

VERA: An Extensible Router Architecture^{*}

Scott Karlin and Larry Peterson

*Department of Computer Science
Princeton University, Princeton, NJ 08544, USA*

Abstract

We recognize two trends in router design: increasing pressure to extend the set of services provided by the router and increasing diversity in the hardware components used to construct the router. The consequence of these two trends is that it is becoming increasingly difficult to map the services onto the underlying hardware. Our response to this situation is to define a virtual router architecture, called VERA, that hides the hardware details from the forwarding functions. This paper presents the details of VERA and reports our preliminary experiences implementing various aspects of the architecture.

Key words: Internet Router, QoS, COTS, Classification, Scheduling

1 Introduction

There is a clear trend towards extending the set of functions that network routers support beyond the traditional forwarding service. For example, routers are programmed to filter packets, translate addresses, make level- n routing decisions, broker quality of service (QoS) reservations, thin data streams, run proxies, support computationally-weak home electronic devices, serve as the front-end to scalable clusters, and support application-specific virtual networks. In general, we expect routers to support a wide assortment of *forwarding functions*, each of which processes and forwards packets in a flow-specific way.

At the same time routers are being programmed to implement new services, emerging hardware is making it possible to build routers from commercial off-the-shelf (COTS) components, including system area network (SAN) switches [9], network

^{*} This is an expanded version of a conference paper by the same name [13]. This work supported in part by NSF grant ANI-9906704, DARPA contract F30602-00-2-0561, and Intel Corporation.

processors [10,25], and programmable line cards [1,22]. In general, we expect non-core routers to be constructed from a rich topology of these components, coupled with general-purpose commodity processors.

The problem such a hardware landscape creates is one of mapping the packet flows that traverse a router—and by implication the forwarding functions that implement services they require—onto a particular hardware configuration. The standard systems response to this situation is to define a virtual router architecture. This paper describes such an architecture, called VERA, that hides the hardware details from the forwarding functions. VERA is designed with the following properties in mind:

Extensible: Because we are implementing *extensible* routers, our design must export an interface and protocol that allows new functionality to be easily added to the router.

Compliant: It can be tempting to overlook some of the requirements of RFC1812 [2] when creating a new router architecture. Our goal is to develop an architecture that supports all of the requirements of a compliant router, with special concern for broadcast and multicast.

Efficient: Subject to the extensibility and compliancy requirements listed above, we want the architecture to support efficient implementations on the given hardware. For example, by taking advantage of the processor on intelligent network interface cards (NICs), many packets can be completely processed and forwarded without involving the main processor at all. By offloading the main processor as much as possible, we leave extra headroom for user-level extensions.

In the development of VERA, we have made a series of design choices that, when taken together, provide a consistent and coherent framework. Nearly every design choice represents a trade-off among performance, complexity, and modularity. Because VERA is designed for implementing extensible internet protocol (IP) routers on a heterogeneous processing environment based on COTS hardware, we have made significantly different design choices than either a general purpose operating system or a router operating system for a single, centralized processor. The main contribution of this paper is to identify and motivate these design choices.

Figure 1 shows how the VERA framework constrains and abstracts the essence of both the routing function space and the hardware configuration space. VERA consists of a router abstraction, a hardware abstraction, and a distributed router operating system. The router abstraction must be rich enough to support the RFC1812 requirements as well as the extensions of interest. The hardware abstraction must be rich enough to support the range of hardware of interest. However, it should expose only enough of the hardware details needed to allow for efficient router implementations. Note that both abstractions must be well “matched” to one another so that the map between them (i.e., the distributed router operating system implementation) is efficient and clean. The abstractions must also be chosen to allow us to model and reason about the system with adequate fidelity without also getting

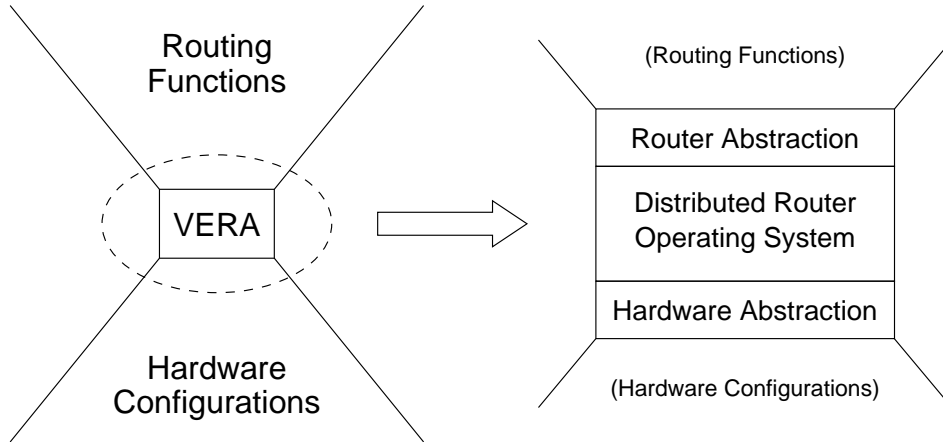


Fig. 1. VERA constrains the space of routing function implementations and hardware exposure to facilitate the mapping between the two.

bogged down by details.

This paper describes each of the three main components: the router abstraction (Section 2), the hardware abstraction (Section 3), and the distributed router operating system (Section 4). For each layer, we both describe the architecture of the layer, and report our preliminary experiences implementing the layer. Section 5 then describes **vera.o**, a Linux device driver that implements the architecture. The paper concludes by discussing related work and offering some early observations.

2 Router Abstraction

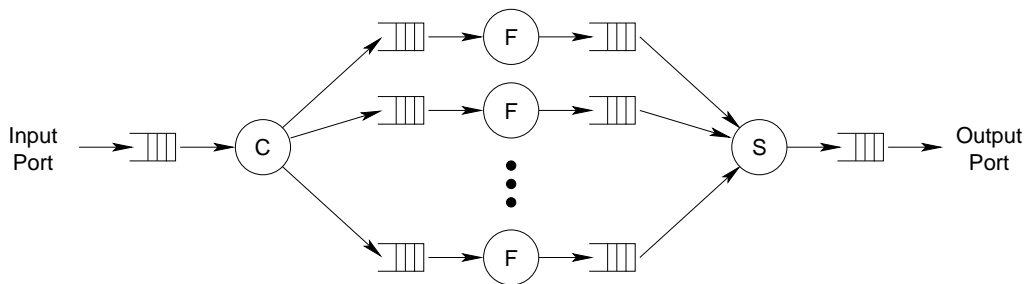


Fig. 2. The classifying, forwarding, and scheduling of IP packets.

This section defines the router architecture visible to programmers writing new forwarding functions. The main attribute of the architecture is that it provides explicit support for adding new services to the router. Figure 2 shows the flow of packets in a router from an input port to an output port. A switching *path* is the instantiation of a forwarder, along with its associated input and output queues. An IP router performs three primary functions on each packet:

Classify: The *classifier* gets packets from an input port, creates an *internal routing header* for the packet, and (based on the contents of the packet) sends the internal routing header to the appropriate path(s). Classifiers map network flows onto switching paths. While classifiers do not modify packets, they can pass information to the path using fields in the internal routing header. The internal routing header is discussed further in Section 4.3. To support multicast and broadcast, classifiers can “clone” packets and send them along multiple paths.

Forward: The *forwarder* gets packets from its single input queue, applies a *forwarding function* to the packet, and sends the modified packet to its single output queue. All transformations of packets in the router occur in forwarders.

Schedule: The *output scheduler* selects one of its non-empty input queues, removes an internal routing header, and sends the associated packet to the output port. The scheduler performs no processing (including link-layer) on the packet.

