國立交通大學

資訊科學與工程研究所

碩士論文

一個基於平行處理的高速 IPv6 位址查表機制

A High Performance Table Lookup Scheme for IPv6 based on Parallel Processing

研 究 生：洪立哲

指導教授：陳耀宗 教授

中華民國 九十五 年八月

一個基於平行處理的高速 **IPv6** 位址查表機制

A High Performance Table Lookup Scheme for IPv6 based on Parallel Processing

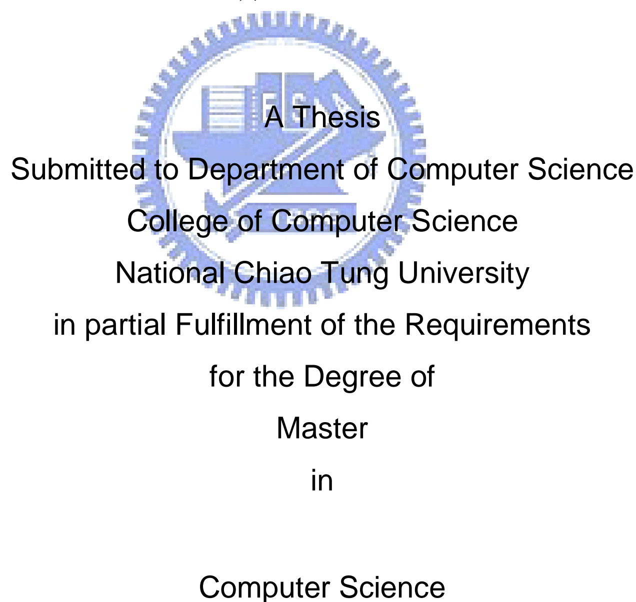
研 究 生：洪立哲　　Student：Li-Che Hung

指導教授：陳耀宗　　Advisor：Yaw-Chung Chen

國 立 交 通 大 學

資 訊 科 學 與 工 程 研 究 所

碩 士 論 文



A Thesis

Submitted to Department of Computer Science

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer Science

August 2006

Hsinchu, Taiwan, Republic of China

中華民國九十五年八月

一個基於平行處理的高速 **IPv6** 位址查表機制

學生：洪立哲　　　　　　　　　　　　　　　　　　指導教授：陳耀宗

國立交通大學資訊工程學系

摘要

網際網路協定第四版（Internet Protocol version 4）早在七零年代末期就被發表出來，並且是目前在網際網路上被廣泛使用的通訊協定。但是隨著網際網路使用者的迅速增加，**IPv4** 的位址數量早已不敷使用。下一代的網際網路通訊協定，也就是 **IPv6**，在 **1996** 年左右被發展出來以解決位址短缺的問題。**IPv6** 將位址的長度從 **32 bits** 擴展到 **128 bits**。

IP 位址查表是基於最長字首比對（Longest prefix matching）。在 **IPv6** 將位址格式擴展到 **128 bits** 的情況下，大部分現有的查表方法難以延伸至 **IPv6**。本篇論文提出一個基於二元搜尋字首長度（Binary search among prefix lengths） [1] 和平行處理的 **IPv6** 查表方法來改善查表效能。首先我們合併 [1] 中的 hash tables 來改善在最糟情況（Worst case）下所需的查表時間。然後我們利用管線（Pipeline）和多執行緒（Multi-threading）的技巧來改善平均情況（Average case）下的流通量（Throughput）。我們將所提出的查表方法在 Intel IXP2400 network processor 上實作。IXP2400 的平行處理架構幫助我們實現管線和多執行緒的設計。模擬結果顯示最大產出可達平均 100 個 cycles 完成一個查表結果，這也表示，在 600 MHz 處理器速度下，我們提出之方法每秒可完成 6 百萬個 IPv6 封包查表。相較於現有之高檔商用產品，我們的方法有明顯較佳的性能改進。

A High Performance Table Lookup Scheme for IPv6 based on Parallel Processing

Student：Li-Che Hung　　　　Adivisor：Yaw-Chung Chen

Department of Computer Science
National Chiao Tung University

Abstract

Internet Protocol version 4 (IPv4) was devised in late 70's and is widely used in Internet nowadays. However, the rapidly increasing number of Internet users leads to the insufficiency of IPv4 addresses. The next generation IP protocol, IPv6, was proposed around 1996 to solve the problem of address shortage. IPv6 extends the IP address length from 32 bits to 128 bits.

IP address lookup is based on longest prefix matching. Most of the existing lookup algorithms scale poorly as IP addresses move to 128 bit addresses. This thesis proposes a table lookup scheme for IPv6 based on binary search among prefix lengths [1] and parallel processing to improve the lookup performance. First, we merge the hash tables in [1] to reduce the lookup complexity of the worst case. Then, we apply the techniques of pipeline and multi-threading to improve the throughput of the average case. We implement our proposed lookup scheme on Intel IXP2400 network processor. The parallel processing architecture of IXP2400 helps us realize the design of pipeline and multi-threading. The simulation results show that the maximum throughput is one lookup result every 100 cycles in average. This means that, under 600 MHz clock rate, our proposed scheme is able to accomplish 6 million table lookups of IPv6 packets. Our proposed method demonstrates better performance obviously comparing with existing high end commercial products.

Acknowledgement

First of all, I would like to express my sincere to the Prof. Yaw-Chung Chen for processing this thesis and giving me some practical suggestions. Besides, I would like to express my thanks to all members in multimedia communication laboratory especially Tzong-Shiun Tsai and Wei-Min Yaw, for their assistances and cheers. Finally, I would like to express my appreciation to my family.

Table of Contents

摘要 i
Abstract ii
Acknoeledgement iii
List of Figures v
List of Tables vi
Chapter 1 Introduction 1
Chapter 2 Related Work 3
2.1 Table Lookup Schemes for IPv4 3
2.1.1 Path-Compressed trie 3
2.1.2 Controlled Prefix Expansion 4
2.1.3 Variants of Multibit Trie 6
2.2 Scalable Table Lookup Sceme 7
2.2.1 Multiway and Multicolumn Search 7
2.2.2 Multiway Range Tree 7
Chapter 3 Proposed Scheme 9
3.1 Binary Search among Prefix Length 9
3.2 Merging Hash Tables 12
3.2.1 Data Structure 14
3.2.2 Lookup Algorithm 15
3.3 Making Lookup Algorithm Pipelined 17
3.4 Using Multi-Threading in the Pipeline Stage 21
Chapter 4 Implementation and Performance Evaluation 23
4.1 Implementation Platform 23
4.2 Some Implementation Issues 25
4.2.1 Simultaneous Memory Accesses 25
4.2.2 Transferring Multiple Words from Memory 28
4.3 Performance Evaluation 31
Chapter 5 Conclusion and Future Works 34
Reference 35

List of Figures

Figure 1.1 The concept of LPM 1

Figure 2.1 A path-compressed trie 3

Figure 2.2 Controlled prefix expansion with the original prefixes and the expanded prefixes 5

Figure 2.3 Expanded trie corresponding to the database of Figure 2.2 6

Figure 2.4 An example of a multiway range tree 8

Figure 3.1 Classifying prefixes to different hash tables 9

Figure 3.2 The binary search tree for IPv4 10

Figure 3.3 A small binary search tree 11

Figure 3.4 Binary search among prefix lengths 12

Figure 3.5 Merging two hash tables into one 13

Figure 3.6 Structure of a hash node 14

Figure 3.7 Details of a flag field 15

Figure 3.8 Modified lookup algorithm 17

Figure 3.9 The binary search tree for IPv6 without merging hash tables 18

Figure 3.10 The modified binary search tree for IPv6 19

Figure 3.11 The concept of pipelined architecture 20

Figure 3.12 The processing cycle of a stage 21

Figure 3.13 The executing order of the 8 threads 22

Figure 4.1 The hardware architecture of IXP2400 23

Figure 4.2 The hardware architecture of the microengine 24

Figure 4.3 The memory latency of SRAM 27

Figure 4.4 The memory latency of DRAM 27

Figure 4.5 The arrangement of hash tables in the three separate memories 28

Figure 4.6 The average latencies of SRAM 29

Figure 4.7 The average latencies of DRAM 29

Figure 4.8 The concept of chaining 30

Figure 4.9 The concept of two contiguous nodes 30

Figure 4.10 The performances of several routers 32

List of Tables

Table 4.1 The maximum size of three separate memories 25
Table 4.2 The latency of SRAM and DRAM 26
Table 4.3 The information of the tested routers 32
Table 4.4 The comparison of the maximum forwarding rates 33

Chapter 1 Introduction

Both traffic and users on the Internet have been growing exponentially and continuously. As a consequence, the 32-bit addresses of IPv4 are consumed rapidly and will be exhausted soon. The next generation IP protocol, IPv6, was proposed around 1996 to solve the problem of address shortage. In IPv6, the address format is 128 bits long.

Because Classless Inter-Domain Routing (CIDR) [3] was deployed to allow for arbitrary aggregation of networks, the process of packet forwarding becomes complicated. When a router receives an IP packet, it performs the operation of Longest Prefix Matching (LPM) to decide the output port to forward the packet. A router has a lookup table (or called forwarding table). Each entry in the lookup table is a 2-tuple (prefix, output port). A prefix is a bit string whose length is between 1 and 128 in IPv6. It represents an aggregation of networks. The operation of LPM is finding the longest prefix that match the destination IP address, so-called the best matching prefix (BMP). Then, the router forwards the IP packet to the output port associated with that BMP. Figure 1.1 shows the concept of LPM：

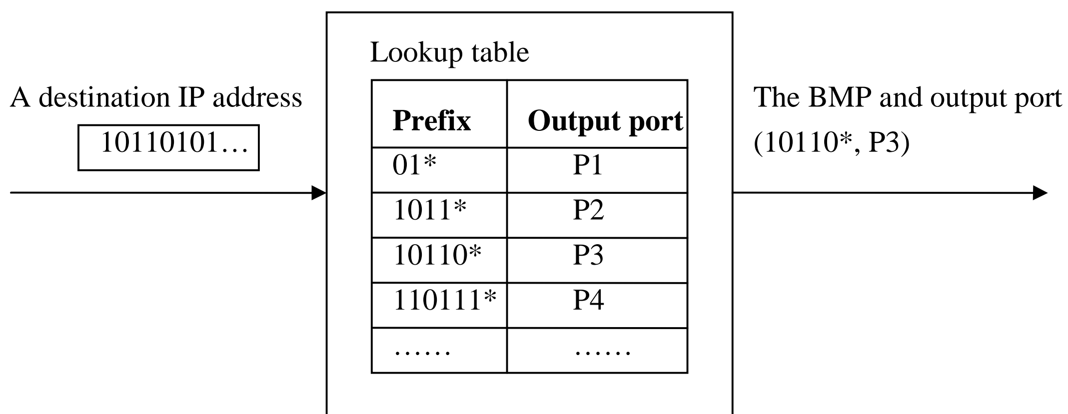


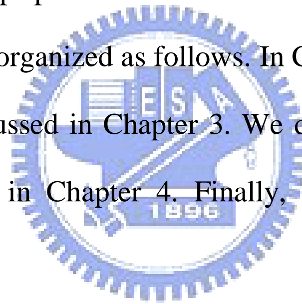
Figure 1.1 The concept of LPM.

