LECTURE 10
Design of Big, Fast Routers

This lecture discusses why high-speed Internet routers are complex and briefly surveys how they are designed. The main ideas to understand are: (1) the design constraints, such as speed, size, and power consumption, (2) the components in a modern (early 2000s) router (such as ports, line cards, packet buffers, forwarding engine, switch arbiter, switch fabric, link scheduler, etc.), and (3) the need for efficient algorithms for lookups, packet queueing, and switch arbitration. We will focus on the general ideas rather than on how a specific router is implemented; these notes are best read together with specific case studies such as BBN's multi-gigabit router (which is a relatively old design) and the somewhat more recent Stanford TinyTera system.

■ 10.1 Introduction

In the first few lectures of this course, we looked at the architecture of packet-switched networks and saw that the *switch* is a fundamental building block in such networks. In this lecture, we will look at the problem of designing an packet switch for IP packet processing that can support links of high speeds.[1] Currently (early 2000s), high-speed Internet routers that process IP packets have links whose speeds are in the range of 1 Gigabit/s to about 40 Gigabits/s per link. A typical high-end router might have between 500 and 1000 *ports*, distributed across perhaps 64 or 128 *line cards*. The ports usually correspond to distinct "next hops".[2]

The conceptual model of a router is quite straightforward: a router has a *data path* and a *control path*. The job of the data path is to move packets that arrive on *input* or *ingress* ports to *output* or *egress* ports; from the outputs, these packets are sent on downstream to the next switch on the path or to an end-host. The job of the control path is to implement algorithms, which are typically distributed, that help the data path figure out the right port to send the packet out on. In high-speed switches, the data path typically requires a hardware implementation, while the control path is almost always done in software.

[1] A switch in the context of Internet datagram delivery is generally called a *router*, because it also runs routing protocol software to build forwarding tables. We will use the terms "switch" and "router" interchangeably in this lecture.

[2] These numbers change every year or two.

Routers are a great example of special-purpose computers built for a specific set of tasks. Routers are big business: according to some market research firms, worldwide revenue from enterprise and service provider router sales in 2003-2004 was about US $5.5 billion (source: Infonetics Research Report, Q2 2004).

Most of this lecture will be concerned with the data path of a router, focusing on how routers can implement the data path efficiently. There are three parts to this lecture. We will start by looking at an abstract model of an Internet router, describing the key components of a "bare-bones" (but functional) router, and giving a broad sense of how routers have evolved over the past several years. Then, we will figure out how to make our bare-bones IP router fast, a process that involves understanding hardware characteristics and trends, some clever algorithms, and will give you a sense for why even simple IP routers are complex. Finally, we will discuss "feature creep"—IP routers today implement much more than our bare-bones router, leading to an daunting degree of complexity.

The earliest routers (until the mid-to-late 1980s) were built by having no packet buffer memory on line cards and by attaching each line card to a shared bus backplane. Packet buffer memory was also attached to the backplane, and all forwarding decisions were made by a central processor also connnected to the backplane. As packets arrived, they were stored in memory by moving the bits over the shared bus, after first passing the packet (or the header) to the processor to determine the egress link for the packet.

This design, while conceptually simple, has a scaling bottleneck: the shared bus. Various caching-based optimizations were developed to alleviate this bottleneck (*e.g.*, caching parts of the forwarding table).

Modern high-speed routers don't use a shared bus: they use a *crossbar*, which is a hardware component that permits more parallelism than a shared bus. Conceptually, a crossbar has N input ports and N output ports (in general, every input port is also an output port). The crossbar has a *scheduler* that typically works in units of time-slots. In each time-slot, the crossbar can move some number of bits between ports across its fabric, subject to a *crossbar constraint*. This constraint states that at any given time, if input i is connected to output j, then no other output can be simultaneously connected to output j, and no other input can be simultaneously connected to input i. In other words, one can think of the crossbar as forming a bipartite graph with the input ports being one partition and the output ports being the other, and an edge from i to j signifies that input i has a packet destined for output j. Then, at any given time, the crossbar scheduler can pick some *matching* along which bits can be moved concurrently across the crossbar.

■ 10.2 High-Speed Routers are Complex

The fundamental problem in the design of high-speed routers is that the designer can't simply sit back and let Moore's law take care of processing packets at the desired speeds. They must solve a number of problems efficiently, and of these many problems, we survey two: fast IP lookups and fast crossbar scheduling.