The router abstraction hides details about the underlying hardware. The abstraction’s model of the hardware is that of a single processor, a single memory pool, and no explicit switching elements. Details of data movement are hidden and all connections appear to be point-to-point. Classifiers, forwarders, and output schedulers run as separate threads. Note that output schedulers are different than the *thread schedulers*. The reason why we require that classifiers and schedulers not modify packets and why we restrict forwarders to a single input queue and a single output queue is motivated by our thread scheduling scheme; we discuss this later in Section 4.2.

At router initialization time, each port has an associated classifier and scheduler. In addition, there is an initial set of pre-established switching paths. To support QoS flows and extensions, our architecture allows paths to be dynamically created and removed by other paths. Section 2.2 gives details about this application programmer interface (API).

2.1 Classification Hierarchy

Our router architecture recognizes that packet classification is not a one-step operation. Instead, we view classification occurring in distinct stages. A simple classification sequence might be:

Sanity Check: the first step of classification is to identify packets which must be ignored or are malformed. Packets that are not identified as malformed are sent to the next level.

Route Cache: at this level, the packet is quickly compared against a cache of known flows to determine the correct path within the router. Packets not in the cache, as well as packets that require special processing (e.g., initial packets of a flow), are sent to the next level.

Prefix Match: most routers run a prefix matching algorithm that maps packets based on some number of bits in the IP header, ranging from just the destination IP address to the source/destination addresses and ports [6,15,24,26].

Full Classification: eventually, packets which have not been classified in early stages will reach a “mop-up” stage which handles all remaining cases, including application-level routing. This stage is often implemented with arbitrary C code.

Figure 3 shows that the internal structure of a classifier is really a hierarchy of sub-classifiers. Once a packet leaves the classifier C , the packets are *fully* classified—a specific path is the target.

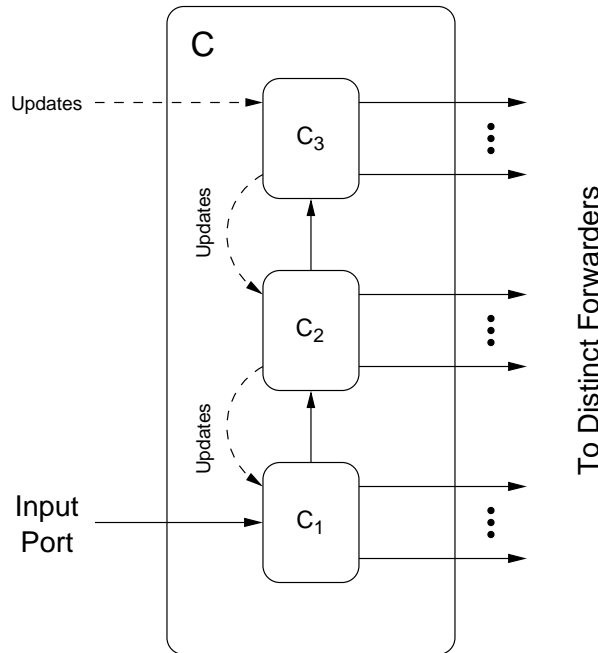


Fig. 3. Classification Hierarchy. Classifier C is composed of partial classifiers C_1 , C_2 , and C_3 . Solid lines indicate packet flow. Dashed lines indicate classification updates (e.g., route table updates).

For our architecture, we impose the restriction that the outputs from a classifier are unique. Referring to Figure 3, this would mean that an arrow from C_1 could not point to the same path as an arrow from C_2 , for example.

Although we believe that this hierarchical structure is a fundamental aspect to packet classification on an IP router [20], the exact form of the hierarchy is often dictated by the processor hierarchy onto which it will be mapped. We return to this issue in Section 4.1.

2.2 Router API

This section outlines some of the router layer API function calls.

$p = \text{createPath}(C, C_parms, F, F_parms, S, S_parms)$

This function creates a new path, p , by instantiating a forwarder, F , and attaching it to the existing classifier, C , and scheduler, S through new queues. (Note that C and S implicitly identify the input and output ports, respectively.) Figure 4 illustrates this process with an example. C_parms include the demultiplexing key needed to identify the flow which should be redirected to this path. F_parms are passed to the forwarder, and include the processor reservation (cycles, memory) required by the forwarder. S_parms include the link reservation needed by the output scheduler.

$\text{removePath}(p, parms)$

This function removes the existing path, p . The additional parameters indicate whether the path should be immediately terminated abandoning any packets in the queues or whether the path should be gracefully shut down by disconnecting it from the classifier first and then letting the packets drain out of the queues.

$\text{updateClassifier}(C, parms)$

This function allows updates (such as new routing tables) to be sent to a classifier.

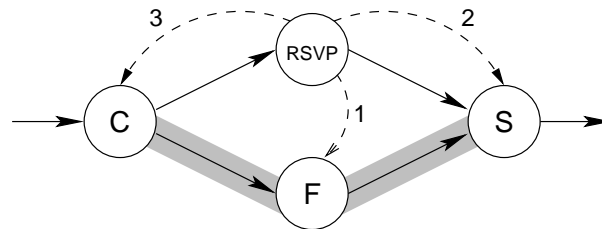


Fig. 4. The basic operations performed by the **createPath** function. Here, an RSVP module (1) instantiates a new forwarder, then (2) attaches to the appropriate output scheduler, and finally (3) attaches to the appropriate classifier.

2.3 Implementation

We have a complete implementation of the router architecture on a single processor with commodity NICs and have used it to build a variety of forwarders. The implementation is based on the Scout operating system [17]. We discuss OS-related issues in Section 4. At this stage, the interesting questions are (1) how Scout fills out the details not specified by the router architecture, and (2) whether or not the resulting system gave us the required flexibility in extending the router's functionality.

Regarding the first question, the most notable way in which Scout enhances the interface visible to the programmer is that it provides a component-based program-

ming model for specifying forwarders. That is, Scout makes it possible to create a forwarder by combining IP modules with “extension” modules that modify standard IP forwarding. For example, we have constructed a forwarder that selectively drops packets during congestion for wavelet-encoded video [14]. This forwarder is specified as **eth/wavelet_drop/ip/eth**, where each of **eth**, **wavelet_drop**, and **ip** are names of modules that can be composed. In addition to defining the function that is applied to each packet, Scout also allows the programmer to specify how many CPU cycles and how much link bandwidth is to be allocated to a forwarder.

As to the second question, we have implemented a variety of forwarding functions, including both “fine-grain” extensions to IP—analogueous to Router Plugins [5] and Click modules [16], and typified by the wavelet dropper example mentioned above—and “coarse-grain” extensions such as TCP proxies (e.g., **eth/ip/tcp/http_proxy/tcp/ip/eth**). Our router abstraction also supports active networking environments. We encapsulate the ANTS execution environment [28] in a forwarder specified as **.../udp/anep/nodeos/ants/nodeos/udp/...**, where **anep** is an active networking protocol layer and **nodeos** defines a wrapper (an execution environment) that isolates untrusted active protocols downloaded into ANTS from the rest of the system [19]. Beyond a variety of forwarding functions, we have also successfully used the interface to establish both best effort and QoS packet flows.

One additional issue that arises from our experience has to do with managing the classification hierarchy. It is often the case that a flow can only be classified by an upper level of the hierarchy, meaning that we need to ensure that lower levels do not. For example, suppose C_i is designed to match layer-4 patterns and C_{i-1} holds a route cache. If a call to **createPath** attaches to the classifier and specifies a level-4 pattern, we need to update C_i to add the new layer-4 pattern and also ensure that there is a miss in the route cache at C_{i-1} . The right side of Figure 5 shows how a cache will obscure higher classification levels; any change to the tree will require that the cache be updated to remain consistent.