The link speed, the router data throughput, and the packet forwarding rate are the three

key factors influencing the transmission rate of the Internet links. The high-speed fiber-optic link and the switching technology are good enough to solve the problem of first two factors. The third factor, the packet forwarding rate is the bottleneck because the operation of LPM is complicated.

Many fast lookup schemes [6] have been proposed, but they almost focus on the processing of IPv4. Their performance degrades when they are scaled to provide lookup for the 128-bit IPv6 addresses. Three scalable lookup schemes [1] [4] [5] were proposed, but they still focus on the processing of IPv4 primarily. We propose a lookup scheme for IPv6 based on [1] and parallel processing to improve the lookup performance of IPv6. In our proposed scheme, the maximum throughput for a lookup result is about 100 cycles in average, which turns out to be 6 million lookups per second.

The rest of this thesis is organized as follows. In Chapter 2, we present the related works. The proposed scheme is discussed in Chapter 3. We discuss the implementation details and show the experiment results in Chapter 4. Finally, the conclusion and future works are presented in Chapter 5.



Chapter 2 Related Work

We discuss some important table lookup schemes in this chapter. First we discuss some important table lookup scheme for IPv4 in section 2.1. The schemes we discuss in section 2.2 are primarily those can be scaled up to IPv6.

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 2.1 shows an example of the path-compressed trie：

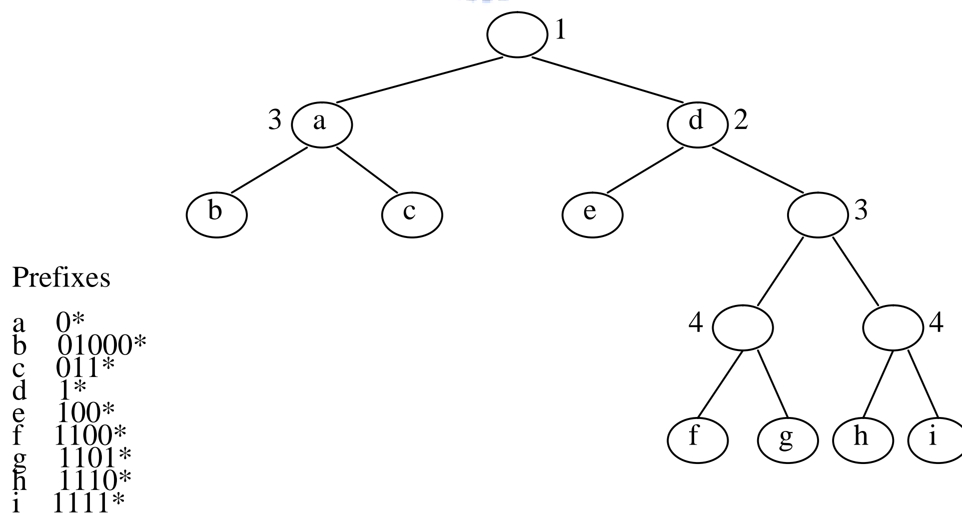


Figure 2.1 A path-compressed trie

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 traversed to decide the branching direction. If the node is with a prefix, we need to compare it with the destination IP address. We record the prefix as the BMP so far if getting matched. We traverse the trie until a leaf is encounter or we fail to get matched.

2.1.2 Controlled Prefix Expansion

Srinivasan *et al*. presented a data structure [7] based on multibit trie. The first idea of the scheme is to reduce a set of prefixes of arbitrary lengths to a predefined set of lengths by using a technique called "controlled prefix expansion". Figure 2.2 shows an example of the original prefixes and the expanded prefixes. Applying dynamic programming can do this, but it also makes the trie construction more time consuming. As Figure 2.3 shows, the 1-bit trie has been divided into three levels, and the expanded trie only has maximum path length of two 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 the 1-bit trie. By using the standard trie representation with arrays of children pointers, insertions and deletions can be supported in the scheme.

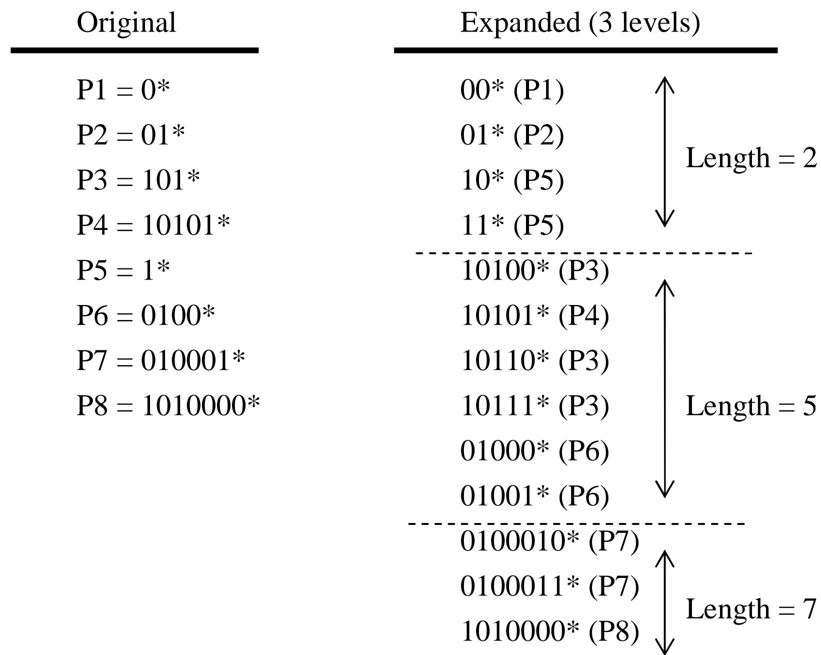
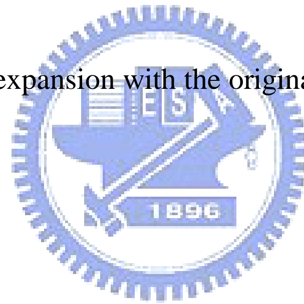


Figure 2.2 Controlled prefix expansion with the original prefixes and the expanded prefixes.



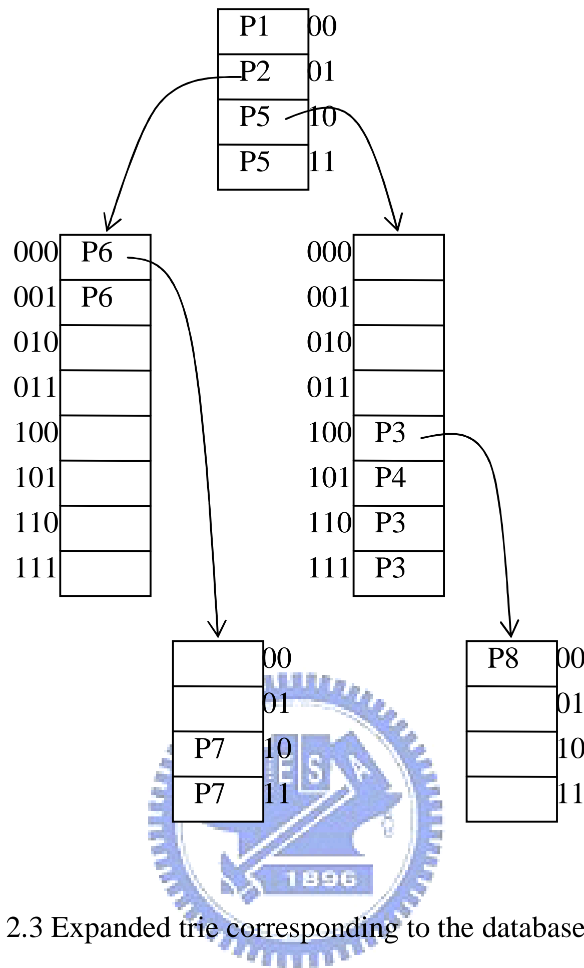


Figure 2.3 Expanded trie corresponding to the database of Figure 2.2.

2.1.3 Variants of Multibit Trie

The basic scheme of Gupta *et al.* [11] uses a two-level multibit trie with fixed strides similar to the one in Figure 2.3. 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 *et al.* [12] recursively transform a binary trie with prefixes into a multibit trie. Starting at the root, they replace a nearly full binary subtrie with a corresponding one-level multibit subtrie. This process is repeated recursively with the children of the multibit subtrie obtained. Actually, 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 the 2^k nodes at level k, where a sufficient fraction of nodes is defined using a single parameter called *fill factor* x, with $0 < x \leqq 1$.

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 precomputing 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 initial precomputed 16-bit array to reduce the number of required memory accesses. The multicolumn search exploits the fact that most processors prefetch an entire cache line when doing a memory access. By using six way branching search, the worst case is five cache line fills in a Pentium Pro with a 32-byte cache line. However, the insertion/deletion of prefixes may result in a table reconstruction due to the recalculation of the pre-computed information.

2.2.2 Multiway Range Tree

This lookup algorithm [5] is the improved one of that described in section 2.2.1 It has faster update speed by using address span. Same as the scheme in section 2.2.1, it encodes each prefix as the start point and the end point of a range. Then it uses the data structure, 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 2.4 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 $O(\log_k N)$, the space complexity is $O(kN\log_k N)$, and the update complexity is $O(k\log_k N)$.

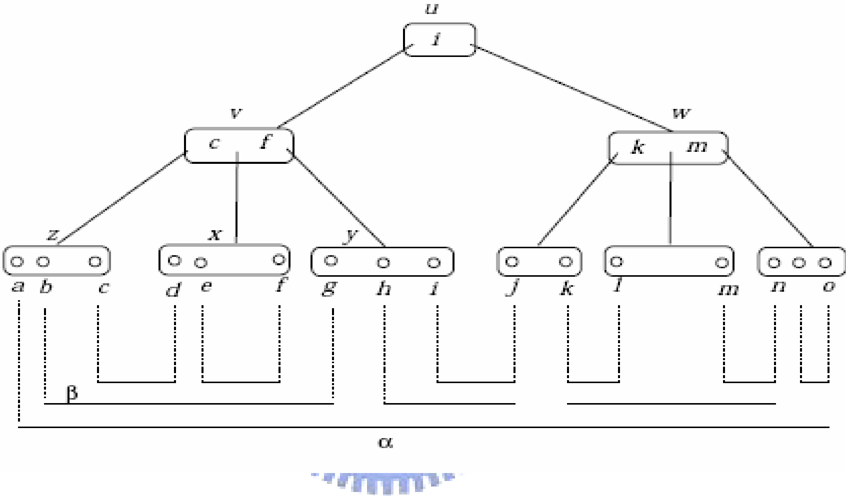


Figure 2.4 An example of a multiway range tree.