■ 10.2.1 Fast Lookups

A router needs to implement a *prefix match* to check if the address being looked-up falls in the range A/m for each entry in its forwarding table. A simple prefix match works when

the Internet topology is a tree and there's only one shortest path between any two networks in the Internet. The toplogy of the Internet is not a tree, however: many networks *multi-home* with multiple other networks for redundancy and traffic load balancing (redundancy is the most common reason today).

The consequence of having multiple possible paths is that a router needs to decide on its forwarding path which of potentially several matching prefixes to use for an address being looked-up. By definition, IP (CIDR) defines the correct lookup response as the *longest prefix* that matches the sought address. As a result, each router must implement a *longest prefix match* (LPM) algorithm on its forwarding path.

LPM is not a trivial operation to perform at high speeds of millions of packets (lookups) per second. For several years, the best implementations used an old algorithm based on *PATRICIA trees*, a trie (dictionary) data structure invented decades ago [**?**]. This algorithm, while popular and useful at lower speeds, does not work in high-speed routers.

To understand the problem of high-speed lookups, let's study what an example high-speed Internet router has to do today. It needs to handle minimum-sized packets (e.g., 40 or 64 bytes depending on the type of link) at link speeds of between 10 Gbits/s (today) and 40 Gbits/s (soon), which gives the router a fleeting 32 ns (for 40-byte packets) or 51 ns (or 64-byte packets) to make a decision on what to do with the packet! (Divide those numbers by 4 to get the 40 Gbits/s latencies.)

Furthermore, a high-speed Internet router today needs to be designed to handle on the order of 250,000 forwarding table entries, and maybe more (today's Internet backbones appear to have around 120,000 routes, and this number has been growing; people expect a commercial router to have a lifetime between five and ten years in the field).

30–50 nanoseconds per lookup is difficult to achieve without being clever. One can't really store the forwarding tables in DRAM since DRAM latencies are on the order of 50ns, and unless one has a prohibitively large amount of memory, doing an LPM in one lookup is impossible.

We are left with SRAM, which has latencies (on the order of 5 ns) that are workable for our needs. Furthermore, over the past couple of years, SRAM densities have approached DRAM densities, allowing router designers to use SRAM for largish forwarding tables.

We can formulate the forwarding table problem solved in Internet routers as follows. Based on the link speed and minimum packet size, we can determine the number of lookups per second. We can then use the latency of the forwarding table memory to determine M, the maximum number of memory accesses allowed per lookup, such that the egress link will remain fully utilized. At the same time, we want all the routes to fit into whatever amount of memory (typically SRAM), S, we can afford for the router. The problem therefore is to maximize the number of routes that fit in S bytes, such that no lookup exceeds M memory accesses.

This problem has been extensively studied by many researchers in the past few years and dozens of papers have been published on the topic. The paper by Degermark *et al.* in the optional readings describes one such scheme (now called the "Lulea" scheme), which uses aggressive ompression to fit as many routes as possible into S bytes. Although their paper was originally motivated by software routers, these are the same optimizations that make sense in fast hardware implementations of the IP forwarding path.

The Lulea scheme observes that a forwarding table viewed as a *prefix tree* that is com-

plete (each node with either 0 or 2 children) can be compressed well. The scheme works in three stages; first, it matches on 16 bits in the IP address, and recursively applies the same idea to the next 8 and the last 8 bits. See the paper for details.

■ 10.2.2 High Throughput in a Crossbar-based Switch Fabric

The key problem is crossbar scheduling: finding a matching in the bipartite graph efficiently.

This requires a number of optimizations, and many approaches have been proposed in the literature (and several designs have been realized in practice).

The first important optimization is *virtual output queueing*. If there is only one queue per input storing all packets regardless of their destination output port, then *head-of-line* blocking ensues. In the presence of such blocking, the maximum throughput of the switch does not exceed $2 - \sqrt{2}$. All crossbar switches now implement virtual output queueing, where each input has (at least) N separate queues, one for each output. (They may have additional queues for each output if they implement QoS.)

Perhaps the earliest crossbar scheduler was the Wavefront arbiter, later implemented in the BBN switch. The idea here is to produce a maximal matching by proceeding in a "wavefront" across the matrix of input-output demands. The rows of the matrix correspond to the inputs, and the columns to the outputs. Each element in the matrix is 0 if there are no packets waiting on that pair, and non-zero otherwise. The wavefront arbiter produces a matching by starting with the (0,0) entry, and moving diagonally across the matrix. It can match all previously unmatched non-zero elements at the same time, because along any (minor) diagonal, there can be no contention for the same input or output. Hence, by doing $2N - 1$ such sweeps across the matrix, it produces a maximal matching. To improve fairness, the order of the sweeps can be randomized.