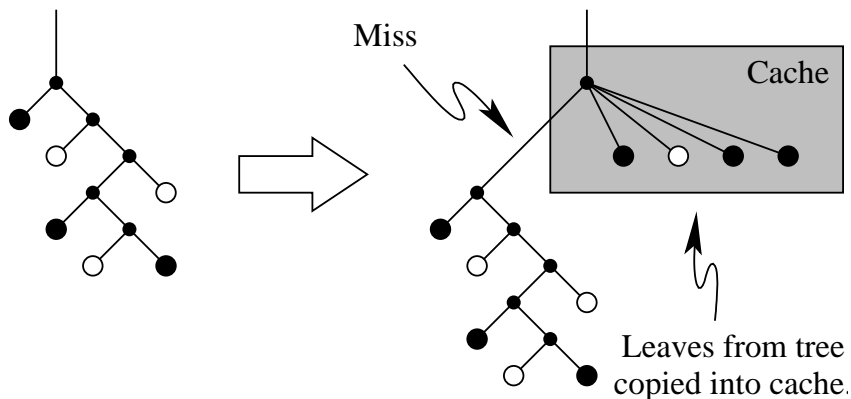


Fig. 5. A partial classifier acting as a route cache. The left side shows a routing decision tree: internal nodes are decision points, leaves are path selections. The right side introduces a route cache. Misses must traverse the master tree.

3 Hardware Abstraction

This section describes the hardware abstraction layer for VERA. The object of any hardware abstraction layer (HAL), is to define an interface between the hardware and the “device independent” upper level software (typically, an operating system). This allows us to support a variety of hardware configurations without rewriting the operating system. Choosing the appropriate level of abstraction is somewhat of an art. We want to choose the abstraction level so that everything below is part of a consistent and uniform HAL and everything above does not directly access the hardware. If we select too high a level of abstraction, each port to a new hardware configuration will require a major effort. If we select too low a level of abstraction, we will not be able to take advantage of higher-level capabilities provided directly by the hardware without breaking through the abstraction layer.

3.1 Example Hardware

Before discussing the architecture in detail, we present block diagrams for two example hardware configurations. This gives us a concrete reference when discussing the abstraction.

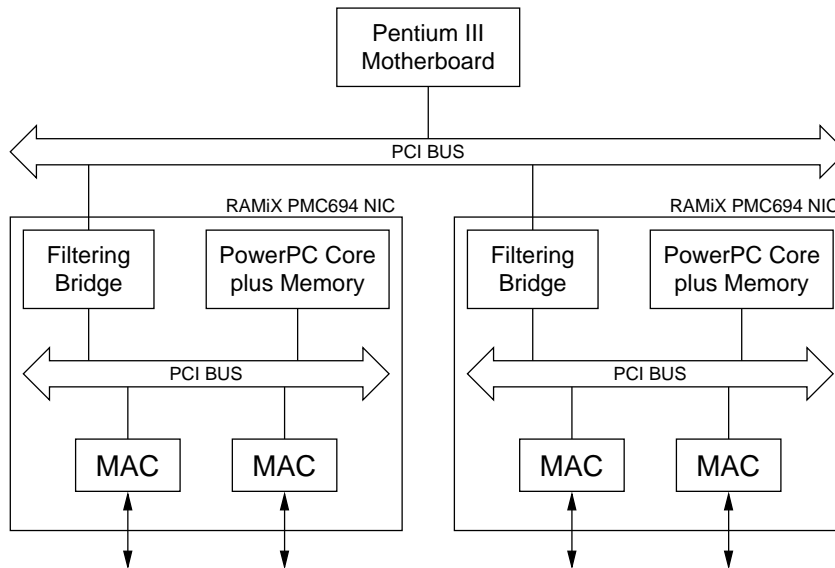


Fig. 6. A four-port router using embedded PowerPC cores as network processors with a Pentium III as the master processor.

Figure 6 shows an example 4-port router based on commercially available components. (Details on the components are in Section 3.4.) While this router configuration is small, it has two especially interesting features: (1) a high processor cycle to port bandwidth ratio, and (2) substantial processing cycles “close” (low-latency) to the ports.

Figure 7 shows a larger configuration with thirty-two 100Mbit/s Ethernet ports based on the IXP1200 network processor [10], the IX Bus, and the IXB3208 bus scaling fabric. By using four IXB3208 chips, this design scales to 128 ports and eight IXP1200 processors.

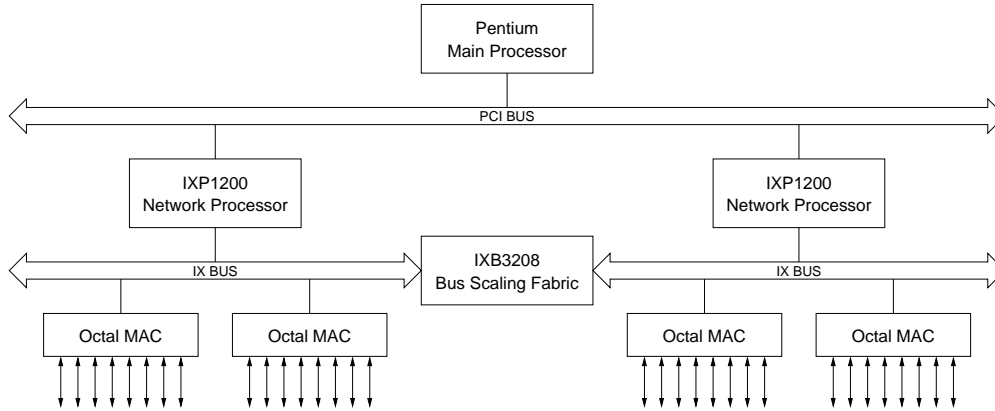


Fig. 7. A hypothetical 32-port router using IXP1200 network processors with a Pentium as the master processor.

In our architecture, we also consider loosely coupled clusters of personal computers that utilize gigabit Ethernet, InfiniBand, or some other SAN technology as the underlying switching technology.

3.2 Hardware Abstraction

The hardware abstraction layer for VERA can be broken down into three major sections: processors, ports, and switches. The abstractions for the ports and processors are fairly standard. The abstraction for the switching elements is more involved and represents the bulk of the hardware abstraction layer. We describe each of the abstractions here:

Processors: The hardware abstraction layer groups the actual processors into virtual processors. Each virtual processor is either a single processor, a symmetric multiprocessor (SMP), or a complex/hybrid processor like the IXP1200 (which has a StrongARM core and six programmable microengines). The relevance is that each virtual processor is its own *scheduling domain* with a single thread pool. Also, any memory local to a processor is consolidated with and owned by that processor.

Ports: A device driver hides the register level details of the media access controller (MAC) chips and provides a uniform interface for upper layers. Each MAC chip is “owned” by a particular processor. The interface exports a scatter/gather capability that can read and write the header and data from separate memory locations. Note that these memory locations must be local to the processor that controls the port.

Switches: The switching elements are modeled as passive (no processing cycles) and shared (all devices connected to a switch share a single bandwidth pool). This also means that there is no explicit control registers accessible to the software to schedule data movement through the switch. VERA’s switch abstraction provides an interface for interprocessor data movement and *distributed queues* whose head and tail are on different processors. Together, these are the primitives needed by the distributed router operating system to implement the interprocessor communication and message passing which is the basis for all the data movement within the router. Sections 3.4.1 and 3.4.2 discuss the details of data movement and queues in greater detail.

In addition to the abstractions of the processors, ports, and switches, the hardware abstraction maintains a static database containing the topology of the system, as well as the capabilities of the components.

The router topology is the connected graph of the *direct* connections among the components of the router. By ignoring the switches and considering only the processors and ports, we can find a spanning tree with the *master processor* at the root and all the ports as leaves. This spanning tree is called the *processor hierarchy*. Figures 8 and 9 show the hardware abstraction of the architectures shown in Figures 6 and 7, respectively. The nodes of the graphs are the processors, switches, and ports (MACs). The graph edges (solid lines) indicate packet switching paths. The dashed arrows indicate the edges of the spanning tree defining the processor hierarchy. Figures 8 and 9 both have three distinct switching elements because both the filtering bridges of Figure 6 and the IXB3208 of Figure 7 segregate the buses and partition the bandwidth.

