Parallel Table Lookup for Next Generation Internet

Li-Che Hung and Yaw-Chung Chen
Department of Computer Science, National Chiao Tung University, Hsinchu, Taiwan
onlyjack@alumni.nctu.edu.tw, ycchen@cs.nctu.edu.tw

Abstract
The rapid growth of Internet population leads to the shortage of IP addresses. The next generation IP protocol, IPv6, which extends the IP address length from 32 bits to 128 bits, was proposed around mid-nineties to accommodate the problem. IP table lookup is based on the longest prefix matching. Most of the existing lookup algorithms scale poorly as IP address space expands to 128 bits. We propose a table lookup scheme for IPv6 using binary search among prefix-length based on parallel processing to improve the performance. We merge hash tables to reduce the lookup complexity of the worst case and use both pipeline and multi-threading to improve the throughput of the average case. Our scheme is realized based on parallel architecture. The simulation results show that the maximum throughput is one lookup per 100 clock cycles in average. This means that, with 600 MHz clock rate, our proposed scheme is able to accomplish 6 million lookups for IPv6 packets. It is more cost effective comparing with existing commercial products in terms of number of lookups per MHz.

1. Introduction
The rapid growth of Internet population leads to the shortage of IP addresses. The next generation internet protocol, IPv6, extends the IP address length from 32 bits to 128 bits. It was proposed around mid-nineties to solve the problem. Since Classless Inter-Domain Routing [3] is deployed to allow for arbitrary network aggregation, the packet forwarding process becomes complicated. When a router receives an IP packet, it performs the operation of Longest Prefix Matching (LPM) to find the longest prefix that matches the destination IP address, called best matching prefix (BMP). Then, the router forwards the IP packet to the output port associated with that BMP.

The link speed, router throughput, and packet forwarding rate are three key factors influencing the actual data rate of the Internet links. The nowadays high-speed fiber-optic link and switching technology are able to accommodate the first two factors. However, the packet forwarding rate is still a bottleneck because the processing of LPM is complicated.

Many fast lookup schemes [6] have been proposed, but these mostly focus on IPv4. Their performance degrades quickly while being scaled up to deal with 128-bit IPv6 addresses. Three scalable lookup schemes [1] [4] [5] have been proposed, but they focus on the processing of IPv4 primarily. We propose a lookup scheme for IPv6 based on [1] with modification, and use parallel processing to improve the lookup performance of IPv6. In our proposed scheme, the maximum throughput for a lookup is about 100 cycles in average, which turns out to be 6 million lookups per second. This means that our scheme is able to accomplish higher than 10G forwarding rate if 256 byte average packet length is assumed.

The rest of this paper is organized as follows. Section 2 briefs the background of the work. The proposed scheme is discussed in Section 3. We address the implementation and show the experiment results in Section 4. Finally, the conclusion and future works are presented in Section 5.

2. Background
2.1. Table lookup schemes for IPv4
2.1.1. Path-compressed trie. A path-compressed trie was originally proposed in [9], but it doesn't support longest prefix matching. Sklower proposed a scheme with modifications for longest prefix matching in [10]. A path-compressed trie is similar to a binary trie, but it removes one-way branch nodes by collapsing them. Figure 1 shows an example of the path-compressed trie. The prefix in a node represents the best matching prefix with this node. The number beside a node represents the bit position in the destination IP address. We check whether the bit in that bit position of the destination IP address is 0 or 1 to decide the branching direction. The search process is as follows. We inspect the bit position of the destination IP address indicated by the number beside the node to decide the branching direction. If the node contains a prefix, we need to compare it with the destination IP address. We record the matched prefix as the BMP so far and traverse the trie until a leaf is encountered.

