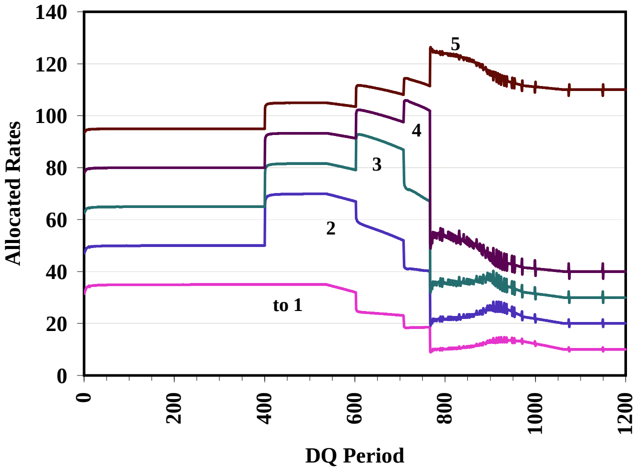


**Figure 20. Input 1 Allocated Rates**

Figure 20 shows three characteristics suggested by the above discussion:

1) The rates $r_{1,j}$, $j = 1..4$ for input 1 are 35 Mb/s which is input 1's equal share of the 140 Mb/s switch capacity. At DQ period 600, $r_{1,1}$ drops dramatically because a sufficient output backlog has developed. In fact, a small drop in rate occurred before this time. These decreases in rates can also be seen for other outputs.

2) Input queues that have no backlog are allocated some non-zero rate. For example, before DQ period 400, there is no input backlog for output 2, but it is still allocated about 15 Mb/s (= (140-35)/7).

3) After all sources have shut off, input 1 gets to send to output 5 at about the link rate of 70 Mb/s. Input 1 continues to drain the other input queues at about 10 Mb/s.

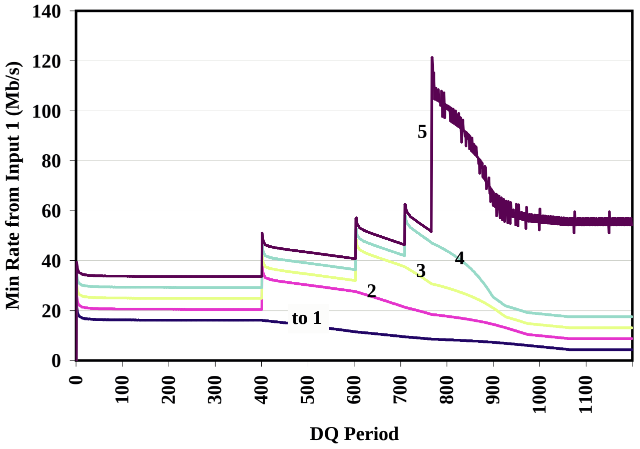
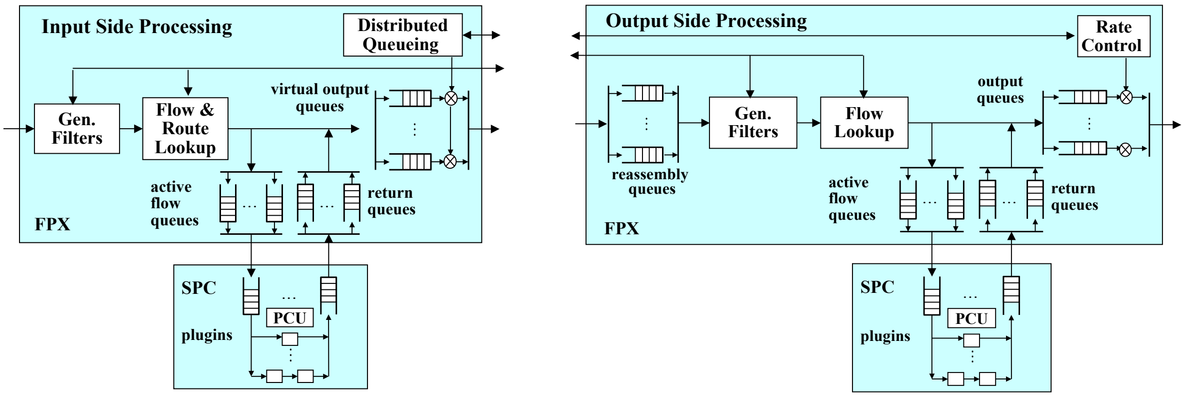