Chapter 3 Proposed Scheme

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

3.1 Binary Search among Prefix Lengths

We address the lookup scheme of [1] in this section. For each possible prefix length, we use a corresponding hash table to store prefixes with that length. We use the notation $Table_i$ to represent the hash table of prefixes of length *i*. Figure 3.1 shows a simple example：

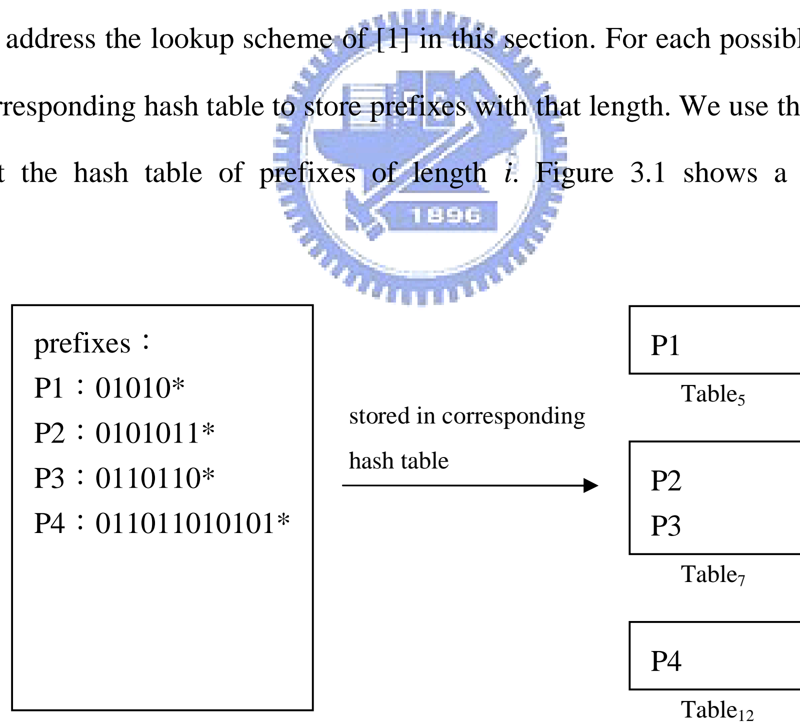


Figure 3.1 Classifying prefixes to different hash tables.

Then, we perform binary search among prefix lengths. It means that we first perform lookup in a hash table of a specific length. According to the lookup result, we decide whether a hash

table of shorter length or of longer length we need to do lookup next. The binary search tree for IPv4 is like that in Figure 3.2：

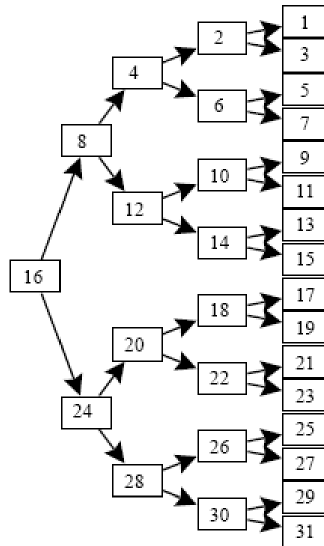


Figure 3.2 The binary search tree for IPv4.

Note that, in Figure 3.2, each node in the binary search tree represents a hash table. Also note that 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 3.2 means a hash table of prefixes of length 16, i.e. $Table_{16}$. In the binary sea rch tree, the path from the root to a certain node represents a possible lookup order of hash tables. According to the binary search tree, we perform lookup in $Table_{16}$ first. Then, according to the lookup result, we decide whether $Table_{8}$ or $Table_{24}$ is the next hash table we need to do lookup and so on.

One characteristic in the problem of longest prefix matching is, if we know that a destination IP address of the destination matches a prefix of a certain length, we only need to look for those matching prefixes of longer length. But if the IP address of the destination 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 feature to make binary search work accurately. That feature is so-called a marker. 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. We don't need all possible markers for a prefix. For a prefix, we know the lookup order of hash tables according to the binary search tree. We 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. Having the feature of marker, we can guarantee that if an IP address matches nothing in a hash table of a certain length, it won't match anything in the hash tables of longer lengths. In other words, it only possibly has matching prefix of shorter length. Let's take a small binary search tree as an example. See the binary search tree in Figure 3.3：

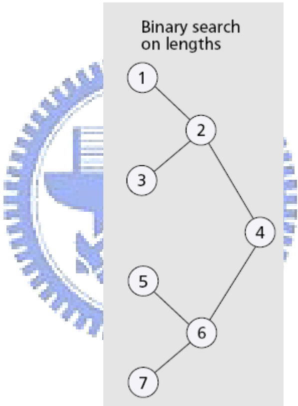


Figure 3.3 A small binary search tree.

If we have a prefix 1001011* in $Table_7$, we insert its marker of length 6 into $Table_6$. It means that we insert 100101 into $Table_6$. We also need to insert the marker of length 4 into $Table_4$. It means that we need to insert 1001 into $Table_4$. Now, each element in the hash tables may be a prefix or a marker. Note that the element may be both a prefix and a marker.

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. Consider Figure 3.3 again. If we have only two prefixes in the forwarding table, say, 1001011* and 10*. The former is in $Table_7$ and the latter is in $Table_2$. 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.

```
Function BinarySearch(D) (* search for address D *)
Initialize search range R to cover the whole array L;
Initialize BMP found so far to null string;
While R is not empty do
    Let i correspond to the middle level in range R;
    Extract the first L[i].length bits of D into D';
    M := Search(D', L[i].hash); (* search hash for D' *)
    If M is nil Then set R := upper half of R; (* not found *)
    Elseif M is a prefix and not a marker
    Then BMP := M.bmp; break; (* exit loop *)
    Else (* M is a pure marker, or marker and prefix *)
        BMP := M.bmp; (* update best matching prefix so far *)
        R := lower half of R;
    Endif
Endwhile
```

Figure 3.4 Binary search among prefix lengths.

Figure 3.4 shows the whole lookup algorithm of binary search among prefix lengths in [1]. Note that in the second line of Figure 3.4, the whole array L means that we have an array of hash tables. Each element in that array is a hash table of a specific length. Also note that in the sixth line, L[i].length means the corresponding prefix length of that hash table, and in the algorithm, searching the upper half of a range means to search the hash tables of shorter lengths, and searching the lower half of a range means searching the hash tables of longer lengths.

This lookup scheme is scalable, and it's complexity is $O(\log_2 W)$. 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 times of table lookup is 5.

3.2 Merging Hash Tables

We apply the lookup scheme described in section 3.1 and do some modification to

improve the performance for the worst case. When we deal with IPv6, the range of binary search has to do lookup of 128 hash tables instead of 32 because the IP address in IPv6 is 128 bits long. We need to perform lookup of 7 different hash tables in the worst case because the lookup complexity is $O(\log_2 128)$. The main idea of modification is merging two hash tables into one. Actually, we merge a hash table of prefixes with even number of bits, say *2n* (*n*=1, 2, 3, 4, etc), with the hash table of prefixes of length *2n+1* (*2n+1* is odd). 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 $Table_{2n}$ and the elements in $Table_{2n+1}$. Assuming we have either prefix *P ·0* or prefix *P ·1* in $Table_{2n+1}$ (*P* is a bit string of length *2n*, and *P ·0* means *P* followed by a bit *0*, and the dot means the operation of concatenation), we should have a marker *P* in $Table_{2n}$. Note that we may have both *P ·0* and *P ·1* in $Table_{2n+1}$. From the discussion above, we associate a marker *P* in $Table_{2n}$ with *P ·0* and *P ·1*. Figure 3.5 shows the concept：

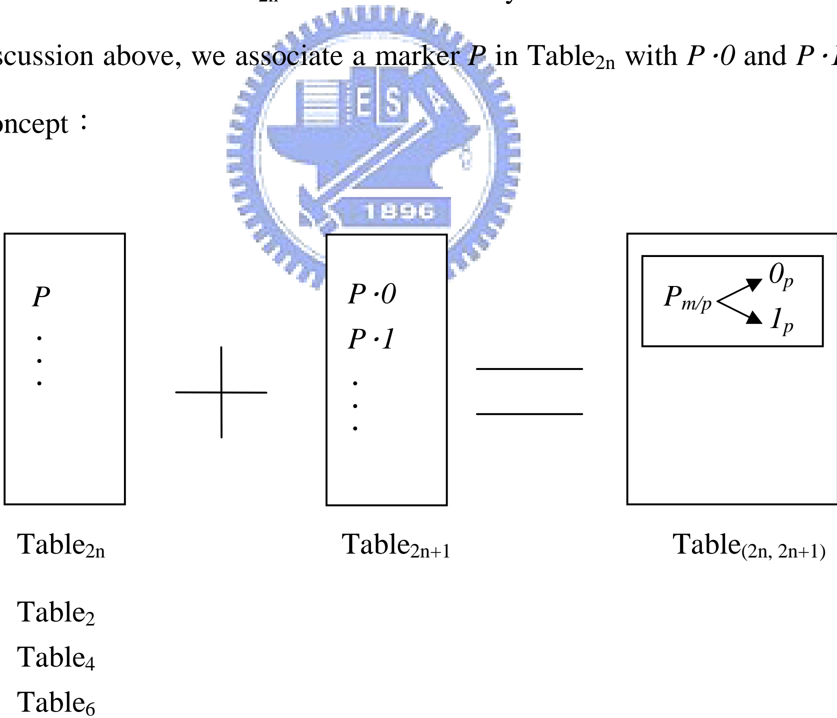


Figure 3.5 Merging two hash tables into one.

