how it works

TODAY'S CORE ROUTERS NOT ONLY HAVE TO MONITOR AND FORWARD TRAFFIC FROM THOUSANDS OF FLOWS, BUT ALSO MUST DETERMINE WHERE TO FORWARD THE DATA PACKETS TO THOUSANDS OF POSSIBLE DESTINATIONS. AND, AT OC-48 RATES, THEY HAVE TO PROCESS THAT DATA IN LESS THAN 65 NSEC. PIECE OF CAKE.

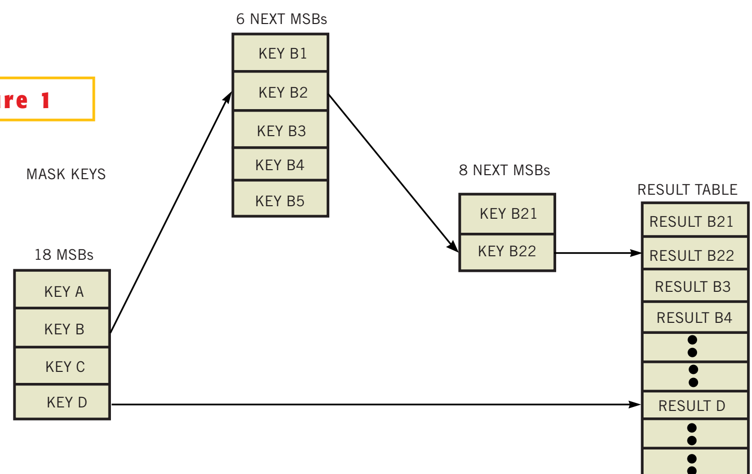
Recipe for success: how fast table look-up makes high-speed communications possible

By Nicholas Cravotta, Technical Editor

IT TAKES A LEAP OF faith to accept without question that packet-forwarding engines can determine the destination port of a packet in just a few clock cycles. After all, an OC-48 router needs to extract the appropriate destination information from a packet, classify and prioritize the packet in relation to other packets, shape traffic at a system level, and forward the packet through the correct egress port. Even with a pipelined architecture, look-ups must be completed within 65 nsec; otherwise, the packets will begin to overrun each other. At OC-192 rates, this threshold drops to 16 nsec. OC-768 offers a scant 4 nsec. (These numbers represent the worst case of a device's receiving 40-byte acknowledgment packets back-to-back indefinitely. For information on packet size versus probability of occurrence, visit moat.nlanr.net or www.vbns.net.)

Depending on the number of functions an application offers, an intelligent router might use tables based on egress port, VLAN (virtual-LAN) connection (Layer 2), MAC (media-access-controller) base address (Layer 2), IP address (Layer 3), VPN (virtual private network (Layer 3), tunnel (Layer 3), QoS (quality of service, Layer 4), or CoS (class of service, Layer 4). It also might use tables based on flow (which means different things to IP and ATM), policy, or URL look-up (Layer 7). Additionally, a packet might require more than one look-up before it can be sent on its way: The IP address of a packet (look-up 1) might reveal that the packet must be encrypted for a VPN (look-up 2) with a certain level of QoS (look-up 3). A router probably won't need to use every one of these tables for every packet, but a router might need to finish several look-ups in that 65-nsec window.

Figure 1



Indexed look-ups separate entries into layered tables based on most significant bits. In this example, several keys share the first 18 bits of Hash Key B, so the next 6 bits resolve the duplication. Several keys share the bits of Hash Key B2, so the next 8 bits resolve the key to locate associated data B22. If the first 18 MSB bits match a unique entry (Key D), the result is the associated data (Result D) without additional look-ups.

Work begins on a packet as the first bits arrive at an ingress port. Using an extraction key, the packet engine masks important fields in the packet, such as destination and CoS. As it extracts each field, the engine can begin appropriate look-ups, even before it receives the complete packet. Thus, by the time it has received the complete packet, the engine could have extracted and possibly already have looked up the appropriate fields. If a packet has an error, the integrity of the extracted fields is questionable, and the packet-management engine must take over to handle discarding or recovering the packet. Note that even an ASIC engine needs to be configurable, say with a rules table, to be able to handle the variety of possible header combinations and to be able to parse new services as they emerge.

The two types of table look-up are "exact" match and "best" match. Once the look-up engine selects a match, the "result" points to a "packet handle," which describes how to process the packet, including such information as transmit priority, channel, and mode. One basic way of organizing data is to store it in a list. To find an entry, you could start at the top of the list and sequence through the list until you find a match. Depending upon where your data falls in the list determines how long it will take to find an entry.

