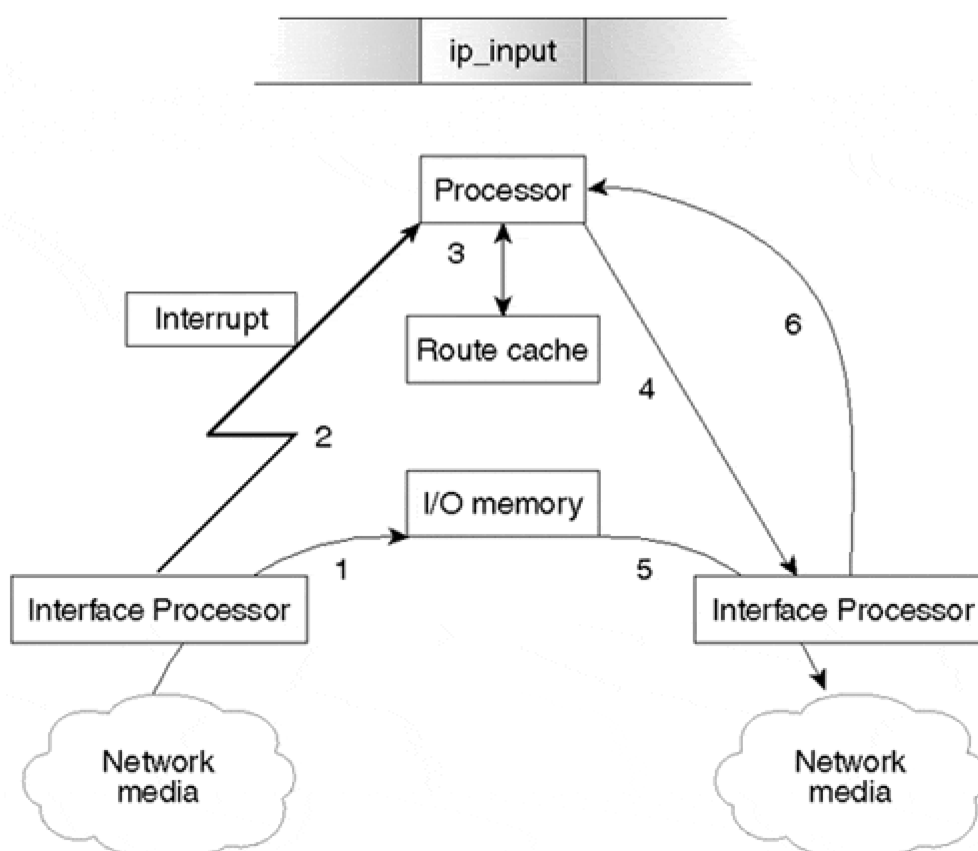
Fast Switching: Caching to the Rescue

The term *cache* in the computer industry usually refers to the concept of storing some frequently used subset of a larger data set in a local storage area with very fast access properties. A computer might keep a local copy of frequently accessed parts of a file stored on disk in RAM, for example, or a CPU might prefetch some instructions into a very fast associative memory area to boost performance. The two key characteristics of such a cache are a relatively small size as compared to the entire data space, and the capability to provide very fast access to any addressable member of its contents.

The IOS developers used these concepts when they created the Fast Cache. The Fast Cache quite simply is a data structure in IOS used to keep a copy of the reachability/interface/MAC-header combinations learned while process switching packets.

To understand how the Fast Cache can be helpful, consider the example of process switching a packet discussed earlier in this chapter. Now add a step to the switching operation performed by the **ip_input** process. After **ip_input** looks up the next hop, output interface, and MAC-header data for a packet, add a step that saves this information in a special data structure that allows very fast access to any cache entry based on a destination IP address. This structure is the Fast Cache. Over time, the **ip_input** process builds a large number of frequently used IP destinations into the cache. Now you'll learn how to put this cache to use in switching packets. Figure 2-4 shows the fast-switched path.

Figure 2-4. The Fast-Switched Path



Consider the process switching method again—only this time the newly built Fast Cache is introduced. Again, the switching process begins with the interface hardware sensing there is a packet on the wire. It receives the packet and transfers it to I/O memory—Step 1 in Figure 2-4.