In Figure 3.5, $Table_{(2n,\ 2n+1)}$ denotes the hash table after merging $Table_{2n}$ and $Table_{2n+1}$. And in $Table_{(2n,\ 2n+1)}$, the subscript *m/p* of *P* denotes that *P* is either a marker or a prefix. *P* is

associated with two prefixes, *P·0* and *P·1*, and we use arrows to represent the associations. So P has two arrows：one point to *0* representing *P·0*, and the other point to *1* representing *P·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 3.6 and 3.7 illustrate the data structure of the node in the hash table：

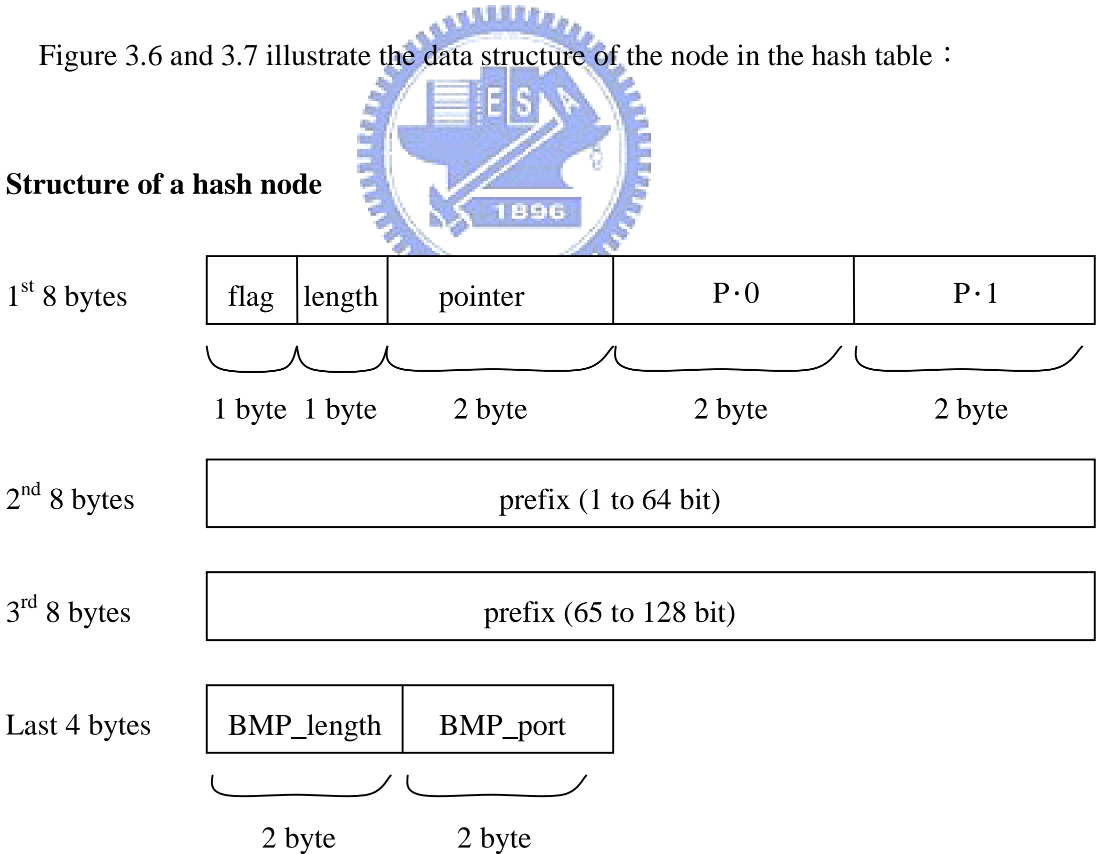


Figure 3.6 Structure of a hash node.

Details of a flag field

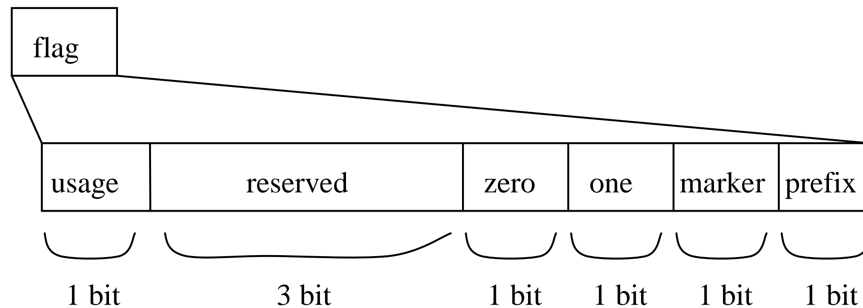


Figure 3.7 Details of a flag field.

The *length* field indica tes the length of this prefix (or this marker). The *pointer* field records the memory location of the next node that has the same hash value. The *prefix* field records this prefix (or this marker). The *P·0* field records the information of the output port associated with the prefix *prefix·0*. The *P·1* field records the information of the output port associated with the prefix *prefix·1*. The *BMP_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, *BMP_length* equals to *length* if this node is a prefix. The *BMP_port* field records the output port associated with the BMP of this marker (or this prefix). The *flag* field of the hash node includes all flags we used. Now, we illustrate those flags in Figure 3.7. The *usage* flag indicates that this node is in use. The *reserved* part is not used. The *zero* flag indicates that whether this node has the prefix *prefix·0* or not. The *one* flag indicates whether this node has the prefix *prefix·1* or not. The *marker* flag indicates that this node is a marker. The *prefix* flag indicates that this node is a prefix.

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 address the modified lookup algorithm and

then give the pseudo procedure. We perform lookup in some hash table, say $Table_{(2i,\ 2i+1)}$. We 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 $Table_{(2i,\ 2i+1)}$. We check whether the hash node of that index in $Table_{(2i,\ 2i+1)}$ is null, if the hash node is null, it will be failed to get matched in $Table_{(2i,\ 2i+1)}$ and then we look for those matching prefixes shorter than *2i*. Otherwise the IP address of the destination get matched in $Table_{(2i,\ 2i+1)}$, we record the *BMP_port* field of that node as the BMP so far. Now, we know the BMP whose length is equal or shorter than *2i*. Then we try if 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·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·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 flag *marker* is set or not and decide if 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 4.2.2. Figure 3.8 shows the pseudo procedure of lookup：

Function ModifiedLookup (*D*) /*lookup for address *D**/

Let *L* be the array of hash tables of all distinct lengths;

Initialize search range *R* to cover the whole array *L*;

Initialize *BMP* found so far to null string;

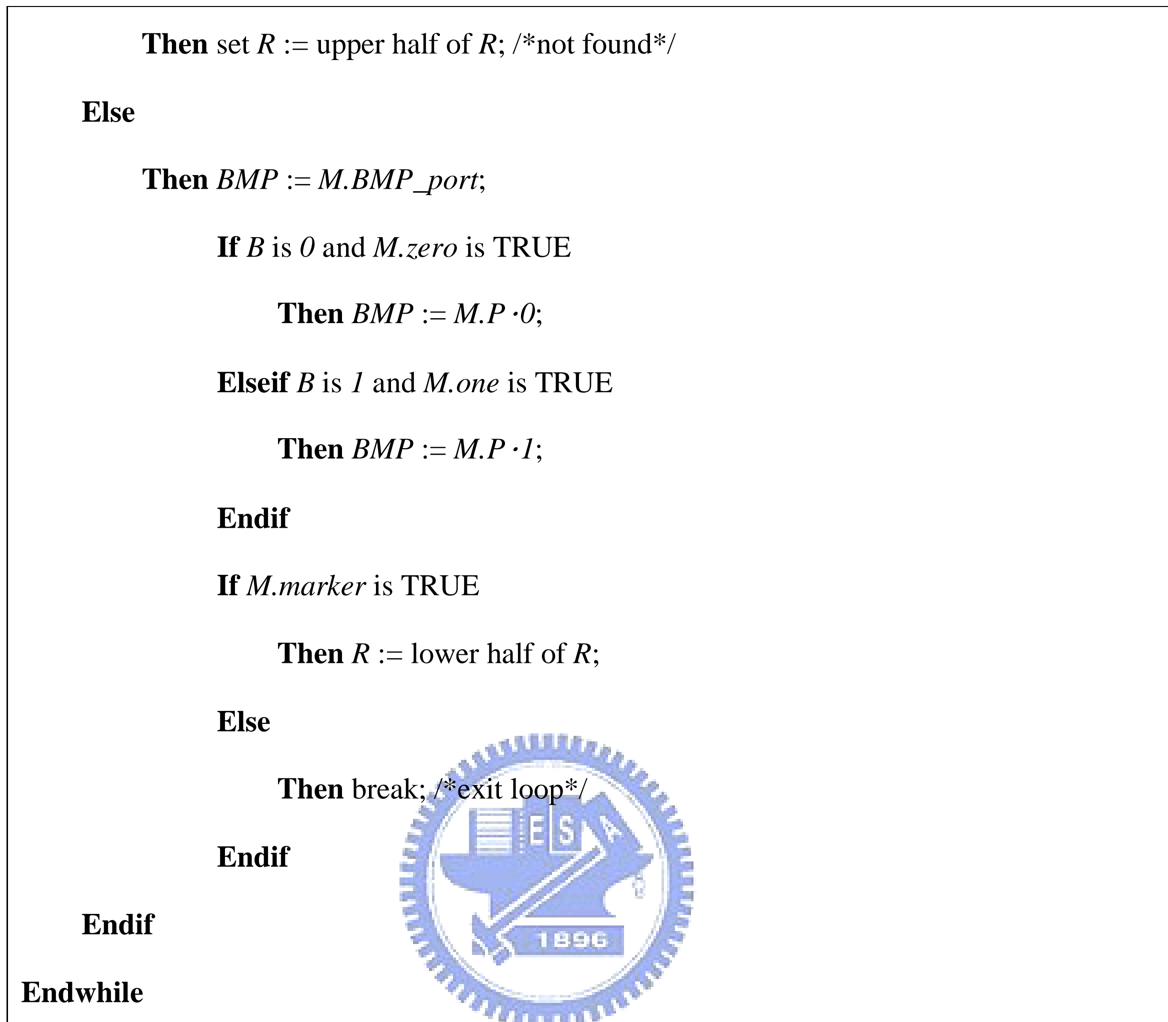
While *R* is not empty do

    Let *n* corresponds to the middle level in range *R*;

    Extract the first *n* bits of *D* into *D'* and let *B* be the *n+1*-th bit;

    M := Hash(*D'*, *L*[*n*]); /*use *D'* to be the key, then do hash in *L*[*n*]*/

    If *M* is nil



```
        Then set R := upper half of R; /*not found*/
    Else
        Then BMP := M.BMP_port;
             If B is 0 and M.zero is TRUE
                 Then BMP := M.P ·0;
             Elseif B is 1 and M.one is TRUE
                 Then BMP := M.P ·1;
             Endif
             If M.marker is TRUE
                 Then R := lower half of R;
             Else
                 Then break; /*exit loop*/
             Endif
    Endif
Endwhile
```

Figure 3.8 Modified lookup algorithm.

