

WASHINGTON UNIVERSITY  
SEVER INSTITUTE OF TECHNOLOGY  
DEPARTMENT OF ELECTRICAL ENGINEERING

---

HARDWARE-BASED INTERNET PROTOCOL PREFIX LOOKUPS

by

WILLIAM N. EATHERTON

Prepared under the direction of Professors Jonathan S. Turner and George Varghese

---

A thesis presented to the Sever Institute of  
Washington University in partial fulfillment  
of the requirements for the degree of  
MASTER OF SCIENCE

May, 1999

Saint Louis, Missouri

WASHINGTON UNIVERSITY  
SEVER INSTITUTE OF TECHNOLOGY

---

ABSTRACT

---

HARDWARE-BASED INTERNET PROTOCOL PREFIX LOOKUPS

by WILLIAM EATHERTON

---

ADVISOR: Professors Jonathan S. Turner and George Varghese

---

May, 1999  
Saint Louis, Missouri

---

Internet Protocol (IP) address lookup is a major bottleneck in high performance routers. IP address lookup is challenging because it requires a *longest matching prefix* lookup for databases of up to 41,000 prefixes for minimum sized packets at link rates of up to 10 Gbps (OC-192c). The problem is compounded by instabilities in backbone protocols like BGP that require very fast update times, and the need for low cost solutions that require minimal amounts of high speed memory. All existing solutions either have poor update times or require large amounts of expensive high speed memory. Existing solutions are also tailored towards software implementations, and do not take into account the flexibility of ASICs or the structure of modern high speed memory technologies such as SDRAM and RAMBUS.

This thesis presents a family of IP lookup schemes that use a data structure that compactly encodes large prefix tables a factor of 10 smaller than schemes with similar update properties. The schemes can be instantiated to require a maximum of 4-7 memory references which, together with a small amount of pipelining, allows wire speed forwarding at OC-192 (10 Gbps) rates. To illustrate this new approach to IP Lookups, a detailed reference design is presented and analyzed.

# TABLE OF CONTENTS

<b>LIST OF TABLES</b> .....	iv
<b>LIST OF FIGURES</b> .....	v
<b>ACKNOWLEDGMENTS</b> .....	vii
<b>1. INTRODUCTION</b> .....	<b>1</b>
1.1. Internetworking .....	1
1.2. Internet Protocol (IP) Router Architectures .....	3
1.3. IP Forwarding .....	10
1.4. Wire Speed .....	16
1.5. Memory Technology .....	18
1.6. Overview of Rest of Thesis .....	23
<b>2. RELATED WORK</b> .....	<b>24</b>
2.1. Avoiding IP Lookups .....	24
2.2. Hardware Content-Addressable Memories (CAMs) .....	26
2.3. Trie Based Schemes .....	28
2.4. Non-Trie Algorithmic Approaches .....	35
2.5. Summary of Existing IP Lookup Algorithms .....	38
<b>3. TREE BITMAP</b> .....	<b>40</b>
3.1. Description of Tree Bitmap Algorithm .....	40

3.2.	Pseudocode for Tree Bitmap Search .....	45
<b>4.</b>	<b>TREE BITMAP OPTIMIZATIONS .....</b>	<b>48</b>
4.1.	Overview of Optimizations .....	48
4.2.	Initial Array Optimization .....	49
4.3.	End Node Optimization .....	49
4.4.	Split Tree Bitmaps .....	51
4.5.	Segmented Bitmaps .....	53
4.6.	CAM Nodes .....	54
4.7.	Empirical size results .....	55
<b>5.</b>	<b>REFERENCE DESIGN.....</b>	<b>58</b>
5.1.	Overview .....	58
5.2.	Reference Design Framework .....	60
5.3.	Memory Management .....	70
5.4.	Updates .....	76
5.5.	Design Details .....	78
<b>6.</b>	<b>REFERENCE DESIGN ANALYSIS .....</b>	<b>82</b>
6.1.	Empirical Memory Usage Results .....	82
6.2.	Projected Worst Case Memory Usage Analysis .....	85
6.3.	Implementation Analysis .....	88
<b>7.</b>	<b>CONCLUSIONS .....</b>	<b>93</b>
	REFERENCES .....	95
	VITA.....	100

## LIST OF TABLES

1-1. Unicast/Multicast Examples .....	15
1-2. Wire Speed Requirements .....	17
1-3. Discrete Memory Technology Comparison for IP Lookups .....	22
2-1. Expanded Range Table .....	36
2-2. Summary of Existing IP Lookup Schemes .....	38
4-1. Tree Bitmap Size Results (Mae-East) .....	55
4-2. Number of Nodes with Initial Arrays & End Nodes (Mae-East) .....	57
5-1. Representative Node Access Patterns .....	68
6-1. Reference Design Empirical Size Results .....	83
6-2. Analysis of Empirical Results .....	84
6-3. Flip Flop Count in Reference Design .....	88
6-4. SRAM Quantity effect on Die Area .....	90

## LIST OF FIGURES

1-1. Internet Topology .....	2
1-2. Simple Bus Based Router .....	4
1-3. Shared Parallel Forwarding Engines .....	5
1-4. Integrated Router Design.....	7
1-5. Scalable Router-Switch Architecture .....	8
1-6. Forwarding Engine Block Diagram.....	11
1-7. IPv4 and IPv6 Packet Formats .....	12
1-8. Structure of a Lookup Engine Design .....	20
2-1. UniBit Trie Example .....	29
2-2. Controlled Prefix Expansion without Leaf Pushing .....	30
2-3. Controlled Prefix Expansion with Leaf Pushing .....	31
2-4. Example with Lulea.....	32
2-5. Binary Search on Ranges.....	36
3-1. Tree Bitmap with Sample Database .....	42
3-2. Multibit Node Compression with Tree Bitmap .....	42
3-3. Reference Tree Bitmap Data Structure.....	45
3-4. Pseudocode of Algorithm for Tree Bitmap .....	46

4-1. Reference Tree Bitmap Data Structure.....	48
4-2. End Node Optimizations .....	50
4-3. Segmented Bitmaps Optimization.....	53
4-4. Histogram of Prefix Count for End and Internal Nodes (Mae-East).....	54
5-1. Block Diagram of OC-192c IP Forwarding ASIC .....	59
5-2. Block Diagram of IP Lookup Engine Core .....	61
5-3. Initial Array data structure.....	62
5-4. Trie Node Data Structure.....	65
5-5. CAM Node Data Structure .....	65
5-6. Internal/End Node Data Structure .....	66
5-7. Result Node Data Structure .....	66
5-8. Allocation Block Header .....	66
5-9. Node Sequence For a full 32 bit search.....	67
5-10. Programmable Pointer Based Memory Management.....	73
5-11. RTL View of Search Engine .....	79
5-12. Memory Access Patterns .....	80
5-13. Reference Design External Interface.....	81

## ACKNOWLEDGEMENTS

I would like to thank Zubin Dittia who co-invented the initial Tree Bitmap algorithm, and Andy Fingerhut who gave me insight into memory management approaches. I would like to thank my advisors Dr. George Varghese and Dr. Jonathan S. Turner for their help and guidance in the writing of this thesis. I would also like to thank Dr. William Richard and Dr. Fred Rosenberger for their guidance in various roles over the past several years and for serving on my thesis committee. Several people assisted in the review of my thesis and helped me to complete it in a timely fashion, this includes A. Viswanathan and Marcel Waldvogel. Finally, I would like to thank my wife, Michaelle, my son, and my parents all who have supported me in every endeavor.

# 1. INTRODUCTION

## 1.1. Internetworking

An internetwork is a term that applies to any group of networks tied together with a common protocol that runs above the data link protocols of the individual networks. The largest and most heavily used internetwork has come to be known as the Internet. The Internet consists a large number of commercial, academic, and research internetworks, and thus is an “internetwork of internetworks”. The most popular internetwork routing protocol, and the one used currently for the Internet, is the Internet Protocol (IP) version 4. Today the Internet has over 2.8 million domains (subnetworks) and over 36 million hosts [36]. Given the current size of the Internet, and the fact that traffic in the Internet is doubling every 3 months, it is crucial to be able to switch IP messages at increasingly high speeds to an increasing number of possible destinations.

Routers are the switch points of an internetwork. Routers use the IP destination address to determine which of the router outputs is the best next hop for each Internet message (packet). A naive approach would require each router to maintain the best next hop for each possible IP address in existence. To reduce the size of routing tables, groups of IP addresses are aggregated into routes, and each route is represented at a router by a single next hop.

A packet will pass through several different deployment classes of routers from source to destination. Figure 1-1 shows an example topology for a portion of the Internet

with the various classes of routers illustrated. The lowest level router, the enterprise router, is characterized as needing low latency (10's of microseconds), medium-high bandwidth interfaces (100 Mb/s to hosts and up to 1 Gb/s to the enterprise backbone) and small forwarding tables (100's of attached hosts and 10's of network routes). The next level up is the backbone enterprise router. These routers typically have higher bandwidth interfaces, and higher total throughput. The number of hosts and routes supported needs to be in the thousands; the uplink for these routers could be Asynchronous Transfer Mode (ATM) or Packet over Sonet (POS) at rates as high as OC-48 (the Sonet specification for 2.4 Gb/s).

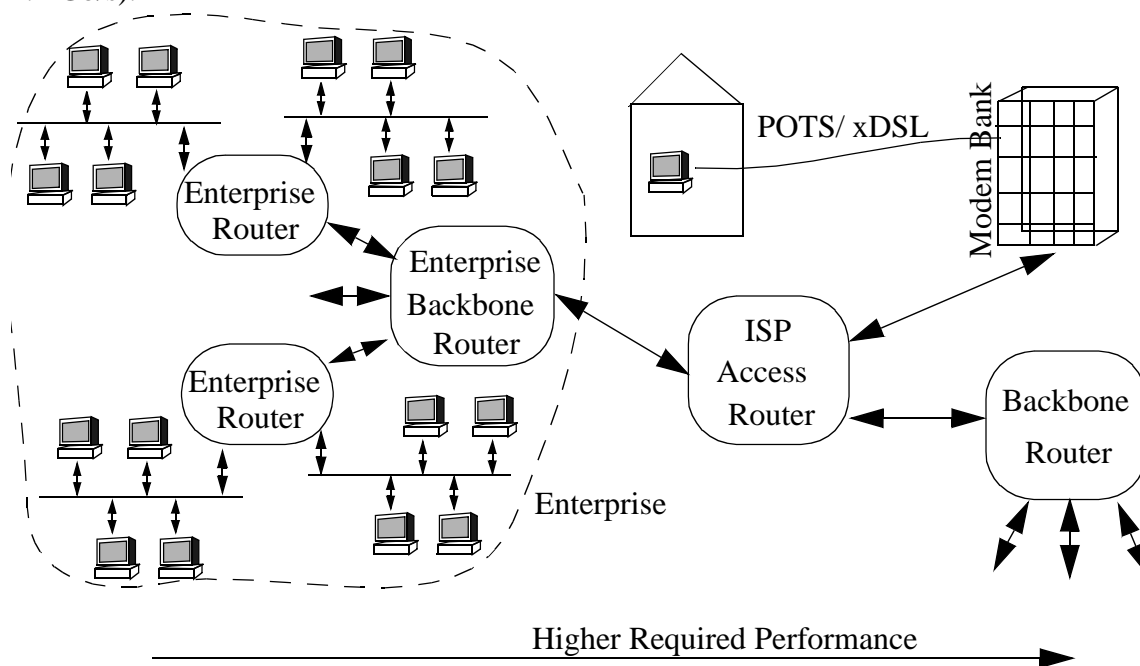


Figure 1-1 Internet Topology

Going further up the hierarchy, the next level is provided by the Access routers of the Internet Service Providers (ISPs). These routers are aggregation points for both corporations and residential services using Plain Old Telephone Service (POTS), or one of the newer Digital Subscriber Loop (DSL) services with higher bandwidth to the home and small office. Access routers must support thousands or tens of thousands of Internet routes. At the high end are the backbone routers of nation wide ISPs. Backbone ISP routers require large numbers of high bandwidth ports (today OC-48c at 2.5 Gb/s is considered

high end, but OC-192c at 9.6 Gb/s is being discussed for the near future), and tens or hundreds of thousands of routes in each forwarding database. There exist routing databases today with more than 50k entries.

This thesis will focus on the design of high end routers for ISP backbones. Specifically, this thesis will explore the problem of Internet address lookup for ISP backbone routers. Before we explore a specific solution, the remainder of this introduction provides background (on router architectures, the router forwarding problem, and memory technologies) that is important for understanding the lookup problem.

## **1.2. Internet Protocol (IP) Router Architectures**

The first several generations of routers were based on general purpose processors, shared memory buffers, and simple bus based switching. Figure 1-2 shows a simple bus based router architecture utilizing general purpose CPUs for routing and forwarding. The family of router architectures using busses and CPUs were able to scale with router bandwidth needs for many years. For switching, the busses got wider, the data rate on each pin of the bus went up, and the architecture was optimized to limit the number of transfers per packet across the bus. For packet forwarding, the forwarding decisions were initially all

made by a central CPU; later, caches of forwarding decisions were kept at each port to provide speedup for a majority of packets.

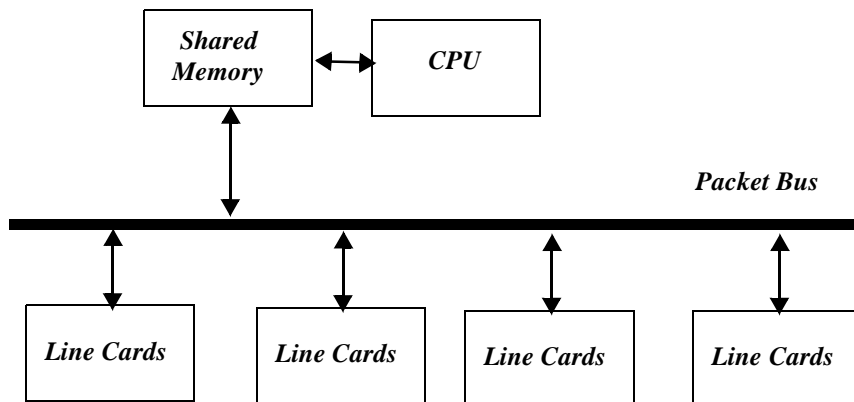


Figure 1-2 Simple Bus Based Router

It became apparent some time ago that bus based routers with general purpose processors were not going to be able to continue scaling at the rate necessary to keep up with exponentially growing bandwidth needs. With the convergence of the networking industry on IP as an internetworking layer, and with advances in IP lookup and packet classification technology, it became feasible to build specialized hardware routers that could meet the bandwidth demands both in the Internet backbone, and in enterprise backbones. The term for routers that employ Application Specific Integrated Circuits (ASICs) and can forward IP packets with the same throughput and deterministic behavior as an Ethernet or ATM switch, is Router-Switch [19][21].