**Figure 21. Input 1 Minimum Rates**

The sum of the minimum rates for input i is an indicator of output underflow. This sum is $L \cdot \sum_h lo_{i,h}$ where $L$ is the link rate. Figure 21 shows this sum for input 1 when $L$ = 70 Mb/s and $S$ = 2 (the switch speed-up). Each curve starting with the "to 1" curve indicates the contribution of each output to this sum. Recall that $L \cdot lo_{i,j}$ is the minimum rate that must be allocated for traffic sent from input $i$ to output $j$ to avoid underflow at output $j$. This will be possible if the sum $L \cdot \sum_h lo_{1,h}$ is no more than $S \cdot L$.

Figure 21 clearly shows that the sum is less than 140 Mb/s and therefore underflow will be avoided. It also shows the response of the DQ algorithm to the stress load. Since there are $K = 4$ sources, the maximum rates $L \cdot hi_{i,j}$ in stages 1..4 is 17.5 Mb/s (= 70/4 Mb/s) for inputs 1..4 going to outputs 1..4. The speed-up of 2 in conjunction with the DQ algorithm allows the DER to move sufficient backlog to outputs 1..4 so that the demand for switch bandwidth is reduced over time. In the last stage, input 1 gets its minimum rate of $L$ = 70 Mb/s ($lo_{1,5} = hi_{1,5} = 1$), and the sum of the minimum rates is well below its underflow limit of 140 Mb/s.



(a) Input Side

(b) Output Side

**Figure 22. FPX IP Processing**

### 8.3. SPC-Only Prototype Experiments

We repeated the stress experiment on an SPC-only prototype and obtained results similar to those shown in the simulation experiments. Three tools were developed to aid in duplicating the stress experiments:

1) A traffic generator that can be remotely configured and controlled.

2) A DQ cell monitor to capture all DQ cells carrying backlog data.

3) A Java DQ visualizer that duplicates the DQ calculations and plots the same graphs we saw earlier in the simulation experiments.

The traffic generator is capable of sending AAL5 frames at nearly 1 Gbps. It consists of three components: 1) a standard *user-space server code* that accepts control and configuration packets (e.g., start sending, stop sending, set pacing rate, set destination, set packet length, stress load); 2) a *user-space library* that directly manipulates the traffic descriptors stored in kernel memory; and 3) an *APIC driver* that sends the AAL5 frames based on the traffic descriptors.

We continue to improve the DQ algorithm and study its operating characteristics. Two fundamental questions are ”How does the algorithm behave when the speed-up $S$ is less than 2 (but greater than 1)?” and ”How low of a speed-up can be practically used?”

## 9. Reconfigurable Hardware Extension

The FPX/SPC system extends the SPC-only system by adding field programmable hardware modules at the ports to assist in port processing. This section describes the logical port architecture of the FPX, briefly describes the FIPL engine, and presents measurements of FIPL's packet forwarding performance under a synthetic load. Reference [38] provides additional details on the FIPL engine.

### 9.1. Logical Port Architecture and the FIPL Engine

In an FPX/SPC system the SPC is used primarily to handle active processing and special IP options. The FPX performs the bulk of the IP processing that the SPC handles in an SPC-only system. Figure 22 shows the logical organization of the FPX in its capacity as an IP packet processor and its interaction with the SPC. The figure is virtually identical to Figure 6 except that the plugins and the Plugin Control Unit remain in its partner SPC.