3.3 Making Lookup Algorithm Pipelined

We apply the pipeline technique to the lookup algorithm. The pipeline technique helps us improve the lookup performance of the average case. After merging hash tables, the binary search tree is changed. Figure 3.9 shows the binary search tree for IPv6 without merging hash tables. Figure 3.10 shows the modified binary search tree for IPv6 after merging hash tables：

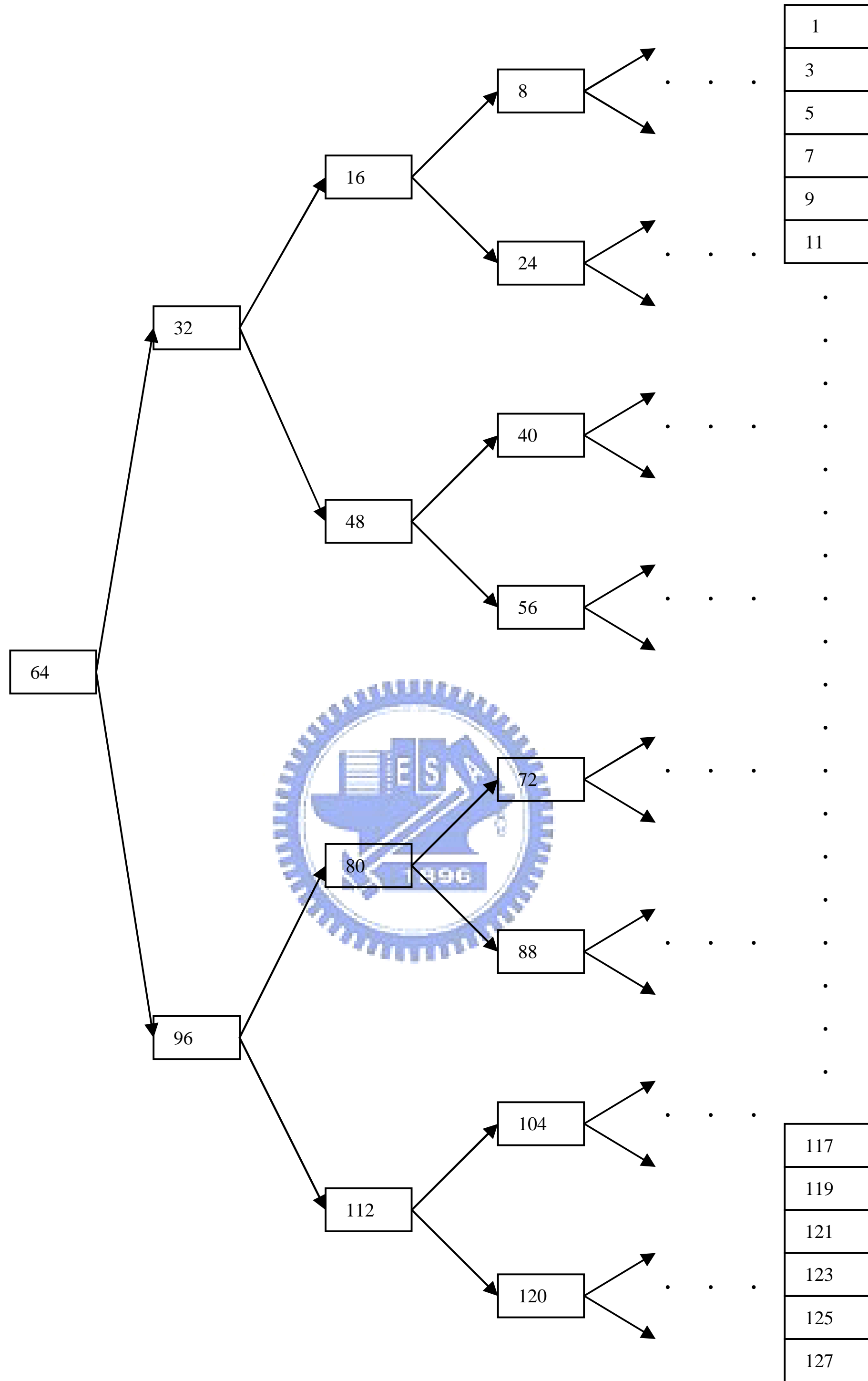


Figure 3.9 The binary search tree for IPv6 without merging hash tables.

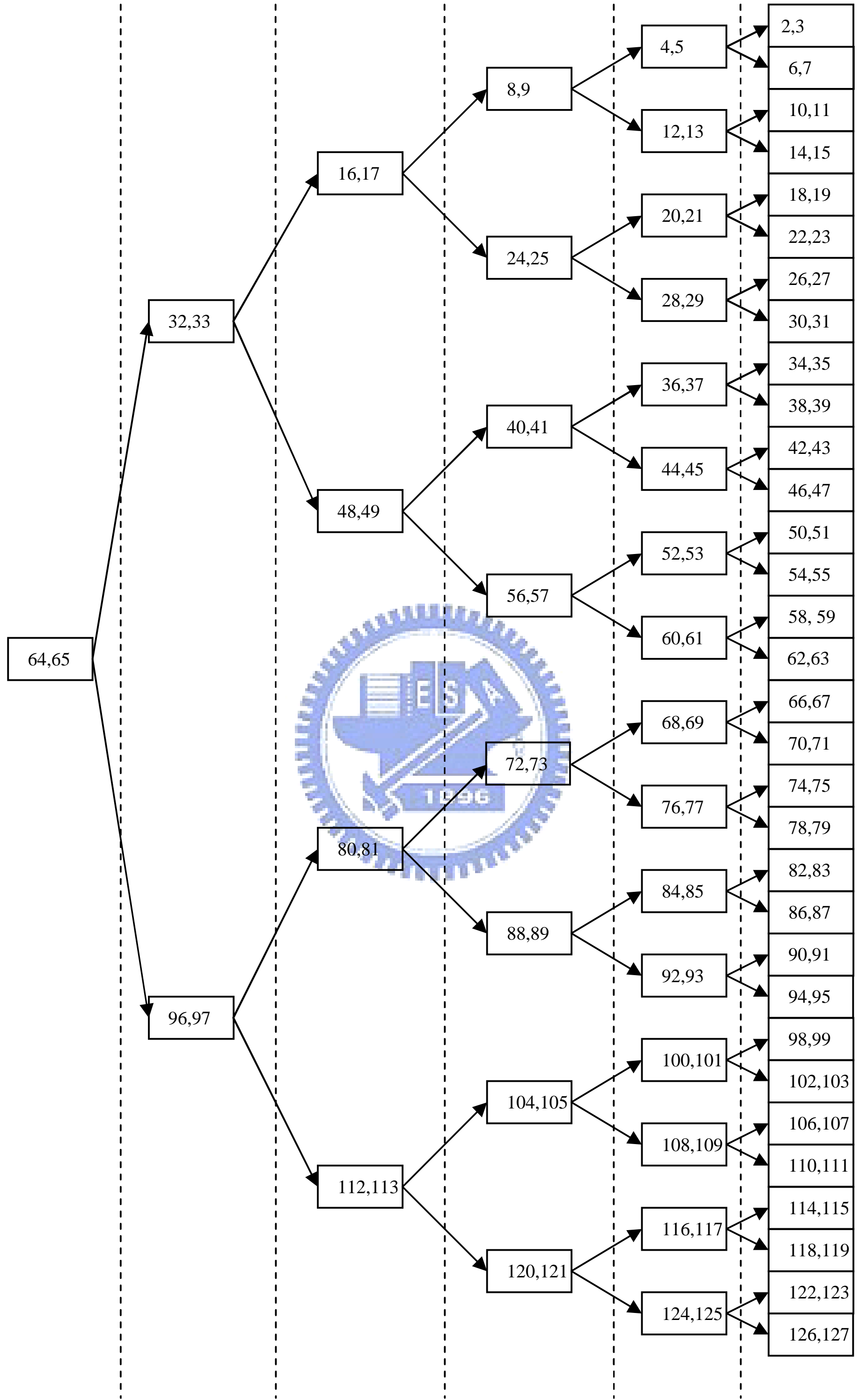


Figure 3.10 The modified binary search tree for IPv6.

In Figure 3.10, each node is a hash table containing the information of prefixes of two consecutive lengths. The modified binary search tree has 6 levels. Considering the lookup algorithm in Figure 3.8, the loop in 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 each iteration 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 that should be searched next, and the skip flag. The skip flag tells the next processing unit that we have already found the BMP, so the next processing unit does not have to do anything. The meaning of the skip flag is same as the operation of break from the loop in the lookup algorithm of Figure 3.8. Figure 3.11 shows the concept of the pipelined lookup：

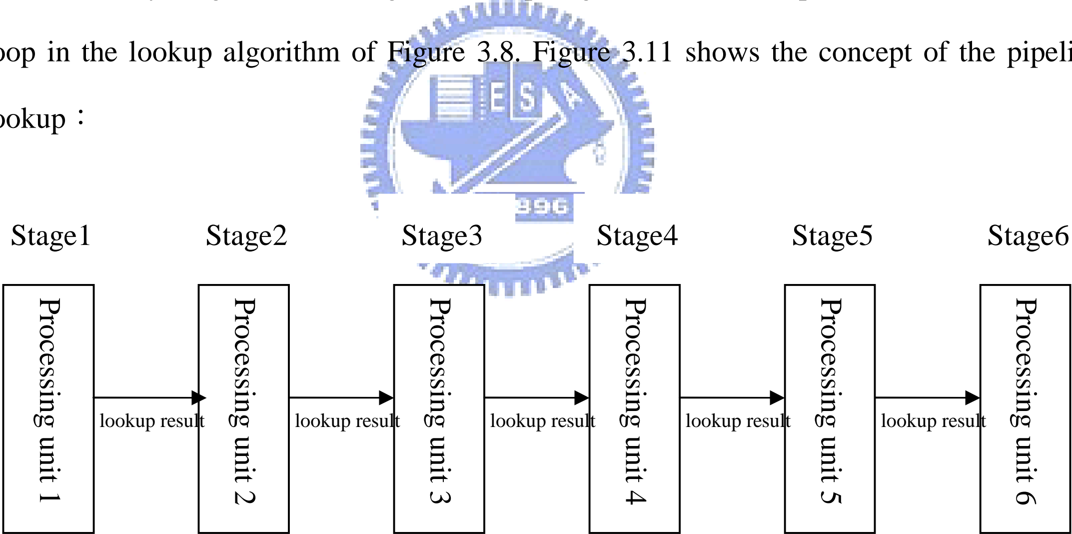


Figure 3.11 The concept of pipelined architecture.

By making the lookup algorithm pipelined, we can get one complete lookup result per processing cycle of a stage. A processing cycle of a stage includes three parts. First we use the destination IP address as the key to do hash. Second we lookup the hash table using the hash value as the index of the hash table. Finally, we do some computation according to the lookup

result. Figure 3.12 shows the processing cycle of a stage：

A processing cycle of a stage

Figure 3.12 The processing cycle of a stage.

We should note one thing, 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. Observing the processing cycle in Figure 3.12, when the processing unit does a lookup on the hash table, it needs to access the memory and waits the memory access to finish because the hash tables are stored in the memory. In the waiting period, the processing unit is idle. We can use a thread to do an IP lookup. When the thread is waiting the memory access to finish, 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 the 8 threads is like that in Figure 3.13. 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.

