A 50-Gb/s IP Router

Craig Partridge, *Senior Member, IEEE*, Philip P. Carvey, *Member, IEEE*, Ed Burgess, Isidro Castineyra, Tom Clarke, Lise Graham, Michael Hathaway, Phil Herman, Allen King, Steve Kohalmi, Tracy Ma, John Mcallen, Trevor Mendez, Walter C. Milliken, *Member, IEEE*, Ronald Pettyjohn, *Member, IEEE*, John Rokosz, *Member, IEEE*, Joshua Seeger, Michael Sollins, Steve Storch, Benjamin Tober, Gregory D. Troxel, David Waitzman, and Scott Winterble

***Abstract*—Aggressive research on gigabit-per-second networks has led to dramatic improvements in network transmission speeds. One result of these improvements has been to put pressure on router technology to keep pace. This paper describes a router, nearly completed, which is more than fast enough to keep up with the latest transmission technologies. The router has a backplane speed of 50 Gb/s and can forward tens of millions of packets per second.**

***Index Terms*—Data communications, internetworking, packet switching, routing.**

I. Introduction

TRANSMISSION link bandwidths keep improving, at a seemingly inexorable rate, as the result of research in transmission technology [26]. Simultaneously, expanding network usage is creating an ever-increasing demand that can only be served by these higher bandwidth links. (In 1996 and 1997, Internet service providers generally reported that the number of customers was at least doubling annually and that per-customer bandwidth usage was also growing, in some cases by 15% per month.)

Unfortunately, transmission links alone do not make a network. To achieve an overall improvement in networking performance, other components such as host adapters, operating systems, switches, multiplexors, and routers also need to get faster. Routers have often been seen as one of the lagging technologies. The goal of the work described here is to show that routers can keep pace with the other technologies and are fully capable of driving the new generation of links (OC-48c at 2.4 Gb/s).

A multigigabit router (a router capable of moving data at several gigabits per second or faster) needs to achieve three goals. First, it needs to have enough internal bandwidth to move packets between its interfaces at multigigabit rates. Second, it needs enough packet processing power to forward several million packets per second (MPPS). A good rule of thumb, based on the Internet's average packet size of approximately 1000 b, is that for every gigabit per second of bandwidth, a router needs 1 MPPS of forwarding power.[1] Third, the router needs to conform to a set of protocol standards. For Internet protocol version 4 (IPv4), this set of standards is summarized in the Internet router requirements [3]. Our router achieves all three goals (but for one minor variance from the IPv4 router requirements, discussed below).

This paper presents our multigigabit router, called the MGR, which is nearly completed. This router achieves up to 32 MPPS forwarding rates with 50 Gb/s of full-duplex backplane capacity.[2] About a quarter of the backplane capacity is lost to overhead traffic, so the packet rate and effective bandwidth are balanced. Both rate and bandwidth are roughly two to ten times faster than the high-performance routers available today.

II. Overview of the Router Architecture

A router is a deceptively simple piece of equipment. At minimum, it is a collection of network interfaces, some sort of bus or connection fabric connecting those interfaces, and some software or logic that determines how to route packets among those interfaces. Within that simple description, however, lies a number of complexities. (As an illustration of the complexities, consider the fact that the Internet Engineering Task Force's *Requirements for IP Version 4 Routers* [3] is 175 pages long and cites over 100 related references and standards.) In this section we present an overview of the MGR design and point out its major and minor innovations. After this section, the rest of the paper discusses the details of each module.

Manuscript received February 20, 1997; revised July 22, 1997; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor G. Parulkar. This work was supported by the Defense Advanced Research Projects Agency (DARPA).

C. Partridge is with BBN Technologies, Cambridge, MA 02138 USA, and with Stanford University, Stanford, CA 94305 USA (e-mail: craig@bbn.com).

P. P. Carvey, T. Clarke, and A. King were with BBN Technologies, Cambridge, MA 02138 USA. They are now with Avici Systems, Inc., Chelmsford, MA 01824 USA (e-mail: phil@avici.com; tclarke@avici.com; allen@avici.com).

E. Burgess, I. Castineyra, L. Graham, M. Hathaway, P. Herman, S. Kohalmi, T. Ma, J. Mcallen, W. C. Milliken, J. Rokosz, J. Seeger, M. Sollins, S. Storch, B. Tober, G. D. Troxel, and S. Winterble are with BBN Technologies, Cambridge, MA 02138 USA (e-mail: skohalmi@bbn.com; milliken@bbn.com; jseeger@bbn.com; sstorch@bbn.com; tober@bbn.com).

T. Mendez was with BBN Technologies, Cambridge, MA 02138 USA. He is now with Cisco Systems, Cambridge, MA 02138 USA.

R. Pettyjohn was with BBN Technologies, Cambridge, MA 02138 USA. He is now with Argon Networks, Littleton, MA 01460 USA (e-mail: ronp@argon.com).

D. Waitzman was with BBN Technologies, Cambridge, MA 02138 USA. He is now with D. E. Shaw and Company, L.P., Cambridge, MA 02139 USA.

Publisher Item Identifier S 1063-6692(98)04174-0.

[1] See [25]. Some experts argue for more or less packet processing power. Those arguing for more power note that a TCP/IP datagram containing an ACK but no data is 320 b long. Link-layer headers typically increase this to approximately 400 b. So if a router were to handle only minimum-sized packets, a gigabit would represent 2.5 million packets. On the other side, network operators have noted a recent shift in the average packet size to nearly 2000 b. If this change is not a fluke, then a gigabit would represent only 0.5 million packets.

[2] Recently some companies have taken to summing switch bandwidth in and out of the switch; in that case this router is a 100-Gb/s router.

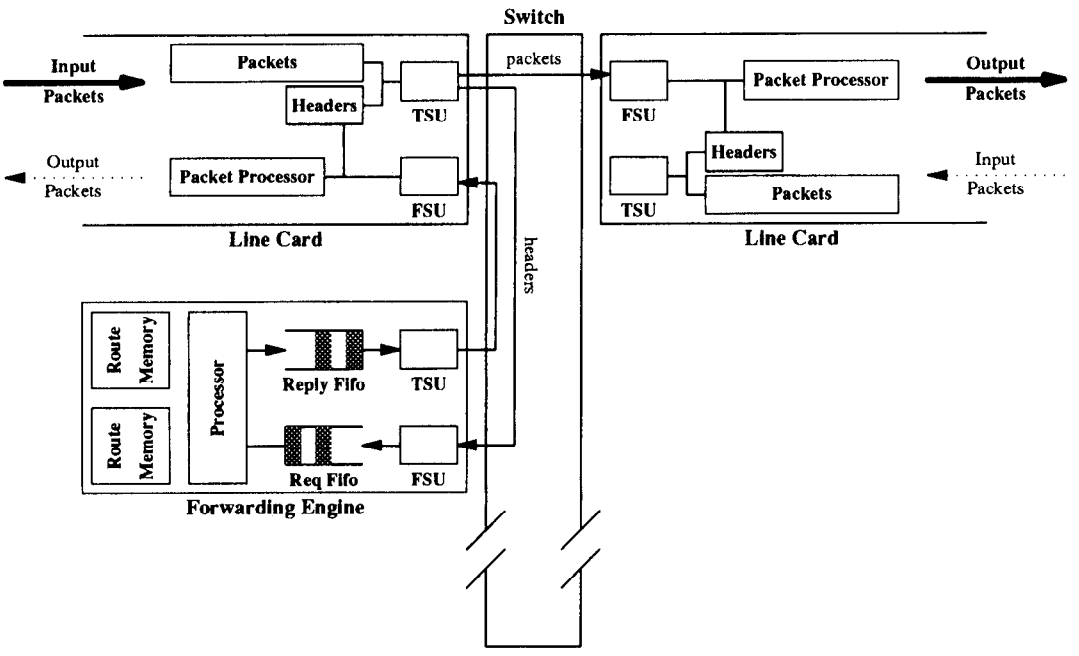

Fig. 1. MGR outline.