![Figure 1. A path-compressed trie](image-url)
2.1.2. Controlled prefix expansion. Srinivasan [7] presented a data structure based on multibit trie. The idea is to convert the set of arbitrary-length prefixes to the set of predefined-length prefixes by using a so-called “controlled prefix expansion” technique. Figure 2 shows an example of the original prefixes and the expanded prefixes. This can be done through dynamic programming, which is time consuming. As Figure 3 shows, the 1-bit trie has been divided into three levels, and the expanded trie only has maximum path length of 2 compared to the 1-bit trie that has maximum path length of 7. Thus the search time can be reduced significantly, and the memory requirement is also smaller than that of 1-bit trie. By using the standard trie representation with arrays of children pointers, insertions and deletions can be supported in the scheme.

2.1.3. Variants of multibit trie. The basic scheme of Gupta [11] uses a two-level multibit trie with fixed strides similar to the one in Figure 3. The first level corresponds to a stride of 24 bits and the second level to a stride of 8 bits. So we at most take two memory accesses to find the BMP. Nilsson [12] recursively transform a binary trie with prefixes into a multibit trie, they replace a nearly full binary subtrie with a multibit subtrie of stride $k$ if the nearly full binary subtrie has a sufficient fraction of $2^k$ nodes at level $k$, where sufficient fraction is defined using a so-called fill factor $x$, with $0 < x \leq 1$.

2.2. Scalable table lookup schemes

2.2.1. Multiway and multicolumn search. By encoding a prefix as the starting point and the end point of a range and pre-computing the best matching prefix associated with a range, the scheme proposed in [4] does a binary search in a sorted array for the longest prefix matching problem. They also use an initially pre-computed 16-bit array to reduce the number of required memory accesses. The multicolumn search exploits the fact that most processors pre-fetch an entire cache line when doing a memory access. By using six-way branching search, the worst case is five 32-byte cache lines filled in a Pentium Pro. However, the insertion/deletion of prefixes may result in a table reconstruction due to the re-calculation of the pre-computed information.

2.2.2. Multiway range tree. The lookup algorithm in [5] is the improved one of that described in [4]. It has faster update speed by using address span. Same as the scheme in [4], it encodes each prefix as the start point and the end point of a range, and uses the B-tree to store these points. So it is called a multiway range tree. It defines the address span of a node in the multiway range tree as the range of addresses that can be reached through the node. Finally, we can find the smallest range covering the destination IP address by traversing the multiway range tree using the destination IP address as the search key. Figure 4 shows an example of a multiway range tree. The address spans of those nodes in the same level of the B-tree form a partition of the range of total addresses. When we want to insert or delete a prefix, we only need to modify the address spans of those nodes in the tree path of the prefix. The lookup complexity is $O(\log N)$, the space complexity is $O(kN \log N)$, and the update complexity is $O(kN \log N)$.

Our proposed scheme is based on the binary search among prefix lengths in [1], which primarily focuses on the processing of IPv4. We modify the lookup scheme of [1] to fit IPv6, and we design three techniques, to reduce the lookup time for the worst case, to improve the throughput for the average case, and to improve the lookup performance.

3. Proposed scheme

Figure 2. Controlled prefix expansion

Figure 3. Expanded trie for the database of Figure 2

Figure 4. An example of a multiway range tree
3.1. Binary search among prefix lengths

We address the lookup scheme of [1] here. For each possible prefix length, we use a corresponding hash table to store prefixes with that length, and use the notation Table\textsubscript{i} to represent the hash table of prefixes of length \textit{i}. Then we perform binary search among prefix lengths. According to the lookup result, we decide whether a hash table of shorter length or of longer length we need to do lookup next. An example of binary search tree for IPv4 is shown in Figure 5 in which each node in the binary search tree represents a hash table, and the number in each node of the binary search tree represents the corresponding prefix length of that hash table. For example, the root of the binary search tree in Figure 5 means a hash table of prefixes of length 16, i.e. Table\textsubscript{16}. In a binary search tree, the path from the root to a certain node represents a possible lookup order of hash tables, and we perform lookup in Table\textsubscript{16} first. Then, depending on the lookup result, we decide whether Table\textsubscript{16} or Table\textsubscript{24} is the next hash table to do lookup and so on.