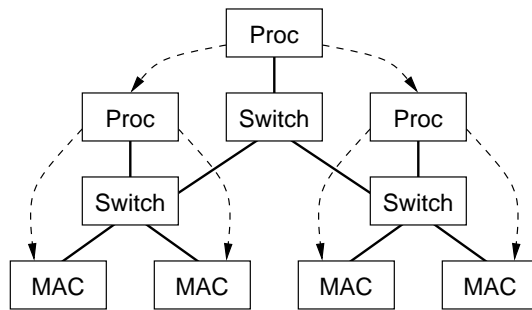


Fig. 8. The hardware abstraction of Figure 6. The solid lines indicate packet flow paths. The dashed arrows show the processor hierarchy.

As mentioned above, the hardware abstraction layer maintains a static database of the component capabilities. The capabilities include the bandwidth of the ports and switches as well as the processing cycles available to the upper software layers. Specifically, this information is used by the distributed router operating system to determine appropriate placement of threads. It is important to note that the HAL only advertises *available* processing cycles and switching capacity; this takes into account any cycles and switching capacity needed to implement the abstraction

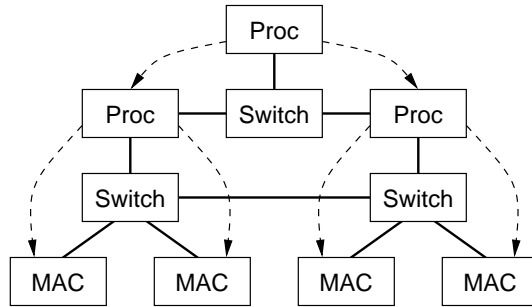


Fig. 9. The hardware abstraction of Figure 7. The solid lines indicate packet flow paths. The dashed arrows show the processor hierarchy.

layer itself.

3.3 Hardware API

This section outlines some of the hardware layer API function calls. The following two functions hide details of how the hardware moves data:

putData(local, remote, size)

This function pushes a block of data of length *size* from a *local* address to a *remote* address.

getData(remote, local, size)

This function pulls a block of data of length *size* from a *local* address to a *remote* address.

The following three functions hide details of how the hardware queues data:

q = allocHWqueue(dir, depth)

This function allocates a distributed queue, *q*, from a fixed-size pool of queues. The queue is configured to hold *depth* entries each of which is a pointer-sized integer. The direction parameter, *dir*, can be set to either **incoming** or **outgoing**. An incoming queue supports multiple remote writers and an outgoing queue supports multiple remote readers. Note that the local processor (which made the call to **allocHWqueue**) cannot write to incoming queues and cannot read from outgoing queues.

push(queue, item)

This function pushes the *item* (a pointer-sized integer) on the given *queue*. The *queue* must be either a locally allocated outgoing queue or a remotely allocated incoming queue.

item = pop(queue)

This function pops the *item* (a pointer-sized integer) from the given *queue*. The *queue* must be either a locally allocated incoming queue or a remotely allocated outgoing queue.

3.4 Implementation Issues

We now discuss two implementation issues—data movement and queue management across the PCI bus—for the PMC694 NIC and the IXP1200 Ethernet Evaluation Board (EEB). The key property of these configurations is that they employ a (PCI) bus-based switch. We later briefly comment on the implementation issues that arise on configurations that use other switching technology.

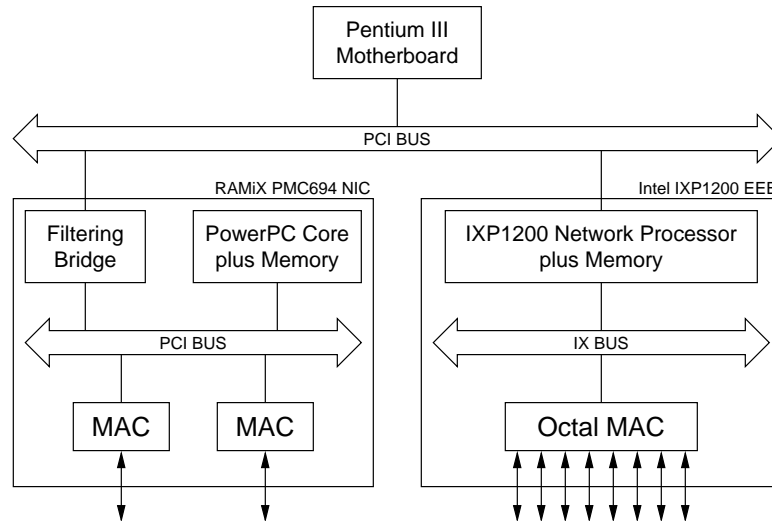


Fig. 10. Testbed based on a Pentium III motherboard with both a PMC694 NIC and an IXP1200 EEB.

Figure 10 shows our development testbed consisting of a commodity motherboard connected to two different off-the-shelf network interface cards using a standard 33MHz, 32bit primary PCI bus. The motherboard is an Intel CA810E with a 133MHz system bus, a 733MHz Pentium III CPU, and 128Mbytes main memory.

The first NIC is a RAMiX PMC694 [22]. It contains its own 266MHz PowerPC processor, two 100Mbit/s Ethernet ports, and 32Mbytes of memory. The primary PCI bus (of the motherboard) is isolated from the secondary PCI bus (of the PMC694) with an Intel 21554 non-transparent PCI-to-PCI bridge. The secondary PCI bus is also 32bits and operates at 33MHz. The PMC694 has a two-channel direct memory access (DMA) engine and queue management hardware registers used to support the Intelligent I/O (I₂O) standard [12].

The second NIC is an Intel IXP1200 Ethernet Evaluation Board (EEB) which is sold as part of an evaluation kit [11]. The board contains a 199MHz IXP1200 network processor, eight 100Mbit/s Ethernet ports, and 32Mbytes of memory. The IXP1200 chip contains a general-purpose StrongARM processor and six special-purpose MicroEngines. Like the PMC694, this boards also has a two-channel DMA engine and I₂O support registers.

3.4.1 Data Movement

The common denominator of all IP routers is that they move data from one network interface to another. The HAL exports an API that allows any of the processors in the router to either push (put) or pull (get) data to or from any of the other processors in the router.

Focusing on PCI-based configurations, we note that the PCI bus efficiency is highest when data is transferred in long *bursts*. This is a direct consequence of the fact that a PCI bus transaction consists of an *address phase* followed by one or more *data phases*. The address phase specifies the address of the transfer for the first data phase; each subsequent data phase within the same bus transaction occurs at the next higher address. The address phase takes one PCI clock cycle; in the best case, each data phase takes one PCI clock cycle. For each data phase, either the master or the target may insert *wait states* (each of which takes one PCI clock cycle). Note that read operations have a mandatory wait state in their first data phase and each transaction must be separated by at least one PCI clock cycle. By transferring data in bursts (bus transactions with many data phases), the overhead (per byte transferred) of the address phase is reduced.

Because processors cannot always consistently generate these efficient bursts, they are often augmented with DMA engines specifically designed to use bursts to transfer blocks of data. Both the PMC694 and the IXP1200 have DMA engines which consistently generate burst transfers. Note that the Pentium motherboard does not have a high-speed PCI DMA engine.

Due to the freedom in the PCI specification, different board designs will exhibit different performance characteristics. We performed a series of measurements of the PCI bandwidth between the motherboard and each of the boards of Figure 10. Both the host processor (the Pentium III) and the on-board processor of the NICs can arbitrate for and then become the PCI bus master which allows them to initiate transfers. By using read or write instructions (or a DMA engine if available) a processor can pull or push data across the bus. Our experimental results are discussed in the following paragraphs.