A. Design Summary

A simplified outline of the MGR design is shown in Fig. 1, which illustrates the data processing path for a stream of packets entering from the line card on the left and exiting from the line card on the right.

The MGR consists of multiple line cards (each supporting one or more network interfaces) and forwarding engine cards, all plugged into a high-speed switch. When a packet arrives at a line card, its header is removed and passed through the switch to a forwarding engine. (The remainder of the packet remains on the inbound line card). The forwarding engine reads the header to determine how to forward the packet and then updates the header and sends the updated header and its forwarding instructions back to the inbound line card. The inbound line card integrates the new header with the rest of the packet and sends the entire packet to the outbound line card for transmission.

Not shown in Fig. 1 but an important piece of the MGR is a control processor, called the network processor, that provides basic management functions such as link up/down management and generation of forwarding engine routing tables for the router.

B. Major Innovations

There are five novel elements of this design. This section briefly presents the innovations. More detailed discussions, when needed, can be found in the sections following.

First, each forwarding engine has a complete set of the routing tables. Historically, routers have kept a central master routing table and the satellite processors each keep only a modest cache of recently used routes. If a route was not in a satellite processor's cache, it would request the relevant route from the central table. At high speeds, the central table can easily become a bottleneck because the cost of retrieving a route from the central table is many times (as much as 1000 times) more expensive than actually processing the packet header. So the solution is to push the routing tables down into each forwarding engine. Since the forwarding engines only require a summary of the data in the route (in particular, next hop information), their copies of the routing table, called *forwarding tables*, can be very small (as little as 100 kB for about 50k routes [6]).

Second, the design uses a switched backplane. Until very recently, the standard router used a shared bus rather than a switched backplane. However, to go fast, one really needs the parallelism of a switch. Our particular switch was custom designed to meet the needs of an Internet protocol (IP) router.

Third, the design places forwarding engines on boards distinct from line cards. Historically, forwarding processors have been placed on the line cards. We chose to separate them for several reasons. One reason was expediency; we were not sure if we had enough board real estate to fit both forwarding engine functionality and line card functions on the target card size. Another set of reasons involves flexibility. There are well-known industry cases of router designers crippling their routers by putting too weak a processor on the line card, and effectively throttling the line card's interfaces to the processor's speed. Rather than risk this mistake, we built the fastest forwarding engine we could and allowed as many (or few) interfaces as is appropriate to share the use of the forwarding engine. This decision had the additional benefit of making support for virtual private networks very simple—we can dedicate a forwarding engine to each virtual network and ensure that packets never cross (and risk confusion) in the forwarding path.

Placing forwarding engines on separate cards led to a fourth innovation. Because the forwarding engines are separate from the line cards, they may receive packets from line cards that

use different link layers. At the same time, correct IP forwarding requires some information from the link-layer packet header (largely used for consistency checking). However, for fast forwarding, one would prefer that the forwarding engines not have to have code to recognize, parse, and load a diversity of different link-layer headers (each of which may have a different length). Our solution was to require all line cards to support the ability to translate their local link-layer headers to and from an abstract link-layer header format, which contained the information required for IP forwarding.

The fifth innovation was to include quality of service (QoS) processing in the router. We wanted to demonstrate that it was possible to build a cutting edge router that included line-speed QoS. We chose to split the QoS function. The forwarding engine simply classifies packets, by assigning a packet to a flow, based on the information in the packet header. The actual scheduling of the packet is done by the outbound line card, in a specialized processor called the QoS processor.

III. The Forwarding Engines

The forwarding engines are responsible for deciding where to forward each packet. When a line card receives a new packet, it sends the packet header to a forwarding engine. The forwarding engine then determines how the packet should be routed.

The development of our forwarding engine design was influenced by the Bell Laboratories router [2], which, although it has a different architecture, had to solve similar problems.

A. A Brief Description of the Alpha 21164 Processor

At the heart of each forwarding engine is a 415-MHz Digital Equipment Corporation Alpha 21164 processor. Since much of the forwarding engine board is built around the Alpha, this section summarizes key features of the Alpha. The focus in this section is on features that impact how the Alpha functions in the forwarding engine. A more detailed description of the 21164 and the Alpha architecture in general can be found in [1] and [31].

The Alpha 21164 is a 64-b 32-register super-scalar reduced instruction set computer (RISC) processor. There are two integer logic units, called E0 and E1, and two floating point units, called FA and FM. The four logic units are distinct. While most integer instructions (including loads) can be done in either E0 or E1, a few important operations, most notably byte extractions, shift operations, and stores, can only be done in E0. Floating point operations are more restricted, with all but one instruction limited to either FA or FM. In each cycle the Alpha attempts to schedule one instruction to each logic unit. For integer register-to-register operations, results are almost always available in the next instruction cycle. Floating results typically take several cycles. The Alpha processes instructions in groups of four instructions (hereafter called quads). All four instructions in a quad must successfully issue before any instructions in the next quad are issued. In practice this means that the programmer's goal is to place either two pairs of integer instructions that can issue concurrently, or a pair of integer instructions plus a pair of floating point instructions, in each quad.

The 21164 has three internal caches, plus support for an external cache. The instruction and data caches (Icache and Dcache) are the first-level caches and are 8 kB each in size. The size of the Icache is important because we want to run the processor at the maximum instruction rate and require that all code fits into the Icache. Since Alpha instructions are 32-b long, this means that the Icache can store 2048 instructions, more than enough to do key routing functions. If there are no errors in branch prediction, there will be no bubbles (interruptions in processing) when using instructions from the Icache. Our software effectively ignores the Dcache and always assumes that the first load of a 32-B cache line misses.

There is a 96-kB on-chip secondary cache (Scache) which caches both code and data. Loads from the Scache take a minimum of eight cycles to complete, depending on the state of the memory management hardware in the processor. We use the Scache as a cache of recent routes. Since each route entry takes 64 b, we have a maximum cache size of approximately 12 000 routes. Studies of locality in packet streams at routers suggest that a cache of this size should yield a hit rate well in excess of 95% [11], [13], [15]. Our own tests with a traffic trace from FIX West (a major interexchange point in the Internet) suggest a 12 000-entry cache will have a hit rate in excess of 95%.

The tertiary cache (Bcache) is an external memory of several megabytes managed by the processor. Loads from the Bcache can take a long time. While the Bcache uses 21-ns memory, the total time to load a 32-B cache line is 44 ns. There is also a system bus, but it is far too slow for this application and shares a single 128-b data path with the Bcache, so we designed the forwarding engine's memory system to bypass the system bus interface.

A complete forwarding table is kept in the Bcache. In our design the Bcache is 16 MB, divided into two 8-MB banks. At any time, one bank is acting as the Bcache and the other bank is being updated by the network processor via a personal computer interface (PCI) bus. When the forwarding table is updated, the network processor instructs the Alpha to change memory banks and invalidate its internal caches.

The divided Bcache highlights that we are taking an unusual approach—using a generic processor as an embedded processor. Readers may wonder why we did not choose an embedded processor. The reason is that, even with the inconvenience of the Bcache, the Alpha is a very good match for this task. As the section on software illustrates below, forwarding an IP datagram is a small process of reading a header, processing the header, looking up a route, and writing out the header plus routing information. The Alpha has four properties that make it a good fit: 1) very high clock speed, so forwarding code is executed quickly; 2) a large instruction cache, so the instructions can be done at peak rate; 3) a very large on-chip cache (the Scache), so that the routing lookup will probably hit in the on-chip route cache (avoiding accesses to slow external memory); and 4) sufficient control on read and write sequencing and buffer management to ensure that we could manage how data flowed through the chip.