The FPX recognizes an active packet destined for its SPC when the packet matches a special filter. The FPX sends an active packet over a special VCI to its SPC for active processing. The SPC returns the packet to its FPX on a different special VCI, and the FPX then forwards it on to the next hop.

A fundamental function in any IP router is *flow lookup*. A lookup consists of finding the longest prefix stored in the forwarding table that matches a given 32-bit IPv4 destination address and retrieving the associated forwarding information.

The Fast Internet Protocol Lookup (FIPL) engine implements Eatherton and Dittia's Tree Bitmap algorithm [25]. The algorithm employs a multibit trie data structure with a clever data encoding that leads to a good hardware implementation:

- Small memory requirement (typically 4-6 bytes per prefix)

- Small memory bandwidth (leads to fast lookup rates)
- Updates have negligible impact on the lookup performance

Also, several concurrent lookups can be interleaved to avoid the impact of external memory latency.

The implementation uses reconfigurable hardware and Random Access Memory (RAM). It is implemented in a Xilinx Virtex-E Field Programmable Gate Array (FPGA) running at 100 MHz and uses a Micron 1 MB Zero Bus Turnaround (ZBT) Synchronous Random Access Memory (SRAM).

In the example shown in Figure 23, the IP address 128.252.153.160 is compared to the stored prefixes starting with the most significant bit. Logically, the Tree Bitmap algorithm starts by storing prefixes in a binary tree. Shaded nodes denote a stored prefix. A search is conducted by using the IP address bits to traverse the trie, starting with the most significant bit of the address. To speed up this searching process, multiple bits of the destination address are compared simultaneously. In order to do this, subtrees of the binary trie are combined into single nodes producing a multibit trie, reducing the number of memory accesses needed to perform a lookup. The depth of the subtrees combined to form a single multibit trie node is called the *stride*.

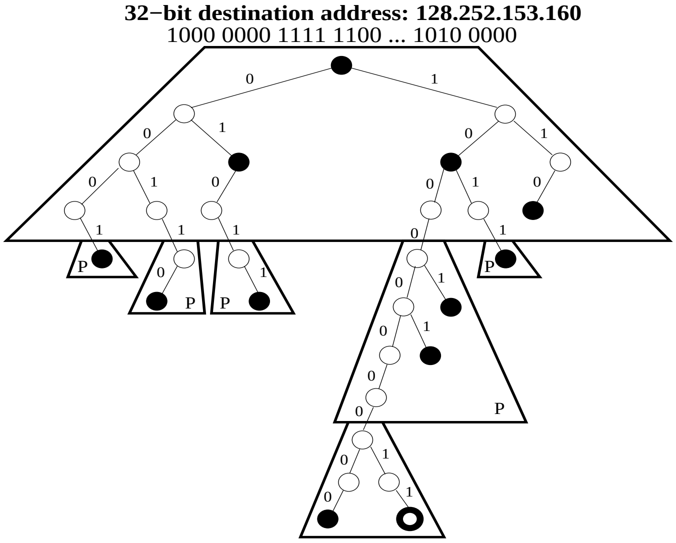


**Figure 23. IP Lookup Table Represented as a Multibit Trie**

The example shows a multibit trie using 4-bit strides. Each trapezoid surrounds a multibit trie node. Figure 24 shows how the Tree Bitmap algorithm codes information associated with each node of the multibit trie using bitmaps. A 1 in the $i$th position of an *Extending Paths Bitmap* indicates that a child multibit trie node exists. For example, the root node initially has a child at every odd position in the first six positions resulting in ”`010101`” in the first six bits. The *Internal Prefix Bitmap* identifies with a `1` the stored prefixes in the binary sub-tree of the multibit node. The Internal Prefix Bitmap of the root multibit node is ”`1 00 0110 00000010`” (reading from left-to-right, the bits have been grouped by tree level) where the leftmost `1` corresponds to the root, the next two `0` bits correspond to the next level of the tree, and so on.