One characteristic in the longest prefix matching is, if we know that a destination IP address matches a prefix of a certain length, we only need to look for those matching prefixes of longer length. But if the destination IP address does not match any prefix of a certain length, it doesn’t mean that we only need to look for those matching prefixes of shorter length. The author of [1] proposed a so-called marker to make binary search work accurately. A marker is an even shorter prefix of a prefix. For example, the prefix 10010* has four possible markers of different lengths, 1, 10, 100 and 1001. For a prefix, we don’t need all possible markers, but only pick those markers whose lengths have appeared in the lookup order. It means that we insert markers into those hash tables in the search path of binary search tree. The meaning of the marker is that we should have a matched prefix longer than this marker. With the feature of marker, we can guarantee that if an IP address matches nothing in a hash table of a certain length, then it won’t match anything in the hash tables of longer lengths. In other words, it only possibly has matching prefix of shorter length.

Figure 5. A binary search tree for IPv4

The other feature in [1] is that we record with a marker the best matching prefix (BMP) of that marker. It means that if the element in the hash table is a marker, the element has to record the information of the BMP of that marker. For example, if we have only two prefixes in the forwarding table, say, 1001011* and 10*. The former is in Table\textsubscript{7} and the latter is in Table\textsubscript{2}. For the prefix 1001011*, we need two markers, 1001 and 100101. Both of them have to record the information of the BMP, that is 10* for both markers. In order to avoid backtracking, the marker is recorded with BMP.

The lookup scheme in [1] is scalable with complexity O(\log{W}). Here W is the length of the IP address. In IPv4, we only need to perform lookup of 5 different hash tables in the worst case. Assuming that we have a perfect hash function, we only need to do lookup for each hash table only one time, so the total number of table lookups is 5.

3.2. Merging hash tables

We modify the lookup scheme described in Section 3.1 to improve the performance for the worst case. When dealing with IPv6, we have 128 hash tables instead of 32, so we need to perform lookup of 7 different hash tables in the worst case because the lookup complexity is O(\log{128}). The key idea of modification is merging two hash tables into one. Actually, we merge a hash table of prefixes with even number of bits, say 2\textit{n}, into another hash table of prefixes with odd number of bits, say 2\textit{n}+1. The hash tables of prefixes of odd length are the last one in all possible lookup order, so they do not have any marker. Consider the relation between the elements in Table\textsubscript{2n} and the elements in Table\textsubscript{2n+1}. Assuming we have either prefix P \cdot 0 or prefix P \cdot 1 in Table\textsubscript{2n+1} (P is a bit string of length 2\textit{n}, and P \cdot 0 means P followed by a bit 0, and the dot means the concatenation), we should have a marker P in Table\textsubscript{2n}. Note that we may have both P \cdot 0 and P \cdot 1 in Table\textsubscript{2n+1}. From the discussion above, we associate a marker P in Table\textsubscript{2n} with P \cdot 0 and P \cdot 1. Figure 6 shows the concept.

In Figure 6, Table\textsubscript{2n+1} denotes the hash table after merging Table\textsubscript{2n} and Table\textsubscript{2n+1}. And in Table\textsubscript{2n+1}, the subscript \textit{m/p} of \textit{P} denotes that \textit{P} is either a marker or a prefix. \textit{P} is associated with two prefixes, P \cdot 0 and P \cdot 1, and we use arrows to represent
the associations. So \( P \) has two pointers, one points to 0, represented by \( P \cdot 0 \); and the other points to 1, represented by \( P \cdot 1 \). The subscript \( p \) of 0 denotes that 0 is a prefix, and the subscript \( p \) of 1 denotes that 1 is a prefix too. By merging two hash tables into one, the total number of distinct hash tables is reduced from 128 to 64. Now, we only need to lookup 6 instead of 7 different hash tables in the worst case.