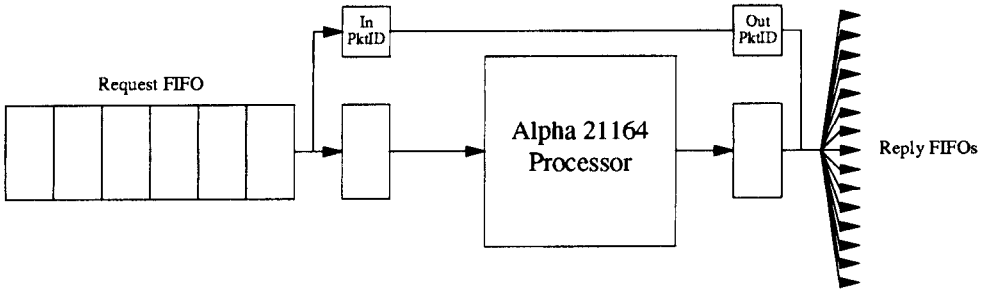

Fig. 2. Basic architecture of forwarding engine.

B. Forwarding Engine Hardware Operation

Once headers reach the forwarding engine, they are placed in a request first-in first-out (FIFO) queue for processing by the Alpha. The Alpha is running a loop which simply reads from the front of the FIFO, examines the header to determine how to route the packet, and then makes one or more writes to inform the inbound and outbound line cards how to handle the packet.

Conceptually, this process is illustrated in Fig. 2. A packet header has reached the front of the request FIFO. The header includes the first 24 or 56 B of the packet plus an 8-B generic link-layer header and a packet identifier which identifies both the packet and the interface it is buffered on. The Alpha software is expected to read at least the first 32 B of the packet header. When the packet is read, the packet identifier is copied into a holding buffer. When the Alpha writes out the updated header, the packet identifier is taken from the holding buffer and combined with the data from the Alpha to determine where the updated header and packet are sent.

The Alpha software is free to read and write more than 32 B of the packet header (if present) and can, if it chooses, read and write the packet identifier registers as well. The software must read and write this information if it is reading and writing packet headers in anything but FIFO order. The motivation for the holding buffer is to minimize the amount of data that must go through the Alpha. By allowing software to avoid reading the packet ID, we minimize the load on the Alpha's memory interface.

When the software writes out the updated header, it indicates which outbound interface to send the packet to by writing to one of 241 addresses. (240 addresses for each of 16 possible interfaces on 15 line cards plus one address indicating that the packet should be discarded.) The hardware actually implements these FIFO's as a single buffer and grabs the dispatching information from a portion of the FIFO address.

In addition to the dispatching information in the address lines, the updated header contains some key routing information. In particular it contains the outbound link-layer address and a flow identifier, which is used by the outbound line card to schedule when the packet is actually transmitted.

A side comment about the link-layer address is in order. Many networks have dynamic schemes for mapping IP addresses to link-layer addresses. A good example is the address resolution protocol (ARP), used for Ethernet [28]. If a router gets a datagram to an IP address whose Ethernet address it does not know, it is supposed to send an ARP message and hold the datagram until it gets an ARP reply with the necessary Ethernet address. In the pipelined MGR architecture that approach does not work—we have no convenient place in the forwarding engine to store datagrams awaiting an ARP reply. Rather, we follow a two-part strategy. First, at a low frequency, the router ARP's for all possible addresses on each interface to collect link-layer addresses for the forwarding tables. Second, datagrams for which the destination link-layer address is unknown are passed to the network processor, which does the ARP and, once it gets the ARP reply, forwards the datagram and incorporates the link-layer address into future forwarding tables.

C. Forwarding Engine Software

The forwarding engine software is a few hundred lines of code, of which 85 instructions are executed in the common case. These instructions execute in no less than 42 cycles,[3] which translates to a peak forwarding speed of 9.8 MPPS per forwarding engines. This section sketches the structure of the code and then discusses some of the properties of this code.

The fast path through the code can be roughly divided up into three stages, each of which is about 20–30 instructions (10–15 cycles) long. The first stage: 1) does basic error checking to confirm that the header is indeed from an IPv4 datagram; 2) confirms that the packet and header lengths are reasonable; 3) confirms that the IPv4 header has no options; 4) computes the hash offset into the route cache and loads the route; and 5) starts loading the next header. These five activities are done in parallel in intertwined instructions.

During the second stage, it checks to see if the cached route matches the destination of the datagram. If not, the code jumps to an extended lookup which examines the routing table in the Bcache. Then the code checks the IP time-to-live (TTL) field and computes the updated TTL and IP checksum, and determines if the datagram is for the router itself. The TTL and checksum are the only header fields that normally change and they must not be changed if the datagram is destined for the router.

In the third stage the updated TTL and checksum are put in the IP header. The necessary routing information is extracted from the forwarding table entry and the updated IP header is written out along with link-layer information from the forwarding table. The routing information includes the

[3]The instructions can take somewhat longer, depending on the pattern of packets received and the resulting branch predictions.

flow classifier. Currently we simply associate classifiers with destination prefixes, but one nice feature of the route-lookup algorithm that we use [34] is that it scales as the log of the key size, so incorporating additional information like the IP type-of-service field into the lookup key typically has only a modest effect on performance.

This code performs all the steps required by the Internet router requirements [3] except one—it does not check the IP header checksum, but simply updates it. The update algorithm is safe [4], [18], [29]. If the checksum is bad, it will remain bad after the update. The reason for not checking the header checksum is that, in the best code we have been able to write, computing it would require 17 instructions and, due to consumer–producer relationships, those instructions would have to be spread over a minimum of 14 cycles. Assuming we can successfully interleave the 17 instructions among other instructions in the path, at minimum they still increase the time to perform the forwarding code by nine cycles or about 21%. This is a large penalty to pay to check for a rare error that can be caught end-to-end. Indeed, for this reason, IPv6 does not include a header checksum [8].

Certain datagrams are not handled in the fast path code. These datagrams can be divided into five categories.

1) *Headers whose destination misses in the route cache.* This is the most common case. In this case the processor searches the forwarding table in the Bcache for the correct route, sends the datagram to the interface indicated in the routing entry, and generates a version of the route for the route cache. The routing table uses the binary hash scheme developed by Waldvogel, Varghese, Turner, and Plattner [34]. (We also hope to experiment with the algorithm described in [6] developed at Lulea University.) Since the forwarding table contains prefix routes and the route cache is a cache of routes for particular destinations, the processor has to convert the forwarding table entry into an appropriate destination-specific cache entry.
2) *Headers with errors.* Generally, the forwarding engine will instruct the inbound line card to discard the errored datagram. In some cases the forwarding engine will generate an internet control message protocol (ICMP) message. Templates of some common ICMP messages such as the TimeExceeded message are kept in the Alpha's Bcache and these can be combined with the IP header to generate a valid ICMP message.
3) *Headers with IP options.* Most headers with options are sent to the network processor for handling, simply because option parsing is slow and expensive. However, should an IP option become widely used, the forwarding code could be modified to handle the option in a special piece of code outside the fast path.
4) *Datagrams that must be fragmented.* Rather than requiring line cards to support fragmentation logic, we do fragmentation on the network processor. Now that IP maximum transmission unit (MTU) discovery [22] is prevalent, fragmentation should be rare.
5) *Multicast datagrams.* Multicast datagrams require special routing, since the routing of the datagram is dependent on the source address and the inbound link as well as the multicast destination. Furthermore, the processor may have to write out multiple copies of the header to dispatch copies of the datagram to different line cards. All of this work is done in separate multicasting code in the processor. Multicast routes are stored in a separate multicast forwarding table. The code checks to see if the destination is a multicast destination and, if so, looks for a multicast route. If this fails, it retrieves or builds a route from its forwarding table.