In Step 2, the interface hardware interrupts the main processor to inform it there is a received packet waiting in I/O memory. The IOS interrupt software inspects the packet's header information and determines it is an IP packet. Now, instead of placing the packet in the input queue for **ip_input** as before, the interrupt software consults the Fast Cache directly to determine whether an outbound interface and a MAC header have been cached for this destination. Assuming there is a cache entry, the interrupt software reads the MAC header from the entry and writes it into the packet. It also reads a pointer to the appropriate outbound interface from the cache entry. Step 3 in Figure 2-4 illustrates the cache read and MAC rewrite.

The main processor (still within the same interrupt) then notifies the outbound interface hardware that a packet in I/O memory is ready to transmit and dismisses the interrupt so other processes can continue—Step 4 in Figure 2-4.

The interface hardware dequeues the packet from I/O memory and transmits it in Step 5, and then interrupts the main processor to update its counters and release the I/O memory, Step 6.

The example just described illustrates how the fast switching method works. Notice the **ip_input** process never gets involved in switching this packet; in fact, no scheduled process gets involved when a packet is fast switched as long as a cache entry exists. With the introduction of the Fast Cache, IOS can now perform the entire packet switching operation within the very brief period of an interrupt! Caching has enabled IOS to separate the resource-intensive task of making a routing decision from the relatively easy task of forwarding a packet. Thus, fast switching introduced the concept of "route once, forward many times."

An important implication of the fast switching process should be noted. As you've seen, Fast Cache entries are built while packets are process switched. Because the process switching operation builds the cache entries, the first packet sent to any given destination *will always be process switched* even when fast switching is enabled. As long as the cache entry exists, future packets to the destination can be fast switched.

This method of using the process switching operation to populate the Fast Cache works well as long as certain assumptions are met: The network must be generally stable with few routes changing and traffic must tend to flow between a particular subset of destinations. These assumptions are true in most cases. In some network environments, however, particularly the Internet backbone, they are not true. In those environments, network conditions can cause an increased number of cache *misses* (an instance where no cache entry matches a packet) and can result in a high number of packets being process switched. It's also possible for a certain set of conditions to cause *cache thrashing*, where older cache entries are continually overwritten with newer cache entries because there just isn't enough room in the cache for all the entries required. Such environments are discussed later in this chapter.

Fast Cache Organization

How does a Fast Cache actually work? How can it provide forwarding data so quickly? For the answer, look at how a typical Fast Cache is organized.

First, to get an idea of exactly what's in the Fast Cache, take a look at the output of the **show ip cache verbose** command in Example 2-3.

Example 2-3. Displaying Fast Cache Contents

router#**show ip cache verbose** IP routing cache 1 entry, 172 bytes 124 adds, 123 invalidates, 0 refcounts Minimum invalidation interval 2 seconds, maximum interval 5 seconds, quiet interval 3 seconds, threshold 0 requests Invalidation rate 0 in last second, 0 in last 3 seconds Prefix/Length Age Interface Next Hop 10.1.1.16/32-24 1w4d Ethernet0 10.1.1.16 14 00403337E5350060474FB47B0800

From the output in Example 2-3, it's obvious the router keeps the destination prefix, the length of the prefix, the outbound interface, the next hop IP address, and a MAC header—just what you'd expect. All the data necessary to switch a packet to a particular destination is contained in this one entry.

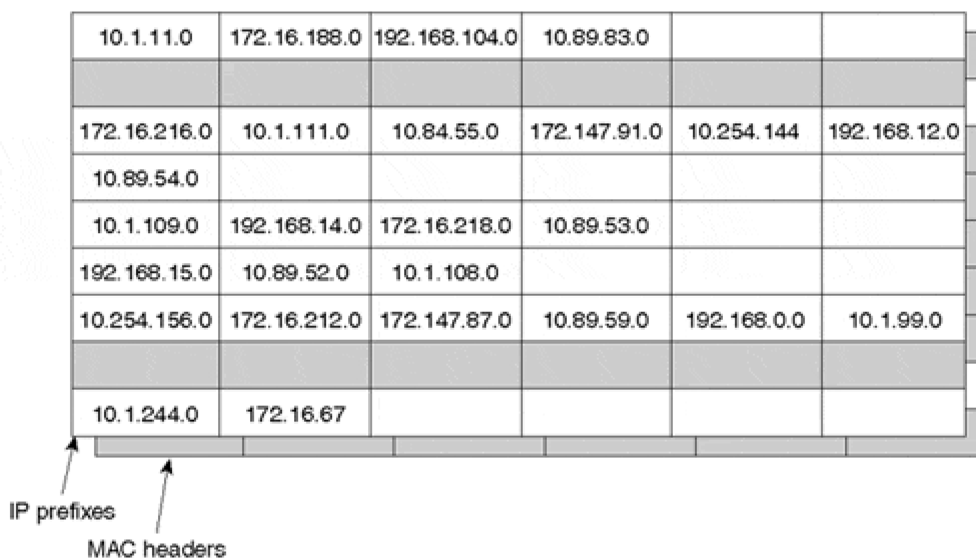
Another characteristic of the cache is not evident from **show** command output. Unlike the master tables used to build them, which are basically just big lists, caches are implemented using special data structures, which allow any particular member to be retrieved with a minimal amount of searching. With the master tables, the searching time increases proportionately with the size of the table. With the cache data