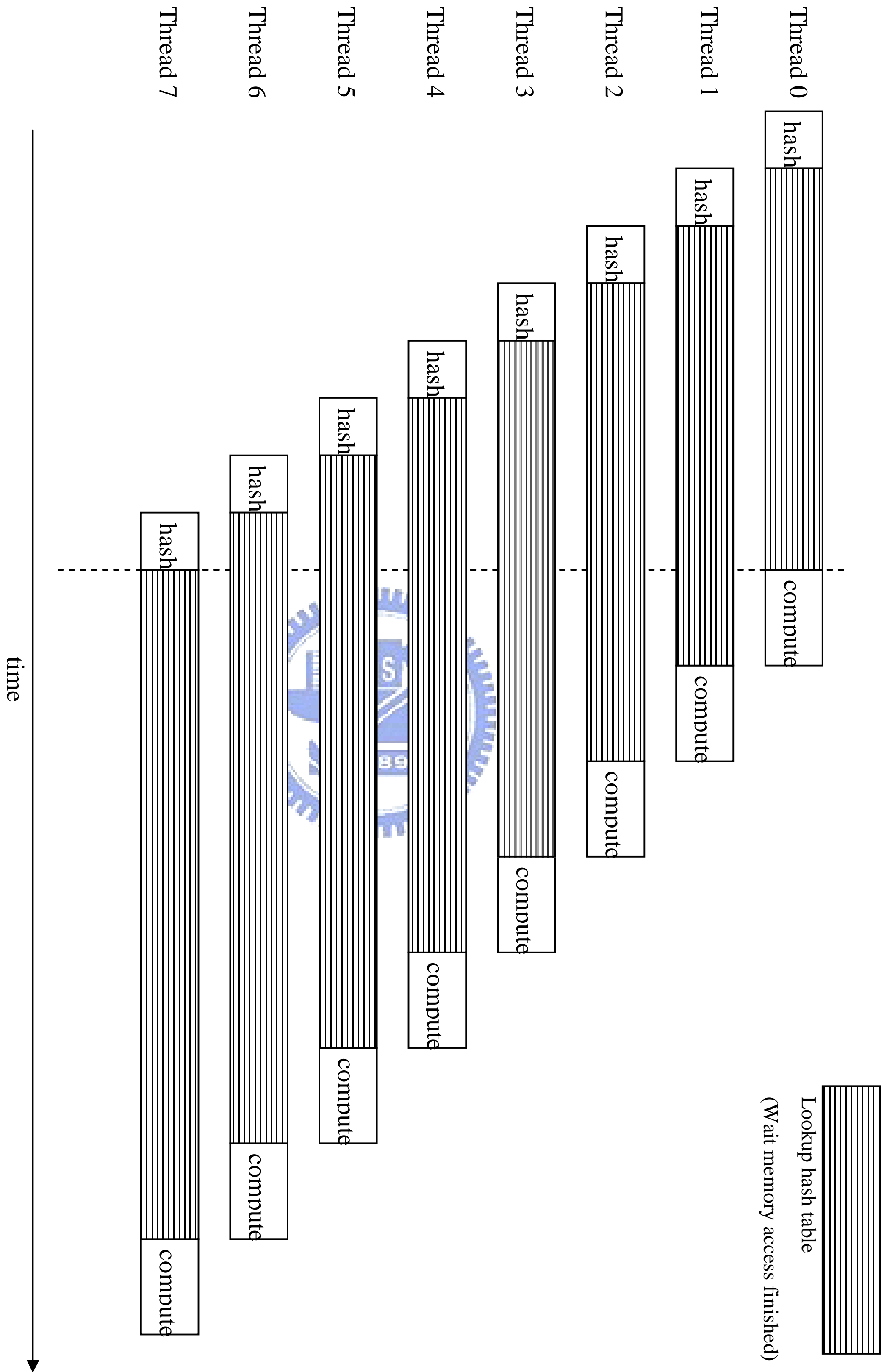


Figure 3.13 The executing order of the 8 threads.

Chapter 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 helps us to realize the design of pipeline and multi-threading. IXP2400 has one Intel XScale core processor and eight co-processors called microengins. Figure 4.1 shows the hardware architecture of IXP2400：

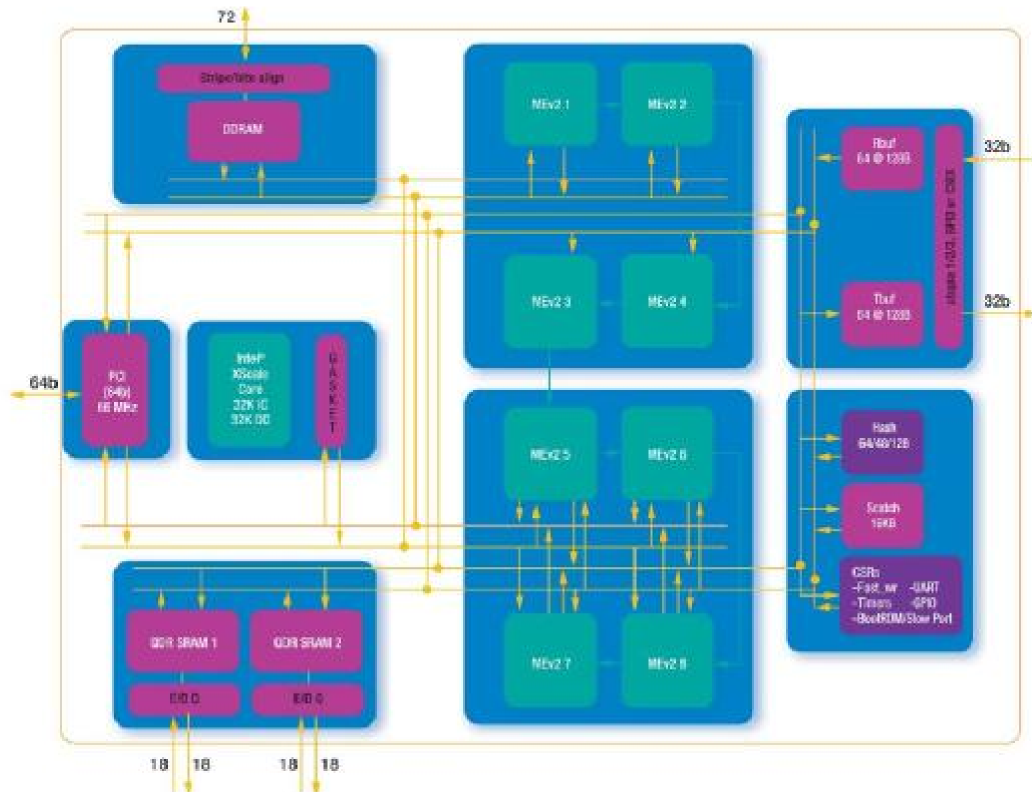


Figure 4.1 The hardware architecture of IXP2400.

The Intel XScale core is just like other embedded general-purpose processors. We can run the ordinary embedded operating system on the Intel XScale core, and then we can run the

ordinary applications on the operating system. The hardware architecture of the microengine is different from the Intel XScale core. The microengine 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. The program running on the microengine can not be too big or too complicated. 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 the program counter. The microengine has special hardware designed for processing the network data, such as the CRC unit doing cyclic redundancy check. And the microengine has an instruction set specifically tuned for processing the network data. So we can utilize the special hardware to speed up processing the network data. Figure 4.2 shows the hardware architecture of the microengine：

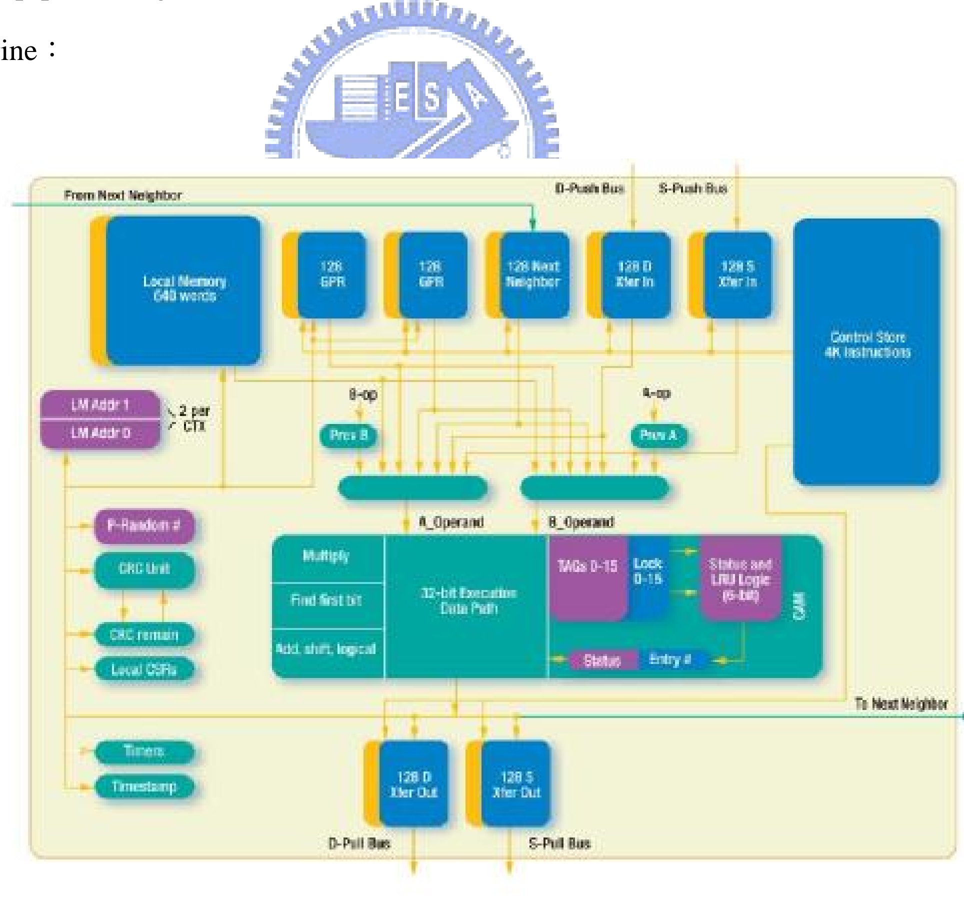


Figure 4.2 The hardware architecture of the microengine.

Due to the differences between the microengines and the Intel XScale core, the microengines handle basic packet processing tasks such as IP lookup. The Intel XScale core acts as a control point managing the microengines and handles the processing tasks of exceptions caused by the network data.

From the discussion above, we use 6 microengines to implement our design of pipeline. And we run 8 threads on each microengine for realizing the design of multi-threading.

4.2 Some Implementation Issues

4.2.1 Simultaneous Memory Accesses

Considering the hardware architecture of IXP2400 shown in Figure 4.1, 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. Distributing the hash tables to separate memories is to alleviate the heavy load of accessing only one memory. Table 4.1 shows the maximum sizes of three separate memories：

Memory	Maximum size
DRAM	1GB
Channel 1 of SRAM	64MB
Channel 2 of SRAM	64MB

Table 4.1 The maximum size of three separate memories.

Now, we examine the access latencies of these memories. Intel provides a development

toolkit called IXA SDK 4.1. IXA SDK 4.1 provides a complete IDE including an editor, a compiler, an assembler, a simulator and a debugger. IXA SDK 4.1 also provides a lot of 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 4.2 shows the average latencies of reading 8 words (a word is 4 bytes long) from SRAM and DRAM respectively：

Memory	Latency
DRAM	137 cycles
SRAM	117 cycles

Table 4.2 The latency of SRAM and DRAM.