A simple way to speed look-up in the list is to sort the data. Now, you can perform a binary search to find the data within N steps where 2^N is the closest value greater than the size of the list. Today's routers may have to search through more than 64,000 entries for destinations or 512,000 flow designators. Using a binary search would require 16 steps in a worst-case scenario, and each step would involve a comparison and conditional branch. Such a scheme would still require too much time for today's high-speed applications. A better approach would be to use algorithms that more quickly converge on the best match.

One alternative is to use CAM (content-addressable memory). In a system using conventional memory, the memory stores the table entries, and a global comparator finds the best match as described above. CAMs, on the other hand, have a comparator for each bit of each entry. For example, in a table with 32-bit-wide entries, the CAM simultaneously compares all 32 bits to the search value, yielding 32 column values. Each entry also has a match line, which is effectively the composite AND of the 32 column values (column 0 AND column 1 AND column 2...AND column 31). If all 32 columns match the search value, the match-line value equals 1. Every entry in the table has a match line, and every match line goes into a PE (priority encoder). The PE then picks the active match line closest to the front of the table. (However, some PEs arbitrarily use the back of the table.) Note that the CAM may find more than one match.

510558891X
510558XXXX
510XXXXXXX

Figure 2

When using longest prefix matching on a CAM, table order defines priority. Entries hold as many bits as necessary to differentiate entries from each other. This table routes all phone numbers for the 510 area code. Numbers starting 510558 route differently from all other 510 numbers, and 510558891 numbers also route differently. For any given number, the longest match (closest to the front of the table) yields the correct destination route for that number.

When a CAM receives a search value to look up, the CAM searches every entry simultaneously. If the value appears in the table, the CAM returns a pointer to the entry. The CAM can then use this pointer as an index to the associated data for that entry (for example, the egress port associated with the destination IP address). Thus, CAMs can quickly complete look-ups. However, because the comparators for every entry in the table are active during a look-up, CAM tends to consume significant amounts of power. Because CAM devices are so expensive, in both cost and power consumption, associated data is usually stored in less expensive conventional memory. Using pointers instead of the actual data result also keeps tables homogenous in size.

However, not all table look-ups create a sorted list. They may instead use *longest prefix matching*, for example, which may yield several matches and require further processing to determine which is the best match. In general, a trade-off exists between memory usage and the number of steps it takes to locate a table entry. You can increase the speed of a table look-up by using *hashing*, indexing, and table compression.

SLINGING HASH

One way of reducing table complexity and memory requirements is to hash entry keys. Conceptually, hashing shrinks or maps a lot information into a smaller bit of information. For example, an IP-address look-up might require a five-tuple (five-field) value representing the IP source/destination, TCP source/destination, and IP type. Such a value could run 100 bits or more. To build a table of values 100 bits wide requires significantly more memory than a table only 32 bits wide. Hashing is effective because of the sparseness or density of the table; that is, only a relatively small subset of the large set of potential values (2^{100}) is ever in use at one time. Thus, you can map the large set of potential values onto a smaller index set (2^{32}).

The hash can take many different forms, such as XORing the two 32-bit IP source and destination fields into a single 32-bit, followed by further hashing of the other fields into a 32-bit result. The hashing engine can yield the same key for many combinations of fields, so some mechanism must be in

place to prevent duplication of hashing keys. One way to accomplish this task is to tie the original fields to the result in the look-up table using layered index tables. When the look-up engine finds a match, the table generally points to an associated result or handle for how to treat a packet. However, if there is more than one entry tied to the hash key (that is, there are duplicated hash keys), the result instead points to another table to resolve the collision. A trade-off exists between table size and the number of layers of tables required, depending upon how you compress the tables.

For example, the first look-up may use only the first 18 bits of the hash key (**Figure 1**). In this approach, *prefix expansion*, the first table sorts by the most significant bits and then expands to work with the rest as necessary. Many hash keys share the first 18 bits, so the associated data for these keys point to another table that may resolve the next 6 bits. At this point, some keys may be uniquely represented, but others may require more resolution, requiring a third index look-up using the last 8 bits. Given that duplicate hash keys should not exist, a fourth index look-up would resolve duplicated hash keys by discarding the hash key and referring to the original fields that share the same hash. CAMs can then find the appropriate entry. Processing engines use algorithms to compute the position in the table based on the hash key or search value.

Although such indexing adds steps to the matching process, the reduced number of bits to look up in the first place is probably enough of a benefit to offset the extra latency. For example, a one-level look-up requires 32 bits of memory for every entry. Using our example, layered indexes require only 18 bits for those entries that can be resolved in 18 bits, an extra 8 bits (plus overhead for the pointer into the second index) for each entry that resolves at the second layer and so on, resulting in less memory overall for storing values. However, given the variable size of each table layer, the look-up engine needs to be flexible in how many bits it can resolve at a time.