PMC694 PCI Bandwidth

Table 1 summarizes the results of our experiments with the PMC694. Here the Pentium processor runs a Linux 2.2 kernel while the PowerPC runs our “raw” code with no operating system. Under programmed I/O (PIO), we measured several different data movement techniques. These differ in the number of bits transferred per loop iteration (Transfer Size). We measured transfer size of 64 bits and 128 bits by unrolling the 32-bit loop two times and four times, respectively. We obtained our best times by using the ANSI C memcpy library routine which uses tuned assembly routines. Our code was written in ANSI C.

Bus Master	Transfer Size	Mode	Direction	64-Byte Packets		1500-Byte Packets	
				Kpps	MByte/sec	Kpps	MByte/sec
host	32 bits	PIO	push	919.9	58.88	39.37	59.06
host	64 bits	PIO	push	975.0	62.40	41.58	62.37
host	128 bits	PIO	push	962.4	61.60	41.53	62.30
host	(memcpy)	PIO	push	1073.4	68.70	44.29	66.44
host	32 bits	PIO	pull	61.1	3.91	2.61	3.91
host	64 bits	PIO	pull	61.8	3.95	2.63	3.95
host	128 bits	PIO	pull	61.9	3.96	2.64	3.95
host	(memcpy)	PIO	pull	62.2	3.98	2.66	3.99
card	32 bits	PIO	push	365.5	23.39	14.84	22.26
card	64 bits	PIO	push	363.4	23.26	14.81	22.21
card	128 bits	PIO	push	363.9	23.29	14.81	22.21
card	—	DMA	push	534.2	34.19	17.57	26.35
card	32 bits	PIO	pull	53.7	3.44	2.29	3.43
card	64 bits	PIO	pull	53.8	3.44	2.28	3.41
card	128 bits	PIO	pull	53.8	3.44	2.28	3.41
card	—	DMA	pull	354.1	22.66	15.74	23.61

Table 1

Raw PCI Transfer Rates Between the PMC694 (card) and the Pentium III Motherboard (host).

IXP1200 PCI Bandwidth

Table 2 summarizes the results of our experiments with the IXP1200 EEB. These experiments are analogous to the PMC694 experiments. As with the PMC694 experiments, the Pentium processor runs a Linux 2.2 kernel while the StrongARM runs our “raw” code with no operating system. We locally modified the firmware and jumper settings to run the board without an operating system and to allow it to be plugged into the PCI bus of a commodity motherboard. Note that the MicroEngines were not used in this experiment. (Details on our MicroEngine software architecture can be found in [23].)

PCI Bandwidth Generalizations and Observations

From Tables 1 and 2, we see that the fastest way to move a packet from the card to the host and back to the card is for the card to push the packet to the host using its DMA controller and then have the host push the packet back to the card using memcpy-based PIO. (The notable exception is that due to the DMA setup overhead, small packets should be sent using programmed I/O on the IXP1200. Our experiments show that the crossover point is at approximately 168 bytes/packet.)

Bus Master	Transfer Size	Mode	Direction	64-Byte Packets		1500-Byte Packets	
				Kpps	MByte/sec	Kpps	MByte/sec
host	32 bits	PIO	push	1030.2	65.93	44.2	66.34
host	64 bits	PIO	push	1028.9	65.85	44.1	66.16
host	128 bits	PIO	push	1030.1	65.93	44.0	66.14
host	(memcpy)	PIO	push	1073.9	68.73	44.2	66.35
host	32 bits	PIO	pull	63.7	4.08	2.7	4.05
host	64 bits	PIO	pull	63.7	4.08	2.7	4.04
host	128 bits	PIO	pull	64.3	4.11	2.7	4.09
host	(memcpy)	PIO	pull	64.4	4.13	2.7	4.12
card	32 bits	PIO	push	349.2	22.35	14.7	22.12
card	64 bits	PIO	push	387.1	24.78	16.5	24.81
card	128 bits	PIO	push	387.1	24.78	16.5	24.81
card	—	DMA	push	204.7	13.10	32.7	49.01
card	32 bits	PIO	pull	69.1	4.43	2.9	4.42
card	64 bits	PIO	pull	69.8	4.47	3.0	4.46
card	128 bits	PIO	pull	71.1	4.55	3.0	4.53
card	—	DMA	pull	179.2	11.47	16.4	24.66

Table 2

Raw PCI Transfer Rates Between the IXP1200 EEB (card) and the Pentium III Motherboard (host).

Since there is only one DMA engine per NIC, this engine becomes a critical resource that is explicitly managed by the HAL. (An additional advantage of using DMA over programmed I/O is concurrency; after issuing a DMA request, a thread can either move on to other computations or yield to another thread.)

Moreover, because the PCI bus is a shared resource and there is no bandwidth reservation mechanism, the HAL must coordinate among the processors when using its DMA engine. For example, suppose processors A, B, C, and D are all connected to the same PCI bus. Suppose there is a QoS flow between processors A and B. In this case the HAL must ensure that a best effort flow between processors C and D does not prevent the QoS flow from meeting its reservation. To effectively support QoS flows in the router, the system must allow reservations on all of the limited, shared resources.

Thus, rather than simply hide the DMA engine beneath a procedural interface, we dedicate a server thread to the DMA engine on each processor. This thread coordinates with its peers to allocate the shared bandwidth available on the bus, as well as manages the local engine and DMA queues. The server supports two DMA work queues: a low-priority queue for packet traffic and a high-priority queue inter-processor control messages (e.g., messages that release packet data buffers.)

Because our hardware model allows multiple intervening processors and switching elements between any two processors, the intervening processors must store-and-forward the packets. While this appears to be the problem that the router as a whole is trying to solve (IP routing), the problem is significantly easier because, unlike the Internet, we have a centralized controller in the master processor. The HAL hides the details of any necessary forwarding.

3.4.2 *Queues*

Another fundamental aspect of IP routers is that the packets of each network flow should be processed in FIFO order. Because we statistically multiplex the processing cycles among the ports, the router must occasionally buffer packets in distributed queues. Recall that a distributed queue has a tail located on one processor and the head located on a second processor.

Our requirement that the distributed queues support multiple readers and multiple writers on the remote side of the queue stems from the fact that we anticipate the number of available queues with hardware support to be limited. (In fact, the PMC694 only supports 2 queues in each direction at the interface to the PowerPC core.) Due to this limitation, we anticipate that these queues will need to be shared. We revisit this issue in Section 4.4.

Without special hardware support, implementing a multiple reader or writer queue over the PCI bus would require the processors to acquire and release semaphores using software techniques (for example, spinning on remote memory locations [7]). This is because the current version of the PCI bus specification [18] no longer supports bus locking by arbitrary bus masters. Fortunately, components that support I₂O have hardware registers that directly implement multiple reader or writer queues. They do this by hiding a FIFO queue behind a single PCI mapped register. When a processor reads the hardware register, there is a side effect of updating the queue pointer. The read and update happen atomically.

Because we take advantage of I₂O support as the basis for **push** and **pop**, we also must live with the restrictions of I₂O. Specifically, the local processor cannot access the registers used by the remote processors and vice versa. This is the reasoning behind the corresponding restrictions in **push** and **pop**. The benefit is that the implementation of **push** and **pop** can be as simple as a memory-mapped register write and read, respectively.

3.4.3 *Other Switching Hardware*

Our implementation effort up to this point has focused on the PCI bus as a switching element, but we considered other technologies when defining the HAL. There are two considerations when using other switches. First, the HAL defines operations for

both pushing and pulling data across the switch. This is a natural match for a bus, which supports both read and write operations, and is consistent with interfaces like VIA [4] (which form the basis for future interfaces like InfiniBand [9]). On a switch component that supports only push (send) operations (e.g., an Ethernet switch), pull will have to be implemented by pushing a request to the data source, which then pushes back the data.