### 3.2.1. Data Structure

Figure 7 and 8 illustrate the data structure of the node in the hash table. The \( \text{length} \) field indicates the length of this prefix (or this marker). The \( \text{point} \) field records the memory location of the next node that has the same hash value. The \( \text{prefix} \) field records this prefix (or this marker). The \( P \cdot 0 \) field records the information of the output port associated with the prefix prefix-0. The \( P \cdot 1 \) field records the information of the output port associated with the prefix prefix-1. The \( \text{BMP} \cdot \text{length} \) field records the length of the BMP of this marker. Note that the BMP of a prefix is the prefix itself. Whether this node is a marker or not, \( \text{BMP} \cdot \text{length} \) equals to \( \text{length} \) if this node is a prefix. The \( \text{BMP} \cdot \text{port} \) field records the output port associated with the BMP of this marker (or this prefix). The \( \text{flag} \) field of the hash node includes all flags we used. We illustrate those flags in Figure 8. Details can be found in [13].

### 3.2.2. Lookup algorithm

In addition to merging the hash tables and modifying the data structure of the node, we also need to modify the lookup algorithm. We perform lookup in some hash table, say Table\( _{2^i, 2^{i+1}} \), and retrieve the first \( 2i \) bits of the destination IP address as the hash key to calculate the hash value, then we use the hash value as the index of Table\( _{2^i, 2^{i+1}} \). We check whether the hash node of index in Table\( _{2^i, 2^{i+1}} \) is null, if so, it will be failing to get matched in Table\( _{2^i, 2^{i+1}} \) and we should look for those matching prefixes shorter than \( 2i \). If the IP address of the destination get matched in Table\( _{2^i, 2^{i+1}} \), we record the \( \text{BMP} \cdot \text{port} \) field of that node as the BMP so far. Now, we know the BMP whose length is equal or shorter than \( 2i \), and we try whether we can match one more bit. If the \( 2i+1 \)-th bit of the destination IP address is 0 and the zero flag is set, we record the \( P \cdot 0 \) field as the BMP so far. If the \( 2i+1 \)-th bit of the destination IP address is 1 and the one flag is set, we record the \( P \cdot 1 \) field as the BMP so far. Now, we know the BMP whose length is equal to or shorter than \( 2i+1 \). Finally, we check whether the marker is set or not and decide whether we need to look for those matching prefixes longer than what we have found so far. Note that we skip the issue of hash collision because we assume a perfect hash function. Actually, we use the conventional method of chaining to resolve the problem of hash collision, and we address the details in Section 4.2. Figure 9 shows the pseudo code of the enhanced lookup algorithm.

### 3.3. Making lookup algorithm pipelined

We apply the pipeline technique to the lookup algorithm for improving the lookup performance of the average case. After merging hash tables, the binary search tree is changed. Figure 10 shows a binary search tree for IPv6 without merging hash tables, while Figure 11 shows the modified binary search tree for IPv6 after merging hash tables. In Figure 11, each node is a hash table containing the information of prefixes of two consecutive lengths. The modified binary search tree has 6 levels. Based on the lookup algorithm in Figure 9, the lookup algorithm will be executed 6 times at most. We perform lookup in the hash table in different levels of the binary search tree for all iterations of the loop. If we have 6 processing units, we can assign each one to do lookup of the hash table in one level, so the pipeline has 6 stages. When a processing unit has finished a lookup operation, the lookup result will be passed to the next processing unit. The next processing unit then uses the received
results to decide what to do. The lookup results include the BMP so far, the hash table to be searched next, and the skip flag for the next processing unit regarding the BMP that has been found, so the next processing unit does not have to do anything. By making the lookup algorithm pipelined, we can get one complete lookup result per processing cycle of a stage, which includes three parts. The first is to use the destination IP address as the key to do hash, the second is to lookup the hash table using the hash value as the index, and the third is to do computation according to the lookup result. Note that using the pipeline technique, we increase the throughput of IP lookup, but we don’t reduce the lookup time for each destination IP address.