Every Router-Switch needs to incorporate the following functions:

- \* **Packet Forwarding:** Packet Forwarding is the process of looking at each incoming packet and deciding what output port it should go out on. This decision can be much more complex with questions about security, Quality-of-Service (QoS), monitoring, and other functions rolled into the packet forwarding process.

- \* **Packet Switching:** Packet Switching is the action of moving packets from one port interface to a different port interface based on the packet forwarding decision.

\* Queueing: Queueing can exist at the input, in the switch fabric, and/or at the output. Output queueing normally has the most capacity and complexity in order to provide quality of service to traffic in the face of a congested output link.

\* Routing: Routing is the process of communicating to other routers and exchanging route information. Internet protocols that implement routing include RIP, OSPF, BGP, DVMRP, PIM. Routing protocol events can result in modifications to the tables used by the packet forwarding function.

Figure 1-3 shows the architecture of the first Router-Switches. A well documented initial router-switch prototype was the BBN Multi-Gigabit Router [40]. These initial Router-Switches still considered IP forwarding an expensive and unstable algorithm. Therefore this architecture still implemented the actual forwarding algorithm in tight low-level software on a general purpose CPU, but used specialized hardware to optimize the data movement into and out of the address space of the forwarding processor. The IP forwarding engines were placed separately from the line cards to spread the cost of a forwarding engine across several ports, and to simplify the upgrading or modification of the forwarding engines.

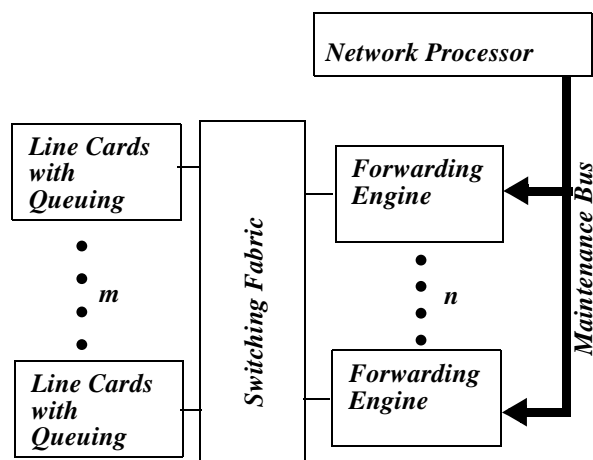


Figure 1-3 Shared Parallel Forwarding Engines

The general idea for this architecture is as follows. Packets enter on the line cards; the line card sends just the data link layer and IP headers to the forwarding engines through the switch fabric. The forwarding engines apply the IP forwarding algorithm to

the packets, and then return next hop information to the line card holding the packet. The packet is finally switched to the appropriate output line card.

The network processor in this architecture has several responsibilities that include:

- \* Running high level routing protocols like BGP, RIP, OSPF, DVMRP, and PIM.
- \* Translating high level protocol events into changes in the forwarding tables of the forwarding engines.
- \* Loading new forwarding tables to the Forwarding Engines through the maintenance bus.
- \* Handling exceptional forwarding cases like trace route IP options, error conditions, etc.

The major flaw in this architecture is that there has been a growing desire to use layers above the inter-networking layer (in the OSI stack this would be levels above 3). Thus the restriction of sending only the data link and internetworking layers was too high. Additionally, there has been increasing pressure for routers to have the ability to forward minimum sized Transmission Control Protocol (TCP) packets at link speed. (This is called wire speed forwarding and is discussed in Section 1.4. on page 16). But forwarding the header of such a minimum size packets followed by the packet itself, can (in the worst case) double the required bandwidth through the switch fabric.

Figure 1-4 is a current popular Router-Switch architecture especially for enterprise backbone routers. The significant change is the move to put ASIC based IP forwarding engines on every port. This move reflects the increasing perception of the IP forwarding algorithm as stable, and a realization that ASICs can do IP forwarding with low cost. An

important point is that this architecture still uses an out of band maintenance bus to provide control between the network processor and the forwarding engines.

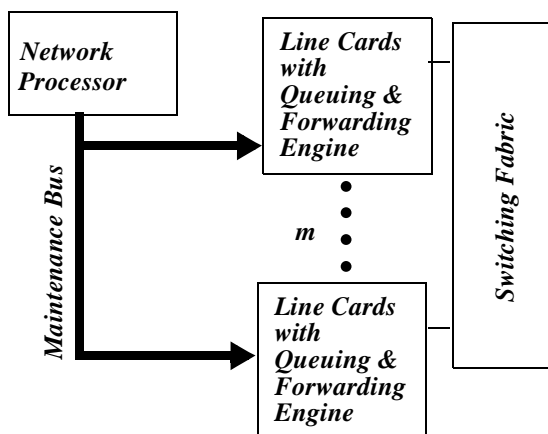


Figure 1-4 Integrated Router Design

Figure 1-5 shows an alternate architecture for Router-Switches that illustrates some interesting points regarding maintenance of the IP forwarding engine. The first thing to note is the change in the control hierarchy for the Router-Switch. In Figure 1-3 and Figure 1-4, a single network processor is responsible for all exceptional/control operations for the entire Router-Switch that can't be handled by the forwarding engines (these were

listed in discussion about Figure 1-3). It is not hard to imagine that this network processor can quickly become overloaded. Therefore in Figure 1-5 a three level hierarchy is shown.

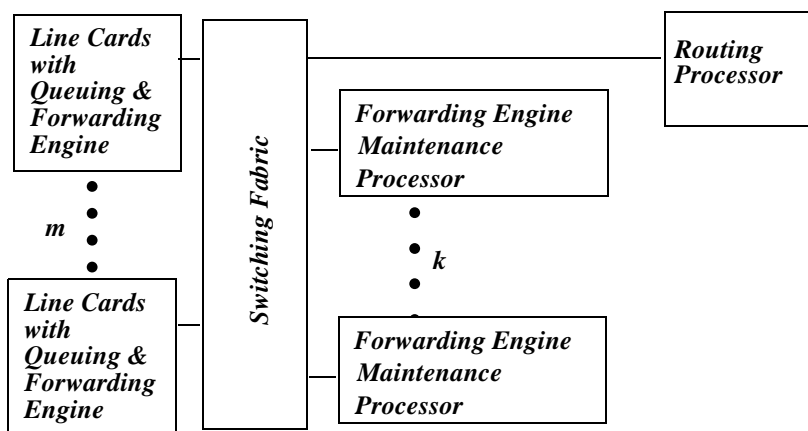


Figure 1-5 Scalable Router-Switch Architecture

In this architecture, a single routing processor is at the top of the hierarchy and runs the unicast and multicast routing protocols, as well as authentication for remote monitoring, management access, etc. Below the routing processor are multiple maintenance processors. The maintenance processors take high-level update commands from the routing processor, and translate these commands into low-level updates to the forwarding engines. This usually requires that the maintenance processor keep local copies of the forwarding tables for each forwarding engine it presides over. In addition to updates, the maintenance processors are responsible for exceptional forwarding cases that the forwarding engines can't handle. A commercial vendor which has implemented a hierarchy similar to the one described here is Berkeley Networks[3].

Another important point is that for a scalable Router-Switch, it becomes technically difficult (if not impossible) to provide a separate high speed control path (referred to as a maintenance bus in Figure 1-4) between one of possibly several forwarding maintenance processors and every forwarding engine. Therefore the answer is to provide in-band control of the forwarding engines. In-band control means that control packets are inter-mixed with the data packets. With in-band control each forwarding engine looks at each

packet that passes through it to check if it is meant to write or read registers within that chip. The major advantage of in-band over out-band is that the same switching network for the data packets can also be used for the control traffic in scalable fashion. When using in-band control it is important to use priorities to insulate control traffic from data traffic, and security mechanisms to insure that devices within the router are not accessed illegally by external devices. This type of in-band control has been implemented in some commercial Router-Switch architectures like the Cisco 1200 series[5].

The point of outlining the scalable Router-Switch architecture is that it has implications on forwarding engine design. The bandwidth between the maintenance processor and the forwarding engine becomes a very important issue since it is part of the switch fabric and because the forwarding engine data path from the switch fabric to the forwarding engine line card contains both data traffic and control traffic. In this situation it is logical to have priorities within the switching fabric itself and to have the maintenance processors pace out control traffic with a maximum bandwidth pacing parameter per output port. The desire to maximize the ratio of maintenance processors to forwarding engines (this would be the ratio of  $k$  to  $m$  from Figure 1-5) and the desire to restrict BW between them puts additional constraints on algorithm design.

Thus IP lookup algorithms are desired that require a minimal amount of maintenance processing and update bandwidth. While this design criteria is strongly illustrated in the above example, it is also a goal of all router-switches in order to maximize update speeds. Note that this goes against a common view (espoused in some research papers) that in a trade-off between forwarding table maintenance complexity and lookup performance, one should always choose lookup performance.

## 1.3. IP Forwarding

To better understand the rest of the thesis, we briefly review some important aspects of IP forwarding. We start with the general concept, and then describe some important aspects of unicast and multicast routing. The next section will provide a precise definition of a commonly desired feature called “wire speed forwarding”.

### 1.3.1. General Concept

Figure 1-6 shows a block diagram of the IP forwarding functionality for an Application Specific Integrated Circuit (ASIC) based router. Each packet is buffered in a small FIFO memory for a short fixed length of time (on the order of a couple of microseconds), while the headers are distributed to multiple classification engines. The classification engines (some times called lookup engines) must be able to accept and process new packet headers at the maximum possible rate that they can be delivered from the data link layer to be considered wire speed. (The advantages for a router to operate the forwarding at wire speed is explored in Section 1.4. on page 16.) After the relevant classifications are completed, the results are presented to the actual forwarding logic. The forwarding logic is simply a state machine based block that follows the IP forwarding algorithm presented in [2]. The result is normally the attachment of an output data link layer header, and also an

internal header with information used by the switch fabric and the output port (e.g., output port number, quality of service designations).

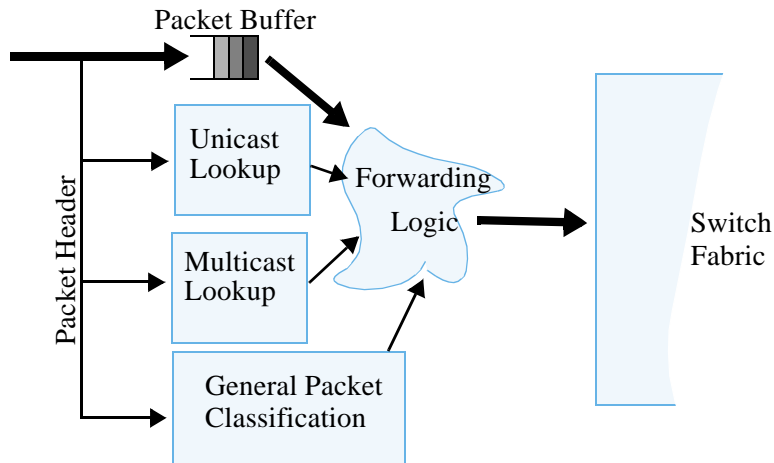


Figure 1-6 Forwarding Engine Block Diagram

Figure 1-7 shows the header format for IPv4 (on the left side) and IPv6 (on the right side). IPv6 is intended to be a replacement for IPv4 in the future with a larger address space and other additional features. However, with additions to the IPv4 protocol to extend its life, it is questionable how long it will take for IPv6 to become widely accepted.

For this reason, this thesis will focus mainly on the IPv4 specification and the problems associated with address lookups of IPv4 addresses.

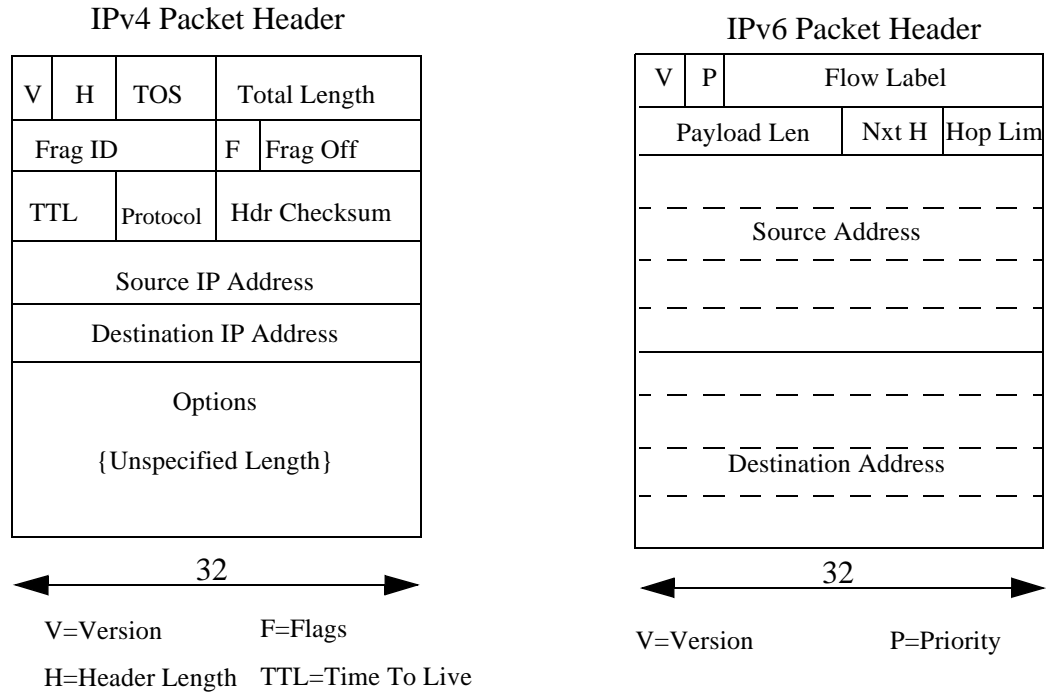


Figure 1-7 IPv4 and IPv6 Packet Formats

For most applications the packet will have a single final destination and is therefore referred to as unicast. Unicast packets are traditionally forwarded based primarily on the destination address field. For IPv4 this field is 32 bits and for IPv6 this field is 128 bits as shown in Figure 1-7. Unicast address lookups are discussed in more detail in Section 1.3.2. on page 13. The other class of routed packets is multicast, with an unspecified number of final destinations. For multicast there are a variety of routing algorithms which have correspondingly different lookup algorithms. Normally multicast packets are forwarded based on the source network and the destination group address. Multicast forwarding and lookups are discussed in more detail in Section 1.3.3. on page 14.

The block marked general packet classification in Figure 1-6 is used for a wide variety of applications like security filters, multi-layer switching (e.g. picking an output port based on fields from the IP and TCP headers), and monitoring/logging. General filters are the most difficult to implement since they require classification of a packet header

based on a matrix of fields such as IP source address, IP destination address, protocol type, application destination port, and application source port. The main complexity of general filters is the fact that both the source and destination address fields in the filter can be prefixes. This thesis will not further explore general packet classification but two recent works discussing packet classification in detail are [21][44].