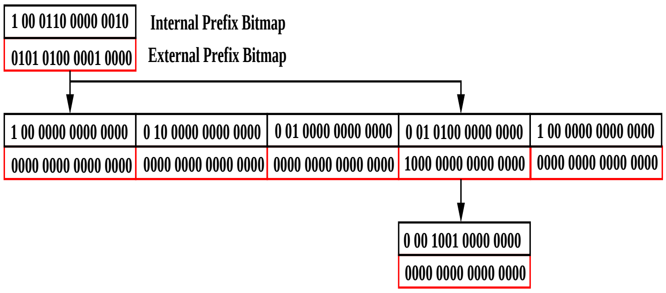


**Figure 24. IP Lookup Table Represented as a Tree Bitmap**

## 9.2. Performance

While the worst-case performance of FIPL is deterministic, an evaluation environment was developed in order to benchmark average FIPL performance on actual router databases. As shown in Figure 25, the evaluation environment includes a modified FIPL Engine Controller, 8 FIPL Engines, and a FIPL Evaluation Wrapper. The FIPL Evaluation Wrapper includes an IP Address Generator which uses 16 of the available on-chip BlockRAMs in the Xilinx Virtex 1000E to implement storage for 2048 IPv4 destination addresses. The IP Address Generator interfaces to the FIPL Engine controller like a FIFO. When a test run is initiated, an empty flag is driven to FALSE until all 2048 addresses are read.

Control cells sent to the FIPL Evaluation Wrapper initiate test runs of 2048 lookups and specify how many FIPL Engines should be used during the test run. The FIPL Engine Controller contains a latency timer for each FIPL Engine and a throughput timer that measures the time required to complete the test run. Latency timer values are written to a FIFO upon completion of each lookup. The FIPL Evaluation Wrapper packs latency timer values into control cells which are sent back to the system control software where the contents are dumped to a file. The throughput timer value is included in the final control cell.

Using a portion of the Mae-West snapshot from July 12,

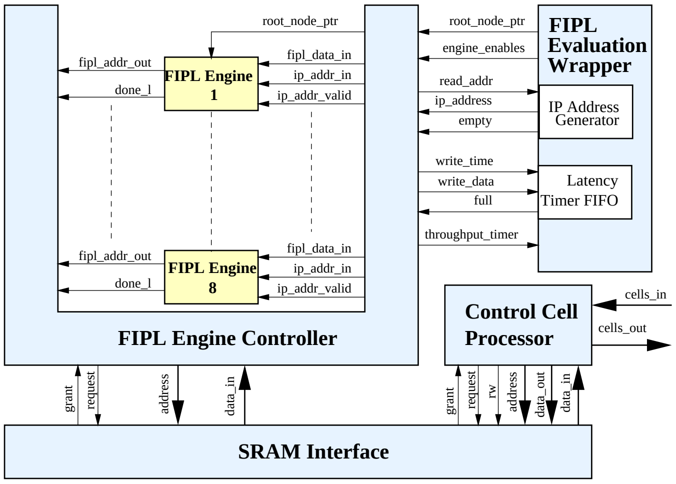


**Figure 25. Block Diagram of FIPL Evaluation Environment.**

2001, a Tree Bitmap data structure consisting of 16,564 routes was loaded into the off-chip SRAM. The on-chip memory read by the IP Address Generator was initialized with 2048 destination addresses randomly selected from the route table snapshot. Test runs were initiated using 1 through 8 engines. With 8 FIPL engines, the entire memory bandwidth of the single SRAM is consumed. Figure 26 shows the results of test runs without intervening update traffic. Note that the left y-axis is the throughput (millions of lookups per second), and the right y-axis is the average lookup latency (nanoseconds). Plots of the theoretical performance for all worst-case lookups is shown for reference. Figure 27 shows the results of test runs with various intervening update frequency. A single update consisted of a route addition requiring 12 memory writes packed into 3 control cells.