3.4. Using multi-threading in the pipeline stage

We use the technique of multi-threading to further improve the lookup performance for the average case. Based on the processing cycles, when a processing unit does a lookup on the hash table, it needs to access the memory and waits the memory access to finish. In the waiting period, the processing unit is idle, so we can use a thread to do IP lookup. When a thread is waiting, it swaps out, so the next thread can utilize the computing resource to do hash for another destination IP address. Suppose we have 8 threads, the executing order of these 8 threads is like that in Figure 12. By using the technique of multi-threading, we can save some latency caused by the memory access. Same as the pipeline technique, we only increase the throughput of IP lookup, but we still don’t reduce the lookup time for each destination IP address.

4. Implementation and performance evaluation

4.1. Implementation platform

We implement our proposed scheme on Intel IXP2400 network processor [2]. The parallel processing architecture of IXP2400 is used to realize the design of pipeline and multi-threading. IXP2400 has one XScale core processor and eight co-processors called microengines. We can run the ordinary embedded operating system on the core, and run the applications on the operating system. The microengine is different from the core, it doesn’t have the hardware assistance for a stack. So we only use macros instead of functions in the programs running on the microengine because the ability of the microengine is restricted. Neither too big nor too complicated program is allowed to run on the microengine. However, the microengine supports multi-threading with maximum of 8 threads. The overhead of context switching between two threads is zero because each thread has its own resources such as registers and program counter. The microengine has an instruction set specifically tuned for processing the network data, such as the CRC unit doing cyclic redundancy check. So we can utilize these special features to speed up the processing.
Due to the differences between the micro engines and the core, we use the micro engines to handle basic packet processing tasks such as IP lookup, while the core acts as a control unit for managing the micro engines and handling the processing tasks of exceptions caused by the network data. In our experiment, we use 6 micro engines to implement our design of pipeline, and run 8 threads on each micro engine for realizing the design of multi-threading.

4.2. Implementation briefs

4.2.1. Simultaneous memory accesses. IXP2400 has three separate memories: DRAM, channel 1 of SRAM and channel 2 of SRAM. We need to know the sizes and the latencies of these memories because we want to distribute the hash tables in our proposed scheme to the three separate memories, so that we can alleviate the bottleneck of accessing only one memory. Table 1 shows the maximum sizes of three separate memories:

We use a development toolkit called IXA SDK 4.1, which provides a complete IDE including an editor, a compiler, an assembler, a simulator and a debugger. It also provides libraries for developers. The simulator in IXA SDK 4.1 can simulate the environment of IXP2400. We use the simulator in IXA SDK 4.1 to run some test programs and observe the simulation results. Table 2 shows the average latencies of reading eight 4-byte words from SRAM and DRAM respectively. We didn’t distinguish channel 1 of SRAM from channel 2 of SRAM in Table 2 because they are same kind of memory. Note that the results in Table 2 are in the circumstance of only one micro engine trying to access the memories. According to Table 2, we can know that the average latency of SRAM is not much different from that of DRAM. Intuitively, we can have three simultaneous memory accesses if three different micro engines access three separate memories independently. The simulation indeed shows the same result of memory latency as that in Table 2 when three different micro engines access three separate memories. Figure 13 shows the average latencies of reading 8 words from a certain channel of SRAM when different numbers of micro engines try to contend for accessing that channel of SRAM. We don’t need to distinguish channel 1 from channel 2 of SRAM in Figure 13. Figure 14 shows the average latencies of reading 8 words from DRAM when different numbers of micro engines try to contend for accessing DRAM. The number of simultaneous memory accesses without increasing latency is more than what we expected, three. As a result, we can allow 8 simultaneous SRAM accesses (4 from each channel) and 3 simultaneous DRAM accesses without increasing the average memory latency.

Those distinct hash tables are located in three separate memories. Hash tables accessed by processing unit (PU) 1 and PU 2 are put in channel 1 of SRAM, and those accessed by PU 3 and PU 4 are put in channel 2 of SRAM. Hash tables accessed by PU 5 and PU 6 are put in DRAM. A PU means a micro engine. Those hash tables accessed by assigned processing units are in the corresponding level of the binary search tree, as shown in Figure 15.