### **1.3.2. Unicast Forwarding**

For traditional unicast forwarding, the decision of which output port of a router a packet should go out on is based only on the destination IP address in the packet. If the IP addresses were small (say 16 bits) a complete table could be constructed at every router with all possible destination IP addresses and corresponding output ports. However, this would severely limit the growth of the Internet. Thus a 32 bit IP address is used in IPv4, and a 128 bit address is used in IPv6. Still, a table could contain every assigned IP address by the use of hashing techniques. This would, however, still require millions of entries since there are millions of hosts. To reduce database size it is desirable to group networks of computers. This can be done by assigning these networks contiguous IP addresses in power of 2 sizes, and then representing the “group” by the common prefix.

The Internet has always used prefixes to aggregate groups of hosts into networks. However, initially the grouping was a simple hierarchy with the prefixes being either 8, 16 or 24 bits in length (Class A, B, and C respectively). As the number of hosts grew, this rigid assignment proved to be restrictive. Specifically, the 16k Class B addresses (too large for most sites, but less restrictive than the 256 hosts of a Class C address) began to run out.

Eventually the aggregation of hosts into network prefixes became flexible to the point that network addresses can now be any length[20]. The terms for the move from the three simple classes to the free range of prefix lengths are supernetting and subnetting[7].

The disadvantage of this new scheme is that the length of the prefixes cannot be deduced from the addresses themselves. This makes the address lookup problem more complex.

### **1.3.3. Multicast Forwarding**

Multicast has traditionally been reserved for special purpose enterprise applications. For applications in which one source is distributing data to many destinations (one-to-many multicast, useful for things like video program broadcasts), and when groups of computers are all sending to each other (many-to-many, for instance in a video conference), multicast can minimize the bandwidth and state kept in the network as a whole. In recent years there has been a growing demand for wide-spread deployment of multicast forwarding not only in the enterprise, but also across the backbone of the Internet.

There are two main types of multicast routing, sender based and receiver based [20]. Sender based is efficient for small groups but does not scale well and therefore is not expected to be deployed in the backbone of the Internet. Examples of the sender-based approach are Stream protocol version II (ST-II), Xpress Transport Protocol (XTP), Multicast Transport Protocol (MTP), and the ATM UNI 3.X. There are also a family of routing protocols that are receiver based and are collectively referred to as IP Multicast.

For IP Multicast, the senders are not aware of the multicast group size or topology. New receivers join a multicast tree by having the closest branch split off a new copy of the data stream to the new receiver without sender involvement. There are two main categories of multicast routing and forwarding, source based trees (e.g. DVMRP, PIM-Dense Mode, and MOSPF) and shared trees (e.g. Core Based Trees, and PIM-Sparse Mode). With source based trees, a tree is constructed with the multicast routing protocol for every source. Source based trees optimize the latency and total network bandwidth for every source tree, but given that every sender must have a distribution tree for every multicast group it is part of, the amount of state in the network grows quadratically. For this and

other reasons it is questionable if source based trees (more specifically the DVMRP protocol) will continue to dominate as multicast groups span backbone networks.

The alternative to source based trees is shared trees where every multicast group has a single distribution tree. In some ways shared based trees are more scalable; for example there is less state in the network with shared trees. However, with shared trees there are potentially bandwidth bottlenecks created because many sources share a common tree. The shared tree may also result in sub-optimal routing compared to per-source trees.

For the forwarding of IP multicast packets, the routing algorithm being used does affect the state kept, and the method of accessing that state. Table 1-1 illustrates an example forwarding table for unicast, and both shared and source tree IP multicast. For this example the IP address is assumed to be 8 bits total in length. For the unicast case, a 4 bit destination address prefix is used. For the shared tree, only the multicast group, which is a full length multicast address (not a prefix) is used for matching a packet with this forwarding entry.

**Table 1-1 Unicast/Multicast Examples**

Filter Type	Destination IP Address	Source IP Address	Destination & Source
Unicast	1011*	*	1011*
Shared Tree Multicast	11111110	*	11111110*
Source-Based Tree (DVMRP)	11110000	1010*	111100001010*
Source-Based Tree (MOSPF)	11001111	01010101	110011110101010

For the third entry, packets matching against a source based tree use the multicast group and the prefix of the source network. The final entry is also a source based tree, but the MOSPF protocol has a separate forwarding entry for every source address rather than for source prefixes. The right most column of Table 1-1 shows the destination and source address fields concatenated together for each entry. Since there is never a prefix in both the source and destination address fields, the forwarding problem for unicast and all the

various multicast protocols can be viewed as a single longest match problem. For IPv4 this longest prefix match is over 64 bits. While a lookup engine doing both unicast and multicast lookups would roughly double the rate of memory accesses (compared to doing just destination address lookups), it is necessary to forward multicast packets at wire speed exactly as in the unicast case.

There have been suggestions in the past for new hierarchical multicast protocols like Hierarchical Distance Vector Multicast Routing (HDVMRP)[20]. Hierarchical multicast protocols would require filters with both source and destination IP address prefixes. However, the Internet drafts for these schemes have long since expired and there are no current proposals for multicast protocols requiring prefixes in both the source and destination address fields. Note that if prefixes did exist in both, the problem would fall under the category of general packet classification[23][44].

## 1.4. Wire Speed

The term “wire speed” means that packet forwarding decisions can be made at the maximum rate that packets can arrive on a given data link layer. The desire for wire speed performance is due to the fact that in a study of Internet-traffic characteristics it was found that 40% of all traffic is TCP/IP acknowledgment packets[22]. An independent study of a trace of an OC-3c backbone link of the MCI network in June of 1997 also showed that 40% of all packets were 40 bytes in length[36]. Also 50% of all packets were less than 44 bytes in length. If a router is not able to process minimum length packets, then the delay at input to the forwarding engine is dependent on the traffic characteristics. This is undesirable since no information is known about the packets as they await processing, therefore all packets must be treated as best effort and FIFO queueing must be used. Also hardware based port design is simpler with deterministic forwarding. Finally, with wire speed

forwarding, an input port can require negligible buffering (depending on the fabric properties).

The maximum packet rate of a given data link level (level 2 in the OSI stack) technology can be limited either by a minimum frame size (like 64 byte ethernet frames), a multiple of a fixed payload block size (48 byte payloads of ATM cells), or by the size of a level 2 encapsulated TCP/IP acknowledgment if no minimum level frame size exists. Table 1-2 shows maximum wire speed forwarding rates for IPv4 and IPv6 over Gigabit Ethernet[16] and two layer 2 technologies running over SONET [4]: IP over PPP (Point to Point Protocol) over Sonet (also known simply as IP over SONET, or Packet over SONET)[25], and IP over ATM over SONET. Note that while PPP as a data link layer technology will probably not scale to OC-192c rates, the estimate of a similar protocol with an 8-byte header is a reasonable one at this time.

**Table 1-2 Wire Speed Requirements**

Data Link Layer	Raw Data Rate (gbps)	User Data Rate	IPv4 Millions of lookups/sec	IPv6 Millions of lookups/sec
Gigabit Ethernet	1.25	1	1.735	1.453
OC-48c ATM/SONET	2.488	2.405	5.672	5.672
OC-48c IP/PPP/SONET	2.488	2.405	6.263	4.420
OC-192c ATM/SONET	9.953	9.621	22.691	22.691
OC-192c IP/PPP/SONET	9.953	9.621	25.054	17.6856

Gigabit Ethernet has a usable bandwidth of 1 gigabit/s. The minimum frame size is 64 bytes, with 18 bytes of overhead per frame and an 8-byte gap between frames.

SONET is a complex protocol and has overhead prior to the data link layer of PPP or ATM<sup>1</sup>. PPP frames have no minimum size and an overhead of 8 bytes per packet. The ATM overhead (we assume the worst case for lookup requirements which is simple IP over AAL5 frames allowing a complete TCP/IPv4 acknowledgment per cell) is any loss due to fixed size cells and 5 bytes for every 48 bytes of payload. For IPv6 the size of the

---

1. For concatenated OC-n, User Rate =  $GrossRate \times \{(87 \div 90) - (1/(90 \times n))\}$

TCP/IPv6 packet is always the limit to maximum packet rate. For simple IPv6/ATM the actual IP packet throughput is actually half that of IPv4/ATM since an IPv6/TCP ACK requires two cells. However, with multiplexed Virtual Circuits it is possible that a long stream of back to back cells will each cause packet reassembly completion and therefore require forwarding at a rate equivalent to the cell rate.

## 1.5. Memory Technology

Memory technology affects IP lookups much more than any other technology factor. The rate of random memory accesses and the bandwidth to memory together set basic limits on the lookup rate. Thus it is crucial to discuss memory technology in detail as it will affect the design of the lookup algorithms discussed later in the thesis. In this section, both on-chip and discrete memory technologies are explored.

### 1.5.1. On-Chip ASIC Memory

Recall that ASICs are custom Integrated Circuits (IC) for (say) IP lookups. For several reasons it may be advantageous to use on-chip as opposed to off-chip memory. Some of these reasons include: 1) memory latency accesses are longer when going off-chip forcing deeper pipelines), 2) most designs are normally pad limited (this means the design is limited by the maximum number of pins on a chip, not logic on the chip), and 3) bandwidth to on-chip memory can be higher than going off-chip. In this section, several commercial vendors of SRAM generators are investigated to quantify the current state of SRAM ASIC technology.

In the IBM SA-12 process (quarter micron drawn process) SRAMs of the size 1024x128 and 4096x128 have ratios of bits to area at consistently about 59 Kbits per

square mm[11]. This means that 60% of a 11 mm by 11 mm die (with 10% overhead for wiring/spacing) can fit around 3.8 Mbits of SRAM. Note the wiring overhead may be higher, or the percentage of die devoted to SRAM may be smaller. Also note that for the SRAM sizes mentioned above, the IBM data sheets quote a worst case SRAM cycle rate of a little over 200 Mhz (a 5 ns period). The IBM SA-12 process will be assumed for the reference design analysis in Section 6. on page 82.

Other commercial vendors for SRAM generators have similar size and speed RAMS. Inventra provides details on their SRAMs targeting the TSMC quarter micron drawn process (3 metal layers)[18]. For 4096x36 SRAMs the density is 87 kbits per square mm and the operating speed is over 200 Mhz.

Another vendor of SRAMs is Artisian Components [1]. This vendor claims speeds as high as 300 Mhz in the quarter micron drawn process. However, the amount of data given about these SRAMs is not enough to evaluate designs based upon these SRAMs.

On-chip DRAM is becoming a standard option from several ASIC foundries, and it appears that in the future it may be as ubiquitous as on-chip SRAM. The promise of embedded DRAM is that it will provide high density (for example 32 Mbits of DRAM with half the die), and high bandwidth interfaces (with interfaces several hundred bits wide, and access times as low as 30-40 ns). However, currently processes that mix DRAM and Logic are not yet mainstream, and the performance of the logic gates can suffer compared to a typical logic process.

### **1.5.2. Discrete Memory Technologies**

Besides the use of conventional off-chip SRAM and DRAM, the major RAM technologies today and in the near future are PC-100 SDRAM [15][17][31], DDR-SDRAM[13][28], SLDRAM[41][30], Direct Rambus[10][12], and Pipelined Burst Synchronous SRAM[32]. Efficient use of the DRAM based memory technologies are based on the idea of memory interleaving to hide memory latency. Consider for example

Figure 1-8 which shows an IP Lookup ASIC that interfaces to a PC-100 SDRAM. The figure shows that the SDRAM internally has two DRAM banks, but a single interface between the ASIC and the SDRAM. The basic picture for other discrete memory technologies is similar except that the number of banks, the width of the interface, and the data rate of the interface can all vary.

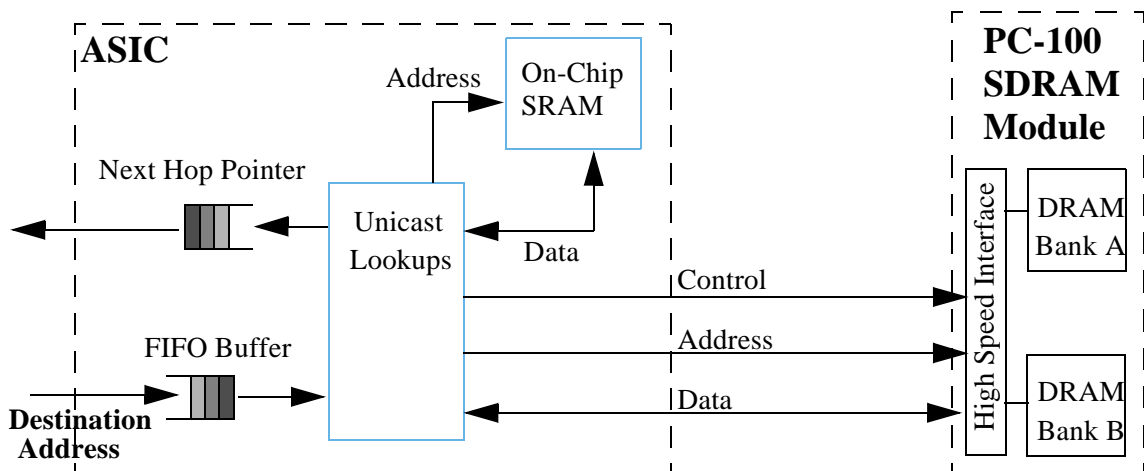


Figure 1-8 Structure of a Lookup Engine Design

The FIFO buffer of destination addresses is needed to allow some pipelining of destination address lookups to fully utilize the memory bandwidth. For example, assume that lookups require 4 accesses to a data structure, starting with a root node that points to a second level node, to a third level node, to a fourth level node. If we have two banks of memories, we can place the first two levels of data structure in the first memory bank and the third and fourth in the second memory bank. If we can simultaneously work on the lookup for two IP addresses (say D1 followed by D2), while we are looking up the third and fourth level for D1 in Bank 2, we can lookup the first and second levels for D2. If it takes 40 nano-second per memory access, it still takes  $4 * 40 = 160$  nano-second to lookup D1. However, we have a higher throughput of two IP lookups every 160 nano-second.

For an algorithm designer, we wish to abstract the messy details of these various technologies into a clean model that can quickly allow us to choose between algorithms.

The most important information required is as follows. First, if we wish to have further interleaving for more performance we need more memory interfaces and so more

interface pins. Because interface pins for an ASIC are a precious resource, it is useful to quantify the number of memory interfaces (or pins) required for a required number of random memory accesses. Other important design criteria are the relative costs of memory per bit, the amount of memory segmentation, and the optimal burst size for a given bus width and random access rate that fully utilizes the memory. As far as cost, when in commodity production, the price of DRAM becomes almost completely based on the process being used and size. The interface complexity quickly becomes negligible in the price.

For example, at the time this thesis is being written 64 megabyte 100 Mhz SDRAM modules are actually cheaper than standard 64 megabyte EDO DRAM. Also at the time of this writing, 200 Mhz synchronous SRAM is about 20 times more expensive than 100 Mhz SDRAM per bit. In the not to distant future it is expected that SLDRAM may be about 15% more expensive than 100 Mhz SDRAM, and Rambus may be 20% more expensive [29]. This clearly illustrates that while all DRAM products have similar price per bit, SRAM is over an order of magnitude more expensive than the most exotic DRAM technology.