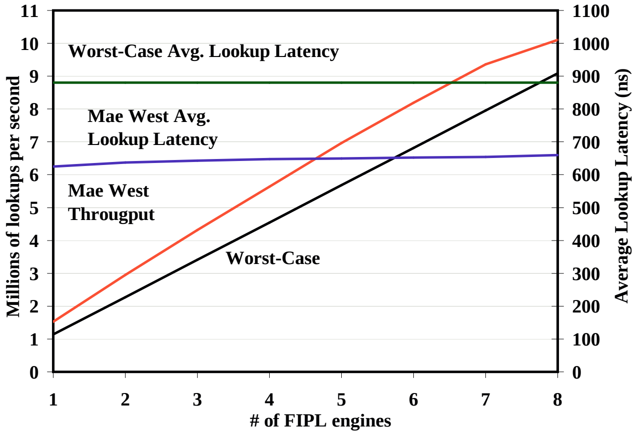


**Figure 26. FIPL Performance**

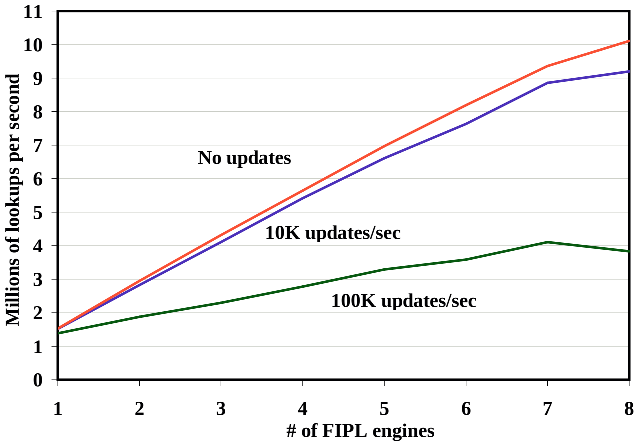


**Figure 27. FIPL Performance Under Update Load**

With no intervening update traffic, lookup throughput ranged from 1,526,404 lookups per second for a single FIPL engine to 10,105,148 lookups per second for 8 FIPL engines. Average lookup latency ranged from 624 ns for a single FIPL engine to 660 ns for 8 FIPL engines. This is less than a 6% increase in average lookup latency over the range of FIPL Engine Controller configurations.

Note that update frequencies up to 1,000 updates per second have little to no effect on lookup throughput performance. An update frequency of 10,000 updates per second exhibited a maximum performance degradation of 9%. Using the near maximum update frequency supported by the Control Processor of 100,000 updates per second, lookup throughput performance is degraded by a maximum of 62%. Note that this is a highly unrealistic situation, as lookup frequencies rarely exceed 1,000 updates per second.

Coupled with advances in FPGA device technology, implementation optimizations of critical paths in the FIPL engine circuit hold promise of doubling the system clock frequency to 200 MHz in order to take full advantage of the memory bandwidth offered by the ZBT SRAMs. Doubling of the clock frequency directly translates to a doubling of the lookup performance to a guaranteed worst case throughput of over 18 million lookups per second. Ongoing research hopes to exploit new FPGA devices and more advanced CAD tools to double the lookup performance by doubling the clock frequency.

## 10. Concluding Remarks

Additional performance measurements of the DER are in progress, and a number of developments and extensions are underway. First, the *integration of the FPX* with the current DER configuration will soon commence. The Fast IP Lookup (FIPL) algorithm has been implemented in reprogrammable hardware using the FPX and partially evaluated in a development environment. In addition, other applications have been ported to the FPX. Second, *SPC 2* boards will be available soon. It will have a faster processor (500 MHz PIII) much higher main memory bandwidth (SDRAM), and a larger memory (256 MB). Third, a *Gigabit Ethernet line card* is being designed around the PMC-Sierra PM3386 S/UNI-2xGE Dual Gigabit Ethernet Controller chipset with plans for availability by summer 2002. This will allow us to interface the DER to routers and hosts that have Gigabit Ethernet interfaces. Fourth, the *Queue State DRR (QSDRR) algorithm* is being evaluated in an SPC-only testbed. Fifth, the *Wave Video active application* will be ported to the DER environment. This will require that it be able to interrogate QSDRR. Sixth, a *lightweight flow setup service* will be developed and demonstrated. This service, which can be implemented largely in hardware, will require no elaborate signaling protocol and no extra round-trip delays normally associated with signaling and resource reservation. Finally, many *CP software components* are in their early prototyping stage. Some of these components include: 1) Automatic multi-level boot process that starts with discovery and ends with a completely configured, running router; 2) Network monitoring components based on active, extensible switch and PP MIBs and probes providing a multi-level view of the DER router; and 3) the Zebra-based routing framework.