<table>
<thead>
<tr>
<th>Table 1. Maximum size of three separate memories</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
</tr>
<tr>
<td>DRAM</td>
</tr>
<tr>
<td>Channel 1 of SRAM</td>
</tr>
<tr>
<td>Channel 2 of SRAM</td>
</tr>
</tbody>
</table>

Figure 13. Memory latency of SRAM
Table 2. Access latencies of SRAM and DRAM

<table>
<thead>
<tr>
<th>Memory</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRAM</td>
<td>137 cycles</td>
</tr>
<tr>
<td>SRAM</td>
<td>117 cycles</td>
</tr>
</tbody>
</table>

Figure 14. Memory latency of DRAM

4.2.2. Transferring multiple words from memory. Each time we access SRAM or DRAM, we can fetch up to 16 words instead of one. Figure 16 and Figure 17 show the average access latencies of SRAM and DRAM respectively when we read different numbers of words. As we read more than one word from either SRAM or DRAM, the memory latency increases only a little bit. We can utilize this characteristic to alleviate the penalty of hash collision. Actually, we do not have a perfect hash function, so we use CRC32 built in the micro engine to handle the hash operation, and use chaining to resolve the hash collision. Figure 18 shows the concept of chaining. In Figure 18, Table(2i, 2i+1) is a hash table. We use a chain to link four prefixes $P_1$, $P_2$, $P_3$ and $P_4$ that have the same hash value. If a destination IP address matches $P_4$, it will need to perform lookup 4 times in Table(2i, 2i+1). This means that we traverse the chain and find that $P_4$ is a matching prefix. So we need 4 memory accesses in this case. If we put $P_2$ and $P_3$ into a contiguous memory location, then when we read the node of $P_2$, we can read the node of $P_3$ together by utilizing the characteristic of transferring multiple words. Figure 19 shows the concept. Each time we encounter a hash collision, we read two contiguous nodes. So the number of lookup times will be reduced by a half when a hash collision happens. But this method wastes some memory space. For each chain, the memory space we may waste is the size of a hash node. Assuming that the probability of wasting memory space in a chain is 50%, the expected memory space we may waste is the number of chains multiplying the size of a hash node.

4.3. Performance evaluation

We use the simulator in IXA SDK 4.1 to run the program of our proposed scheme. We use randomly generated prefixes and destination IP addresses to perform IP lookup. To get the maximum throughput, we ensure that we won’t have any hash collision by letting the number of prefixes be small. We generate 10,000 random IP addresses, and calculate the number of total cycle counts required to perform lookup. Then we can get the maximum throughput by dividing the number of total cycle counts by 10,000. The maximum throughput for a lookup result is about one per 100 cycles, i.e. 167 ns, in average.

In [8], the author picks four router products and tests their performances. Table 3 shows the information of the routers and Figure 20 shows the results. We are interested in the packet size of 64 bytes. In this circumstance, three of them can not achieve the OC48 line rate. So we can calculate their forwarding rates. For the other one, we can only know its minimum forwarding rate. Table 4 shows their forwarding rates and ours. The performance of our proposed scheme can compete against any of these routers, with much lower cost.
5. Conclusion and Future Works

We propose a table lookup scheme for the next generation Internet protocol, IPv6. It is based on binary search among prefix length and parallel processing. We design three techniques, merging hash tables, pipelining and multithreading to improve the lookup performance. The maximum throughput is about one lookup result per 100 cycles, i.e. 167 ns, in average. The performance of our proposed scheme depends on the hash function. In our scheme, we use CRC32 as the hash function. We can alleviate the penalty of hash collision by transferring multiple words from the memory, but we cannot guarantee the number of hash collision in the worst case. So doing a more complete experiment to find a better hash function is what we need to do in the future. Since doing insertions or deletions may cause the reconstruction of the lookup table, we still need to develop a fast update scheme in the future. Further, since more advanced processor architectures such as multi-core are available on the market, we may modify our scheme to run on these state-of-the-art platforms for even better cost-effective performance.

6. References