structures, because of the way they are organized, the searching time can be kept small and fairly constant regardless of the total number of entries.

Fast Cache Data Structures: Hash Tables and Radix Trees

The IP Fast Cache was originally implemented as a data structure called a *hash table*; Figure 2-5 illustrates.

Figure 2-5. Fast Cache Organization



In the hash table, each IP prefix points to a particular place in the table. A particular hash table entry is found by computing some XOR operations on the upper and lower 16 bits of the 32-bit IP address being searched. The result of the calculation points to the desired hash table location, called a *hash bucket*. Each hash bucket contains a cache entry, including a pre-built MAC header for the next hop.

A hash calculation (or just *hash*) does not always produce a unique hash bucket address for every IP address. The case where more than one IP address points to the same hash location is called a *collision*. When collisions occur, IOS links each of the colliding cache entries together in a list within the bucket up to a maximum of six entries. This way, no more than six entries in the cache need to be searched to find a particular match.

In Cisco IOS Release 10.2, the hash table was replaced with another data structure called a *2-way radix tree* (a form of a binary tree). The MAC headers are still stored as part of the cache in this implementation.

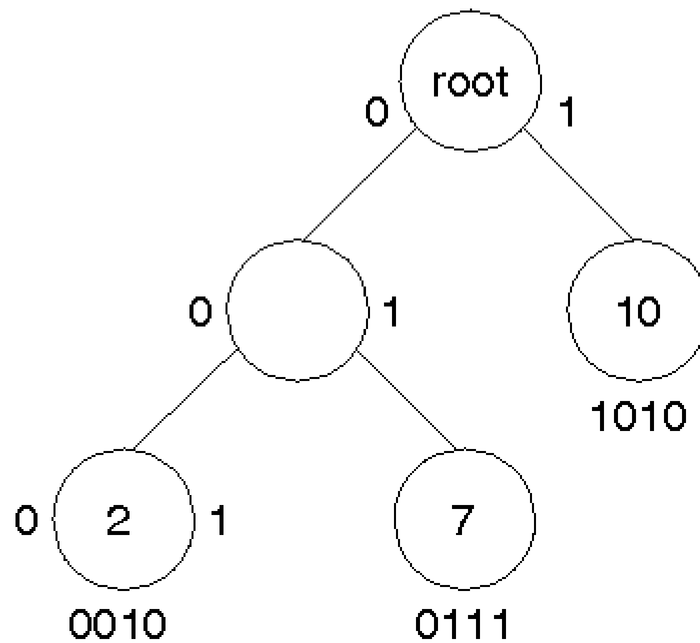
The radix tree, like the hash table, is another special data structure used to improve retrieval time of the member data.

A radix tree gets its name from the way in which data is stored—by its *radix*. In practice, this means the information is stored in a tree structure based on the binary representation of the key (the unique field that uniquely identifies each set of data). For example, to store the following numbers:

- 10, which is 1010 in binary
- 7, which is 0111 in binary
- 2, which is 0010 in binary

you could store them in a radix tree structure, as shown in Figure 2-6.

Figure 2-6. A Radix Tree



The tree branches are based on the binary digits within the number at any given level. To store, or find, the number 10, for example, you begin at the root with the first bit in 1010, which is 1.

This first 1 causes you to choose the right branch, which takes you to a node on the tree. You find this node doesn't have any children, so you compare the number in the node to the number you are searching for to see whether you have a match. In this case, there is a match.