Second, the bus offers only best-effort, shared bandwidth. On a switching element that supports either dedicated point-to-point bandwidth (e.g., a crossbar) or virtual point-to-point bandwidth (e.g., InfiniBand channels), the role of the DMA server thread diminishes. In effect, the hardware supports the point-to-point behavior that the HAL requires. Of course a best effort switch (e.g., Ethernet switch) will have to be managed much like a PCI bus.

4 Distributed Router Operating System

As shown in Figure 1, the job of the distributed router operating system is to provide an execution environment for the forwarding functions which ties together the router abstraction and hardware abstraction. A case in point is that the OS must bridge the semantic gap between the high-level router abstraction and the low-level HAL; the OS *implements* the **createPath** call using the data movement and hardware queue support functions of the HAL. In addition to tying together these core abstractions, the OS provides a computation abstraction in the form of a thread API and a memory abstraction in the form of both a buffer management API and an internal routing header datatype.

We use the Scout operating system as a prototype implementation of VERA for the special case of a single switch connecting a single processor to multiple MAC devices. In the following subsections, we outline the major features and abstractions provided by the operating system. We are in the process of extending Scout with VERA support for distributed configurations.

4.1 Processor Hierarchy

In our abstraction, the processors are organized into a hierarchy. The dashed arrows in Figures 8 and 9 shows how the master processors directly control the subordinate processors which eventually control the MAC devices. At router initialization time, administrative message queues are created from the master processor to each of its child processors, and so on down the hierarchy. The single, master processor maintains the master copies of the routing tables and controls the overall operation of the router. Since each processor operates independently and has its own thread

scheduler, the control the master processor has over the other processors is, by necessity, coarse grained.

The processor hierarchy nicely maps to the classification hierarchy of Figure 3. The first partial classifier, C_1 , always runs on the processor controlling the MAC for that input port. The last partial classifier, C_n , always runs on the master processor. Intermediate classification stages can be mapped to either the current processor, or the next processor up the processor hierarchy. Each classifier exports a single interface. This interface is only exported on the master processor. Updates to the classifier are made via this interface and then trickle down the classification hierarchy as needed. This means that the OS must propagate router abstraction level calls to **updateClassifier** through the processor hierarchy to the master processor where they can be fed into the classifier at the top level.

4.2 Thread Assignment and Scheduling

The OS provides an abstraction of computation in the form of a thread API. At the router abstraction level, each forwarder and output scheduler runs as a separate thread on a particular processor. As mentioned above, classifiers are multithreaded with the first level of classification running on the processor which owns the input port and the last level of classification running on the master processor. The output scheduler threads run on the processor that owns the output port. Forwarder threads and intermediate classification threads can run on *any* processor in the router. Intermediate classification threads are statically assigned to processors at initialization time.

In Figure 4 we saw an example of the **createPath** instantiating a new path. An important part of path creation is identifying on which processor to instantiate the forwarder. The OS supports this decision by maintaining a database of resource reservations for processor cycles and internal switching cycles. In addition to the traditional object code, a *forwarder object file* also contains per packet processing and switching requirements for each processor architecture supported by the forwarder object file. During a **createPath** call, the OS uses the resource database to determine a list of eligible processors that have enough processing cycles to run the function. With the internal router graph, the OS attempts to find a path that has enough switching cycles to support the request. Paths are greedily considered from shortest to longest. If no path is found which can support the request, the call to **createPath** fails.

After new forwarder threads are instantiated, they must be scheduled along with all the other classifier, forwarder, and output scheduler threads. Because our architecture supports a heterogeneous distributed processing model, the OS must provide support for scheduling computations across the entire router. In Section 3.2 we de-

fined a *scheduling domain* as a tightly bound set of processors (usually, a single processor) running a single thread scheduler. Because the amount of processing required on each packet is small, we have chosen not to have a single, fine-grained scheduler for all the threads of the router. Instead, each processor runs a completely independent scheduler. The master scheduler (running on the master processor) provides only coarse grain adjustments to each scheduling domain to guide their independent scheduling decisions.

The scheduler running in each scheduling domain builds upon our existing scheduling work [21], which in turn, is derived from the WF²Q+ scheduler [3]. We have added hooks to support coarse grain influence from the master processor. In brief, the scheduler assigns a processor share to each forwarder according to the F_parms passed to the `createPath` operation. Processor shares are also assigned to the scheduler and classifier threads. The share-based scheduler is augmented to use information about the states of the queues to determine when a forwarder or output scheduler thread should run. In the case of an output scheduler, packets in the queue are assumed to be in local memory so that when an output scheduler thread is activated it will not immediately block attempting to fetch the packet data from a remote processor. This helps keep the output link full. We explore the issue of ensuring that the packet data is local to the output scheduler in the next section.

In Section 2 we stated several design rules, including: (1) classifiers must completely classify a packet and not modify the packet, (2) forwarders must have a single input and a single output, and (3) output schedulers do not modify packets. These design rules are motivated by the need to support QoS in our router. When a QoS reservation is made, certain resources are reserved for the corresponding path. The thread scheduler must ensure that the threads belonging to particular QoS paths get the share of the processor they need to meet their reservation. Because the classification threads must be scheduled based on link bandwidth (because the flow for a packet is unknown until *after* classification), we want the classifiers to perform the minimum amount of processing. (This allows the thread scheduler to have control over the maximum amount of processing.) Once packets are classified, forwarder threads can be scheduled based on the QoS reserved for the corresponding flow. Our design stipulates that output schedulers not modify packets. This allows the scheduler to make its thread scheduling decision based on the state of the queues going into and coming from the output scheduler without needing to estimate the amount of processing which might need to be performed on the packet. Because no packet processing occurs in the classifier or output scheduler, all the processing must occur in the forwarder. We have chosen not to perform link-layer processing in the output scheduler, the forwarder can perform arbitrary processing at the link layer. The downside is that if there are N different media types in the router, each processor must be capable of handling all N link layers. However, we expect N to be small enough that $N \times N$ translation will not be an issue.

4.3 Internal Packet Routing

It is well known that routers should internally copy data as little as possible. Our architecture helps reduce the amount of data copying by sending an *internal routing header* (rather than the entire packet) from place to place. This internal routing header contains the initial bytes of the packet plus an ordered scatter/gather list of pointers to blocks containing the data of the packet. A reference count is associated with each block; the OS uses this count to determine when a block can be recycled. (The classifier changes the reference count from its nominal value of one when a multicast or broadcast packet is detected; the reference count is set to the number of destinations.)

An interesting property of IP routers is that in most cases only the header need be modified by the forwarder; the body/payload of the packet remains unchanged. When activated by the thread scheduler, a forwarder first reads the next internal routing header from its input queue, fetches any (remote) packet data it needs and then performs the computation. After processing the packet, the forwarder sends the updated internal routing header to its output queue (which is connected to the appropriate output scheduler). It is the responsibility of the queue API to make sure all the packet data is local to the output scheduler before the internal routing header is placed on the queue. Because the classification hierarchy and the forwarder on the invoked path may not have needed the entire packet to complete the classification and requisite forwarding function, the packet may be scattered in several blocks across several processors. Anytime a block is moved, the internal routing header is updated with the new location. When the internal routing header reaches the output scheduler's processor, the data must be copied to local memory before the internal routing header can be placed on the output scheduler's inbound queue. Figure 11 illustrates the sequence.

- (1) The forwarder, F , send an internal routing header (IRH) to the output scheduler, S .
- (2) The *queue server* (QS), preprocesses the IRH to determine the location of the packet data.
- (3) The QS uses the HAL data movement primitives to fetch the packet data.
- (4) The QS modifies the IRH to indicate that the data is now local and places it on the input queue where it is visible to S .
- (5) S reads the IRH from the queue.
- (6) S directs the data from local memory to the device queue.