An additional design criterion relates to the fact that memory technologies have standard memory module packaging. To make scalable designs, these memory module characteristics (most importantly the data path width) must be taken into account. When discrete memory chips are soldered directly onto the Printed Circuit Board (PCB) to customize the data bus width, it is similar to on-chip memory in its inflexibility. SDRAM and DDR-SDRAM come in 64 bit wide modules, Rambus is 16 bits wide, synchronous SRAM is typically 32 bits (for processor cache memory), and SLDRAM will probably be available in 32 bit wide memory modules.

Table 1-3 shows a table comparing the more quantifiable differences between the discrete memory technology choices. In the first column is the name of each memory technology, the bandwidth per data pin, and the standard data path width. In the second column is the number of ASIC pads required to interface to the standard memory width of that technology. Note that in the case of Rambus, the number of equivalent pads is significantly larger than the number of actual pins since the Rambus core is in the pad ring. The

third column gives the logical number of banks needed to maximize the number of random memory accesses. Note that in the case of Rambus the number of logical banks needed (8) is half the number of physical banks(16). This is because the time between accessing the same bank is 70ns which means that with 8 logical banks random memory accesses can be maximized. The number of logical banks is important because (for technologies based on DRAM banks) it controls the degree of pipelining required to fully utilize the memory bandwidth (while maximizing number of random accesses). The number of logical banks used is also important because it indicates memory segmentation which will have an effect on worst case table size bounds.

**Table 1-3 Discrete Memory Technology Comparison for IP Lookups**

Memory Technology with (Data path Width, BW per data pin)	ASIC Pads	Logical # of Banks	# of Random Memory Accesses every 160 ns	ASIC Pins/ Random Memory Access	Block Size (bytes)	Cost (Normalized)
PC-100 SDRAM (64 bit, 100 Mbps)	80	2	4	20	32	1.00
DDR-SDRAM (64 bit, 200 Mbps)	80	4	8	10	32	1.05
SLDRAM (32 bit, 400 Mbps)	50	8	16	2.125	16	1.15
Direct Rambus(16 bit, 800 Mbps)	76	8 <sup>a</sup>	16	4.75	16	1.30
External SyncSRAM(32 bit, 200 Mbps)	52	1	32	1.625	4	20

a. Direct Rambus actually has 16 or 32 banks, but due to the fixed burst size of 8 data ticks, we only need to logically segment the Rambus memory into 8 banks.

The fourth column shows the number of random memory accesses possible for OC-48 minimum packet times (160 ns). One cost measure of these random accesses is given in the fifth column by showing the ratio of ASIC pins to random memory accesses. The sixth column shows the optimal burst sizes when accessing the memory at the highest random rate possible. The final column shows the normalized cost per bit of each memory technology with PC-100 SDRAM as the baseline.

Why should an IP lookup ASIC designer care about this table? First, the designer can use this table to choose between technologies based on dollar cost and pins/random accesses. Having chosen a technology, the designer can then use the table to determine the optimal burst size. The data structure can then be designed to have every access to the data structure access the optimal burst size. Using bigger burst sizes will be suboptimal (would require more banks or more accesses), and not utilizing the full burst size would use the memory system inefficiently. For example, we observe from the table that the optimal block size for DRAM based memories (e.g., 32 bytes for PC-100 SDRAM) is much larger than for SRAM based technologies (4 for SRAM). For example, this indicates that for DRAM implementations we should use algorithms that minimize the number of memory accesses but use wide data structures.

## **1.6. Overview of Rest of Thesis**

Chapter 2 describes related work in the area of IP address lookups. Chapter 3 presents the Tree Bitmap algorithm for IP lookups. Chapter 4 elaborates on the Tree Bitmap algorithm with optimizations to reduce storage, improve worst case analysis, and provide the ability to tweak the algorithm for different implementation constraints. Chapter 5 describes a reference design for doing IP lookups using the tree bitmap algorithm. Chapter 6 gives performance analysis and circuit complexity estimation. The last chapter summarizes the thesis and gives suggestions for future work.

## 2. Related Work

There are numerous methods to approaching the problem of address resolution for IPv4 forwarding. For each approach, the advantages and disadvantages are discussed in terms of system implementation for lookup performance, storage, and update complexity. There are four broad classes of solutions as described in this chapter. Avoiding IP Lookups, Hardware CAMs for prefix search, trie based algorithmic solutions and non-trie based algorithmic solutions. The algorithmic solutions use standard SRAM or DRAM memories.

### 2.1. Avoiding IP Lookups

#### 2.1.1. Caching

First, the entire IP lookup problem would go away if we could cache mappings from 32 bit addresses to next hops in line cards. However, caching has not worked well in the past in backbone routers because of the need to cache full addresses (it is not clear how to cache prefixes). This potentially dilutes the cache with hundreds of addresses that map to the same prefix. Also typical backbone routers may expect to have hundreds of thousands of flows to different addresses. Some studies have shown cache hit ratios of around 80 percent (possibly caused by web traffic that has poor locality due to surfing). Hence most vendors today are aiming for full IP lookups at line rates.

A recent lookup scheme based on caching is the Stony Brook scheme [46]. They use a software scheme but integrate their lookup cache carefully with the processor cache to result in high speeds (they claim a lookup rate of 87.7 million lookups per second when every packet is a cache hit.). While they showed good cache hit rates on a link from their intranet to the Internet, they provide no evidence that caching would work well in backbones. There are also no worst case numbers, especially in the face of cache misses.

### **2.1.2. Label Switching**

There have been numerous proposals for and commercial implementations of label switching. The basic concept behind all of the label switching approaches is that instead of forwarding packets across the entire Internet based on the final address, a series of labels are used along the way. Normally, a label represents a tunnel that can pass through one or more routers. At each router, the label is used to simply index into a table that will provide output port, QoS, and possibly a replacement label. There are two methods for establishing these tunnels, either control driven or data driven establishment [47].

The most well known implementation of data driven label switching is the Ipsilon Flow Management Protocol (IFMP). Based on traffic patterns and traffic type (ftp traffic versus telnet traffic), a background software process can decide to initiate an ATM virtual circuit set-up between the upstream router and itself. If an entire ATM cloud is implementing the IFMP protocol it is possible for an IP packet to pass through the cloud as switched cells which reduces latency, and load on the forwarding engines throughout the network. Note that the IP forwarding efforts at the edge of the network are unchanged in complexity. In fact some argue that these schemes simply push the complexity to the edges, since the edges are connected at the IP level to a much larger number of neighbors than if every hop implemented full routing and forwarding.

For control driven establishment of label switched paths, MPLS (Multi-Protocol Label Switching) is the most popular protocol and is currently undergoing standardization

by the IETF. With MPLS labels are distributed either external to unicast and multicast routing protocols (via the RSVP mechanisms) or label distribution and routing are combined (with the Label Distribution protocol or LDP) [8]. Labels can represent IP prefixes, Egress Routers, or an application flow.

In summary, label switching approaches do not remove the need for IP lookups completely from the network they simply push the lookups to the edges. Also there are some questions whether label switching will find wide spread acceptance. Label switching adds new and unproven protocols to an already fragile system. Additionally, with current advances in hardware forwarding without the use of labels, it is unclear what advantages label switching brings to the picture.

## **2.2. Hardware Content-Addressable Memories (CAMs)**

CAMs are special memories in which accesses are done by comparing a search value against all the entries in the CAM simultaneously [35]. The order of the CAM entries imply priority such that if multiple CAM entries match the search string, the entry closest to the “high priority” end of the CAM is selected. The return value of a CAM is normally the address of the best match; some CAMs also have an internal SRAM memory to store associative data which is indexed by the winning CAM address. For a given CAM memory, the width of the entries scales linearly (and without any inherent implications on speed). The depth of a CAM is not as simple a scaling proposition. Since each compare must be done across all CAM entries, each CAM entry adds additional delay in the rate of CAM searches.

From a systems perspective, CAMs can normally be cascaded to create deeper tables. This does require the CAM memories have hooks for this cascading or for the use

of an additional priority decoder to choose among the multiple CAM memories. Cascading CAMs to extend the width of the compares is not possible since the cascaded CAMs would need an interconnection for every entry.

Traditionally CAM entries have been binary --- that is each bit in the search value is either a '1' or a '0'. An entry either matches or it doesn't. Recently to support prefix searching, ternary CAMs have become more popular. Ternary CAMs have a third additional state of "Don't Care" for each bit using a mask. To use Ternary CAMs for IP lookups, all that is necessary is to organize the CAM memory from shortest to longest entries so that the longer prefixes have higher CAM priorities. Netlogic Components and Music Semiconductor are two companies that are commercializing CAM technology for networking applications [34][37].

Music has announced a Ternary CAM for doing IP longest prefix matching called the MUAC. The MUAC can do 32 bit longest prefix match at the rate of 14.3 million per second (each search takes 70 ns). The table depth will go up to 8k entries in a single CAM memory, and hooks are provided to cascade up to 4 MUAC chips together providing a 32k entry table. For each entry the MUAC provides the 32 bit compare value, a 32 bit mask (to provide the don't care functionality), and a single bit valid field to indicate to the priority decoding logic if this entry participates or not. Each search returns the index of the highest priority match which is output on a separate set of pins from which the search value is placed on. The priority is based on location in the table as in most CAMs. The fact that priority is not directly tied to prefix length (as is the case for IP address lookups) means a processor must maintain careful control over the CAMs organization and juggle entry locations when a new entry needs to be inserted. There are no associative values stored in the MUAC but the output index could directly drive the address pins of an SRAM containing associative data.

Netlogic has announced a Ternary CAM called the IPCAM which will range from 1k to 8k entries and operates at 20 million searches per second. Each of the IPCAM entries is 128 bits wide instead of 32 bits. The wider entry allows simultaneous longest prefix match searches to be done on both IP source and destination addresses. Additional fields

(for example the application port numbers) can also be added into the search. The IPCAM outputs the index number of the best matching entry (which implies entry juggling for insertions) but also allows a portion of each CAM entry to be used as associative data instead of compare data. A final interesting note about Netlogic's CAMs is that they are offering them as a hard core for networking vendors to include in their ASICs.

In summary, modern CAMs have low latency (in both examples there is no required pipeline latency), deterministic and high lookup rates, and deterministic number of entries. While updates do require entry moving, and micromanagement, there are update algorithms for CAMs that bound the delay for any insertion to a reasonable number [33]. The main drawback of CAMs is that the deterministic number of entries (8k for non cascaded and up to 32k cascaded) is enough for enterprise routers but is no longer enough for backbone routers. The commonly cited Mae-East database has over 40 thousand entries and is still growing [27]. A second disadvantage is that the lookup rate for CAMs is determined by technology and the table size. Even the highest speed CAM reviewed (20 million lookups per second) is not sufficient for OC-192c wire speed rates (25 million lookups per second) as calculated in Section 1.4. on page 16.

## **2.3. Trie Based Schemes**

### **2.3.1. Unibit Tries**

For updates, the one bit at a time trie (unibit trie) is an excellent data structure. The unibit trie for a sample database is shown in Figure 2-1. With the use of a technique to compress out all one-way branches, both the update times and storage needs can be excellent. Its main disadvantage is its speed because it needs 32 random reads in the worst case. The resulting structure is somewhat different from Patricia tries which uses a skip count

compression method that requires backtracking. For example imagine a search for the address '1110100', this address matches P1, P2, P5, and P7. The goal of prefix matching is to find the longest match. For this example, P7 matches 5 bits of the address which is the largest number of match bits.

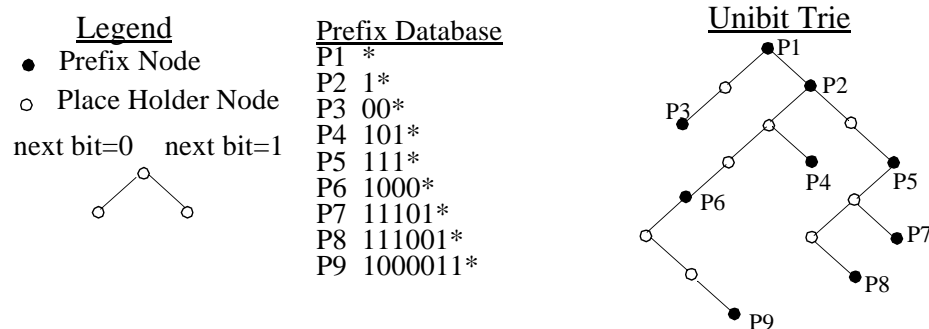


Figure 2-1 UniBit Trie Example

### 2.3.2. Controlled Prefix Expansion (CPE)

For CPE [43], the main idea is to use tries that go several bits at a time (for speed) rather than just one bit at a time as in the unibit trie. Suppose we want to do the database in Figure 2-1 three bits at a time. We will call the number of bits we traverse at each level of the search the “stride length”.

A problem arises with prefixes like P2 = 1\* that do not fit evenly in multiples of 3 bits. The main trick is to expand a prefix like 1\* into all possible 3 bit extensions (100, 101, 110, and 111) and represent P1 by four prefixes. However, since there are already prefixes like P4 = 101 and P5 = 111, clearly the expansions of P1 should get lower preference than P4 and P5 because P4 and P5 are longer length matches. Thus the main idea is to expand each prefix of length that does not fit into a stride length, into a number of prefixes that fit into a stride length. Expansion prefixes that collide with an existing longer length prefix are discarded.

Figure 2-2 shows the expanded trie for the same database as Figure 2-1 but using expanded tries with a fixed stride length of 3 (i.e., each trie node uses 3 bits). Notice that each trie node element in a record has two entries: one for the next hop of a prefix and one for a pointer. (Instead of showing the next hops, we have labelled the next hop field with the actual prefix value in Figure 2-2.) We need to have both pointers and next hop information. Consider for example entry 100 in the root node. This contains a prefix (P1 = 100) and must also contain a pointer to the trie node containing P6 and pointing to P9.

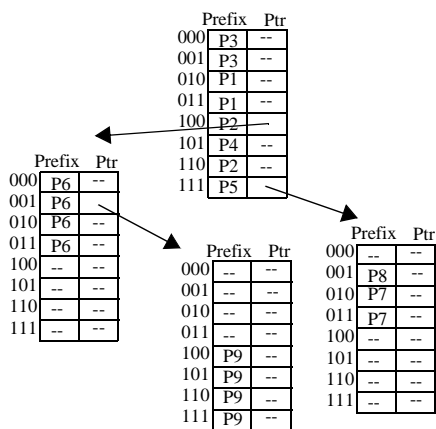


Figure 2-2 Controlled Prefix Expansion without Leaf Pushing

Notice also the down side of expansion. Every entry in each trie node contains information. For example, the root trie node has its first two elements filled with expansions of P3, and the next two entries with expansions of P1. In general, using larger strides requires the use of larger trie nodes with more wasted space. Reference [43] does show how to use variable stride tries (where the strides are picked using dynamic programming to reduce memory) while keeping good search times. Variable stride lengths are not appropriate for hardware design with the goal of deterministic lookup times (wire speed). Also, even with dynamic programming and variable length strides, the best case storage requirements for the Mae East database (after the most sophisticated optimizations) are around 500 Kbytes.