To find or store 7, you begin at the root of the tree again. The first bit in the binary representation of the number 7 is a 0, so you take the left branch. You find yourself at a node with children, so you branch based on the second digit in the number, which is 1. This causes you to branch right.

As another example, let's work through finding or storing the number 2. The first binary digit in the number is 0, so you branch left. The second binary digit is 0 as well, so the second branch is to the left as well. At this point, you find yourself at a node with no children, so you compare the number the node represents with the number you are searching for, and find a match.

Fast Cache Limitations for IP Routing

There is one major limitation on the way the Fast Cache stores IP prefixes: Overlapping cache entries are not possible. For example, consider building cache entries for the following IP prefixes:

- 172.31.46.0/24
- 172.31.46.128/25
- 172.31.46.129/32

Because the Fast Cache doesn't use the subnet mask, or prefix length, when looking up routing information, there isn't any way to know that the entry for 172.31.46.129 uses a 32-bit prefix, and the 172.31.46.128 entry uses a 25-bit prefix.

A simple way to overcome this limitation is to build a cache entry for each destination host. This is impractical for several reasons, including the processing required to build this many cache entries and the amount of memory a cache containing all this information would consume.

So how does IOS solve this problem? It builds cache entries according to the following set of rules:

- If the destination is directly attached, cache with a 32-bit prefix length.
- If there are multiple equal-cost paths for this destination, cache with a 32-bit prefix length.
- If it's a supernet, cache using the prefix length of the supernet.
- If it's a major network with no subnets, cache using the prefix length of the major network.
- If it's a major network with subnets, cache using the longest length prefix in this major network.

Given the snippet of an IP routing table from a Cisco router in Example 2-4, you could determine what prefix lengths the router would use for various destinations.

Example 2-4. Determining Destination Prefix Lengths Used by a Router

router#**show ip route** …. O 172.31.0.0 [110/11] via 172.25.10.210, 2d01h, Ethernet0 [110/11] via 172.25.10.215, 2d01h, Ethernet0 172.16.0.0/16 is variably subnetted, 2 subnets, 2 masks D EX 172.16.180.0/25 [170/281600] via 172.25.10.210, 3d20h, Ethernet0 D EX 172.16.180.24/32 [170/281600] via 172.25.10.210, 3d20h, Ethernet0 O 10.0.0.0 [110/11] via 172.25.10.210, 2d01h, Ethernet0 O 192.168.0.0/16 [110/11] via 172.25.10.210, 2d18h, Ethernet0 172.25.0.0/24 is subnetted, 1 subnet C 172.25.10.0 [0/0] via connected, Ethernet0

The following list explains how the destination prefix lengths that appear in Example 2-4 will be cached:

- Every destination in the 172.31.0.0/16 network will be cached with a 32-bit prefix length because there are two equal-cost paths for this network installed in the routing table.
- Every destination in the 172.16.0.0/16 network will be cached with a 32-bit prefix length because there is a host route within this range.
- Network 10.0.0.0/8 will have one cache entry, because it is a major network route with no subnets, host routes, equal-cost paths, and so on.
- 192.168.0.0/16 will have one cache entry because it is a supernet route with no subnets.
- All destinations within the 172.25.10.0/24 network will be cached using a 32-bit prefix because this destination is directly connected to the router.

Maintaining the Cache

Any time data is cached, it's important to somehow maintain the cached data so it doesn't become outdated or out of sync with the master data used to build the cache in the first place. There are two issues to address in this area: cache invalidation and cache aging.

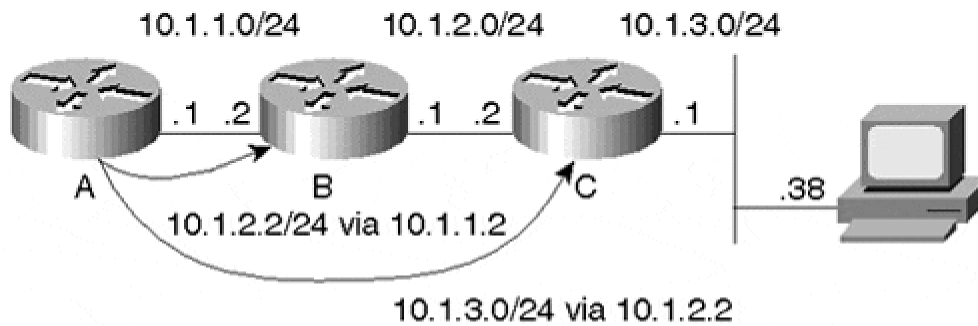
In the case of fast switching, the issue is how to keep the Fast Cache updated so it contains the same information as the routing table and the ARP cache (or other tables from which the MAC headers are built).