Figure 3

The radix tree looks like an upside-down tree. Branches represent various entries but only to the point at which the branch becomes unique (analogous to the longest prefix). For example, A represents entry 11101, B represents 101xx, and C represents 0011x. To speed look-up using a radix tree, you can compress several levels into one look-up step.

THE CLUMPS

An important aspect of sparseness requires that you spread data evenly throughout a table to avoid having large clumps of data. Thus, you can more easily manage tables. As connections come and go, so do entries in a table. When you change a table, however, you can't simultaneously use it for table look-up. Therefore, the more drastic a change to a table, the longer you can't use the table for look-ups. Additionally, you may need to change the rules of a table, which might change the structure of the table. You need to be able to dynamically update the table without shutting down the device for look-up.

For example, if you want to add an entry to the middle of a large table, you have to shift every entry past the entry where you want to add the new entry to make space for it. To avoid this massive overhaul of a table, it is general practice to leave spaces in the table, so you can add entries with little overall impact on the table. The challenge, then, is guessing how many extra entries an index table or a section of a table will need over time and where to put them. When data clumps, however, these empty entries disappear quickly. Finally, when an entry needs a place and no empty entries are available, you must adjust the table. In a worst-case scenario, you would have to adjust all 64,000 entries in the table, which is simply not feasible.

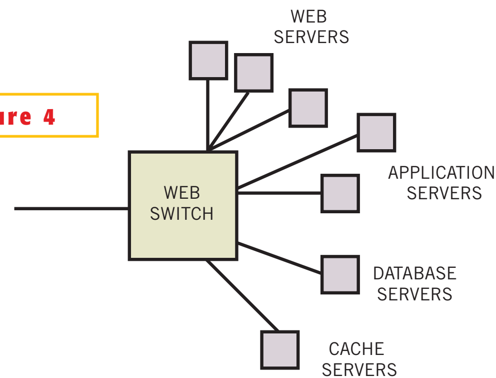
One method of minimizing table-management delays is to keep tables small by layering them. Therefore, you may need to change only a small subset of a table. In general, the more layers, the less significant the table adjustments. This advantage comes at the cost of an extra look-up step and the associated latency for each layer. One difficulty to keep in mind is that tables may spread across several chips, such as cascaded CAMs or several memory banks, and the table may not appear contiguous between the chips. Table maintenance becomes more challenging because the RISC or CPU updating the table has to monitor which device holds which parts of the table. Additionally, table adjustments across chips can be tricky. Testing how effective a management scheme is like slow-cooking a stew: It takes awhile to know if you did it right.

The most robust method of table management is to manage two complete tables with the same memory image (shadow, virtual, or dual tables). One image is in the foreground and performs the actual look-ups. When a new connection arrives, the table-management engine adds the entry to the background image and then swaps the foreground with

the background. The table-management engine then adjusts entries in the background image to "declump" the data and make room for future entries without affecting real-time processing of packets. Thus, you can fix the table for clumping problems before rather than after clumping becomes a problem. Although this scheme offers absolute coherency (that is, table management at wire rate), it takes additional memory (read extra cost, power, and real estate). Note that the required memory is most likely not double because you don't have to leave as many empty entries as you might with a single table.

It's also important to note that memory access plays a crucial role in efficient table look-up. Given that memory access is a serious bottleneck at these speeds, look-up algorithms have focused on using the fewest memory accesses at the expense of processing time and cost (that is, using of large amounts of memory). If memory accesses are long enough, it may make sense to use *concurrent execution*, in which the look-up engine switches context and works on another packet while it waits. Note that even CAMs require additional memory to store complex results; given the cost of CAM, it is more efficient to store a short pointer to a longer result stored in less expensive memory technologies.

Figure 4



Web switches enable servers to hold distinct parts of a site, instead of requiring each server to mirror the entire site. Such switches, however, need to be able to look inside the data payload to determine how to route the packet.

LONGEST PREFIX MATCHING

As noted above, a CAM or processing engine can flag multiple matches during a table look-up. In an exact-match table, multiple matches indicate that a catastrophic error has occurred and that the table is corrupt or invalid.