TABLE I
DISTRIBUTION OF INSTRUCTIONS IN FAST PATH

Instructions	Count	Percentage	E0/E1/FP
`and, bic, bis, ornot, xor`	24	28	E0/E1
`ext*, ins*, sll, srl, zap`	23	27	E0
`add*, sub*, s*add`	8	9	E0/E1
`branches`	8	9	E1
`ld*`	6	7	E0/E1
`addt, cmpt*, fcmov*`	6	7	FA
`st*`	4	5	E0
`fnop`	4	5	FM
`wmb`	1	1	E0
`nop`	1	1	E0/E1

Observe that we have applied a broad logic to handling headers. Types of datagrams that appear frequently (fast path, destinations that miss in the route cache, common errors, multicast datagrams) are handled in the Alpha. Those that are rare (IP with options, MTU size issues, uncommon errors) are pushed off to the network processor rather than using valuable Icache instruction space to handle them. If the balance of traffic changes (say to more datagrams with options), the balance of code between the forwarding engine and network processor can be adapted.

We have the flexibility to rebalance code because the forwarding engine's peak forwarding rate of 9.8 MPPS is faster than the switch's maximum rate of 6.48 million headers per second.

Before concluding the discussion of the forwarding engine code, we would like to briefly discuss the actual instructions used, for two reasons. First, while there has been occasional speculation about what mix of instructions is appropriate for handing IP datagrams, so far as we know, no one has ever published a distribution of instructions for a particular processor. Second, there has been occasional speculation about how well RISC processors would handle IP datagrams.

Table I shows the instruction distribution for the fast path instructions. Instructions are grouped according to type (using the type classifications in the 21164 manual) and listed with the count, percentage of total instructions, and whether instructions are done in integer units E0 and E1 or both, or the floating point units FA and FM.

Probably the most interesting observation from the table is that 27% of the instructions are bit, byte, or word manipulation instructions like `zap`. The frequency of these instructions largely reflects the fact they are used to extract various 8-, 16-, and 32-b fields from 64-b registers holding the IP and link-layer headers (the `ext` commands) and to zero byte-wide fields

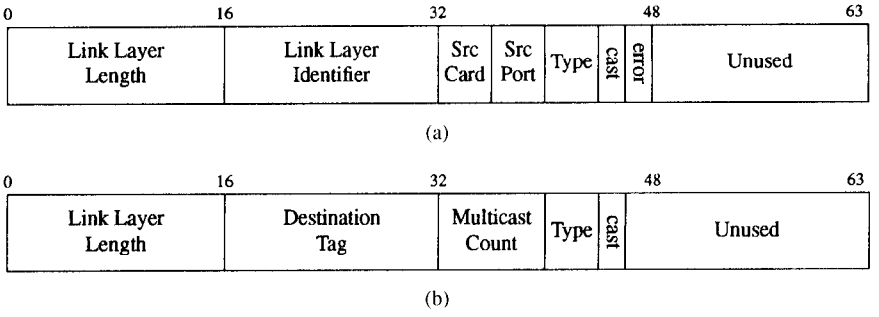

Fig. 3. Abstract link layer headers. (a) Inbound. (b) Outbound.

(zap) in preparation for inserting updated information into those registers. Oddly enough, these manipulation instructions are some of the few instructions that can only be done in the logic unit E0, which means that some care must be taken in the code to avoid instruction contention for E0. (This is another reason to avoid checking the header checksum. Most of the instructions involved in computing the header checksum are instructions to extract 16-b fields, so checking the header checksum would have further increased the contention for E0.)

One might suspect that testing bits in the header is a large part of the cost of forwarding, given that bit operations represent 28% of the code. In truth, only two instructions (both xors) represent bit tests on the headers. bis is used to assemble header fields, and most of the remaining instructions are used to update the header checksum and compute a hash into the route cache.

The floating point operations, while they account for 12% of the instructions, actually have no impact on performance. They are used to count simple network management protocol (SNMP) management information base (MIB) values and are interleaved with integer instructions so they can execute in one cycle. The presence of four fnop instructions reflects the need to pad a group of two integer instructions and one floating point instruction so the Alpha can process the four instructions in one cycle.

Finally, observe that there is a minimum of instructions to load and store data. There are four loads (ld*) to load the header, one load to load the cached forwarding table entry, and one load to access the MTU table. Then there are four stores (stq) to store the updated header and an instruction to create a write memory barrier (wmb) and ensure that writes are sequenced.

1) Issues in Forwarding Engine Design: To close the presentation of the forwarding engine, we address two frequently asked questions about forwarding engine design in general and the MGR's forwarding engine in particular.

a) Why not use an ASIC?: The MGR forwarding engine uses a processor to make forwarding decisions. Many people often observe that the IPv4 specification is very stable and ask if it would be more cost effective to implement the forward engine in an application specific integrated circuit (ASIC).

The answer to this issue depends on where the router might be deployed. In a corporate local-area network (LAN) it turns out that IPv4 is indeed a fairly static protocol and an ASIC-based forwarding engine is appropriate. But in an internet service provider's (ISP's) backbone, the environment that the MGR was designed for, IPv4 is constantly evolving in subtle ways that require programmability.

b) How effective is a route cache?: The MGR uses a cache of recently seen destinations. As the Internet's backbones become increasingly heavily used and carry traffic of a greater number of parallel conversations, is such a cache likely to continue to be useful?

In the MGR, a cache hit in the processor's on-chip cache is at least a factor of five less expensive than a full route lookup in off-chip memory, so a cache is valuable provided it achieves at least a modest hit rate. Even with an increasing number of conversations, it appears that packet trains [15] will continue to ensure that there is a strong chance that two datagrams arriving close together will be headed for the same destination. A modest hit rate seems assured and, thus, we believe that using a cache makes sense.

Nonetheless, we believe that the days of caches are numbered because of the development of new lookup algorithms—in particular the binary hash scheme [34]. The binary hash scheme takes a fixed number of memory accesses determined by the address length, not by the number of keys. As a result, it is fairly easy to inexpensively pipeline route lookups using the binary hash algorithm. The pipeline could be placed alongside the inbound FIFO such that a header arrived at the processor with a pointer to its route. In such an architecture no cache would be needed.

D. Abstract Link Layer Header

As noted earlier, one innovation for keeping the forwarding engine and its code simple is the abstract link-layer header, which summarizes link-layer information for the forwarding engine and line cards. Fig. 3 shows the abstract link-layer header formats for inbound (line card to forwarding engine) and outbound (forwarding engine to line card) headers. The different formats reflect the different needs of reception and transmission.

The inbound abstract header contains information that the forwarding engine code needs to confirm the validity of the IP header and the route chosen for that header. For instance, the link-layer length is checked for consistency against the length in the IP header. The link-layer identifier, source card, and source port are used to determine if an ICMP redirect must be sent. (ICMP redirects are sent when a datagram goes in and out of the same interface. The link-layer identifier is used

in cases where multiple virtual interfaces may coexist on one physical interface port, in which case a datagram may go in and out of the same physical interface, but different virtual interfaces, without causing a redirect.)

The outbound abstract header contains directions to the line cards about how datagram transmission is to be handled. The important new fields are the multicast count, which indicates how many copies of the packet the inbound line card needs to make, and the destination tag, which tells the outbound line card what destination address to put on the packet, what line to send the packet out, and what flow to assign the packet to. For multicast packets, the destination tag tells the outbound line card what set of interfaces to send the packet out.[4]

IV. The Switched Bus

Most routers today use a conventional shared bus. The MGR instead uses a 15-port switch to move data between function cards. The switch is a point-to-point switch (i.e., it effectively looks like a crossbar, connecting one source with one destination).

