A Fifty Gigabit Per Second IP Router

C. Partridge, P. Carvey, E. Burgess, I. Castineyra, T. Clarke, L. Graham, M. Hathaway, P. Herman, A. King, S. Kohlami, T. Ma, J. Mcallen, T. Mendez, W. Milliken, R. Osterlind, R. Pettyjohn, J. Rokosz, J. Seeger, M. Sollins, S. Storch, B. Tober, G. Troxel, D. Waitzman and S. Winterble

BBN Technologies (a part of GTE Corporation)

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.

1. 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 1,000 bits, is that for every gigabit per second (Gb/s) of bandwidth, a router needs a million packets per second (1 MPPS) of forwarding power.[1] Third, the router needs to conform to a set of protocol standards. For 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 million packet per second 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 2 to 10 times faster than the high-performance routers available today.

2. 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

Most of the work described in this paper is funded by the Defense Advanced Research Projects Agency (DARPA). The author list includes all major contributors to the final design of the router. Partridge and Carvey developed the router's architecture and led its implementation.

[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 bits long. Link-layer headers typically increase this to approximately 400 bits. 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 2,000 bits. 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.

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 one hundred 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.

2.1. Design Summary

A simplified outline of the MGR design is shown in Figure 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 Figure 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.

2.2. 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 detailed 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 the central table is many times (as much as 1,000 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 100KB 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 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 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 allow 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 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 processing in the router. We wanted to demonstrate that it was possible to build a cutting edge router that included line-speed quality of service. We chose to split the quality of service 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.

3. The Forwarding Engines

The forwarding engines are responsible for deciding where to forward each packet. When an 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 Labs router [2], which, although it has a different architecture, had to solve similar problems.

3.1. 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, 31].

The Alpha 21164 is a 64-bit, 32-register, super-scalar 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 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 3 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-bits long, this means the Icache can store 2,048 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-byte cache line misses.

There is a 96KB on-chip secondary cache (Scache) which caches both code and data. Loads from the Scache take a minimum of 8 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 bits, we have a maximum cache size of approximately 12,000 routes. Studies of locality in packet streams at routers suggest a cache this size should yield a hit rate well in excess of 95% [11, 15, 13]. Our own tests with a traffic trace from FIX West (a major inter exchange 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 21ns memory, the total time to load a 32-byte 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-bit 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 the 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 didn't we 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.

3.2. Forwarding Engine Hardware Operation

Once headers reach the forwarding engine, they are placed in a request 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 Figure 2. A packet header has reached the front of the request FIFO. The header includes the first 24 or 56 bytes of the packet plus an 8-byte 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 bytes 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 bytes 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 the packet should be discarded.) The hardware actually implements these FIFOs 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, used for Ethernet [28]. If a router gets a datagram to an IP address whose Ethernet address it doesn't 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 doesn't 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 ARPs 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.

3.3. 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

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

20 to 30 instructions (10 to 15 cycles) long. The first stage (a) does basic error checking to confirm that the header is indeed from an IPv4 datagram, (b) confirms that the packet and header lengths are reasonable, (c) that the IPv4 header has no options, (d) computes the hash offset into the route cache and loads the route; and (e) starts loading the next header. These five activities are done in parallel in intertwined instructions.

During the second stage 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 Bache. Then the code checks the IP 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 flow classifier. Currently we simply associate classifiers with destination prefixes, but one nice feature of the route-lookup algorithm 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 doesn't 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 9 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 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 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 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.

Observe that we've 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 use 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's been occasional speculation about how well RISC processors would handle IP datagrams.

Table 1 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-bit fields from 64-bit registers holding the IP and link-layer headers (the `ext` commands) and to zero byte-wide fields (`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 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-bit 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 SNMP 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 (`ldq`) to load the header and two loads to load the cached forwarding table entry. Then there are four stores (`stq`) to store the updated header and a call to create a write memory barrier (`wmb`) to ensure writes are sequenced.

3.3.1. 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.

3.3.1.1. 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 would it be more cost effective to implement the forward engine in an ASIC?

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

3.3.1.2. 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 using a cache makes sense.

Nonetheless, we believe 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 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.