Until now we have discussed the issues with moving the data through the router. Eventually, these buffers must be recycled when the packet data is no longer needed. The HAL data movement commands are really data *copying* commands. It is the responsibility of the OS to manage the buffers. When we wish to *move* a buffer referenced by an IRH, we send a release message to the source processor. The pro-

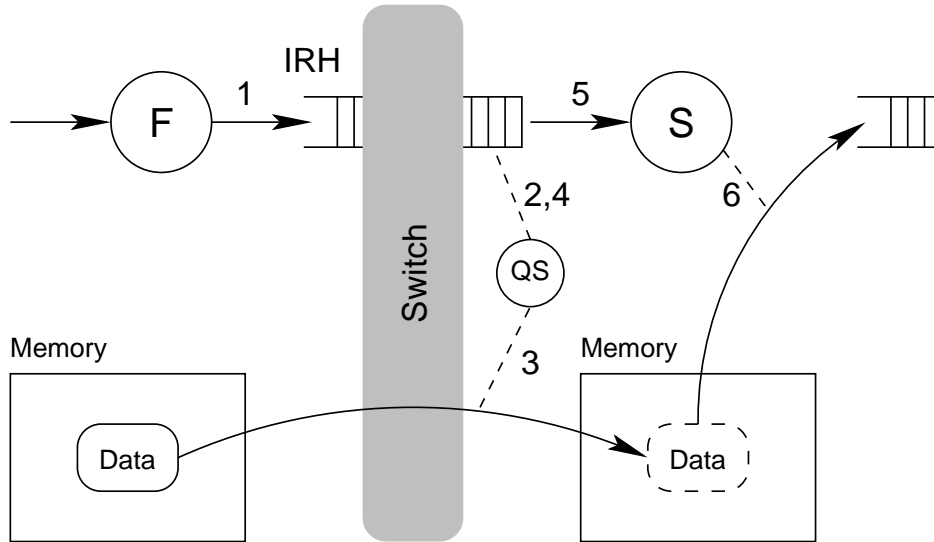


Fig. 11. This shows the steps performed by the OS when an internal routing header (IRH) is sent from a forwarder, F , on one processor to an output scheduler, S , on another processor. (See text.)

cessor decrements the reference count associated with the block(s) and reuses any whose count has reached zero.

4.4 Queues

Recall that our queuing system is modeled after I_2O . This means that we are pushing and popping *pointers* to IRHs. In order to effect the transfer of an IRH, we use two hardware-level queues to implement the high-level queues of the router abstraction. To simplify the explanation of the process, we will temporarily refer to the low-level queues as (circular) buffers. One buffer contains pointers to empty IRH frames, and one buffer contains pointers to to-be-processed IRH frames. Putting an IRH onto a queue involves first pulling a pointer to a free frame from the free-frame buffer, filling the frame using the data movement primitives in the HAL, and then placing the pointer on the to-be-processed buffer. Getting an IRH from a queue involves first retrieving a pointer from the to-be-processed buffer, processing the data, and then returning the block to the free pool by placing its pointer on the free-block buffer. We elect this method because I_2O gives us hardware support for managing these circular buffers and we want to take advantage of this support when it is available.

With the path of each flow requiring two high-level queues (one of which will usually span a switching element) and the router handling 1000's of flows, the OS must be able to efficiently create and destroy queues. Because the limitation imposed by the hardware (Section 3.4.2) will generally be much less than the number of queues the router must support, the OS must use the limited hardware to create

a larger number of virtual queues. These virtual queues are multiplexed on top of the hardware queues.

When a virtual queue is established, it is given an identification key. Each queue element is tagged with the key of the destination virtual queue. A demultiplexing step occurs on the receiving end to assign the incoming item to the correct queue. Note that this step can be combined with the queue server thread of Figure 11.

5 Realization in Linux

We have implemented a Linux kernel module which is a unified device driver for both the PMC694 and the IXP1200 EEB. (By choosing to develop a Linux kernel module, our device driver will also work with the communication-oriented operating system, Scout [17].) Figure 12 shows the driver instantiated with one IXP1200 EEB and two PMC694 boards. In this router configuration, there are a total of twelve 100Mbit/s Ethernet ports.

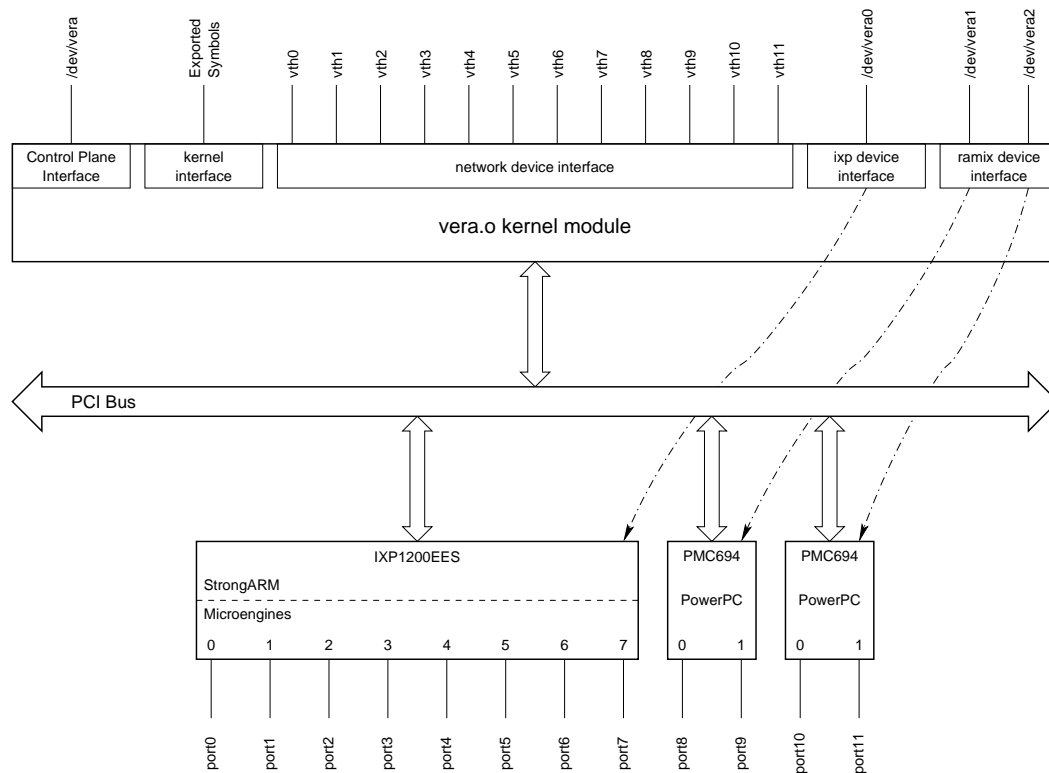


Fig. 12. The `vera.o` kernel module. In this figure, the module has been instantiated in a system containing one IXP1200 EEB and two PMC694 boards.

As shown in Figure 12, there are four types of interfaces to the driver:

Control Plane Interface: The driver exports a character device interface as `/dev/vera`. This is used to perform updates (from user space) which do not apply to a specific hardware device or network interface. (For example, routing table updates made via the classification hierarchy as discussed in Section 2.1.)

Kernel Interface: The module exports symbols which allow kernel modules which are dynamically loaded (after `vera.o` is loaded) to extend the `ioctl` interface to either `/dev/vera` or `/dev/veraN`.

Virtualized Network Interfaces: A key element of the module is that the physical network interfaces are *virtualized*. Because we want the intelligent NICs to route the packets directly between hardware devices (either within a board or from board to board), many packets will never arrive at the network driver interface to the Linux kernel. However, packets which do arrive on a particular port and which are to be handled by the kernel are sent to their corresponding virtual interface. In this way, packets which are not handled by VERA can be processed normally by the Linux kernel.