The major limitation of a point-to-point switch is that it does not support the one-to-many transfers required for multicasting. We took a simple solution to this problem. Multicast packets are copied multiple times, once to each outbound line card. The usual concern about making multiple copies is that it reduces effective switch throughput. For instance, if every packet, on average, is sent to two boards, the effective switch bandwidth will be reduced by half. However, even without multicast support, this scheme is substantially better than a shared bus.[5]

The MGR switch is a variant of a now fairly common type of switch. It is an input-queued switch in which each input keeps a separate FIFO and bids separately for each output. Keeping track of traffic for each output separately means that the switch does not suffer head-of-line blocking [20], and it has been shown by simulation [30] and more recently proven [21] that such a switch can achieve 100% throughput. The key design choice in this style of switch is its allocation algorithm—how one arbitrates among the various bids. The MGR arbitration seeks to maximize throughput at the expense of predictable latency. (This tradeoff is the reverse of that made in many asynchronous transfer mode (ATM) switches and is why we built our own switch, optimized for IP traffic.)

A. Switch Details

The switch has two pin interfaces to each function card. The data interface consists of 75 input data pins and 75 output data pins, clocked at 51.84 MHz. The allocation interface consists of two request pins, two inhibit pins, one input status pin, and one output status pin, all clocked at 25.92 MHz. Because of the large number of pins and packaging constraints, the switch is implemented as five identical data path cards plus one allocator card.

A single switch transfer cycle, called an *epoch*, takes 16 ticks of the data clock (eight ticks of the allocation clock). During an epoch, up to 15 simultaneous data transfers take place. Each transfer consists of 1024 bits of data plus 176 auxiliary bits of parity, control, and ancillary bits. The aggregate data bandwidth is 49.77 Gb/s (58.32 Gb/s including the auxiliary bits). The per-card data bandwidth is 3.32 Gb/s (full duplex, not including auxiliary bits).

The 1024 bits of data are divided up into two transfer units, each 64 B long. The motivation for sending two distinct units in one epoch was that the desirable transfer unit was 64 B (enough for a packet header plus some overhead information), but developing an field programmable gate array (FPGA)-based allocator that could choose a connection pattern in eight switch cycles was difficult. We chose, instead, to develop an allocator that decides in 16 clock cycles and transfers two units in one cycle.

Both 64-B units are delivered to the same destination card. Function cards are not required to fill both 64-B units; the second one can be empty. When a function card has a 64-B unit to transfer, it is expected to wait several epochs to see if another 64-B unit becomes available to fill the transfer. If not, the card eventually sends just one unit. Observe that when the card is heavily loaded, it is very likely that a second 64-B unit will become available, so the algorithm has the desired property that as load increases, the switch becomes more efficient in its transfers.

Scheduling of the switch is pipelined. It takes a minimum of four epochs to schedule and complete a transfer. In the first epoch the source card signals that it has data to send to the destination card. In the second epoch the switch allocator decides to schedule the transfer for the fourth epoch. In the third epoch the source and destination line cards are notified that the transfer will take place and the data path cards are told to configure themselves for the transfer (and for all other transfers in fourth epoch). In the fourth epoch the transfer takes place.

The messages in each epoch are scheduled via the allocation interface. A source requests to send to a destination card by setting a bit in a 15-b mask formed by the two request pins over the eight clock cycles in an epoch.[6] The allocator tells the source and destination cards who they will be connected to with a 4-b number (0–14) formed from the first four bits clocked on the input and output status pins in each epoch.

The switch implements flow control. Destination cards can, on a per-epoch basis, disable the ability of specific source cards to send to them. Destination cards signal their willingness to receive data from a source by setting a bit in the 15-b mask formed by the two inhibit pins. Destinations are allowed to inhibit transfers from a source to protect against certain pathological cases where packets from a single source could

[4] For some types of interfaces, such as ATM, this may require the outbound line card to generate different link-layer headers for each line. For others, such as Ethernet, all of the interfaces can share the same link-layer header.

[5] The basic difference is that a multicast transfer on a shared bus would monopolize the bus, even if only two outbound line cards were getting the multicast. On the switch, those line cards not involved in the multicast can concurrently make transfers among themselves while the multicast transactions are going on. The fact that our switch copies multiple times makes it less effective than some other switch designs (e.g., [23]), but still much better than a bus.

[6] Supporting concurrent requests for multiple line cards plus randomly shuffling the bids (see Section IV-B) ensures that even though the MGR uses an input-queued switch, it does not suffer head-of-line blocking.

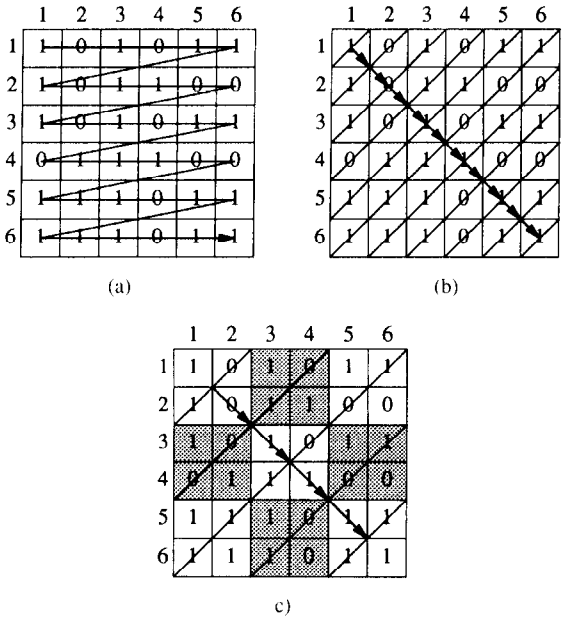

Fig. 4. Simple and wavefront allocators. (a) Simple. (b) Wavefront. (c) Group wavefront.

consume the entire destination card's buffer space, preventing other sources from transmitting data through the destination card.[7]

B. The Allocator

The allocator is the heart of the high-speed switch in the MGR. It takes all of the (pipelined) connection requests from the function cards and the inhibiting requests from the destination cards and computes a configuration for each epoch. It then informs the source and destination cards of the configuration for each epoch. The hard problem with the allocator is finding a way to examine 225 possible pairings and choose a good connection pattern from the possible 1.3 trillion (15 factorial) connection patterns within one epoch time (about 300 ns).

A straightforward allocator algorithm is shown on the left side of Fig. 4. The requests for connectivity are arranged in a 5 × 5 matrix of bits (where a 1 in position x, y means there is a request from source x to connect to destination y). The allocator simply scans the matrix, from left to right and top to bottom, looking for connection requests. When it finds a connection request that does not interfere with previous connection requests already granted, it adds that connection to its list of connections for the epoch being scheduled. This straightforward algorithm has two problems: 1) it is clearly unfair—there is a preference for low-numbered sources and 2) for a 15 × 15 matrix, it requires serially evaluating 225 positions per epoch—that is one evaluation every 1.4 ns, too fast for an FPGA.

[7]The most likely version of this scenario is a burst of packets that come in a high-speed interface and go out a low-speed interface. If there are enough packets, and the outbound line card's scheduler does not discard packets until they have aged for a while, the packets could sit in the outbound line card's buffers for a long time.