3.4. 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. Figure 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 the same interface. The link-layer identifier is used in cases where multiple virtual interfaces may co-exist on one physical interface port, in which case a datagram may go in and out 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 both 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]

4. 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 to 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 the switch does not suffer Head-of-Line blocking [20] and it has been shown by simulation [30] and more recently proved [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

[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 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 card 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.

reverse of that made in many ATM switches and is why we built our own switch optimized for IP traffic.)

4.1. 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 2 request pins, 2 inhibit pins, 1 input status pin and 1 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 (8 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 bytes long. The motivation for sending two distinct units in one epoch was that the desirable transfer unit was 64 bytes (enough for a packet header plus some overhead information), but developing an FPGA-based allocator that could choose a connection pattern in 8 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-byte units are delivered to the same destination card. Function cards are not required to fill both 64-byte units. The second one can be empty. When a function card has a 64-byte unit to transfer, it is expected to wait a several epochs to see if another 64-byte 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-byte 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 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-bit mask formed by the 2 request pins over the 8 clock cycles in an epoch.[6] The allocator tells the source and destination cards who they will be connected to with a 4-bit number (0 to 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-bit mask formed by the 2 inhibit pins. Destinations are allowed to inhibit transfers from a source to protect against certain pathological cases where packets from a single source could consume the entire destination card's buffer space, preventing other sources from transmitting data through the destination card.[7]

4.2. The Allocator

The allocator is the heart of the high-speed switch in the MGR. It takes all 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 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 nanoseconds).

A straightforward allocator algorithm is shown on the left side of Figure 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

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

[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.

allocator simply scans the matrix, left to right, 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's a preference for low-numbered sources; and (2) for a 15×15 matrix, it requires serially evaluating 225 positions per epoch − that's one evaluation every 1.4 ns, too fast for an FPGA.

There is an elegant solution to the fairness problem: randomly shuffle the sources and destinations. The allocator has two 15-entry shuffle arrays. The sources array is a permutation of the values 1 to 15, and value of position `s` of the array indicates what row in the allocation matrix should be 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, `M`, one generates a shuffled matrix `SM` according to the following rule:

```
SM[x,y] = M[rowshuffle[x],colshuffle[y]]
```

and uses `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 Figure 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 Wave Front Allocation [16, 32] and is illustrated in the middle of Figure 4. However, for a 15×15 matrix, Wave Front 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 Figure 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 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.

5. Line Card Design

A line card in the MGR can have up to sixteen 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 Gb/s and the 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 sixteen 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 processing is 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

[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 uneveness 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.

[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

processing is rather simple. Outbound packet processing is much more complex.

5.1. 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-byte 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-byte payload, which is less than the 64-byte 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-byte full ATM cell in a page and ship the cell directly to an outbound interface.

5.2. 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 Quality of Service (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 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 VLIW programmable state machine implemented in a 52 MHz ORCA 2C04A FPGA. (ORCA FPGAs 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 schedule it. The second event occurs when the transmission interface is ready for another packet to send. The transmission 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 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 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 scheduling, after that 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 8,000 ATM VCs per interface, each with its own quality of service parameters for CBR, VBR or UBR service.

6. The Network Processor

The network processor is a commercial PC motherboard with a PCI interface. This motherboard uses a 21064 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.

switch and the inbound side of the card in the same cycle). That's painful to implement at high speed.

The processor runs the 1.1 NetBSD release of UNIX. NetBSD is a freely available version of UNIX based on the 4.4 BSD software 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.

7. 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 who 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, do not have to affect router throughput. The network processor can delay updating the forwarding tables on the forwarding engines until the flapping has subsided.

8. 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-byte datagram entering an otherwise idle router, would experience a delay of between 7 and 8 microseconds. A 1 kilobyte datagram would take 0.9 microseconds longer. These delays assumes 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 the instruction cache and branch predictions are correctly made).

9. Related Work and Conclusion

Many of the features of the MGR have been influenced by prior work. The Bell Labs 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 how parallel IP header processing might be. So too, several people have looked at ways to adapt switches to support IP traffic [27, 24].

Beyond the innovations outlined in section 2.2, we believe 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 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.

Acknowledgments

Many people have contributed to or commented on the ideas in this paper including Jeff Mogul and others at Digital Equipment Corporation's Western Research Lab, Steve Pink, Joe Touch, Dennis Ferguson, Noel Chiappa, Mike St Johns, Gary Minden and members of the Internet End-To-End Research Group chaired by Bob Braden. The anonymous TON reviewers and the TON technical editor, Guru Parulkar, also provided very valuable comments which substantially improved this paper.