We didn't distinguish channel 1 of SRAM from channel 2 of SRAM in Table 4.2 because they are the same kind of memory, SRAM. Note that the results in Table 4.2 are in the circumstance of only one microengine trying to access the memories. According to Table 4.2, 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 microengines access the three separate memories respectively. The simulation indeed shows the same result of memory latency as that in Table 4.2 when three different microengines access the three separate memories respectively. Figure 4.3 shows the average latencies of reading 8 words from a certain channel of SRAM when different numbers of microengines try to contend for accessing that channel of SRAM. We still don't need to distinguish channel 1 of SRAM from channel 2 of SRAM in Figure 4.3. Figure 4.4 shows the average latencies of reading 8 words from DRAM when different numbers of microengines try to contend for accessing DRAM. The number of simultaneous memory accesses is more than what we imagine, three. We can have 8 simultaneous SRAM accesses (4 from channel 1 of SRAM and

4 from channel 2 of SRAM) without increasing the average memory latency. In addition, we can have 3 simultaneous DRAM accesses without increasing the average memory latency.

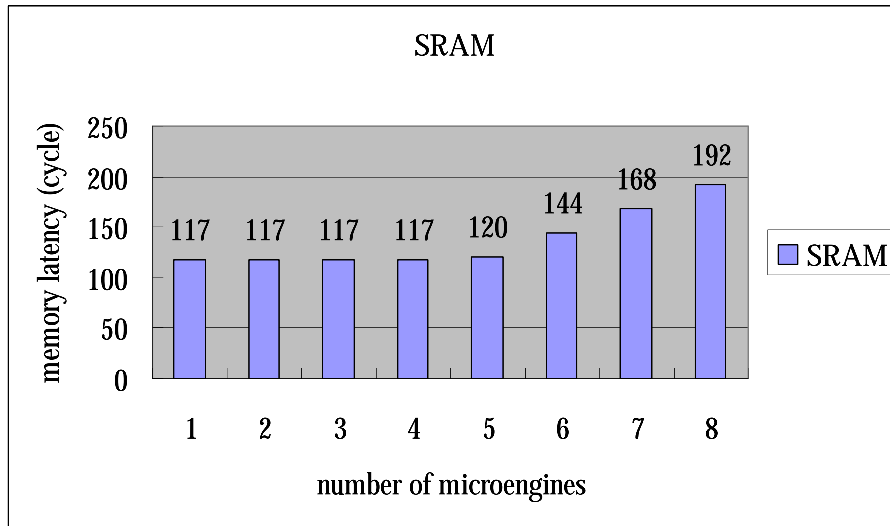


Figure 4.3 The memory latency of SRAM.

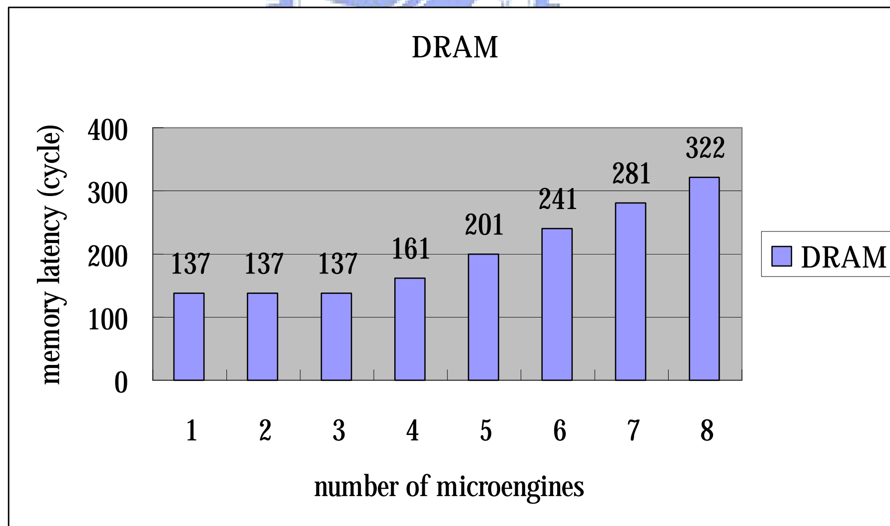


Figure 4.4 The memory latency of DRAM.

Now, we extend Figure 3.11 and show the arrangement of those distinct hash tables in the three separate memories. Figure 4.5 is the extension of Figure 3.11. Those hash tables accessed by processing unit 1 and processing unit 2 are put in channel 1 of SRAM. Those

hash tables accessed by processing unit 3 and processing unit 4 are put in channel 2 of SRAM. Those hash tables accessed by processing unit 5 and processing unit 6 are put in DRAM. A processing unit means a microengine. Those hash tables accessed by certain processing units are in the corresponding level of the binary search tree as shown in Figure 4.5.

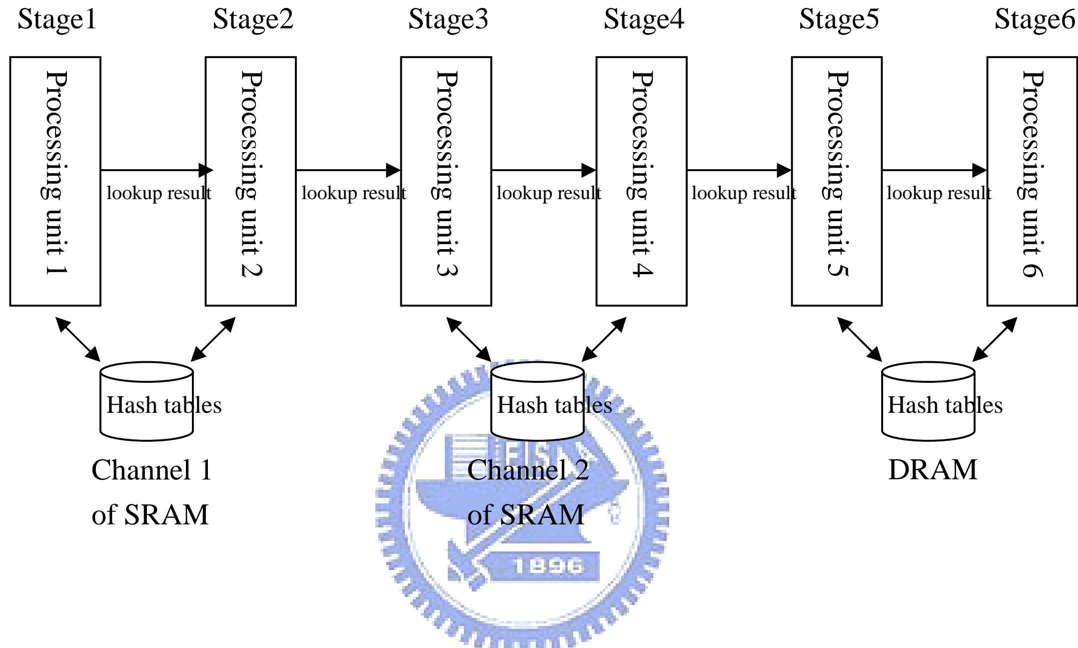


Figure 4.5 The arrangement of hash tables in the three separate memories.

4.2.2 Transferring Multiple Words from Memory

Each time we access SRAM or DRAM, we can transfer multiple words instead of only one word. The maximum number of words we can transfer is 16. Figure 4.6 shows the average latencies of SRAM when we read different numbers of words. Figure 4.7 shows the average latencies of DRAM when we read different numbers of words.

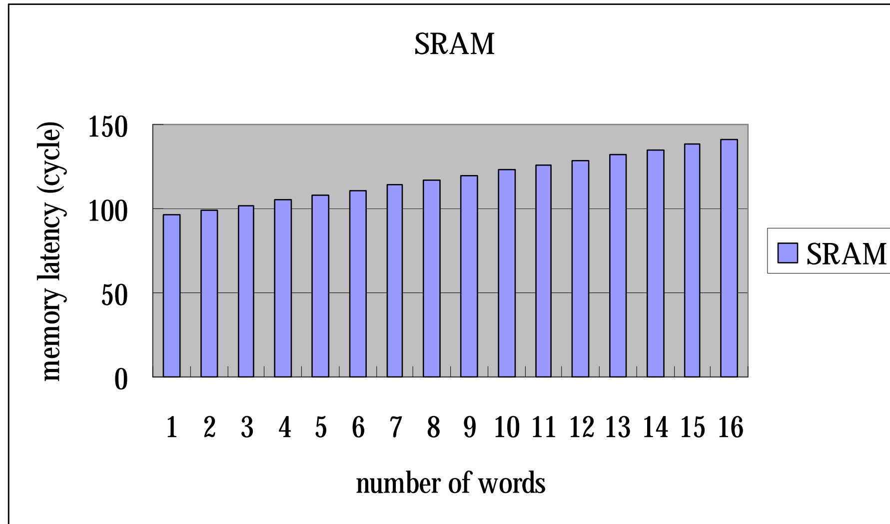


Figure 4.6 The average latencies of SRAM.

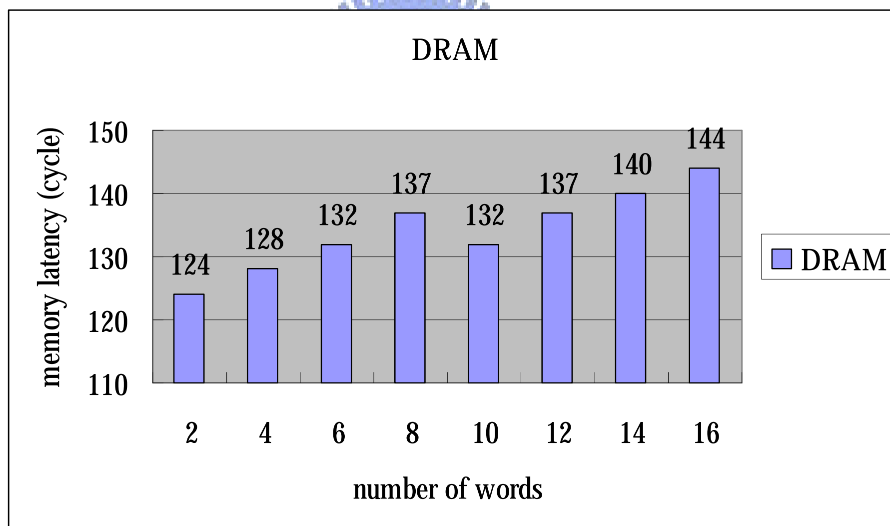


Figure 4.7 The average latencies of DRAM.