There is an elegant solution to the fairness problem—randomly shuffle the sources and destinations. The allocator has two 15-entry shuffle arrays. The source array is a permutation of the values 1–15, and value of position s of the array indicates what row in the allocation matrix should be filled with the bids from source s. The destination array is a similar permutation. Another way to think of this procedure is if one takes the original matrix $\boldsymbol{M}$, one generates a shuffled matrix $\boldsymbol{SM}$ according to the following rule:

$$\boldsymbol{SM}[x, y] = \boldsymbol{M}[\text{rowshuffle}[x], \text{colshuffle}[y]]$$

and uses $\boldsymbol{SM}$ to do the allocation.[8]

The timing problem is more difficult. The trick is to observe that parallel evaluation of multiple locations is possible. Consider Fig. 4 again. Suppose we have just started a cycle and examined position (1,1). On the next cycle, we can examine both positions (2,1) and (1,2) because the two possible connections are not in conflict with each other—they can only conflict with a decision to connect source 1 to itself. Similarly, on the next cycle, we can examine (3,1), (2,2), and (1,3) because none of them are in conflict with each other. Their only potential conflicts are with decisions already made about (1,1), (2,1), and (1,2). This technique is called wavefront allocation [16], [32] and is illustrated in the middle of Fig. 4. However, for a 15 × 15 matrix, wavefront allocation requires 29 steps, which is still too many. But one can refine the process by grouping positions in the matrix and doing wavefront allocations across the groups. The right side of Fig. 4 shows such a scheme using 2 × 2 groups, which halves the number of cycles. Processing larger groups reduces the time further. The MGR allocator uses 3 × 5 groups.

One feature that we added to the allocator was support for multiple priorities. In particular we wanted to give transfers from forwarding engines higher priority than data from line cards to avoid *header contention* on line cards. Header contention occurs when packets queue up in the input line card waiting for their updated header and routing instructions from the forwarding engines. In a heavily loaded switch with fair allocation one can show that header contention will occur because the forwarding engines' requests must compete equally with data transfers from other function cards. The solution to this problem is to give the forwarding engines priority as sources by skewing the random shuffling of the sources.

[8]Jon C. R. Bennett has pointed out that this allocator does not always evenly distribute bandwidth across all sources. In particular, if bids for destinations are very unevenly distributed, allocation will follow the unevenness of the bids. For instance, consider the bid pattern in the figure below:

	1	2	3	4	5	6
1	1	1	1	1	1	1
2	0	0	0	0	0	1
3	0	0	0	0	0	1
4	0	0	0	0	0	1
5	0	0	0	0	0	1
6	0	0	0	0	0	1

Line card 1 has only a 1-in-36 chance of transferring to line card 6, while the other line cards all have a 7-in-36 chance of transferring to line card 6.

V. Line Card Design

A line card in the MGR can have up to 16 interfaces on it (all of the same type). However, the total bandwidth of all interfaces on a single card should not exceed approximately 2.5 Gb/s. The difference between the 2.5- and 3.3-Gb/s switch rate is to provide enough switch bandwidth to transfer packet headers to and from the forwarding engines. The 2.5-Gb/s rate is sufficient to support one OC-48c (2.4-Gb/s) SONET interface, four OC-12c (622-Mb/s) SONET interfaces, or three HIPPI (800-Mb/s) interfaces on one card. It is also more than enough to support 16 100-Mb/s Ethernet or FDDI interfaces. We are currently building a line card with two OC-12c (622-Mb/s) SONET/ATM interfaces and expect to build additional line cards.

With the exception of handling header updates, the inbound and outbound packet processes are completely disjoint. They even have distinct memory pools. For simplicity, packets going between interfaces on the same card must be looped back through the switch.[9] Inbound packet processing is rather simple; outbound packet processing is much more complex.

A. Inbound Packet Processing

As a packet arrives at an inbound line card, it is assigned a packet identifier and its data is broken up (as the data arrives) into a chain of 64-B pages, in preparation for transfer through the switch. The first page, which includes the summary of the packet's link-layer header, is then sent to the forwarding engine to get routing information. When the updated page is returned, it replaces the old first page and its routing information is used to queue the entire packet for transfer to the appropriate destination card.

This simple process is complicated in two situations. First, when packets are multicast, copies may have to be sent to more than one destination card. In this case the forwarding engine will send back multiple updated first pages for a single packet. As a result, the inbound packet buffer management must keep reference counts and be able to queue pages concurrently for multiple destinations.

The second complicating situation occurs when the interfaces use ATM. First, ATM cells have a 48-B payload, which is less than the 64-B page size, so the ATM segmentation and reassembly (SAR) process that handles incoming cells includes a staging area where cells are converted to pages. Second, there are certain situations where it may be desirable to permit an operations and maintenance cell to pass directly through the router from one ATM interface to another ATM interface. To support this, the ATM interfaces are permitted to put a 53-B full ATM cell in a page and ship the cell directly to an outbound interface.

B. Outbound Packet Processing

When an outbound line card receives pages of a packet from the switch, it assembles those pages into a linked list representing the packet and creates a packet record pointing to the linked list. If the packet is being multicast to multiple interfaces on the card, it will make multiple packet records, one for each interface getting the packet.

After the packet is assembled, its record is passed to a line card's QoS processor. If the packet is being multicast, one record is passed to each of the interfaces on which the multicast is being sent. The purpose of the QoS processor is to implement flow control and integrated services in the router. Recall that the forwarding engine classifies packets by directing them to particular flows. It is the job of the QoS processor to actually schedule each flow's packets.

The QoS processor is a very large instruction word (VLIW) programmable state machine implemented in a 52-MHz ORCA 2C04A FPGA. (ORCA FPGA's can be programmed to use some of their real-estate as memory, making them very useful for implementing a special purpose processor). The QoS processor is event-driven and there are four possible events. The first event is the arrival of a packet. The processor examines the packet's record (for memory bandwidth reasons, it does not have access the packet data itself) and based on the packet's length, destination, and the flow identifier provided by the forwarding engine, places the packet in the appropriate position in a queue. In certain situations, such as congestion, the processor may choose to discard the packet rather than to schedule it. The second event occurs when the transmission interface is ready for another packet to send. The transmission interface sends an event to the scheduler, which in turn delivers to the transmission interface the next few packets to transmit. (In an ATM interface each virtual channel (VC) separately signals its need for more packets.) The third event occurs when the network processor informs the scheduler of changes in the allocation of bandwidth among users. The fourth event is a timer event, needed for certain scheduling algorithms. The processor can also initiate messages to the network processor. Some packet handling algorithms such as random early detection (RED) [12] require the scheduler to notify the network processor when a packet is discarded.

Any link-layer-based scheduling (such as that required by ATM) is done separately by a link-layer scheduler after the packet scheduler has passed the packet on for transmission. For example, our OC-12c ATM line cards support an ATM scheduler that can schedule up to 8000 ATM VC's per interface, each with its own QoS parameters for constant bit rate (CBR), variable bit rate (VBR), or unspecified bit rate (UBR) service.

VI. The Network Processor

The network processor is a commercial PC motherboard with a PCI interface. This motherboard uses a 21 064 Alpha processor clocked at 233 MHz. The Alpha processor was chosen for ease of compatibility with the forwarding engines. The motherboard is attached to a PCI bridge, which gives it access to all function cards and also to a set of registers on the switch allocator board.

The processor runs the 1.1 NetBSD release of UNIX. NetBSD is a freely available version of UNIX based on the

[9] If one allows looping in the card, there will be some place on the card that must run twice as fast as the switch (because it may receive data both from the switch and the inbound side of the card in the same cycle). That is painful to implement at high speed.

4.4 Berkeley Software Distribution (BSD) release. The choice of operating system was dictated by two requirements. First, we needed access to the kernel source code to customize the IP code and to provide specialized PCI drivers for the line and forwarding engine cards. Second, we needed a BSD UNIX platform because we wanted to speed the development process by porting existing free software such as gated [10] to the MGR whenever possible and almost all this software is implemented for BSD UNIX.