References

1. *Alpha 21164 Microprocessor; Hardware Reference Manual,* Digital Equipment Corporation (April 1995).
2. Asthana, A., C. Delph, H.V. Jagadish, and P. Krzyzanowski, "Towards a Gigabit IP Router," *Journal of High Speed Networks,* 1, 4, pp. 281-288 (1992).
3. Baker, F., "Requirements for IP Version 4 Routers; RFC-1812," *Internet Request For Comments,* 1812 (June 1995).
4. Braden, B., D. Borman, and C. Partridge, "Computing the Internet Checksum; RFC 1071," *Internet Request for Comments,* 1071 (September 1988).
5. Bradner, S., and A. Mankin, *IPng: Internet Protocol Next Generation,* Addison-Wesley (1995).
6. Brodnik, A., S. Carlsson, M. Degermark, and S. Pink, "Small Forwarding Tables for Fast Routing Lookups," *Proc. ACM SIGCOMM '97,* Cannes (September 1997).
7. Chiappa, N., *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)* (28 September 1993).
8. Deering, S., and R. Hinden, "Internet Protocol, Version 6 (IPv6); RFC-1883," *Internet Requests for Comments,* 1883 (January 1996).
9. Demers, A., S. Keshav, and S. Shenker, "Analysis and Simulation of a Fair Queueing Algorithm," *Internetwork: Research and Experience,* 1, 1, pp. 3-26, John Wiley & Sons (September 1990).
10. Fedor, M., "Gated: A Multi-routing Protocol Daemon for UNIX," *Proc. 1988 Summer USENIX Conference,* pp. 365-376, San Francisco, CA (1988).
11. Feldmeier, D.C., "Improving Gateway Performance with a Routing-Table Cache," *Proc. IEEE INFOCOM '88,* pp. 298-307, New Orleans (March 1988).
12. Floyd, S., and V. Jacobson, "Random Early Detection Gateways for Congestion Avoidance," *IEEE/ACM Transactions on Networking,* 1, 4, pp. 397-413 (August 1993).
13. Heimlich, S.A., "Traffic Characterization of the NSFNET National Backbone," *Proc. Winter 1990 USENIX Conference,* pp. 207-227, Washington DC (January 1990).
14. Jain, R., "A Comparison of Hashing Schemes for Address Lookup in Computer Networks," *IEEE Trans. on Communications,* 40, 10, pp. 1570-1573 (October 1992).
15. Jain, R., and S. Routhier, "Packet Trains: Measurements and a New Model for Computer Network Traffic," *IEEE Journal on Selected Areas of Communication,* 4, 6, pp. 986-995 (September 1986).
16. Lamaire, R.O, and D.N. Serpanos, "Two Dimensional Round Robin Schedulers for Packet Switches with Multiple Input Queues," *IEEE/ACM Trans. on Networking,* pp. 471-482. (October 94).
17. Lewis, H.R., and L. Denenberg, *Data Structures & Their Algorithms,* p. Harper Collins (1991).
18. Mallory, T., and A. Kullberg, "Incremental Updating of the Internet Checksum; RFC-1141," *Internet Requests for Comments,* 1141 (January 1990).
19. McKeown, N., M. Izzard, A. Mekkittikul, B. Ellersick, and M. Horowitz, "The Tiny Tera: A Packet Switch Core," *IEEE Micro* (January 1997).
20. Karol, M.J., M.G. Hluchyj, and S.P. Morgan, "Input Versus Output Queueing on a Space-Division Packet Switch," *IEEE Trans. Communications,* 35, 12, pp. 1347-1356 (December 1987).
21. McKeown, N., V. Anantharam, and J. Walrand, "Achieving 100% Throughput in an Input-Queued Switch," *Proc. IEEE Infocom '96,* San Francisco (March 1996).
22. Mogul, J.C., and S.E. Deering., "Path MTU Discovery; RFC-1191," *Internet Requests for Comments,* 1191 (November 1990).
23. Prabhakar, B., N. McKeown, and R. Ahuja, "Multicast Scheduling for Input-Queued Switches," *IEEE Journal on Selected Areas of Communications* (May 1997).
24. Newman, P., "IP Switching and Gigabit Routers," *IEEE Communications Magazine* (February 1997).
25. Partridge, C., "How Slow Is One Gigabit Per Second?," *ACM Computer Communication Review,* 21, 1, pp. 44-53 (January 1990).
26. Partridge, C., *Gigabit Networking,* Addison Wesley Publishers (1994).