One way to improve on the storage requirements of CPE is to subject the expanded trie scheme to the following optimization that we call “leaf pushing”. The idea behind leaf

pushing is to cut in half the memory requirements of expanded tries by making each trie node entry contain either a pointer or next hop information but not both. Thus we have to deal with entries like 100 in the root node of Figure 2-2 that have both a pointer and a next hop. The trick is to push down the next hop information to the leaves of the trie. Since the leaves do not have a pointer, we only have next hop information at leaves [43]. This is shown in Figure 2-3. Notice that the prefix P2 associated with the 100 root entry in Figure 2-3 has been pushed down to several leaves in the node containing P6.

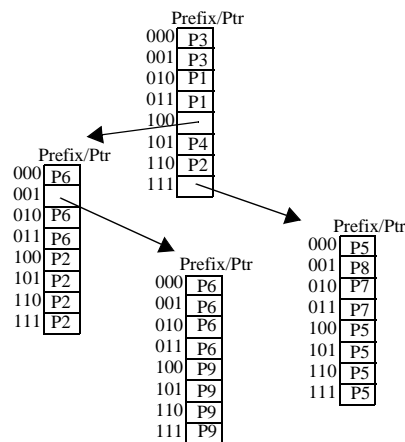


Figure 2-3 Controlled Prefix Expansion with Leaf Pushing

### 2.3.3. Lulea Compressed Tries

The Lulea scheme [6] does much better in memory storage than any variant of CPE, using only 200 Kbytes of memory to store the Mae East database.

The Lulea scheme starts with a conceptual leaf pushed expanded trie and (in essence) replaces all consecutive elements in a trie node that have the same value with a single value. This can greatly reduce the number of elements in a trie node. To allow trie indexing to take place even with the compressed nodes, a bit map with 0's corresponding to the removed positions is associated with each compressed node.

For example consider the root node in Figure 2-4. After leaf pushing the root node has the sequence of values P3, P3, P1, P1, ptr1, P4, P2, ptr2 (where ptr1 is a pointer to the trie node containing P6 and ptr2 is a pointer to the trie node containing P7). After we replace each string of consecutive values by the first value in the sequence we get P3, -, P1, -, ptr1, P4, P2, ptr2. Notice we have removed 2 redundant values. We can now get rid of the original trie node and replace it with a bit map indicating the removed positions (10101111) and a compressed list (P3, P1, ptr1, P4, P2, ptr2). The result of doing this for all three trie nodes is shown in Figure 2-4.

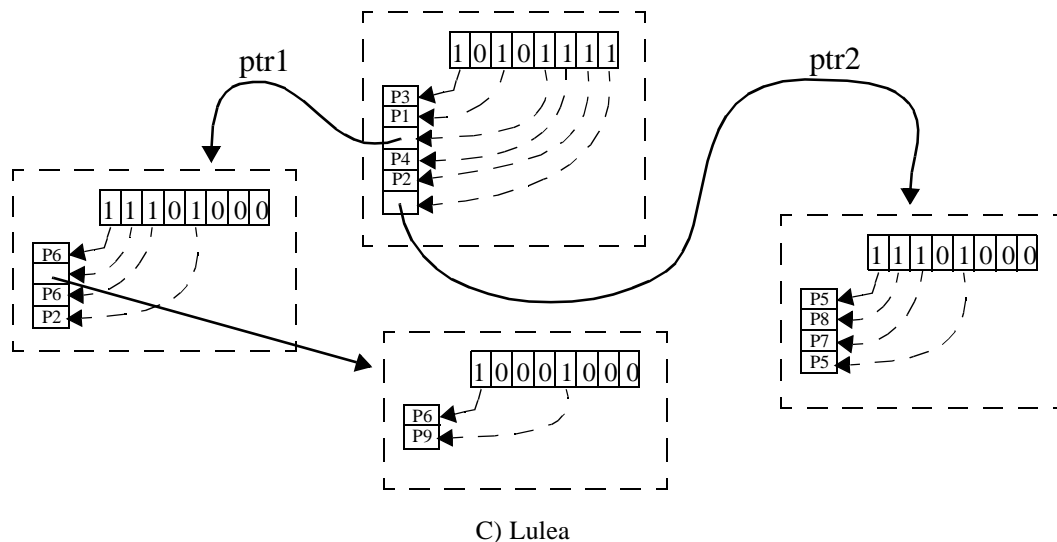


Figure 2-4 Example with Lulea

The search of a trie node now consists of using a number of bits specified by the stride (e.g., 3 in this case) to index into each trie node starting with the root, and continuing until a null pointer is encountered. On a failure at a leaf, we need to compute the next hop associated with that leaf. For example, suppose we have the data structure shown in Part C and have a search for an address that starts with 111111. We use the first three bits (111) to index into the root node bit map. Since this is the sixth bit set (we need to count the bits set before a given bit), we index into the sixth element of the compressed node which is a pointer to the right most trie node. Here we use the next 3 bits (also 111) to index into the eighth bit. Since this bit is a 0, we know we have terminated the search but

we must still retrieve the best matching prefix. This is done by counting the number of bits set before the eighth position (4 bits) and then indexing into the 4th element of the compressed trie node which gives the next hop associated with P5.

The Lulea scheme is used in [6] for a trie search that uses strides of 16, 8, and 8. Without compression, the initial 16 bit array would require 64K entries of at least 16 bits each, and the remaining strides would require the use of large 256 element trie nodes. With compression and a few additional optimizations, the database fits in the extremely small size of 200 Kbytes.

Fundamentally, the implicit use of leaf pushing (which ensures that only leaves contain next hop information) makes insertion inherently slow in the Lulea scheme. Consider a prefix P0 added to a root node entry that points to a sub-trie containing thirty thousand leaf nodes. The next hop information associated with P0 has to be pushed to thousands of leaf nodes. In general, adding a single prefix can cause almost the entire structure to be modified making bounded speed updates hard to realize.

If we abandon leaf pushing and start with the expanded trie of Figure 2-2, when we wish to compress a trie node we immediately realize that we have two types of entries, next hops corresponding to prefixes stored within the node, and pointers to sub-tries. Intuitively, one might expect to need to use two bit maps, one for internal entries and one for external entries. This is indeed one of the ideas behind the tree bit map scheme presented in Section 3. on page 40.

### **2.3.4. Stanford Hardware Trie Lookups**

The only literature thus far with discussion of hardware implementation is from Stanford [9]. The Stanford scheme is actually the controlled prefix expansion algorithm with a 24 bit first stride and an 8 bit second stride. There are several schemes put forth for updates. One such scheme is to have dual table memories and load one, while using the other. For this scheme using 100 MHZ SDRAM and linearly loading 32 MB of memory,

will take 160 ms. Therefore this solution is slow and expensive. The only reasonable solution involves the ASIC assisting with the updates. This means the ASIC is reading and writing up to 64k memory locations in the worst case (change of an 8 bit prefix). Even with linear accesses, 10 ns SDRAM, and giving the update process 50% of the memory accesses this is about 12.8 ms per update. The paper presents some simulation results to show that circa 1996 an ISP router using his scheme would have reasonable update times since the prefixes are not heavily fragmented. The problem with this is that it does not necessarily hold in any other routing tables, or routing table update patterns of the future.

Another weakness of the Stanford scheme is that in the worst case every additional prefix with length  $> 24$  bits results in the addition of another 512 byte node (assuming a 16 bit next hop pointer). This of course is true in the CPE scheme in general, but since the first 24 bits use 32-megabytes of memory, the inefficiency of later stages is felt even greater here. It is interesting to note that if in the future there were 33k prefixes in an ISP database with length greater than 24, the size of each entry in the 24 bit initial array would have to be increased an extra byte to 24 bits (since the presented scheme uses 15 bit pointers), and the total database size would be in the worst case ( $16 \text{ million} * 3 \text{ bytes} + 33\text{k} * 512 = 64 \text{ Megabytes}$ ). This is unlikely; but if it did occur not only would the memory usage be even more atrocious but the hardware would have to be redesigned which would be a much greater cost. Additionally the way the hardware implementation is described in the paper, the suggested level 2 size is 1 megabyte with an 8 bit wide DRAM. This restricts the number of prefixes greater than 24 bits to a total of 4k.

### **2.3.5. LC-Tries**

LC-Tries are essentially another form of compressed multi-bit tries. Each path in an LC-Trie can be compressed to use a different set of stride lengths [39]. The single main idea behind LC-Tries is to recursively pick strides (and therefore multi-bit tries) that result in having real prefixes at all of the leaves. Having fully populated leaves means that none

of the internal prefixes need to be pushed to the leaves of the multi-bit tree or to lower trees (no leaf pushing). In addition to this strategy, LC-Tries use path compression to save space on non-branching paths. The final results of LC-Tries is a solution that does not have deterministic number of memory accesses or size of accesses. This indicates it is not well suited for hardware implementation. In addition, prefix insertions/deletions in the worst case require the entire table to be rebuilt.

## **2.4. Non-Trie Algorithmic Approaches**

### **2.4.1. Binary Search on Ranges**

This algorithmic approach to IP Lookups views a database of prefix addresses as a set of intermingled ranges [24]. Figure 2-5 shows first a sample prefix database (a subset of the database used in prior examples) and then a graphical representation of the ranges shown. For the graphical representation we show on the y-axis the ranges of addresses assuming a 6 bit address length. As can be seen in Figure 2-5, a database of prefixes consists of ranges that can be nested. The smaller the range of a prefix, the longer the prefix is.

When searching for the longest match, graphically you want to find the smallest range that encompasses the search value.

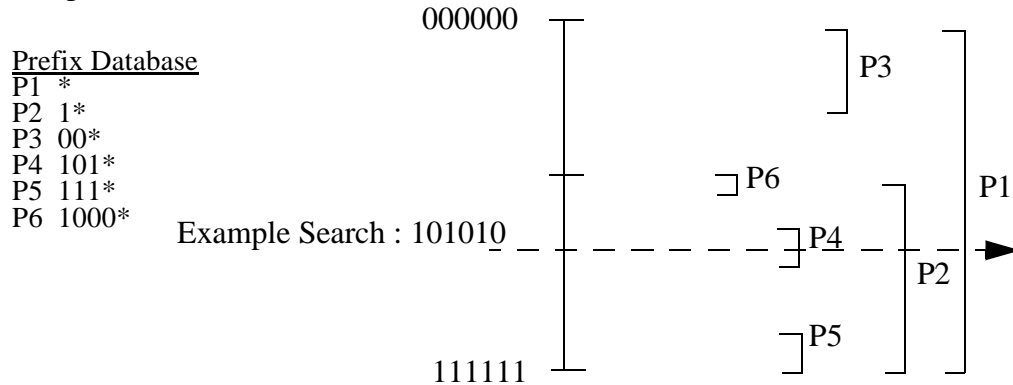


Figure 2-5 Binary Search on Ranges

Table 2-1 shows an expanded range table based on Figure 2-5. Without looking yet at the search method, the table can be used by simply scanning the entries until the address being searched for is found to be between two entries. The top of the two entries contains the prefix that is the longest match for the search. For example consider the address 101010 which is illustrated by the dotted line in Figure 2-5. As can be seen in the figure, there are 3 ranges which encompass this address. Looking at both the table and the figure, it is clear that P4 is the longest matching prefix for this search address.

**Table 2-1 Expanded Range Table**

Address	Prefix
000000	P3
001111	P1
100000	P6
100011	P2
101000	P4
101111	P2
111000	P5

There are several methods to search a range table once it is computed like binary search or a B-Tree. The major drawback to any of these schemes is that the

pre-computation of the expanded range database is  $O(n)$  (requires computation of the whole database for every update). The major advantage to this approach is a bounded and attractive ratio of memory to prefixes in the database. The number of memory accesses is comparable to other schemes and is based on the size of the prefix table and the B parameter. An initial array can be used to reduce the size of each range value stored and get all prefixes of 16 bits and less with the initial array index. This decreases the storage and reduces the size of the memory access while in the B-Tree portion of the search, and so reduces implementation bandwidth cost.

### **2.4.2. Binary Search on Prefix Lengths**

One of the first new IP lookup algorithms (since the Patricia trie approach) is based on the idea of modifying a prefix table to make it possible to use hashing. As mentioned when discussing caching of prefixes, there do not exist any methods for efficiently hashing on prefixes. The approach in [48] is to use hashing when searching among all entries of a given prefix length. Instead of searching every possible prefix length and picking the longest prefix length with a match, binary search is used to reduce the number of searches to be logarithmic in the size of the address being searched. To do this effectively and prevent backtracking, precomputation is needed. The precomputation involves the use of markers to point searches in the right direction, and adding precomputed information to these markers. Compared to more recent lookup schemes found in the literature, Binary search on Prefix Length has large memory foot prints, poor insertion time, but moderate requirements for the required number and size of memory accesses.

## 2.5. Summary of Existing IP Lookup Algorithms

Table 2-2 compares most of the discussed methods for Algorithmic IP Lookups. Note that caching, lookup avoidance schemes (MPLS), and CAMs are not part of the comparison. The first column shows the worst case number of memory accesses and the second column shows the approximate sizes of these memory accesses. While at first pass it may seem that all that is important is the worst case number of bytes fetched, that is not true. As discussed in Section 1.5. on page 18, memory technologies can have generous bandwidth but restricted random memory accesses. In this case some of the multi-bit trie based schemes are attractive since they minimize the number of random memory accesses. Also the Binary Search on Range scheme exhibits the property that the number of random memory accesses is small, but the size of each is large.

**Table 2-2 Summary of Existing IP Lookup Schemes**

Name (section discussed)	Worst Case # Mem Accesses	Estimated Memory Access Size	Worst Case Update Complexity	Max Memory Utilization <sup>a</sup>
Patricia Trie (N/A)	64	24-bytes	O(n)	4 MB
Lulea(2.3.3)	9	2-Bytes	O(n)	200KB
CPE w/leaf push <sup>b</sup> (2.3.2)	4	4-Bytes	O(n)	2.1MB
CPE wo/leaf push (2.3.2)	4	4-Bytes	O(1)	4.2 MB
Stanford Hardware Trie (2.3.4)	2	4-Bytes	O(memory)	33 MB
L-C Tries(2.3.5)	32	4-Bytes	O(n)	800 KB
Binary Search on Ranges <sup>c</sup> (k=6) (2.4.1)	5	32-bytes	O(n)	875 KB
Binary Search on Prefix Length (2.4.3)	5	8-bytes	O(n)	1.9 MB

- a. Some sizes have been linearly normalized from the original papers to 40k database size
- b. With strides of 8 bits
- c. For this scheme, worst case number of memory accesses is dependent on the database size so 5 accesses is for approximately 40k prefixes

In the 4th column is the worst case update complexity. Most of the schemes are  $O(n)$  in update complexity which means that as the database grows, the time to insert, delete, and modify entries also grows proportionately. Update performance is becoming more of an issue since some routers have experienced attempts to update as frequently as once per millisecond [22]. Additionally, it is speculated in the future that routing will be done more dynamically to take into account QoS considerations and the current congestion of the network. For these types of schemes to work a forwarding database might have to be updated in the 10's or 100's of microseconds.