The Washington University DER provides an open, flexible, high-performance active router testbed for advanced networking research. Its parallel architecture will allow researchers to deal with many of the same real design issues faced by modern commercial designers. Finally, its reprogrammability in combination with its open design and implementation make it an ideal prototyping environment for exploring advanced networking features.

## References

[1] T. Chaney and A. Fingerhut and M. Flucke and J. S. Turner, "Design of a Gigabit ATM Switch," in *IEEE INFOCOM '97*, (Kobe, Japan), IEEE Computer Society Press, April 1997.

[2] J. S. Turner and A. Staff, "A Gigabit Local ATM Testbed for Multimedia Applications ," Tech. Rep. ARL-94-11, Applied Research Laboratory, Washington University in St. Louis, 1994.

[3] W. D. Richard, "The Gigabit Switch Link Interface Specification," Tech. Rep. ARL-94-17, Washington University Applied Research Laboratory,St. Louis, Missouri, January 1995.

[4] A. T1.106-1988, *Telecommunications - Digital Hierarchy Optical Interface Specifications: Single Mode*, 1988.

[5] H.-P. Corporation, "HDMP-1022 Transmitter/HDMP-1024 Receiver Data Sheet," 1997.

[6] DARPA Active Network Working Group, *Architectural Framework for Active Networks Version 1.1*, Dec 2001.

[7] DARPA Active Network Node OS Working Group, *NodeOS Interface Specification*, Jan 2001.

[8] 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.

[9] 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.

[10] D. S. Alexander, M. W. Hicks, P. Kakkar, A. D. Keromytis, M. Shaw, J. T. Moore, C. A. Gunter, J. Trevor, S. M. Nettles, and J. M. Smith in *The 1998 ACM SIGPLAN Workshop on ML / International Conference on Functional Programming (ICFP)*, 1998.

[11] Z. D. Dittia, "ATM Port Interconnect Chip." www.arl.wustl.edu/apic.html.

[12] "NetBSD." http://www.netbsd.org.

[13] Intel Corporation, *Intel 430HX PCISET 82439HX System Controller (TXC) Data Sheet*. Mt. Prospect, IL, 1997.

[14] Intel Corporation, *822371FP (PIIX) and 82371SB (PIIX3) PCI ISA IDE Xcelerator Data Sheet*. San Jose, CA, 1997.

[15] Xilinx, Inc., *Xilinx 1999 Data Book*. San Jose, CA, 1999.

[16] 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," Department of Computer Science, Technical Report WUCS-01-18, Washington University, St. Louis, 2001.

[17] Z. D. Dittia, J. R. Cox, Jr., and G. M. Parulkar, "Design of the APIC: A High Performance ATM Host-Network Interface Chip," in *IEEE INFOCOM '95*, (Boston, USA), pp. 179–187, IEEE Computer Society Press, April 1995.

[18] Z. D. Dittia, G. M. Parulkar, and J. R. Cox, Jr., "The APIC Approach to High Performance Network Interface Design: Protected DMA and Other Techniques," in *Proceedings of INFOCOM '97*, (Kobe, Japan), pp. 179–187, IEEE, April 1997.