27. Parulkar, G., D.C. Schmidt, and J. Turner, “IP/ATM: A Strategy for Integrating IP with ATM,” *Proc. of ACM SIGCOMM ’95 (Special Issue of ACM Computer Communication Review),* 25, 4, pp. 49-59 (October 1995).

28. Plummer, D., “Ethernet Address Resolution Protocol: Or converting network protocol addresses to 48.bit Ethernet address for transmission on Ethernet hardware,” *Internet Request for Comments,* 826 (November 1992).

29. Rijsinghani, A., “Computation of the Internet Checksum via Incremental Update; RFC-1624,” *Internet Request for Comments,* 1624 (May 1994).

30. Robinson, J., “The Monet Switch,” *Internet Research Steering Group Workshop on Architectures for Very-High-Speed Networks; RFC-1152,* DDN Network Information Center, Cambridge, MA (24-26 January 1990).

31. Sites, Richard L., *Alpha Architecture Reference Manual,* Digital Press (1992).

32. Tamir, Y., and H.C. Chi, “Symmetric Crossbar Arbiters for VLSI Communications Switches,” *IEEE Trans. Parallel and Distributed Systems,* 4, No. 1 (1993).

33. Tantawy, A., and M. Zitterbart, “Multiprocessing in High-Performance IP Routers,” *Protocols for High-Speed Networks, III (Proc. IFIP 6.1/6.4 Workshop),* Elsevier, Stockholm (13-15 May 1992).

34. Waldvogel, M., G. Varghese, J. Turner, and B. Plattner, “Scalable High Speed IP Routing Lookups,” *Proc. ACM SIGCOMM ’97,* pp. 25-37, Cannes (14-18 September 1997).