However, a table need not necessarily list every possible combination uniquely. One look-up scheme uses *longest prefix matching*, similar to the concept of prefix expansion except that there is only one layer of table. By analogy, consider classifying phone numbers. It might be that every phone call to the 510 area code leaves through the same egress port. In this case, if a phone number matches the 510 prefix, the call goes through this port. Thus, one entry (510xxxxxxx, where x is a "don't-care" digit) represents many possible entries and potentially achieves a high compression ratio for the table. If a new phone number, say 5105588906, comes through that uses a different egress port, a new entry joins the table. The entry represents the longest prefix necessary to differentiate among entries. In this case, if there have been no calls to 558, the new entry might be 510558xxxx. Say that you add an entry, 5105588914, that goes to a port different from all the other 510558 entries. You would then add an entry, 510558891x, to the table with the new egress port. Notice that the table stores the number only as far as the first difference; this prefix is the longest one necessary to differentiate the number from all other numbers already in the table.

Now when you look up 5105588914, you get three CAM matches (**Figure 2**). Because the table lists the longest prefix match first, you would use the first entry, 510558891x. Looking up 5105588906 yields two matches, and 510558xxxx is tied to the correct egress port. As you can see, the longest match is the "best" match. Note that longest prefix match works just as well for QoS and other types of look-up. One of the key advantages of this approach is that it requires less memory to comprehensively represent a table because an entry can actually represent a subset of entries.

Similar to longest prefix matching but using a different model is the *radix tree*, which looks like an upside-down tree (**Figure 3**). At each node, you go left if the current bit is 0 and right if it is 1. Note that the radix tree is similar to the longest prefix matching in that you have to plot a branch only until it represents a unique table entry. Obviously, you need to compress the tree; otherwise, a 32-bit key could require 32 look-ups. One way of compressing the tree is to "walk down" it multiple levels at a time; for example, you could use 32 bits to represent four layers of the tree as one step. Thus, the worst-case look-up would be eight rather than 32 steps.

One challenge with longest prefix matching is table management. Because the table order represents priority, you may have to move entries so that longest prefixes appear before associated shorter prefixes. To avoid the delay that these adjustments would cause, the table must provide sufficient empty entries to accommodate new prefixes as they become necessary. Thus, whereas a table may have room for 64,000 entries, the actual number of entries it uses is significantly smaller. Again, the tradeoff is between memory usage and processing efficiency.

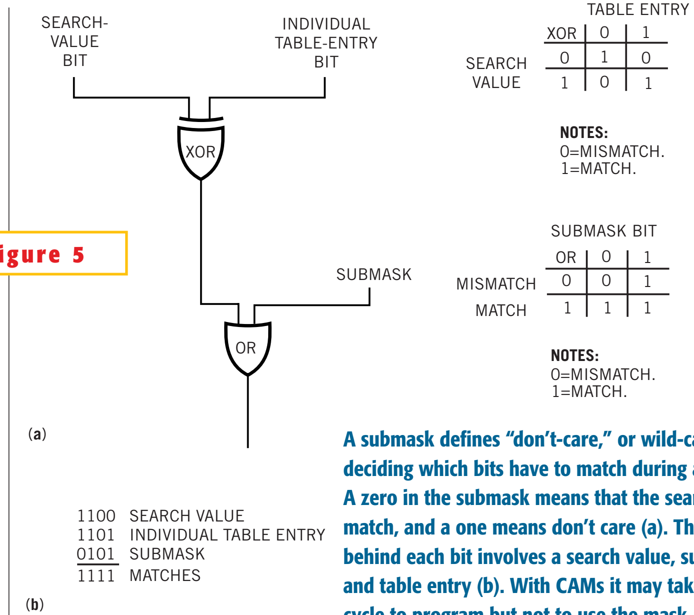
Traditional routers worked only with information

from layers 2 and 3. Today's routers, however, may look as deep into a packet as L7 (Layer 7). An example of an L7 look-up is routing a packet based on a URL or cookie Why might you route a packet based on a URL? Web switches, for example, can improve server efficiency by enabling servers to support segments of a Web site instead of the entire site (**Figure 4**). This feature is possible because the Web switch can look at the requested URL and determine which Web (or cache) server has that information. Other applications include intrusion detection, in which the Web switch tracks suspicious sequences of ports that might signal an attack on the network, and filtering, in which the switch identifies packets using key words, such as "bomb" or "terrorism."

Extracting the URL field can be difficult for several reasons. First, it resides past the header, which means it takes longer to collect and begin looking up. Additionally, some URLs are unanchored or not tied to a location in the packet. In this case, the engine has to find the URL, based on context, the protocol type in use, various tokens, and non-case-sensitive variations (for example, hOst=host). Then, because the URL's length varies, the engine has to extract a maximum length. These types of look-ups are more complex and require significantly more horsepower to execute than some of the simple IP-address look-ups.