VII. Managing Routing and Forwarding Tables

Routing information in the MGR is managed jointly by the network processor and the forwarding engines.

All routing protocols are implemented on the network processor, which is responsible for keeping complete routing information. From its routing information, the network processor builds a forwarding table for each forwarding engine. These forwarding tables may be all the same, or there may be different forwarding tables for different forwarding engines.

One advantage of having the network processor build the tables is that while the network processor needs complete routing information such as hop counts and whom each route was learned from, the tables for the forwarding engines need simply indicate the next hop. As a result, the forwarding tables for the forwarding engines are much smaller than the routing table maintained by the network processor.

Periodically, the network processor downloads the new forwarding tables into the forwarding engines. As noted earlier, to avoid slowing down the forwarding engines during this process, the forwarding table memory on the forwarding engines is split into two banks. When the network processor finishes downloading a new forwarding table, it sends a message to the forwarding engine to switch memory banks. As part of this process, the Alpha must invalidate its on-chip routing cache, which causes some performance penalty, but a far smaller penalty than having to continuously synchronize routing table updates with the network processor.

One of the advantages of decoupling the processing of routing updates from the actual updating of forwarding tables is that bad behavior by routing protocols, such as route flaps, does not have to affect router throughput. The network processor can delay updating the forwarding tables on the forwarding engines until the flapping has subsided.

VIII. Router Status

When this paper went to press, all of the router hardware had been fabricated and tested except for the interface cards, and the majority of the software was up and running. Test cards that contained memory and bidding logic were plugged into the switch to simulate interface cards when testing the system.

Estimating latencies through the router is difficult, due to a shortage of timing information inside the router, the random scheduling algorithm of the switch, and the absence of external interfaces. However, based on actual software performance measured in the forwarding engine, observations when debugging hardware, and estimates from simulation, we estimate that a 128-B datagram entering an otherwise idle router would experience a delay of between 7 and 8 μs. A 1-kB datagram would take 0.9 μs longer. These delays assume that the header is processed in the forwarding engine, not the network processor, and that the Alpha has handled at least a few datagrams previously (so that code is in the instruction cache and branch predictions are correctly made).

IX. Related Work and Conclusions

Many of the features of the MGR have been influenced by prior work. The Bell Laboratories router [2] similarly divided work between interfaces, which moved packets among themselves, and forwarding engines, which, based on the packet headers, directed how the packets should be moved. Tantawy and Zitterbart [33] have examined parallel IP header processing. So too, several people have looked at ways to adapt switches to support IP traffic [24], [27].

Beyond the innovations outlined in Section II-B, we believe that the MGR makes two important contributions. The first is the MGR's emphasis on examining every datagram header. While examining every header is widely agreed to be desirable for security and robustness, many had thought that the cost of IP forwarding was too great to be feasible at high speed. The MGR shows that examining every header is eminently feasible.

The MGR is also valuable because there had been considerable worry that router technology was failing and that we needed to get rid of routers. The MGR shows that router technology is not failing and routers can continue to serve as a key component in high-speed networks.

Acknowledgment

The authors would like to thank the many people who have contributed to or commented on the ideas in this paper, including J. Mogul and others at Digital Equipment Corporation's Western Research Lab, S. Pink, J. Touch, D. Ferguson, N. Chiappa, M. St. Johns, G. Minden, and members of the Internet End-To-End Research Group chaired by B. Braden. The anonymous ToN reviewers and the ToN Technical Editor, G. Parulkar, also provided very valuable comments which substantially improved this paper.

References

[1] "Alpha 21164 Microprocessor," Hardware Reference Manual, Digital Equipment Corporation, Burlington, MA, Apr. 1995.

[2] A. Asthana, C. Delph, H. V. Jagadish, and P. Krzyzanowski, "Toward a gigabit IP router," *J. High Speed Networks*, vol. 1, no. 4, pp. 281–288, 1992.

[3] F. Baker, "Requirements for IP version 4 routers; RFC-1812," *Internet Request For Comments*, vol. 1812, June 1995.

[4] B. Braden, D. Borman, and C. Partridge, "Computing the internet checksum; RFC 1071," *Internet Request for Comments*, vol. 1071, Sept. 1988.

[5] S. Bradner and A. Mankin, *IPng: Internet Protocol Next Generation*. New York: Addison-Wesley, 1995.

[6] A. Brodnik, S. Carlsson, M. Degermark, and S. Pink, "Small forwarding tables for fast routing lookups," in *Proc. ACM SIGCOMM'97*, Cannes, France, Sept. 1997, pp. 3–14.

[7] N. Chiappa, "Data packet switch using a primary processing unit to designate one of a plurality of data stream control circuits to selectively handle the header processing of incoming packets in one data packet stream," US Patent 5 249 292, Sept. 28, 1993.
[8] S. Deering and R. Hinden, "Internet protocol, version 6 (IPv6); RFC-1883," *Internet Request for Comments*, vol. 1883, Jan. 1996.
[9] A. Demers, S. Keshav, and S. Shenker, "Analysis and simulation of a fair queueing algorithm," in *Internetwork: Research and Experience*, vol. 1, no. 1. New York: Wiley, Sept. 1990, pp. 3–6.
[10] M. Fedor, "Gated: A multi-routing protocol daemon for UNIX," in *Proc. 1988 Summer USENIX Conf.*, San Francisco, CA, 1988, pp. 365–376.
[11] D. C. Feldmeier, "Improving gateway performance with a routing-table cache," in *Proc. IEEE INFOCOM'88*, New Orleans, LA, Mar. 1988, pp. 298–307.
[12] S. Floyd and V. Jacobson, "Random early detection gateways for congestion avoidance," *IEEE/ACM Trans. Networking*, vol. 1, pp. 397–413, Aug. 1993.
[13] S. A. Heimlich, "Traffic characterization of the NSFNET national backbone," in *Proc. Winter 1990 USENIX Conf.*, Washington, DC, Jan. 1990, pp. 207–227.
[14] R. Jain, "A comparison of hashing schemes for address lookup in computer networks," *IEEE Trans. Commun.*, vol. 40, pp. 1570–1573, Oct. 1992.
[15] R. Jain and S. Routhier, "Packet trains: Measurements and a new model for computer network traffic," *IEEE J. Select. Areas Commun.*, vol. 4, pp. 986–995, Sept. 1986.
[16] R. O. Lamaire and D. N. Serpanos, "Two dimensional round robin schedulers for packet switches with multiple input queues," *IEEE/ACM Trans. Networking*, vol. 2, pp. 471–482, Oct. 1994.
[17] H. R. Lewis and L. Denenberg, *Data Structures & Their Algorithms*. New York: Harper Collins, 1991.
[18] T. Mallory and A. Kullberg, "Incremental updating of the Internet checksum; RFC-1141," *Internet Requests for Comments*, vol. 1141, Jan. 1990.
[19] N. McKeown, M. Izzard, A. Mekkittikul, B. Ellersick, and M. Horowitz, "The tiny tera: A packet switch core," *IEEE Micro*, vol. 17, pp. 26–33, Jan. 1997.
[20] M. J. Karol, M. G. Hluchyj, and S. P. Morgan, "Input versus output queueing on a space-division packet switch," *IEEE Trans. Commun.*, vol. COM-35, pp. 1347–1356, Dec. 1987.
[21] N. McKeown, V. Anantharam, and J. Walrand, "Achieving 100% throughput in an input-queued switch," in *Proc. IEEE INFOCOM'96*, San Francisco, CA, Mar. 1996, pp. 296–302.
[22] J. C. Mogul and S. E. Deering, "Path MTU discovery; RFC-1191," *Internet Request for Comments*, vol. 1191, Nov. 1990.
[23] R. Ahuja, B. Prabhakar, and N. McKeown, "Multicast scheduling for input-queued switches," *IEEE J. Select. Areas Commun.*, vol. 15, pp. 855–866, May 1997.
[24] P. Newman, "IP switching and gigabit routers," *IEEE Commun. Mag.*, vol. 30, pp. 64–69, Feb. 1997.
[25] C. Partridge, "How slow is one gigabit per second?," *ACM Comput. Commun. Rev.*, vol. 21, no. 1, pp. 44–53, Jan. 1990.
[26] ——, *Gigabit Networking*. New York: Addison-Wesley, 1994.
[27] G. Parulkar, D. C. Schmidt, and J. Turner, "IP/ATM: A strategy for integrating IP with ATM," in *Proc. ACM SIGCOMM'95 (Special Issue of ACM Comput. Commun. Rev.)*, vol. 25, no. 4, pp. 49–59, Oct. 1995.
[28] D. Plummer, "Ethernet address resolution protocol: Or converting network protocol addresses to 48.bit Ethernet address for transmission on ethernet hardware," *Internet Request for Comments*, vol. 826, Nov. 1992.
[29] A. Rijsinghani, "Computation of the Internet checksum via incremental update; RFC-1624," *Internet Request for Comments*, vol. 1624, May 1994.
[30] J. Robinson, "The Monet switch," in *Internet Research Steering Group Workshop Architectures for Very-High-Speed Networks; RFC-1152*, Cambridge, MA, Jan. 24–26, 1990, p. 15.
[31] R. L. Sites, *Alpha Architecture Reference Manual*. Burlington, MA: Digital Press, 1992.
[32] Y. Tamir and H. C. Chi, "Symmetric crossbar arbiters for VLSI communications switches," *IEEE Trans. Parallel Distrib. Syst.*, vol. 4, pp. 13–27, Jan. 1993.
[33] A. Tantawy and M. Zitterbart, "Multiprocessing in high-performance IP routers," in *Protocols for High-Speed Networks, III (Proc. IFIP 6.1/6.4 Workshop)*. Stockholm, Sweden: Elsevier, May 13–15, 1992.
[34] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, "Scalable high speed IP routing lookups," in *Proc. ACM SIGCOMM'97*, Cannes, France, Sept. 14–18, 1997, pp. 25–37.