Device Interfaces: When the module is instantiated, a character device of the form `/dev/veraN` is assigned to each PMC694 or IXP1200 EEB device. This interface allows the boards to be initialized, code to be downloaded, and gives access to the memory and registers of each board.

6 Related Work

Many currently available general-purpose operating systems such as Unix or Windows can be configured to route Internet packets. However, as general-purpose systems, they were not necessarily designed with packet forwarding in mind. For example, we have measured Linux to be up to six times slower forwarding vanilla IP packets than Scout [21]. More importantly, however, such general-purpose systems provide no explicit support for adding new forwarders. Unfortunately, router operating systems in commercial products are closed, and the extent to which they are extensible is not clear.

Recent systems like Click [16] do support extensibility, but do not consider distributed heterogeneous environments. Other efforts to define architectures for active network also support extensibility [8,28], but pay little attention to either performance or compliance. In contrast, our focus is on how to make a compliant IP router extensible.

The Router Plugin [5] system is the closest to our effort in that it considers both extensibility and high-performance distributed architectures. The main difference is that we envision the extensible forwarders running across a hierarchy of commodity processors, while Router Plugins are restricted to the processor on the network in-

terface card. A second difference is that our architecture explores the design space of extensible internet routers implemented on the heterogeneous processing environment of commodity PC components, while the Router Plugins effort is focused on the design of custom hardware.

Walton, et. al. [27] describe how to make efficient use of the PCI bus on a PC-based router by exploiting peer-to-peer DMA. We take advantage of the same technique, but extend their work in several ways. For example, we use intelligent NICs with much higher processor cycle to port bandwidth ratios than they consider; we assume the possibility of moving non-standard forwarders onto the interface cards. We also pay particular attention to how the various resources (bus, CPU) are scheduled, and the synchronization problems that occur when multiple sources are trying to access the same sink devices.

The hardware abstraction's switch API is similar to the VIA [4] and I₂O [12] interfaces. VERA's interface is significantly smaller and specifically tailored to the needs of a router. However, as discussed in Section 3.4, components designed to support VIA and I₂O are particularly well suited to support many of VERA's hardware abstractions directly in hardware.

7 Concluding Remarks

This paper describes a virtual router architecture, called VERA, that bridges the gap between the new services being written to run on IP routers and the COTS hardware that we expect will soon be used to build non-core routers. The paper's main contribution is to identify the design decisions we made while defining and implementing VERA.

The design consists of three layers. First, VERA defines an upper interface, called the router abstraction, that has explicit support for extensibility. We have used a prototype implementation of this interface to build a variety of new forwarders. Second, VERA defines a lower interface, called the hardware abstraction, that hides the details of the underlying hardware. At this level, we have uncovered the many details involved in data movement and queue management in a distributed environment. Third, a distributed OS ties the upper and lower abstractions together. At this layer we have preliminary experience with scheduling and resource allocation based on a prototype implementation in Scout.

References

- [1] Alteon WebSystems, Inc., San Jose, California, ACEnic Server-Optimized 10/100/1000 Mbps Ethernet Adapters Datasheet (August 1999).
- [2] F. Baker, Requirements for IP Version 4 Routers; RFC 1812, Internet Request for Comments .
- [3] J. C. R. Bennett, H. Zhang, Hierarchical Packet Fair Queueing Algorithms, in: Proceedings of the ACM SIGCOMM '96 Conference, 1996, pp. 143–156.
- [4] Compaq Computer Corporation, Intel Corporation, Microsoft Corporation, Virtual Interface Architecture Specification, Version 1.0 (December 1997).
- [5] D. Decasper, Z. Dittia, G. Parulkar, B. Plattner, Router Plugins: A Software Architecture for Next Generation Routers, in: Proceedings of the ACM SIGCOMM '98 Conference, 1998, pp. 229–240.
- [6] M. Degermark, A. Brodnik, S. Carlsson, S. Pink, Small Forwarding Tables for Fast Routing Lookups, in: Proceedings of the ACM SIGCOMM '97 Conference, 1998, pp. 3–14.
- [7] A. N. Habermann, Introduction to Operating System Design, Science Research Associates, Inc., 1976, pp. 72–75.
- [8] M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, S. Nettles, PLAN: A packet language for active networks, in: Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming Languages, 1998, pp. 86–93.
- [9] InfiniBandSM Trade Association, InfiniBandTM Architecture Specification, Release 1.0 (October 2000).
- [10] Intel Corporation, IXP1200 Network Processor Datasheet (September 2000).
- [11] Intel Corporation, IXP12EB Intel IXP1200 Network Processor Ethernet Evaluation Kit Product Brief (2000).
- [12] Intelligent I/O (I₂O) Special Interest Group, Intelligent I/O (I₂O) Architecture Specification, Version 2.0 (March 1999).
- [13] S. Karlin, L. Peterson, VERA: An Extensible Router Architecture, in: Proceedings of the 4th International Conference on Open Architectures and Network Programming (OPENARCH), 2001, pp. 3–14.
- [14] R. Keller, S. Choi, D. Decasper, M. Dasen, G. Fankhauser, B. Plattner, An Active Router Architecture for Multicast Video Distribution, in: Proceedings of the IEEE INFOCOM 2000 Conference, 2000, pp. 1137–1146.
- [15] T. V. Lakshman, D. Stiliadis, High Speed Policy-based Packet Forwarding Using Efficient Multi-dimensional Range Matching, in: Proceedings of the ACM SIGCOMM '98 Conference, 1998, pp. 203–214.

- [16] R. Morris, E. Kohler, J. Jannotti, M. F. Kaashoek, The Click Modular Router, in: Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP), 1999, pp. 217–231.
- [17] D. Mosberger, L. L. Peterson, Making Paths Explicit in the Scout Operating System, in: Proceedings of the Second USENIX Symposium on Operating System Design and Implementation (OSDI), 1996, pp. 153–167.
- [18] PCI Special Interest Group, Hillsboro, Oregon, PCI Local Bus Specification, Revision 2.2 (December 1998).
- [19] L. Peterson, Y. Gottlieb, M. Hibler, P. Tullmann, J. Lepreau, S. Schwab, H. Dandelkar, A. Purtell, J. Hartman, An OS Interface for Active Routers, IEEE Journal on Selected Areas in Communications 19 (3) (2001) 473–487.
- [20] L. L. Peterson, S. C. Karlin, K. Li, OS Support for General-Purpose Routers, in: Proceedings of the 7th Workshop on Hot Topics in Operating Systems (HotOS–VII), 1999, pp. 38–43.
- [21] X. Qie, A. Bavier, L. Peterson, S. Karlin, Scheduling Computations on a Programmable Router, in: Proceedings of the ACM SIGMETRICS 2001 Conference, 2001, pp. 13–24.
- [22] RAMiX Incorporated, Ventura, California, PMC/CompactPCI Ethernet Controllers Product Family Data Sheet (1999).
- [23] T. Spalink, S. Karlin, L. Peterson, Y. Gottlieb, Building a Robust Network-Processor-Based Router, in: Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP), 2001, pp. 216–229.
- [24] V. Srinivasan, G. Varghese, S. Suri, M. Waldvogel, Fast and Scalable Level Four Switching, in: Proceedings of the ACM SIGCOMM '98 Conference, 1998, pp. 191–202.
- [25] Vitesse Semiconductor Corporation, Longmont, Colorado, IQ2000 Network Processor Product Brief (2000).
- [26] M. Waldvogel, G. Varghese, J. Turner, B. Plattner, Scalable High Speed IP Routing Lookups, in: Proceedings of the ACM SIGCOMM '97 Conference, 1997, pp. 25–36.
- [27] S. Walton, A. Hutton, J. Touch, High-Speed Data Paths in Host-Based Routers, IEEE Computer 31 (11) (1998) 46–52.
- [28] D. Wetherall, Active network vision and reality: lessons from a capsule-based system, in: Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP), 1999, pp. 64–79.