The final column shows the memory size to implement a lookup table for the Mae-East database with 40k entries. The smallest footprint is the Lulea scheme. Lulea is attractive in memory footprint but has poor update complexity and requires a large number of memory accesses. The Controlled Prefix Expansion scheme without leaf pushing seems the most desirable in update complexity, number of memory accesses, and memory access size. The only area that CPE (without leaf pushing) suffers is in memory utilization.

The Tree Bitmap presented in the next chapter has  $O(1)$  update complexity, memory foot prints on the order of Lulea, and a worst case number of memory accesses on the order of CPE w/out leaf pushing. Memory access size with Tree Bitmap can be directly traded off against number of memory accesses, but will tend to be higher than Lulea or CPE without leaf pushing. However, as we will argue, this is perfectly reasonable given the characteristics of modern memories.

## 3. Tree Bitmap

In this chapter, the algorithm called Tree Bitmap is presented. Tree Bitmap is a multi-bit trie algorithm that allows fast searches (one memory reference per trie node unlike 3 memory references per trie node in Lulea) and allows much faster update times than existing schemes (update times are bounded by the size of a trie node; since we only use trie nodes with at most 8 bits, this requires only around  $256 + C$  operations in the worst case where  $C$  is small). It also has memory storage requirements comparable to (and sometimes slightly better than) the Lulea scheme.

### 3.1. Description of Tree Bitmap Algorithm

The Tree Bitmap design and analysis is based on the following observations:

- \* A multi-bit node (representing multiple levels of unibit nodes) has two functions: to point at child multi-bit nodes, and to produce the next hop pointer for searches in which the longest matching prefix exists within the multi-bit node. It is important to keep these purposes distinct from each other.

- \* With burst based memory technologies, the size of a given random memory access can be very large (e.g. 32 bytes for SDRAM, see Section 1.5.2. on page 19). This is because while the random access rate for core DRAMs has improved very slowly, high speed synchronous interfaces have evolved to make the most of each random access. Thus the trie node stride sizes can be determined based on the optimal memory burst sizes.

\* Hardware can process complex bitmap representations of up to 256 bits in a single cycle. Additionally, the mismatch between processor speeds and memory access speeds have become so high that even software can do extensive processing on bitmaps in the time required for a memory reference.

\* To keep update times bounded it is best not to use large trie nodes (e.g., 16 bit trie nodes used in Lulea). Instead, we use smaller trie nodes (at most 8 bits). Any small speed loss due to the smaller strides used is offset by the reduced memory access time per node (1 memory access per trie node versus three in Lulea). Also limiting the maximum size of the multi-bit nodes improves the update characteristics of the algorithm.

\* To ensure that a single node is always retrieved in a single page access, nodes should always be power of 2 in size and properly aligned (8 byte nodes on 8-byte boundaries etc.) on page boundaries corresponding to the underlying memory technology.

Based on these observations, the Tree Bitmap algorithm is based on four key ideas.

The first idea in the Tree Bitmap algorithm is that all child nodes of a given trie node are stored contiguously. This allows us to use just one pointer for all children (the pointer points to the start of the child node block) because each child node can be calculated as an offset from the single pointer. This can reduce the number of required pointers by a factor of two compared with standard multi-bit tries. More importantly it cuts down the size of trie nodes, as we see below. The only disadvantage is that the memory allocator

must, of course, now deal with variable sized allocation chunks. Using this idea, the same prefix database used in past examples is redrawn in Figure 3-1.

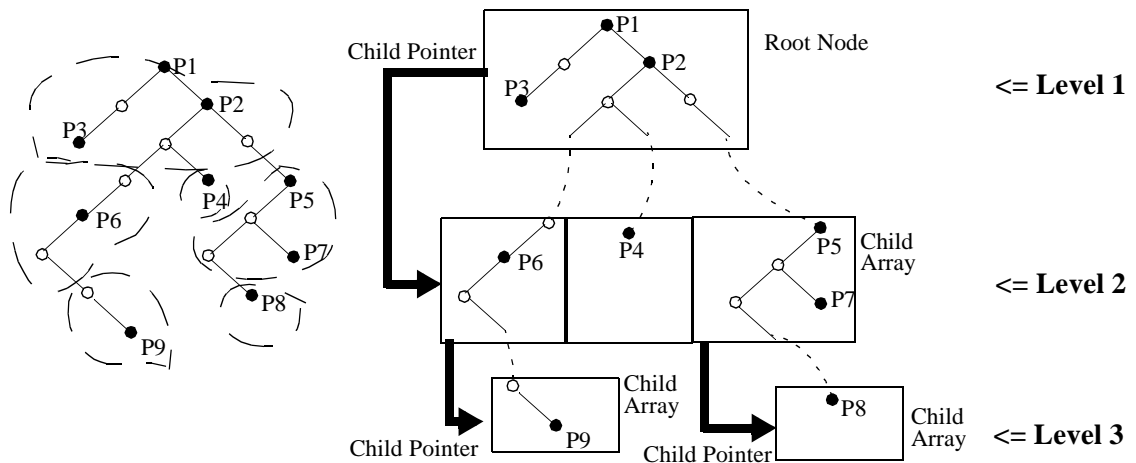


Figure 3-1 Tree Bitmap with Sample Database

The second idea is that there are two bit maps per trie node, one for all the internally stored prefixes and one for the external pointers. See Figure 3-2 for an example of the internal and extending bitmaps for the root node. The use of two bit maps allows us to avoid leaf pushing. The internal bit map is very different from the Lulea encoding, and has a 1 bit set for every prefix stored within this node. Thus for an  $r$  bit trie node, there are  $2^{r-1}$  possible prefixes of lengths  $< r$  and thus we use a  $2^r - 1$  bit map.

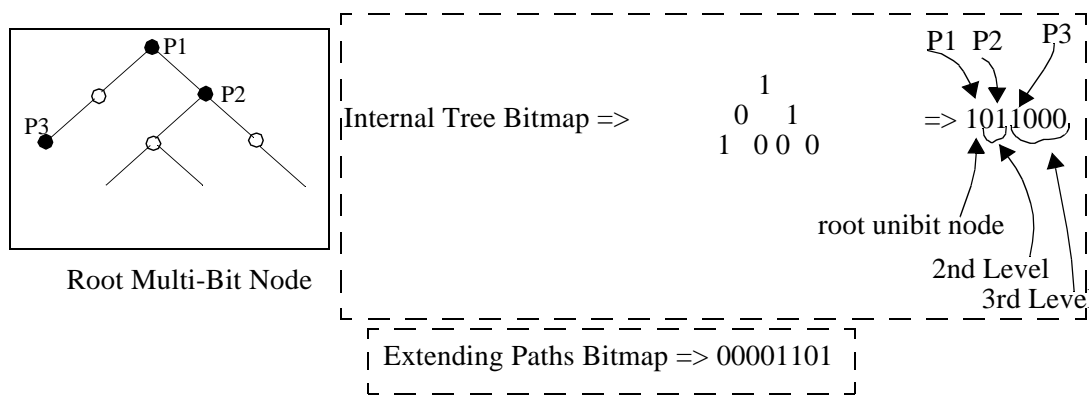


Figure 3-2 Multibit Node Compression with Tree Bitmap

For the root trie node of Figure 3-1, we see that we have three internally stored prefixes:  $P1 = *$ ,  $P2 = 1*$ , and  $P3 = 00*$ . Suppose our internal bit map has one bit for

prefixes of length 0, two following bits for prefixes of length 1, 4 following bits for prefixes of length 2 etc. Then for 3 bits the root internal bit map becomes 1011000. The first 1 corresponds to P1, the second to P2, the third to P3. This is shown in Figure 3-2. In summary, the internal bitmap represents a tree in a linearized format: each row of the tree is captured top-down from left to right. The bit list of prefixes represented by an internal bitmap are in order: \*, 0\*, 1\*, 00\*, 01\*, 10\*, 11\*, 000\*, 001\*, 010\*, 011\*, 100\*, 101\*, 110\*, 111\*. Recall that these are *prefix extensions* from whatever search address was used to get to this multi-bit node.

The extending bitmap contains a bit for all possible  $2^f$  child pointers. Thus in Figure 3-2, we have 8 possible leaves of the 3 bit sub-trie. Only the fifth, sixth, and eighth leaves have pointers to children. Thus the extending paths bit map shown in Figure 3-2 is 00011001.

The third idea is to keep the trie nodes as small as possible to reduce the required memory access size for a given stride. Thus a trie node is of fixed size, and only contains an extending bitmap, an internal bit map, and a single pointer to the block of child nodes. But what about the next hop information associated with any stored prefixes?

The trick is to store the next hops associated with the internal prefixes stored within each trie node in a *separate* array associated with this trie node. For memory allocation purposes result arrays are normally an even multiple of the common node size (e.g. with 16 bit next hop pointers, and 8-byte nodes, one result node is needed for up to 4 next hop pointers, two result nodes are needed for up to 8, etc.) Putting next hop pointers in a separate result array potentially requires two memory accesses per trie node (one for the trie node and one to fetch the result node for stored prefixes). However, we use a simple lazy strategy to not access the result nodes till the search terminates. We then access the result node corresponding to the last trie node encountered in the path that contained a valid prefix. This adds only a single memory reference at the end besides the one memory reference required per trie node.

The search algorithm is now quite simple. We start with the root node and use the first bits of the destination address (corresponding to the stride of the root node, 3 in our

example) to index into the extending bitmap at the root node at say position  $P$ . If we get a 1 in this position there is a valid child pointer. We count the number of 1s to the left of this 1 (including this 1) as say  $I$ . Since we know the pointer to the start position of the child block (say  $y$ ) and the size of each trie node (say  $S$ ), we can easily compute the pointer to the child node as  $y + (I * S)$ .

Before we move on to the child, we also check the internal bit map to see if there are one or more stored prefixes corresponding to the path through the multi-bit node to the extending path position  $P$ . Conceptually we want to take the internal bitmap and reconstruct the tree. Looking at Figure 3-2 this is conceptually taking the bitmap at the right of the figure and reconstructing the tree on the right side of the figure. For example, suppose  $P$  is 101 and we are using a 3 bit stride at the root node bit map of Figure 3-2. We first look at prefix 10\*. Since 10\* corresponds to the sixth bit position in the internal bit map (recall the internal bitmap is organized as a linearized tree, as described above), we check if there is a 1 in that position (there is not in Figure 3-2). Next we wish to look at the next highest node on the tree which would be for prefix 1\*. Since 1\* corresponds to the third position in the internal bit map, we check for a 1 there. In the example of Figure 3-2, there is a 1 in this position, so our search ends. (If we did not find a 1, however, we would next check for a \* in the first entry of the internal bit map).

This search algorithm appears to require a number of iterations proportional to the logarithm of the internal bit map length. However, in hardware for bit maps of up to 512 bits or so, this is just a matter of simple combinational logic (which intuitively does all iterations in parallel and uses a priority encoder to return the longest matching stored prefix).

Once we know we have a matching stored prefix within a trie node, we do not immediately retrieve the corresponding next hop info from the result node associated with the trie node. We only count the number of bits before the prefix position in the internal bitmap (more combinational logic) to indicate its position in the result array. Accessing the result array immediately would take an extra memory reference per trie node. Instead, we move to the child node while remembering the stored prefix position and the

corresponding parent trie node. The intent is to remember the last trie node  $T$  in the search path that contained a stored prefix, and the corresponding prefix position. When the search terminates (because we encounter a trie node with a 0 set in the corresponding position of the extending bitmap), we have to make one more memory access. We simply access the result array corresponding to  $T$  at the position we have already computed to read off the next hop info.

## 3.2. Pseudocode for Tree Bitmap Search

This section presents pseudocode for the Tree Bitmap search algorithm. Below in Figure 3-3 is a generic data structure for the tree bitmap algorithm used in the pseudocode example. The data structure does not assign sizes to the fields since the sizes will vary depending on implementation specifics like the maximum amount of memory supported and the stride length.

```

1.  trie_node {
2.  extending_bitmap /* The bitmap for the extending paths*/
3.  child_address   /* Pointer to children node array */
4.  internal_bitmap /* The bitmap of internal prefixes */
5.  results_address /* Pointer to results array */
6.  }

```

Figure 3-3 Reference **Tree Bitmap Data Structure**

The pseudocode for search in Figure 3-4 assumes 3 arrays set up prior to execution of the search code. The first array is called `stride[]` contains the search address broken into the stride length. So with a stride of 4 bits, and a search address length of 8 bits, the array `stride[]` will have 8 entries each 4 bits in length. The designation `stride[i]` indicates the  $i$ 'th entry of the array. The second array that is required is `node_array[]` which contains all of the trie nodes. The third array is `result_array` which contains all the results (next hop

pointers). In practice the next hop pointers and node data structures would probably all be in the same memory space with a common memory management tool.

1. node:= node\_array[0]; /\* node is the current trie node being examined; so we start with root as the first trie node which is assumed to be at location 0 of node\_array \*/
2. i:= 1; /\* i is the index into the stride array; so we start with the first stride \*/
3. do forever
4. if (treeFunction(node.internal\_bitmap, stride[i]) is not equal to null) then
5. /\* there is a longest matching prefix, update pointer \*/
6. LongestMatch:= node.results\_address + CountOnes(node.internal\_bitmap,
7. treeFunction(node.internalBitmap, stride[i]));
8. if (extending\_bitmap[stride[i]] = 0) then /\* no extending path through this trie node for this search \*/
9. NextHop:= result\_array[LongestMatch]; /\* lazy access of longest match pointer to get next hop pointer \*/
10. break; /\* terminate search)
11. else /\* there is an extending path, move to child node \*/
12. node:= node\_array[node.child\_address + CountOnes  
(node.extending\_bitmap, stride[i]));
13. i=i+1; /\* move on to next stride \*/
14. end do;

Figure 3-4 Pseudocode of Algorithm for Tree Bitmap

There are two functions assumed to be predefined by the pseudocode. The first function ‘treeFunction’ can find the position of the longest matching prefix, if any, within a given node by consulting the internal bitmap. The function treeFunction takes in an internal bitmap and the value of stride[i] where i is for the current multi-bit node. The second function CountOnes simply takes in a bit vector and an index into the bit vector and returns the number of ‘1’ bits to the left of the index.

There are several variables assumed. The first is “LongestMatch” which keeps track of an address into the result\_array of the longest match found so far. Another variable is ‘i’ which indicates what level of the search we are on. A final variable is the node which maintains the current trie node data structure under investigation.

The loop terminates when there is no child pointer (i.e., no bit set in extending bitmap of a node) upon which we still have to do our lazy access of the result node pointed to by LongestMatch to get the final next hop.