Craig Partridge (M'88–SM'91) received the A.B., M.Sc., and Ph.D. degrees from Harvard University, Cambridge, MA.

He is a Principal Scientist with BBN Technologies, Cambridge, MA, where he is the Technical Leader for the MultiGigabit Router. He is also a Consulting Assistant Professor at Stanford University, Stanford, CA.

Philip P. Carvey (M'79) received the B.S.E.E. degree from the Illinois Institute of Technology, Chicago, and the M.S.E.E. degree from the Massachusetts Institute of Technology, Cambridge.

He is Vice President of Engineering with Avici Systems, Inc., Chelmsford, MA, where he is building a terabit switch router. Previously, he was with BBN Technologies, Cambridge, MA, as Division Engineer, and with Ztel as Vice President of Research.

Ed Burgess, photograph and biography not available at the time of publication.

Isidro Castineyra received the M.S. and Ph.D. degrees from the Massachusetts Institute of Technology, Cambridge.

He is currently with the Internet Research Department, BBN Technologies, Cambridge, MA, as a Senior Scientist. His research centers on network resource management and routing.

Tom Clarke received the B.S. degree from the Massachusetts Institute of Technology, Cambridge.

He is currently with Avici Systems, Inc., Chelmsford, MA, as a Principal Engineer, working on high-performance Internet routers. Previously he was with BBN Technologies, Cambridge, MA, as a Principal Engineer, working on the MultiGigabit router.

Lise Graham, photograph and biography not available at the time of publication.

Michael Hathaway, photograph and biography not available at the time of publication.

Phil Herman, photograph and biography not available at the time of publication.

Allen King received the B.A., M.A., and E.E. degrees from the Massachusetts Institute of Technology, Cambridge.

He is currently with at Avici Systems, Inc., Chelmsford, MA, as a Consulting Engineer, where he is the Architect for the toroidal mesh interconnect of their terabit switch router. Prior to this, he was with BBN Technologies, Cambridge, MA, as a Senior Scientist, where he designed portions of the MultiGigabit Router and butterfly-type multiprocessor systems.

Steve Kohalmi received the B.S. degree from Syracuse University, Syracuse, NY, and the M.S. degree from the University of New Hampshire, Durham.

He is with BBN Technologies, Cambridge, MA, as a Senior Engineer, where he is responsible for IP and ATM QoS implementation of the MultiGigabit router.

Tracy Ma, photograph and biography not available at the time of publication.

John Mcallen received the A.B degree from Harvard University, Cambridge, MA.

He is a Software Engineer with BBN Technologies, Cambridge, MA, developing software and firmware for the MultiGigabit Router Forwarding Engine. Prior to this, he was with Digital and Sun Microsystems, where he worked on various network, processor, and cluster design projects.

Trevor Mendez received the B.S. degree from the Massachusetts Institute of Technology, Cambridge, MA.

He is currently with Cisco Systems, Cambridge, MA, as a Software Engineer. He was with BBN Technologies, Cambridge, MA, as a Scientist.

Walter C. Milliken (S'74–M'78) received the B.S. degrees (E.E. and C.S.) from Washington University, St. Louis, MO, and the M.S. degree from Stanford University, Stanford, CA.

He is with BBN Technologies, Cambridge, MA, as a Division Scientist, where he is the Software and System Design Leader for the MultiGigabit Router.

Ronald Pettyjohn (M'96) received the S.B.E.E. and S.M.E.E. degrees from the Massachusetts Institute of Technology, Cambridge.

He is presently with Argon Networks, Littleton, MA. Prior to this, he was with BBN Technologies, Cambridge, MA, as a Lead Scientist, where he led the group responsible for the implementation of the quad-SONET-OC12C/ATM line-card.

John Rokosz (S'92–M'95) received the A.B. degree from Harvard University, Cambridge, MA.

He is with the Internet Research Department, BBN Technologies, Cambridge, MA, where he leads hardware research activities and is the Hardware Manager for the MultiGigabit Router.

Joshua Seeger received the B.A. degree from the University of Rochester, Rochester, NY, and the Ph.D. degree in mathematics from Temple University, Philadelphia, PA.

He is with BBN Technologies, Cambridge, MA, as Director of Internetwork Research. His research interests have included multivariate probability distributions, error correction coding, routing algorithms for large networks, and network performance management.

Michael Sollins, photograph and biography not available at the time of publication.

Steve Storch received the B.S. degree in physics from the State University of New York, Stony Brook, in 1972.

He is with the Internetwork Research Department, BBN Technologies, Cambridge, MA, as an Area Manager with management responsibilities for research and development programs in such areas as high-speed routing and advanced satellite networking.

Benjamin Tober received the B.A. degree from Brandeis University, Waltham, MA.

He is with BBN Technologies, Cambridge, MA, as a a Staff Engineer.

Gregory D. Troxel received the S.B., S.M., E.E., and Ph.D. degrees from the Massachusetts Institute of Technology, Cambridge.

He is a Senior Scientist with BBN Technologies, Cambridge, MA, where he is the Technical Leader for the QoS-Capable Bilevel Multicast Router project.

David Waitzman received the B.S. degree from Carnegie-Mellon University, Pittsburgh, PA.

He is with D.E. Shaw & Company, L.P., Cambridge, MA, as a Software Developer on their Farsight online brokerage project. He was previously with BBN Technologies, Cambridge, MA, where he concentrated on network management.

Scott Winterble, photograph and biography not available at the time of publication.