Cache Invalidation

The problem of keeping the route cache synchronized with the routing table is complicated when switching IP packets because of recursion or dependencies within the routing table.

Because recursion is such an important concept when considering the operation of route caches, it's helpful to walk through an example of recursive routing. Figure 2-7 provides an example of what recursion looks like in a network.

Figure 2-7. A Recursive Route



In Figure 2-7, Router A needs to look up a route for the host 10.1.3.38 and find out what next hop and MAC header to use. When Router A examines its routing table, it finds this destination is reachable through 10.1.2.2.

This destination isn't directly attached to Router A, so it needs to look into the routing table again to determine how to reach the next hop. Router A looks up the route to 10.1.2.2, and discovers it is reachable through 10.1.1.2, which is directly connected.

Router A will send all traffic destined to 10.1.3.38 to 10.1.1.2 for further processing.

In fast switching, recursion is resolved when the cache entry is built rather than when the packet is switched. So, the route cache contains the MAC header and outbound interface of the actual next hop toward the destination being cached. This disconnects the cache entry from both the routing table entry and the ARP cache (or other MAC-layer table) entry.

Because recursion is resolved when a cache entry is built, there is no direct correlation between the Fast Cache and the routing table or the ARP cache. So, how can the cache be kept synchronized with the data in the original tables? The best way to solve this problem is to just invalidate, or remove, cache entries when any corresponding data changes in the master tables.

Cache entries can be removed from the Fast Cache, for example, because:

- The ARP cache entry for the next hop changes, is removed, or times out.
- The routing table entry for the prefix changes or is removed.
- The routing table entry for the next hop to this destination changes.

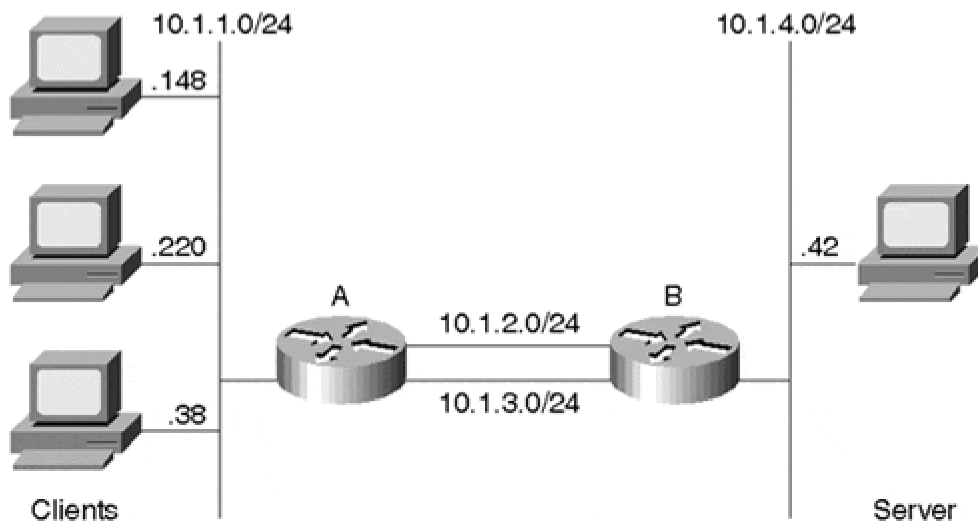
Cache Aging

IOS also ages entries in the Fast Cache periodically. A small portion of the Fast Cache is invalidated every minute to make certain the cache doesn't grow out of bounds, and to resynchronize entries with the routing and ARP tables periodically. When the amount of available memory is greater than 200 KB, the *cache ager* process randomly invalidates 1/20th of the cache entries every minute. If free memory is lower than 200 KB, the cache ager process starts aging entries more aggressively—1/5th of the cache entries are invalidated every minute.

Traffic Load Sharing Considerations with Fast Switching

Unlike process switching, fast switching does not support load sharing on a per-packet basis. This restriction can lead to uneven link utilization in some situations where multiple paths exist between two destinations. The reason for this restriction has to do with the separation of routing and forwarding that occurs when fast switching. To understand this reason and the possible disadvantages, consider the example illustrated in Figure 2-8.