So far, we have implicitly assumed that processing a trie node takes one memory access. This can be done if the size of a trie node corresponds to the memory burst size (in hardware) or the cache line size (in software). That is why we have tried to reduce the trie node sizes as much as possible in order to limit the number of bits accessed for a given stride length. In the next chapter, we describe some optimizations that can reduce the trie node size even further.

## 4. Tree Bitmap Optimizations

### 4.1. Overview of Optimizations

For a stride of 8, a data structure for the core algorithm in C-like pseudo-code might look like:

```

1.  trie_node {
2.  extending_bitmap   : 255 /* The bitmap for the extending paths*/
3.  child_pointer      : 256 /* Pointer to children node array */
4.  internal_bitmap    : 20  /* The bitmap of internal prefixes */
5.  results_pointer    : 20  /* Pointer to results array */
6.  Reserved           :473 /* These bits keep the size of the node a power of 2 */
7.  }

```

Figure 4-1 Reference Tree Bitmap Data Structure

Note that the number of used bits is 551 which means the next larger power of 2 node size is 1024 bits or 128-bytes. From our table in Section 1.5.2. on page 19. we can see that this is a much larger burst size than any technology optimally supports. Thus we seek optimizations that can reduce the access size. Additionally, the desire for node sizes that are powers of two means that in this case only 54% of the 128-byte node is being used. So we also seek optimizations that make more efficient use of space. We also desire optimizations that reduce total memory storage, and reduce worst case number of memory accesses required. The optimizations presented in this chapter do not all need to be used in every implementation of Tree Bitmap.

## 4.2. Initial Array Optimization

Before we reduce required burst size, we note that almost every IP lookup algorithm can be sped up by using an initial array (e.g., [6],[26],[46],[48]). The idea is that at the top of the trie we permanently sacrifice a small amount of memory in order to reduce the worst case number of lookups. It is important to realize that this simple optimization (if applied over-zealously) can ruin the update properties of an algorithm like Tree Bitmap. Array sizes of say 13 bits or higher could have poor update times. A example of initial array usage would be an implementation that uses a stride of 4, and an initial array of 8 bits. This means that the first 8 bits of the destination IP address would be used to index into a 256 entry array. Each entry is a dedicated node possibly 16 bytes in size which results in a dedicated initial array of 4k bytes. This is a reasonable price in bytes to pay for the savings in memory accesses. In a hardware implementation, this initial array can easily be placed in on chip memory even if the rest of the table is placed off-chip.

## 4.3. End Node Optimization

We have already seen an irritating feature of the basic algorithm in Figure 3-1. Single prefixes that just extend to the root of a new multi-bit node (like P8 in Figure 3-1) will require a separate trie node to be created (with bitmaps that are almost completely unused). While this cannot be avoided in general, it can be mitigated by picking strides carefully (see Section 4.7. on page 55). In a special case, we can also avoid this waste completely. Suppose we have a trie node that has no external pointers, or only has external pointers that point to nodes that have a single internal prefix at the root. Consider Figure 4-2 as an example that only has children (in this case one child) which have a single prefix at the root.

In that case, we can make a special type of node called an endnode (the type of node can be encoded with a single extra bit per node) in which the extending bitmap is eliminated and substituted with an internal bit map of twice the length. The endnode now has room in its internal bit map to indicate prefixes that were previously stored in the root of its children. Thus in Figure 4-2, P8 is moved up to be stored in the upper trie node which has been modified to be an endnode with a larger bit map. It is important to note that in Figure 3-1, P4 appears to be similar to P8 in that they both begin as roots of multi-bit nodes. However, the multi-bit node for P4 cannot be pulled up to the root multi-bit node since the root has other children that would not be eliminated by doubling the length of the internal bitmap.

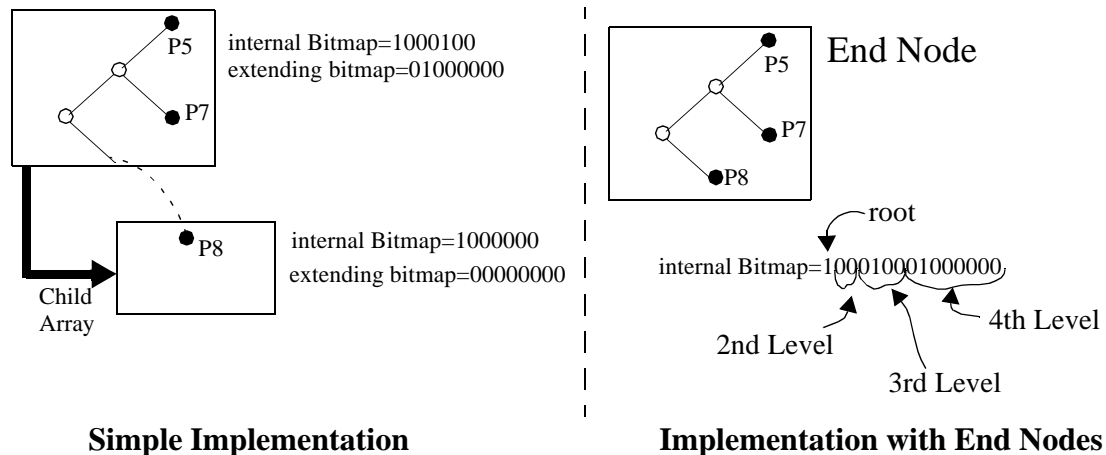


Figure 4-2 End Node Optimizations

An important reason for the use of endnodes (besides the savings in number of nodes explained in the last paragraph). is that without them for typical strides of say 4 or 8 bits, every 32-bit address will require a multi-bit node of it's own. The reason for this is that if we are using an 8 bit stride for example, the first level will contain prefixes from 0 bits in length (a complete wildcard) to 7 bits in length, the 2nd level is 8 bits to 15 bits, the third level 16 to 23 bits, and the final level 24 to 31 bits. So the 32nd address bit would require a 5th level without the end node optimization. While this additional level is wasteful in space, it is (more importantly) an additional memory access in the worst case. The

use of endnodes will result in all 32 bit addresses being in the bottom most level of an end-node and will not require an additional memory access.

## 4.4. Split Tree Bitmaps

Keeping the stride constant, one method of reducing the size of each random access is to split the internal and extending bitmaps. This is done by placing only the extending bitmap in each “Trie” node and placing the internal bitmap separately in another node (we will call this other node an internal node). The idea behind this optimization is the idea that we can divide the longest matching prefix problem into sub-problems. The first sub-problem is finding which multi-bit node contains the longest prefix match for a given search. The second sub-problem is given the multi-bit node which contains the longest prefix match, finding exactly where the longest match is in that multi-bit node.

To make this optimization work, each child must have a bit indicating if the parent node contains a prefix that is a longest match so far. This may sound similar to leaf pushing (discussed in Section 2.3.2. on page 29 and Section 2.3.3. on page 31) but note all we are doing is helping to solve the problem of which multi-bit node contains the longest prefix match. The exact details of the prefixes are maintained in an internal node just for the trie node in which the prefix is found.

When each trie node is being evaluated if the parent had a prefix that was a match for this search, the lookup engine records the location of the internal node (the possible methods for pointing to an internal node are explored in the next paragraph) as containing the longest matching prefix thus far. Then when the search terminates, we first have to fetch the internal node corresponding to the trie node with the longest prefix match and then from the internal node the results node can be fetched corresponding to the internal node. Notice that the core algorithm accesses the next hop information lazily; the split tree algorithm accesses even the internal bit map lazily.

If there is no memory segmentation (all nodes are stored in the same physical memory) then a simple way to keep pointers to internal nodes simple is to place the children and the internal nodes of the same trie node in a contiguous array in memory. This means that if an internal node exists, its location is simply the `child_address` plus the number of children (found by counting the number of '1's set in the extending bitmap) plus an additional one. If memory segmentation does exist (segmentation could be used to increase lookup rate by putting nodes of each level in separate memories or banks) this pointer method is problematic since the internal nodes would be scattered across different memories.

A nice benefit of split tree bitmaps is that if a node contained only paths and no internal prefixes, no space will be wasted on the internal bitmap since the internal node for the trie node will simply not exist. For the simple Tree Bitmap scheme with an initial stride of 8 and a further stride of 4, over 50% of the 2971 trie nodes in the Mae-East database do not have any internally terminating prefixes.

A side effect of the split tree bitmap optimization is the removal of the benefit from the end node optimization (see Section 4.3. on page 49). With the internal bitmap removed from the trie node data structure, the advantages of the end node optimization are not present since the internal bitmap is no longer in the trie data structure to be removed by the end node optimization. However, when a trie node has internal prefixes and no extending paths, there is still a benefit from treating that trie node directly as an internal node. This means replacing the extending bitmap with an internal bitmap. We will refer to this type of node (containing a internal bitmap but not mirroring a trie node) as an end node (similar to the end node in Section 4.3. on page 49 since they both replace trie nodes and contain internal bitmaps.)



## 4.6. CAM Nodes

Empirical results show that a significant number of multi-bit nodes contain only a few prefixes (both trie nodes and end nodes). Figure 4-4a shows a histogram for the number of prefixes per multi-bit node of the end type for the Mae-East<sup>1</sup> database from IPMA [27]. This figure shows that while most typical trie nodes have no internal prefixes (and therefore no internal nodes), the bin with a single prefix is larger than all other bins for greater numbers of prefixes combined. Figure 4-4b shows a histogram for the number of prefixes per internal node (recall internal nodes mirror a trie node with extending paths). This figure is also for the Mae-East database and also has the property that the bin for internal nodes with a single prefix is larger than all other bins combined.

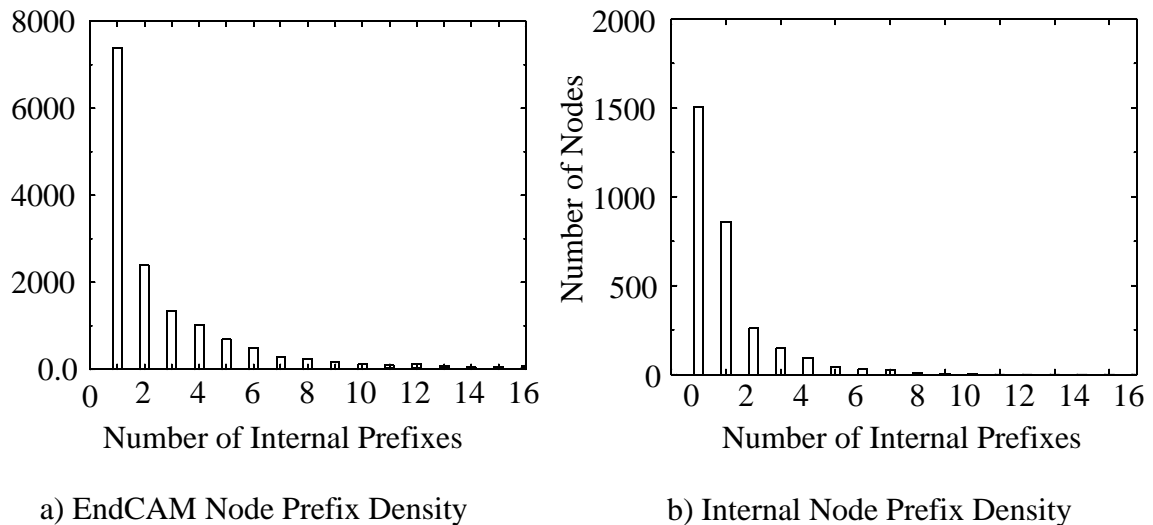


Figure 4-4 Histogram of Prefix Count for End and Internal Nodes<sup>1</sup> (Mae-East)

1. Initial Stride of 8, with a further stride of 4.

For either end nodes or internal nodes in the cases where there are small numbers of prefixes in a multi-bit node, the space normally occupied by internal bitmaps and a pointer to a results array, can be replaced by simple CAM type entries that have match bits, match length, and next hop pointers. For a stride of 4, the match bits need to be 4 bits, and the

1. The main backbone database used in this thesis is the MAE-East database taken on Jun. 24, 1998

match length needs to be 2 bits. The gain is that the next hop pointers are in the CAM nodes and not in a separate results array taking up space. Even a single entry CAM node will result in over half of the next hop pointers moving from results arrays, to inside CAM end nodes. However, there are quickly diminishing returns.

From Figure 3-1 the multi-bit node containing P4 could be made into a CAM node (recall P8 from Figure 3-1 was pulled into it's parent node with the endnode optimization). So while we cannot eliminate the multi-bit node that contains P4 (like we did for the multi-bit node containing P8 in Section 4.3. on page 49), we can move the next hop pointer into the P4 multi-bit node so that an additional results array is not needed.

## 4.7. Empirical size results

In this subsection we explore empirically the size characteristics of the simple core algorithm and the effects on size of both end nodes and initial array optimizations. Since the split tree and segmented bitmap optimizations are primarily used to alter memory access sizes, the empirical results of these optimizations are not explored here. The empirical investigation in this section also reveals some interesting characteristics of real databases that should be taken into account when picking the initial and the normal stride.

Table 4-1 gives size results for the Mae-East database with the raw Tree Bitmap algorithm and no optimizations.

**Table 4-1 Tree Bitmap Size Results (Mae-East)**

Stride	# Nodes	Size of node (bytes)	Total Size (with next hop pointers)
1	131058	8	1.12 MB
2	72313	8	658 KB
3	51451	8	491 KB
4	43446	8	427 KB
5	17190	16	355 KB

**Table 4-1 Tree Bitmap Size Results (Mae-East)**

Stride	# Nodes	Size of node (bytes)	Total Size (with next hop pointers)
6	31129	32	1.07 MB
7	18165	64	1.96 MB
8	30698	128	4.7 MB

As the stride grows, the number of nodes to represent the Mae-East database generally decreases since each node covers a larger area in the prefix trie. Note that there are anomalies to the general decrease, such as the minima at stride of 5 for the simple case. For a stride of 5, the end of each internal bitmap is at lengths 4, 9,14,19,24,29. Since 24 bits length is a class C address from the original Internet classes, this address length still has the largest number of prefixes in most backbone databases. Therefore it is logical that stride of 5 in the simple case has a small number of nodes.

With the end node optimization (but not using the split tree optimization), the lengths 4,6, and 8 are all significantly below the numbers expected by the lengths 1,2,3,5, and 7. This is because all three of these strides will have end nodes that can terminate with the bottom level of the end node at 24 bits. The reason the end node optimization moves the strides with a minima is that it allows an extra bit to be picked up due to use of the extending bitmap space for the internal bitmap.

As can be seen from Table 4-2, for a given stride the minima occur when initial stride added to a multiple of the stride is equal to 24. The cells that satisfy this condition have been lightly shaded. Note the for strides that already are optimal for Class C addresses without initial arrays (strides of 4 for example), the initial array makes little difference to the total storage. For cases in which the initial array combined with the stride are optimized for class C addresses (the shaded cells), there can be quite an improvement in size compared to the case without the initial array. An interesting note is that for the

Mae-East database the minimal number of nodes is obtained using an initial stride of 10, and then a stride of 7.