Because the Wavefront might be too slow for some systems, other schemes were also developed. An early switch scheduling scheme was *Parallel Iterative Matching* (PIM), pioneered in DEC's AutoNet switch (Anderson, Owicki, et al.). PIM is a randomized matching scheme that operates in three phases: request, grant, and accept.

In the request phase, each input port sends "requests" to all outputs for which it has packets. In the grant phase, an output picks an input *at random*, and sends a grant to it. At this stage, no output is committed to more than one input, but an input may receive a grant from more than one output. Hence, in the final "accept" phase, an input picks one of the granting outputs *at random*.

It is easy to see that this three-phase approach produces a matching, but the matching is not maximal: *i.e.*, there may an input and output that are each unmatched, and which have packets for each other. To produce a more complete matching, PIM runs the above three-phase procedure a number of times, eliminating at each stage all previously matched nodes. They show that doing that about $\log N$ times results in maximal matchings most of the time.

The problem with PIM is that it can lead to unfairness over short time scales. McKeown's iSLIP algorithm overcomes this problem by removing the randomness. Each input maintains a round-robin list of outputs, and each output maintains a round-robin list of inputs. The request phase is the same as in PIM. In the grant phase, rather than pick at random, each output picks the first input in the round-robin sequence from the previously

matched input for that output, and updates this state only if this input is picked in the accept phase. Similarly, in the accept phase, an input picks the first output in round-robin sequence following the previously matched output for that input.

Simulations show that this scheme has good fairness properties. It is implemented in many commercial routers.

For several years, it was not clear whether these simple schemes achieve 100% throughput in a switch. Several complex scheduling schemes were shown to achieve 100% throughput, but they were impossible to implement. Dai and Prabhakar showed a breakthrough result a few years ago, proving that *any maximal matching* scheme running in a crossbar switch with 2× speedup can achieve 100% throughput. (The speedup of a switch is the relative speed of the internal links of the crossbar to the input and output link speeds.)

■ 10.3 Death by a Thousand Cuts: Feature Creep in Modern Routers

The data path often does more than simply forward packets in the order in which they came in. The following is a sampling of the "check-box" data-path functions that today's high-speed routers tend to have to support:

1. Packet classification, involving processing higher-layer protocol fields.
2. Various counters and statistics for measurement and debugging.
3. IPSec (security functions), especially at "edge" routers. More sophisticated security services such as Virtual Private Networks (VPNs). Firewalls.
4. Quality-of-service, QoS (I): Rate guarantees and traffic class isolation.
5. Packet "shaping" and "policing" functions.
6. IPv6.
7. Denial-of-service remediation schemes (the simplest of which is access control based on ingress filtering of packets that don't have a valid source address for the incoming interface that the packet came in on) and traceback (figuring out the Internet path along which a given packet or stream of packets came).
8. IP multicast.
9. Packet-drop policies ("active queue management") such as RED.
10. QoS: Differentiated Services (differentiating traffic classes using suitable scheduling).
11. QoS: Integrated Services (end-to-end delay/rate guarantees).

We won't discuss these features in detail here, nor will we pass judgement on which of these is actually a feature users care about and which are simply "check box" items. Some of these features have been covered in previous lectures in the course, either because they are intellectually interesting or because some users care about them.

■ Appendix A: A "cheat sheet" summarizing the main ideas in the BBN 50 Gb/s router

■ 10.3.1 Why bother?

- Trends
  1. Faster and faster link bandwidths.
  2. Larger and larger network size (hosts, routers, users).
  3. Users want more and more (15% increase in per-user demand in 1 year!).
  4. Not just because of Web: for example, Internet size has been growing at 80% per year since at least 1984!
  5. Conclusion: Unlike other areas in computer science that can rely on Moore's law to achieve progress, many aspects of networking cannot!
- *Conventional "wisdom":* IP routers are inherently slow. They cannot possibly forward packets fast enough, where "fast" refers to multi-gigabit (or higher) rates.
- This paper's motivation, at least in part, was to refute this belief.

■ 10.3.2 How does it work?