[19] J. W. Lockwood, N. Naufel, J. S. Turner, and D. E. Taylor, "Reprogrammable Network Packet Processing on the Field Programmable Port Extender (FPX)," in *ACM International Symposium on Field Programmable Gate Arrays (FPGA'2001)*, (Monterey, CA, USA), pp. 87–93, Feb. 2001.

[20] D. E. Taylor, J. W. Lockwood, and N. Naufel, "Generalized RAD Module Interface Specification of the Field-programmable Port eXtender (FPX)," tech. rep., WUCS-01-15, Washington University, Department of Computer Science, July 2001.

[21] J. W. Lockwood, "Evolvable internet hardware platforms," in *The Third NASA/DoD Workshop on Evolvable Hardware (EH'2001)*, pp. 271–279, July 2001.

[22] F. Kuhns, D. C. Schmidt, and D. L. Levine, "The Design and Performance of a Real-time I/O Subsystem," in *Proceedings of the $5^{th}$ IEEE Real-Time Technology and Applications Symposium*, (Vancouver, British Columbia, Canada), pp. 154–163, IEEE, June 1999.

[23] B. Braden, D. Clark, and S. Shenker, "Integrated services in the internet architecture," *Network Information Center RFC 1633*, June 1994.

[24] The Zebra Organization, "GNU Zebra." http://www.zebra.org.

[25] W. Eatherton, "Hardware-Based Internet Protocol Prefix Lookups," Master's thesis, Department of Electrical Engineering, Washington University in St. Louis, 1998.

[26] E. Leonardi, M. Mellia, F. Neri, and M. A. Marsan, "On the stability of input-queued switches with speed-up," *IEEE Trans. on Networking*, vol. 9, pp. 104–118, Feb 2001.

[27] T. Anderson, S. Owicki, J. Saxe, and C. Thacker, "High speed switch scheduling for local area networks," *ACM Trans. on Computer Systems*, vol. 11, pp. 319–352, Nov 1993.

[28] N. McKeown, V. Anantharam, and J. Walrand, "Achieving 100% throughput in an input-queued switch," *IEEE Trans. Communication*, vol. 47, pp. 1260–1267, 1999.

[29] N. McKeown, M. Izzard, A. Mekkittikul, W. Ellersick, and M. Horowitz, " The Tiny Tera: A Packet Switch Core," in *Hot Interconnects*, 1996.

[30] S. Floyd and V. Jacobson, "Random Early Detection Gateways for Congestion Avoidance," *IEEE/ACM Transactions on Networking*, vol. 1, pp. 397–413, Aug. 1993.

[31] W. Feng, D. Kandlur, D. Saha, and K. Shin, "Blue: A New Class of Active Queue Management Algorithms," Tech. Rep. CSE-TR-387-99, University of Michigan, Apr. 1999.

[32] Keller, R., S. Choi, M. Dasen, D. Decasper, G. Fankhauser, and B. Plattner, "An Active Router Architecture for Multicast Video Distribution," in *Proceedings of IEEE INFOCOM 2000*, (Tel Aviv, Israel), March 2000.

[33] E. Hashem, "Analysis of random drop for gateway congestion control," Tech. Rep. LCS TR-465, Laboratory for Computer Science, MIT, 1989.

[34] R. Morris, "Scalable TCP Congestion Control," in *IEEE INFOCOM 2000*, March 2000.

[35] A. Kantawala and J. Turner, "Efficient Queue Management for TCP Flows," Tech. Rep. WUCS-01-22, Washington University, Department of Computer Science, September 2001.

[36] M. Shreedhar and G. Varghese, "Efficient Fair Queueing using Deficit Round Robin," in *ACM SIGCOMM '95*, Aug. 1995.

[37] P. McKenney, "Stochastic Fairness Queueing," *Internetworking: Research and Experience*, vol. 2, pp. 113–131, Jan. 1991.

[38] D. E. Taylor, J. W. Lockwood, T. Sproull, J. Turner, and D. B. Prlour, "Scalable IP Lookup for Programmable Routers," Tech. Rep. WUCS-01-33, Washington University, Department of Computer Science, October 2001.