When we read more than one word from the memory, either SRAM or DRAM, the memory latency increases only a little bit. It is not multiple increasing in memory latency. We can utilize this characteristic to alleviate the penalty of hash collision. In section 3.2.2, we skipped the issue of hash collision and assumed a perfect hash function. Actually, we do not have a perfect hash function. We use the special hardware in the microengine, the CRC unit, and use

CRC32 built in the CRC unit as the hash function. As mentioned in section 3.2.2, we use the method of chaining to resolve the problem of hash collision. Figure 4.8 shows the concept of chaining：

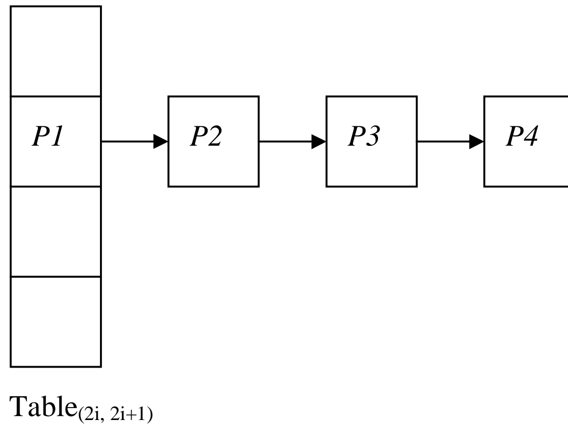


Figure 4.8 The concept of chaining.

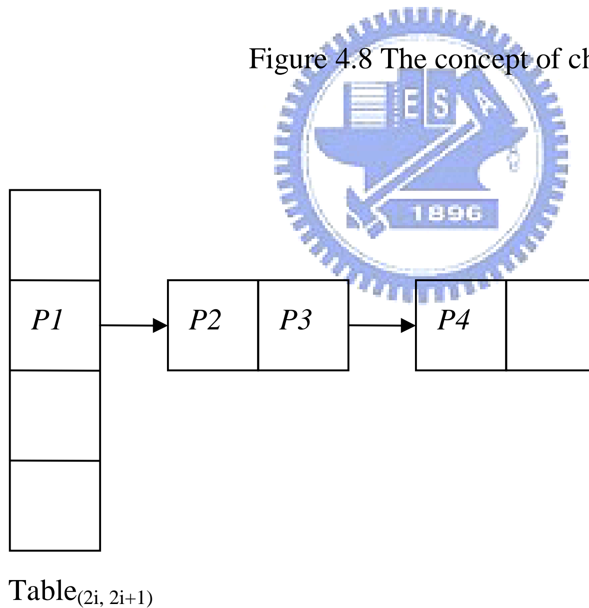
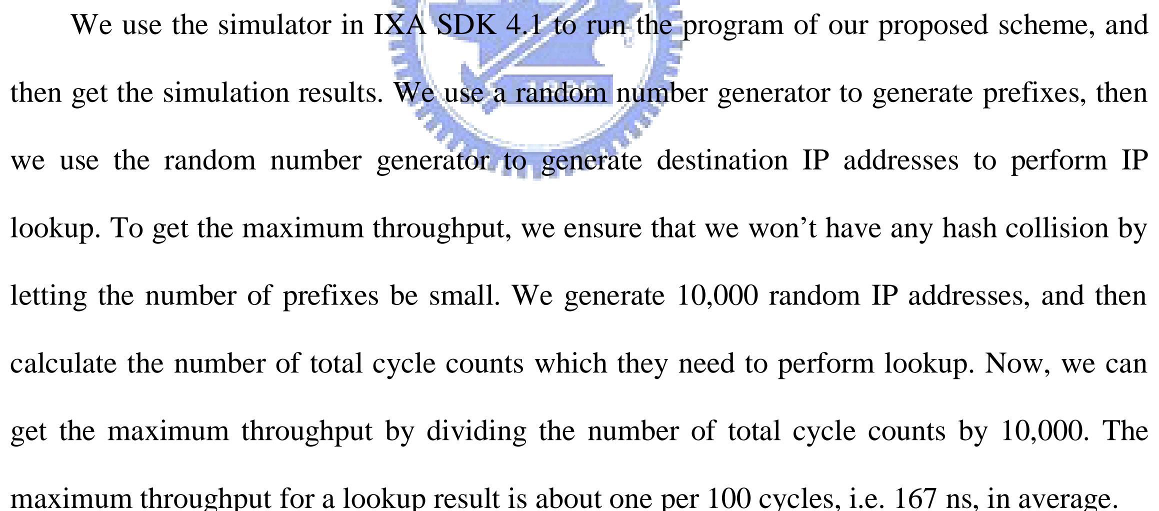


Figure 4.9 The concept of two contiguous nodes.

In Figure 4.8, $Table_{(2i,\ 2i+1)}$ is a hash table. *P1*, *P2*, *P3* and *P4* are prefixes that have the same hash value in $Table_{(2i,\ 2i+1)}$. So we use a chain to link these four prefixes. If a destination IP address matches *P4*, it will need to perform lookup 4 times in $Table_{(2i,\ 2i+1)}$. It means that we

traverse the chain and then find that *P4* is a matching prefix. So we need 4 memory accesses in this case. We can put *P2* and *P3* together. It means that we put *P2* and *P3* at a contiguous memory location. So when we read the node of *P2*, we can read the node of *P3* together by utilizing the characteristic of transferring multiple words. Figure 4.9 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 half when hash collision is happened. But this method wastes the 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 of a chain is 50%, the expected memory space we 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, and then get the simulation results. We use a random number generator to generate prefixes, then we use the random number generator to generate 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 then calculate the number of total cycle counts which they need to perform lookup. Now, 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 4.3 shows the information of the routers and Figure 4.10 shows the results. We are interested in the circumstance that the packet size is 64 bytes. In this circumstance, three of them can not achieve the line rate, OC48. So we can calculate their forwarding rates. For the other one, we can only know its minimum forwarding rate. Table 4.4 shows their forwarding rates and ours.

The performance of our proposed scheme can compete against that of these routers.

Company	**Model**	**Operating System Version**
Fujitsu	Geostream R920	E10V02L03C44
Hitachi	GR2000-20H	S-9181-61 07-01
Juniper	M20	5.5R1.2
NEC	CX5210	--------------------

Table 4.3 The information of the tested routers.

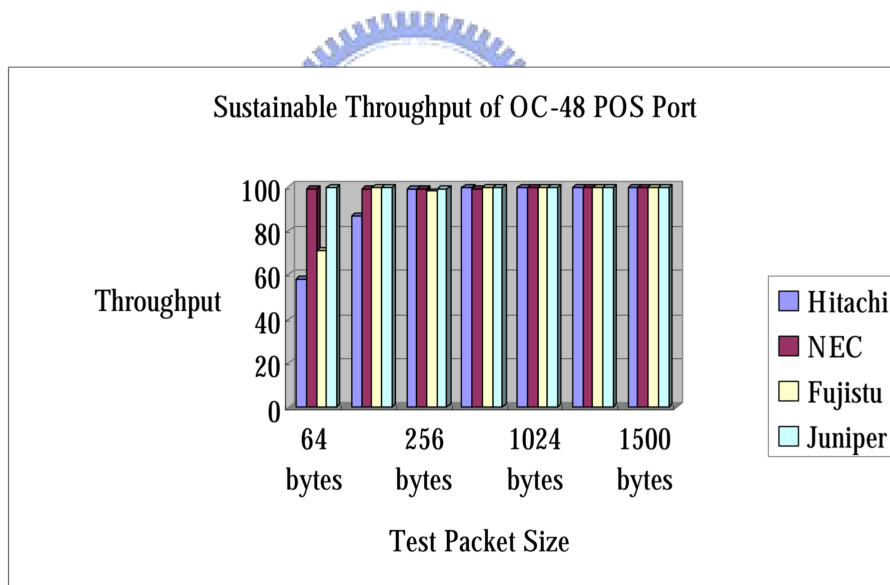


Figure 4.10 The performance of several routers.

	Maximum forwarding rate (Million packets per second)
Hitachi	2.82
NEC	4.81
Fujistu	3.45
Juniper	Exceed 4.86
Our proposed scheme	Up to 6

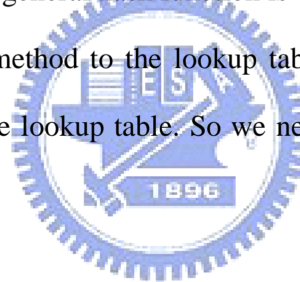
Table 4.4 The comparison of the maximum forwarding rates.

Chapter 5 Conclusion and Future Works

In this thesis, we propose a lookup scheme for IPv6. This scheme is based on binary search among prefix length and parallel processing. We propose three techniques, merging hash tables, pipeline 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 we use. In the proposed scheme, we use CRC32 as the hash function. But we can not guarantee the number of hash collision in the worst case. We only alleviate the penalty of hash collision by using the technique of transferring multiple words from the memory. So doing a more complete experiment to find a good and general hash function is what we need to do in the future.

We lack a fast update method to the lookup table. Doing insertions or deletions may cause the reconstruction of the lookup table. So we need to develop a fast update scheme in the future work.



References

[1] M. Waldvogel, G. Varghese, J. Turner, and B.Plattner, "Scalable high speed IP routing lookups," Proc. ACM SIGCOMM, vol. 27, Oct. 1997, pp. 25-36.

[2] Intel Network Processor IXP2400,

http://www.intel.com/design/ntwork/products/npfamily/ixp2400.htm

[3] V. Fuller, T. Li, J. Yu, and K. Varadhan, "Classless Inter-Domain Routing (CIDR)： An Address Assignment and Aggregation Strategy," IETF, RFC 1519 http://www.ietf.org/rfc/rfc1519.txt, 1993

[4] B. Lampson, V. Srinivasan, and G. Varghese, "IP lookups using multiway and multicolumn search," Proc. IEEE INFOCOM, San Francisco, CA, 1998, pp. 1248-1256.

[5] S. Suri, G. Varghese, and P. R. Warkhede, "Multiway range tree： scalable IP lookup with fast update," Tech. Rep. 99-28, Washinton Univ, 1999.

[6] M. A. Ruiz-Sanchez, E. W. Biersack, and W. Dabbous, "Survey and taxonomy of IP address lookup algorithms," IEEE Network, vol. 15, no. 2, pp. 8-23, Mar.-Apr. 2001.

[7] V. Srinivasan, G. Varghese, "Fast address lookups using controlled prefix expansion," Proc. ACM SIGMETRICS, Jun. 1998, pp. 1-11.

[8] http://www.unstrung.com/document.asp?site=unstrung&doc_id=28937&page_number=1

[9] D. R. Morrison, "PATRICIA – practical algorithm to retrieve information coded in alphanumeric," J. ACM, vol. 15, no. 4, Oct. 1968, pp514-534.

[10] K. Sklower, "A tree-based routing table for Berkeley unix," Proc. Winter Usenix Conf., 1991, pp. 93-99.

[11] P. Gupta, S. Lin, and N. McKeown, "Routing lookups in hardware at memory access speeds," Proc. IEEE INFOCOM, vol. 3, San Francisco, CA, 1998, pp. 1240-1247.

[12] S. Nilsson and G. Karlsson, "IP-Address Lookup Using LC-Tries," IEEE JSAC, June 1999, vol. 17, no.6, pp. 1083-92.