- Routers do two things: participate in routing protocols with their neighbors, and forward packets. We are concerned primarily with the latter, since it's the data path.
- At first sight, forwarding is straightforward:
  1. Router gets packet.
  2. Looks at packet header for destination.
  3. Looks up routing table for output interface.
  4. Modifies header (ttl, IP header checksum).
  5. Passes packet to output interface.
- But there's a lot of stuff to take care of: RFC 1812 is 175 pages long!
- **Challenge.** How to do all this *fast.* In particular, in the common case.
- **Architecture.** Consists of:
  1. Multiple line cards.
  2. Multiple forwarding engines.
  3. High-speed switch fabric and processor.
  4. Network processor.
- Salient features of architecture:
  1. Separate forwarding engine from line card. From input line card, send only header to engine. Engine returns modified header to input line card, after which it gets vectored to output line card.

2. Each forwarding engine has its own routing table (cache). Notice that only a subset of the complete routing table, which gives prefix-to-output-interface mapping is needed. This is therefore a *forwarding table* rather than complete routing table.
3. Use a switched backplane rather than a shared bus. Custom-designed for IP rather than an ATM switch. The big advantage of this is parallelism.
4. QoS processing in router (on output line card).
5. An abstract link-layer to handle different link technologies.

- **Forwarding engines:**
  - Key design principle: *optimize the common case.*
  - Not ASIC but based on Alpha 21164 at 415 MHz.
  - 8 KB I and D caches: forwarding code.
  - 96 KB L2 cache (Scache): cache of routing table. Only relevant table entries. 12000 routes (95% hit rate).
  - 16 MB L3 cache (Bcache): complete forwarding table in 2 8 MB chunks. [Why is it banked like this?]
  - Why not an ASIC or embedded processor? Want software. And chip has a very high clock speed and large Icache and Scache.
  - Optimize common case in custom assembler. 85 instructions to forward a packet.
  - Note: Does not check IP header checksum. Q: is this a problematic drawback?
  - Not-so-common cases deferred to network processor. This includes route cache misses, multicast, IP options, packets requiring ICMP (uses templates), packets needing fragmentation.
  - Q: Do we need a route cache in the future? A: depends on what algorithm you choose to forward.
- **Switching fabric:** (Draw pic. of switch)
  - Advantage over bus? Parallelism.
  - Disadvantage? Multicast is hard. Need scheduling machinery. (They call this "allocation").
  - Input-queued switch. Q: what's the problem with standard input-queued switch? A: h-o-l blocking.
  - How to fix this? Keep track of per-output port traffic. Each input port bids for different outputs. In each epoch, match an input to output port.
  - Allocator arbitrates amongst bids to do this. Maximize switch throughput (rather than bounded latency).
  - Scheduling is pipelined (4 stages): get bids, allocate schedule, notify pairings, ship bits.

  - Q: What graph problem does this correspond to? A: bipartite matching.
  - Flow control by destination ports. Prevents an input from hogging an output.

- **Allocation algorithm:**

  - Heavily studied problem. Approaches include Parallel Iterative Matching (from DEC), wavefront scheduling, etc.
  - Use *randomization* to alleviate unfairness issues.
  - Essentially a greedy algorithm that progresses along diagonal of allocation matrix.
  - Prioritize forwarding engine inputs over line cards (by skewing shuffling of ports).

- **Line cards:**

  - Input: simple, except for ATM (small cells) and multicast (copies have to be made).
  - Output: QoS processor for scheduling and queue management. A VLIW FPGA.

- **Network processor:**

  - Alpha processor running NetBSD. (Another nice example in this architecture of using off-the-shelf components.)
  - Handles "not-so-common" cases of packet processing.
  - Handles routing protocols (gated).
  - Builds forwarding tables and populates table for each forwarding engine using a subset of each entry.
  - Using 2 banks in forwarding engines reduces disruptions and flapping effects; allows network processor to upload a forwarding table while forwarding is still in progress.

■ 10.3.3 Fast lookups

- Two approaches from Sigcomm 97 papers (current state-of-the-art)

  1. Make better use of caching via compression (Degermark et al.).
  2. Constant-time lookups via clever data structures (Waldvogel et al.).

- Problem: Best-matching prefix.

- Basic idea: use hash tables for each prefix length. Then, apply binary search techniques to get to the longest matching prefix to find output interface.

10.3.4 Performance

- Peak performance of about 50 Gigabits/s, assuming 15 simultaneous transfers take place through the switch in a single transfer cycle (called an *epoch*, equal to 16 clock ticks). (Notice that it includes the line card to forwarding engine transfers too!)
- Note: this is only an estimate, not much deployment experience/measurements reported in paper.