Figure 2-8. Load Sharing Using the Fast Cache



In Figure 2-8, several workstations are connected to a network segment attached to Router A. Each workstation communicates with a single server on another network segment attached to Router B. Router A has multiple parallel paths to Router B. Assume both parallel paths have the same cost in this example. You would like the traffic between the client and the server networks to be evenly distributed over those two paths. Now, check out what happens.

Starting with an empty Fast Cache, Router A receives a packet from client 10.1.1.220, destined toward the server, 10.1.4.42. As you've seen, this first packet will be process switched and Router A will build a Fast Cache entry for the 10.1.4.42 destination. Because there are two equal-cost paths to the 10.1.4.42 destination (via Router B), Router A will have to choose one of these when it builds the cache entry. It will use the per-packet, load-balancing algorithm described earlier for process switching to make the decision.

Next, Router A receives another client packet destined to server 10.1.4.42. This time the packet is fast switched because there is already an entry in the Fast Cache. Because the transmit interface pointer is built in to the cache entry, Router A switches this second packet through the same path as the first. As you can see, Router A will continue to send all packets destined for the 10.1.4.42 server through this same path until the cache entry either ages out or is invalidated for some reason. If the cache entry does happen to get removed, the other path might be chosen—but packets will still only be forwarded out one path for the 10.1.4.42 server.

If there happened to be multiple servers on the 10.1.4.0/24 network, this same process would apply to each of them. The path cached for each server might vary, but for any given server all packets coming from the clients would take the same path.

Now consider traffic originating from the other direction. Router B will also cache destinations on the client network in the same manner Router A did. In this case, Router B will build two cache entries along one path, and the third along the other path. It could turn out Router B will choose to send the traffic for two clients along the path not chosen by Router A. This would result in a fairly well distributed traffic load on the parallel paths. It's just as likely Router B will choose to cache two client entries along the same path Router A chose for the server traffic, however, resulting in unbalanced utilization of the paths.

This lack of a deterministic load-balancing scheme is an area of concern for many network designers. In response, newer switching methods now support load-balancing schemes that help overcome this problem. One such method, Cisco Express Forwarding (CEF), is covered later in this chapter.

Index terms contained in this section

2-way radix trees
    Fast Cache 2nd
aging cache entries
    Fast Cache
cache
    Fast Cache 2nd 3rd
        cache maintenance 2nd 3rd
        hash buckets 2nd
        hash tables 2nd
        load sharing 2nd
        radix trees 2nd
        storing IP prefixes 2nd
cache misses
commands
    show ip cache verbose 2nd
Fast Cache 2nd 3rd
    cache maintenance 2nd 3rd
    hash buckets 2nd
    hash tables 2nd
    load sharing 2nd
    radix trees 2nd
    storing IP prefixes 2nd
Fast switching, see Fast Cache
hash buckets
    Fast Cache 2nd
hash tables
    Fast Cache 2nd
I/O memory
    Fast Cache
invalidating cache entries
    Fast Cache
IP prefixes
    Fast Cache 2nd
load sharing
    Fast Cache packet switching 2nd
misses, cache
output
    show ip cache verbose command 2nd
packet switching
    Fast Cache 2nd 3rd
        cache maintenance 2nd 3rd
        hash buckets 2nd
        hash tables 2nd
        load sharing 2nd
        radix trees 2nd
        storing IP prefixes 2nd
process switching
    populating cache
radix trees
    Fast Cache 2nd
recursive routing
routing
    recursive routing

sharing loads
Fast Cache packet switching 2nd
show ip cache verbose command 2nd
speed
Fast Cache 2nd 3rd
cache maintenance 2nd 3rd
hash buckets 2nd
hash tables 2nd
load sharing 2nd
radix trees 2nd
storing IP prefixes 2nd
switching packets
Fast Cache 2nd 3rd
cache maintenance 2nd 3rd
hash buckets 2nd
hash tables 2nd
load sharing 2nd
radix trees 2nd
storing IP prefixes 2nd
tables
hash tables
Fast Cache 2nd
thrashing, cache
trees
radix trees
Fast Cache 2nd

About Us | Advertise On InformIT | Contact Us | Legal Notice | Privacy Policy

© 2001 Pearson Education, Inc. InformIT Division. All rights reserved. 201 West 103rd Street, Indianapolis, IN 46290