Table 1: 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

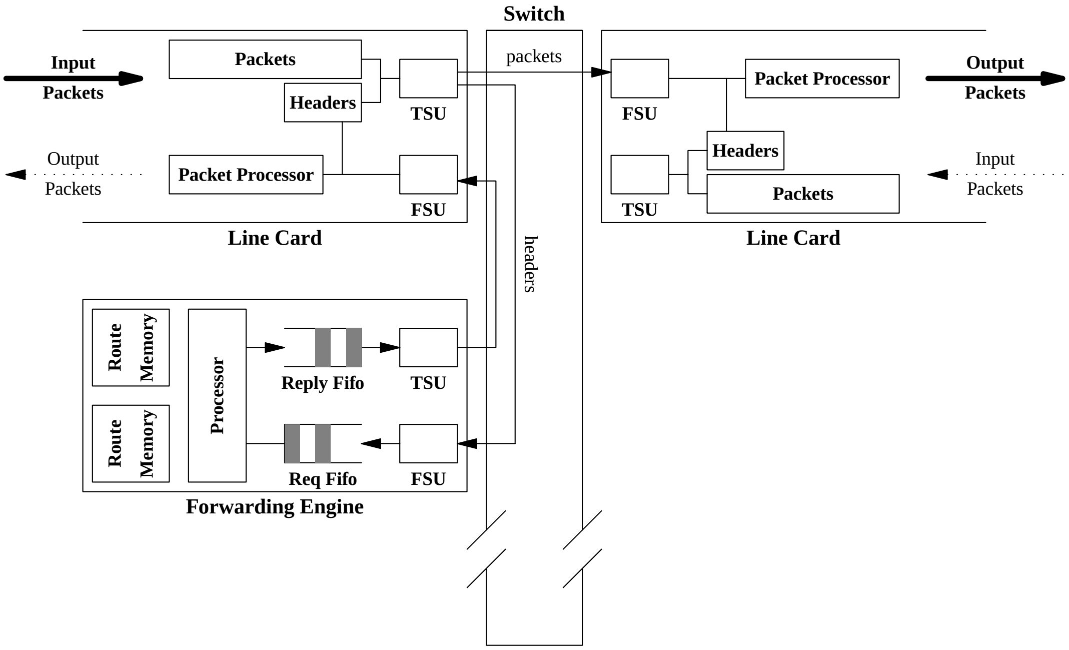


Figure 1: MGR Outline

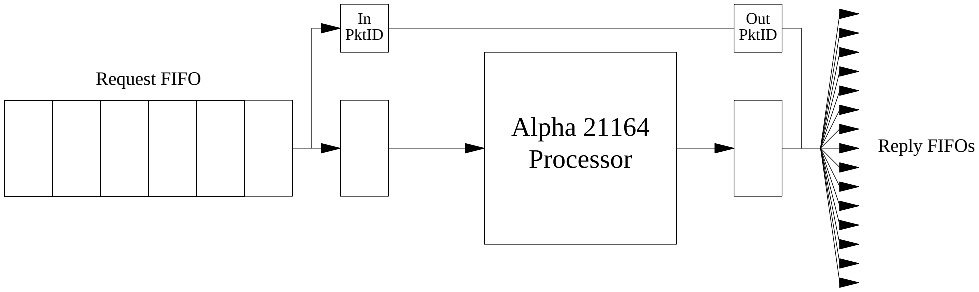
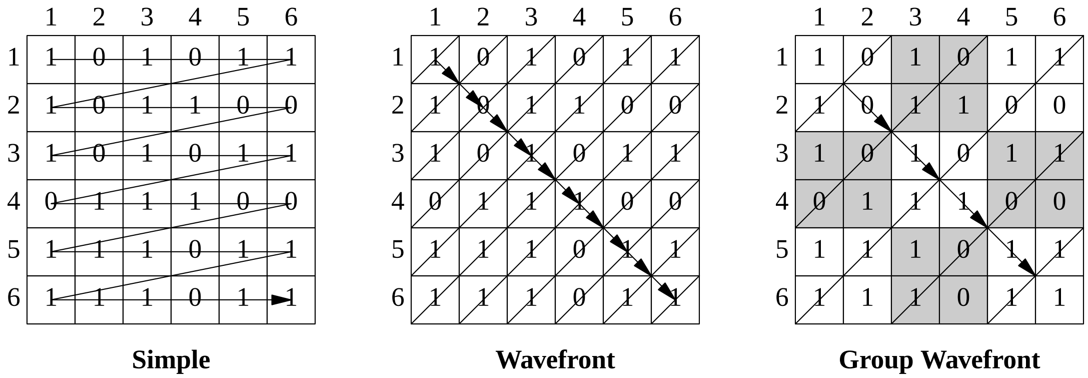


Figure 2: Basic Architecture of Forwarding Engine



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

Simple

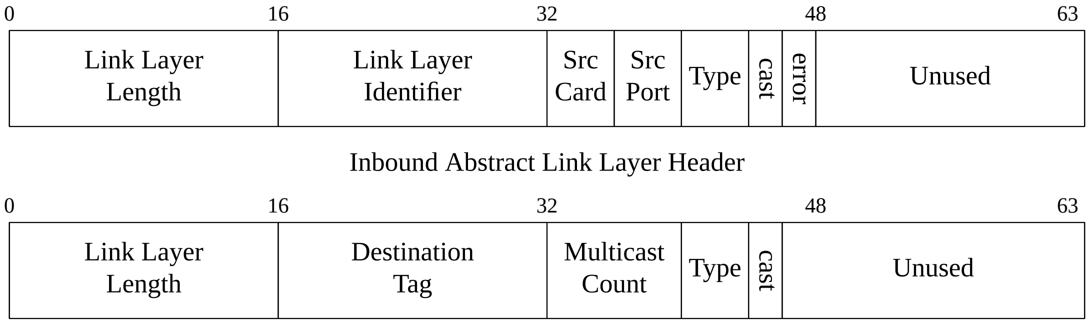
	1	2	3	4	5	6
1	1	0	1	0	1	1
2	1	0	1	1	0	0
3	1	0	1	0	1	1
4	0	1	1	1	0	0
5	1	1	1	0	1	1
6	1	1	1	0	1	1

Wavefront

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

Group Wavefront

Figure 4: Simple and Wavefront Allocators



Inbound Abstract Link Layer Header

Outbound Abstract Link Layer Header

Figure 3: Abstract Link Layer Headers