Tables can also work both ways. For example, in an ARP (address-resolution-protocol) table, you use a MAC address to find an associated IP address. Sometimes, however, you might want a reverse ARP: You know the IP address but need the MAC address. Together the 48-bit MAC address and the 32-bit IP address comprise at least an 80-bit value. To use the table for ARP, you would use a submask that cares only about the first 48 bits, and the look-up result would point to the appropriate entry so that you could read off just the last 32 bits for the IP address. Conversely, for reverse ARP, you could set the submask to care only about the last 32 bits, resulting in a pointer to the appropriate entry in which the first 48 bits are the value you are looking for. CAMs that support such don't-care, or wild-card, bits in the submask are *ternary CAMs* (**Figure 5**).

Figure 5



A submask defines "don't-care," or wild-card, bits by deciding which bits have to match during a look-up. A zero in the submask means that the search bit must match, and a one means don't care (a). The logic behind each bit involves a search value, submask, and table entry (b). With CAMs it may take an extra cycle to program but not to use the mask.

THE PROOF IS IN THE PUDDING

Both CAMs and processing engines can provide efficient table look-up, depending on your application. CAM vendors claim that CAMs can do everything, and engine vendors point out CAM's deficiencies. Many CAM vendors are now altogether abandoning the term "CAM"; look under the hood of a "packet-forwarding engine," and you just might find a CAM in disguise.

CAMs are typically more expensive bit for bit, but CAMs let you use these bits in ways different from those of processing engine-based systems. Because vendors partition some CAMs into blocks, you can implement several table types on a device. However, you may find yourself with larger empty spaces than you'd like if your tables can't efficiently fill the blocks. CAMs can be as flexible as processing engines, and you need flexibility for processing layers 4 to 7, but this flexibility is useless unless you attach a processing engine to the CAM. A CAM alone cannot manage a longest-prefix-matching table. You need a network processor or equivalent device to handle the computational side of managing the table unless you find a CAM that integrates this function.

One issue of contention in table look-up is determinism. You must have some upper limit on the time it takes to complete a look-up, and you must address the issue of several worst-case look-ups occurring consecutively. CAMs offer a set determinism: a look-up always takes x cycles. Engines using algorithms, however, may complete a look-up over a range of times. The challenge with this variation of time is that table look-up is often a concurrent operation, meaning that the engine that initiated a table look-up can switch contexts and work on another packet while waiting for a result. If times for results to arrive can vary, context switching can become complex, meaning that an interrupt must signal that the look-up is complete, or wasteful, meaning that the initiating engine assumes worst-case times, and the look-up engine sits idle.

One common misconception about CAMs is that they can perform a look-up in a single cycle. Instead, CAMs are usually pipelined. Parts from Lara Networks, for example, have a four-stage pipeline. Additionally, if you cascade as many as eight engines, an extra cycle of latency occurs as the engines ne-

gotiate which of them has the best match. Cascade more than eight, and you add another cycle for a maximum of six cycles. The one-cycle-look-up claim comes from the fact that, once you primed the pipeline, look-ups still take four to six cycles, but you get a result every cycle. This result is possible only if the CAM can sustain back-to-back look-ups.

One important disadvantage of CAMs is their high power consumption. Because every access to the CAM compares every cell, every cell draws power. CAMs address this problem through several power-management strategies. For applications that store several tables in the same CAM, the CAM can save power by only activating those cells associated with a table. CAMs can associate cells with a table on a block level, so that only the blocks reserved for a table consume power. However, by nature of their structure, CAMs have many empty entries to leave room for new entries. These empty cells draw power just as filled cells do. Another strategy is to reduce the process technology and thus reduce overall power consumption. The challenge then transfers to supplying many 5, 3.3, 1.8, and 1.5V voltages on the same board.

You can reach Technical Editor Nicholas Cravotta at 1-510-558-8906, fax 1-510-558-8914, e-mail ednnick@pacbell.net.

How you partition the various blocks of a table-look-up engine affects performance. For example, if the blocks reside on different chips, you have to account for the increased latency across the interface. The more stages in the process, the longer the overall latency.

In addition to table look-up, an engine might also track statistical information about the packets it passes—for network management, say. The engine may track every packet or a sample of packets. Some of today's engines have 250,000-flow counters. Given the conceivable flood of data available, many applications simply dump the data onto a hard drive for later averaging and evaluation. It's easy to see, then, that an opportunity exists for statistical-management coprocessors.□

Acknowledgments

Special thanks to the folks at C-Port (a Motorola company, www.cport.com), Entridia (www.entridia.com), Lara Networks (www.laranetworks.com), SwitchOn (www.switchon-net.com), and Vitesse (www.vitesse.com) for technical assistance with this article.