**Table 4-2 Number of Nodes with Initial Arrays & End Nodes (Mae-East)**

Initial Array	stride=4	stride=5	stride=6	stride=7	stride=8
4	17541				
5	21556	17107			
6	28356	21025	8383		
7	35732	23821	11662	18116	
8	17515	29750	14870	23290	7941
9	21536	13056	20233	28233	5074
10	28321	17085	24781	5439	7578
11	35672	20988	26313	7791	10531
12	17424	23750	8334	10883	12706

## 5. Reference Design

### 5.1. Overview

The reference design presented in this chapter is a single chip implementation of an IP Lookup engine using embedded SRAM. The goal of this reference design is to explore the limits of a single chip design for backbone Internet routers. The targets of this design are:

- \* Deterministic lookup rates supporting worst cast TCP/IP over OC-192c link rates (25 million lookups per second).
- \* Empirical evidence of support for the largest backbone databases of today and room for future growth
- \* Reasonable worst case bounds for the ratio of prefixes to memory
- \* Guaranteed worst case update rates
- \* Transparent updates

Figure 5-1 shows a block diagram of an ASIC that does IP Forwarding with IP address Lookups embedded. Packets enter the IP Forwarding chip either as raw IP Packets or encapsulated in a data link layer frame (The details for handling IP/ATM are not shown in this example). The Parser block takes the packets and extracts the IP Destination Address and passes it to the Lookup Engine block. The packets themselves are sent to the Packet Buffer to wait the completion of the IP address lookup. Since the address lookup has a short deterministic latency, the Packet Buffer can be a small FIFO. For example,

with a deterministic 500 ns lookup latency, the Packet Buffer need only be  $\{(9.6 \text{ gigabits / second}) * 500 \text{ ns}\}$  or 4.8 Kbits. When the address lookup completes, the Lookup Engine passes the next hop pointer to the Packet Manipulation block. At this final stage the next hop pointer can index into a table of next hops and use the stored data for manipulating the packet for transmission through the fabric and onto the output link.

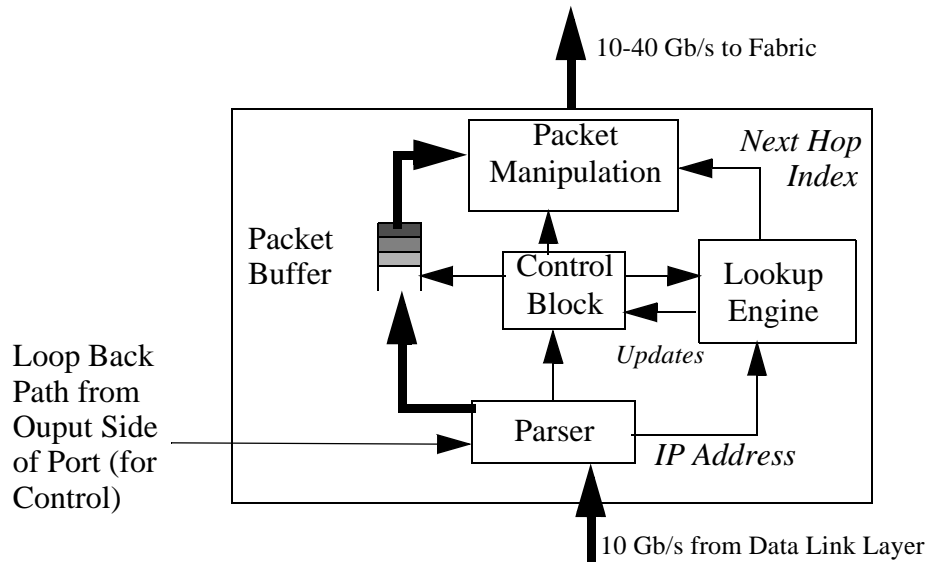


Figure 5-1 Block Diagram of OC-192c IP Forwarding ASIC

For the control functions of the Forwarding ASIC in Figure 5-1, control messages can originate from the data link layer or through a loop back port from the output side of the link card. The Parser block determines which messages are control and passes them to the Control Block. The Control Block issues forwarding table updates to the Lookup Engine and next hop table updates to the Packet Manipulation block. When the Control Block receives an acknowledgment for the update from either block, a response message is placed into the Packet Buffer. Note the update messages to the next hop pointer table are of the type:

For Next Hop Pointer 235: Output Port = 154, Fabric Priority = 2

For Forwarding Table Updates in the Lookup Engine updates are of the form:

Insert Prefix 0xabcdefg, length = 25, Next Hop Pointer = 235

The first section of this chapter describes the framework for the reference design. This framework will be based on the description of the Tree Bitmap algorithm in earlier

chapters. Additional sub-sections will cover memory management and the update algorithm for the reference design. The final sub-section will go into low level detail of the reference design. Chapter 6 will contain analysis of the reference design that was presented in this chapter.

## **5.2. Reference Design Framework**

### **5.2.1. Lookup Engine Overview**

Below in Figure 5-2 is a block diagram of the IP Lookup Engine. The left side of the engine has the lookup interface with a 32-bit search address, and a return next hop pointer. The lookups are pipelined so that each next hop pointer is returned exactly 360 ns after the search address enters the Lookup Engine (72 clock cycles). On the right side of the lookup engine is an update interface. Updates take the form: type {insert, change, delete}, prefix, prefix length, and for insertions or table changes a Next Hop Pointer. When an update is completed the Lookup Engine returns an acknowledgment signal to conclude the update.

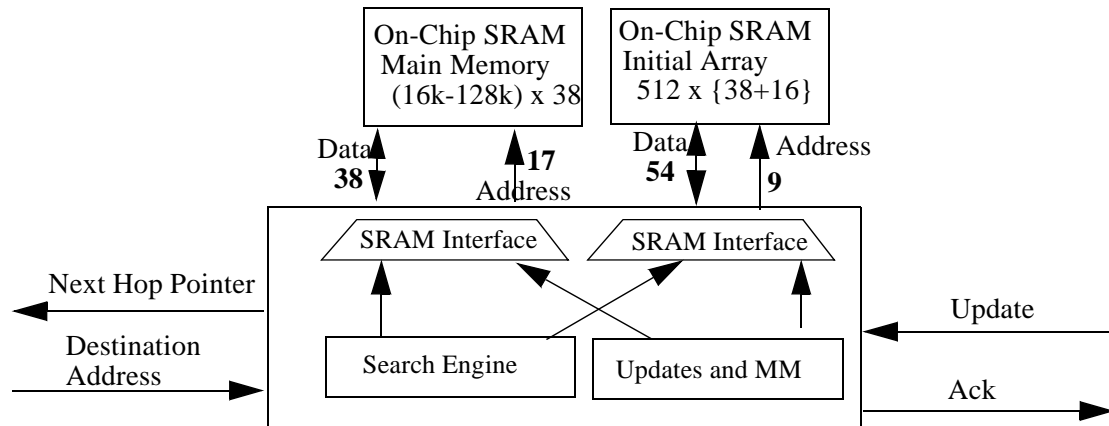


Figure 5-2 Block Diagram of IP Lookup Engine Core

Looking inside the block diagram of Figure 5-2, there are two SRAM Interface blocks. The right SRAM Interface block connects to an SRAM containing the initial array. The initial array optimization (see Section 4.2. on page 49) trades off a permanently allocated memory for a reduction in the number of memory accesses in the main memory. For the reference design the initial array SRAM is 512 entries and 54 bits wide. The left SRAM Interface block connects to the main memory of the lookup engine. The reference design has a 38 bit wide interface to the main memory and supports implementations of up to 128k memory entries. For all analysis of the reference design, the main memory depth is varied from 16k entries up to the maximum supported size of 128k entries. Note that with 128k entries the main memory is nearly 5 Mbits in size which would occupy significant die area even in a cutting edge technology (a detailed discussion about ASIC implementation details can be found in Section 6.3. on page 88). The design could be easily be modified to support more than 128k entries if desired, the address pointers would need to be increased and therefore the width of the entries (and the SRAMs) would also need to be increased.

The final two blocks of Figure 5-2 are the Search Engine, and the block for updates and memory management (Updates and MM block). The Updates and MM block interleaves memory accesses with the Search Engine for both SRAM interfaces. Therefore

the Search Engine must be engineered to leave bandwidth for control memory accesses. The Updates and MM block must do control operations in an incremental fashion that never allows a search to get an incorrect result or result in an error state. Note that it is allowed for searches to retrieve the “old” result that an update will change when the update is completed.

The design parameters based on the terminology of previous chapters are:

- \* 9 bit initial stride (So first 9 bits of search address used to index into initial array)
- \* 4 bit regular stride
- \* Split Tree Bitmap Optimization
- \* End nodes (Note the lowest level of the trie must use end nodes and will encompass the last 3 search address bits)
- \* CAM optimization for end nodes and internal nodes

For each entry of the initial array, a trie node is stored; additionally a next hop pointer and a prefix length are stored. For each entry in the initial array, the longest matching prefix for bits 0-8 will be stored with the length of the prefix. When an update is received with a prefix length of 9 bits or less, every initial array entry is checked that has the update prefix as a subset (for instance an update for the prefix 0000\* could possibly effect initial array entries 000000000 to 000011111). If an initial entry checked for a possible update has a prefix with a length less the length of the update prefix, then the next hop pointer and length in the entry are changed to match the update. This means that in the worst case a prefix of length  $k$  (for  $k < 10$ ) will require  $2 \times 2^{9-k}$  memory operations to the initial array memory (the factor of 2 reflects the read and write necessary in the worst case where every entry must be changed). Below is the data structure for the initial array entries. Note that the node data structures will be defined in future sections.

1. struct initial\_array\_struct {
2. node : 38;
3. next\_hop\_pointer : 12;
4. prefix\_length : 4;
5. }

Figure 5-3 Initial Array data structure

## 5.2.2. Path Compression

In Chapter 6, one of the properties of the reference design analyzed is the worst case prefix to memory ratio. This analysis of the design as presented so far, yields quite poor results due to one-way branches. The pathological worst case is that every prefix is near maximum length (29-32 bits) and every prefix is at the end of a series of trie-nodes that each have one extending path and no internal prefixes. In this case the storage for  $P$  prefixes is approximately  $\{P \times \text{NumberLevels}\}$ . For the reference design this means the storage is roughly 6 nodes for every prefix. A simple way to improve this analysis is to eliminate the storage of trie nodes that have no internal prefixes and only one extending path. The resulting storage properties with this optimization will be further explored in Section 6.2. on page 85.

The method for eliminating one-way branches in the reference design is based on a skip length embedded in every node that can be encountered during the search down the trie (trie, CAM or end). For the reference design with a stride of 4 bits, a node with a skip length of 1 would indicate that the node is actually 1 level below the expected level and therefore 4 bits of the search address should be skipped. The complexity of a scheme like this is related to the fact that when search bits are skipped, it is possible that the skipped search bits do not match the corresponding bits of the prefixes below the skip point. When a search arrives at a node without any relevant extending paths, we would like to check the full search address for this node to verify the prefixes below the skip point are valid for this search. If we find that there is a mismatch in skipped bits, we then we want to conceptually back track to the last trie node that contained an internal prefix that is a match for this search.

With hardware implementation and the Tree Bitmap algorithm, this backtracking can be done without any additional memory accesses. As we traverse the trie, we store each node in a separate register. When we reach the bottom of the search (no more extending paths) we check a 32 bit address representing the last node. If the last node is a failed match, we use the 32 bit address representing the last node's address, to find the last valid

node that was fetched. Since all the nodes fetched in the search are in registers, we can directly calculate the address of the internal node that contains the longest prefix match. It may seem that a cost of this approach is the storage with each node of the address used to get to it. However, as described in Section 5.3. on page 70 this address is already needed with every allocation block for memory management purposes. The additional memory access to fetch the full address of the allocation block the final search node resides still has to be taken into account. However, as seen in Section 5.2.4. on page 67, we only need to fetch the full node address when at least one node has been skipped (thereby eliminating at least 1 memory access) therefore the worst case number of memory accesses is not affected by path compression.

### **5.2.3. Reference Design Data Structures**

This section presents all the data structures used in the reference design. The trie data structure is shown below in Figure 5-4. It is important to note that all addressing is done relative to the node size of 38 bits (While 38 bits is a non-standard memory size, it is not a problem for ASIC SRAM memory generators). With a 17 bit address, there is a maximum of 128k nodes (not counting the initial array). Figure 5-4 has a 16 bit extending bitmap but no internal bitmap since the split tree optimization is used for the reference design which puts internal bitmaps in internal nodes. The bit parent\_has\_match is used to indicate if the parent has any prefixes that matched the search through it. When each node is fetched, the type of node is not known in advance so the type field indicates if the node is

trie, end, or end-CAM. The field `skip_length` indicates how many levels were skipped between the parent and the child.

```

1. struct trie_node {
2.   skip_length      : 2;   /* Number of levels skipped */
3.   type             : 1;   /* type=1 means trie node, 0 indicates other */
4.
5.   xtending_bitmap  : 16;  /* the 16 bit bitmap for the extending paths*/
6.   child_address    : 17;  /* pointer to children node array */
7.   parent_has_match : 1;   /* a 1 indicates the parent has a matching prefix */
8.   reserved         : 1;   /* reserved */

```

Figure 5-4 Trie Node Data Structure

Figure 5-5 below contains the data structure for CAM nodes. CAM nodes can replace end nodes or internal nodes with less than 2 prefixes. The fields `type` and `skip_length` are the same as for the trie node. Note that the `skip_length` field will be non-used in internal-CAM nodes and will always be set to zero. For each stored prefix there is a 3 bit address field, 2 bit prefix length and a 12 bit next hop pointer.

```

1. struct cam_node {
2.   skip_length      : 2;   /*
3.   type             : 2;   /* '00' is CAM Node */
4.   address_prefix_a : 3;   /* address bits for prefix a */
5.   length_prefix_a  : 2;   /* length of prefix a */
6.   next_hop_pointer_a : 12; /* next hop pointer for prefix a */
7.   address_prefix_b : 3;   /* address bits for prefix b */
8.   length_prefix_b  : 2;   /* length of prefix b */
9.   next_hop_pointer_b : 12; /* next hop pointer for prefix b */
10. }

```

Figure 5-5 CAM Node Data Structure

Below in Figure 5-6 is the data structure for both an internal node (mirroring a trie node) or an end node. Once again the `skip_length` and `type` fields have the same definition as trie nodes except that for internal nodes the `skip_length` field is always set to zero since internal nodes mirror a trie node and the field has no significance. A 15 bit internal bitmap

represents 4 unibit levels with up to 15 prefixes represented. The 17 bit result\_address points to a child array of result nodes.

```

1. struct internal_node {
2.   internal_bitmap   : 15; /* Simply the 16 bit bitmap for the extending paths*/
3.   result_address    : 17; /* Pointer to children node array */
4.   type              : 2; /* type=01 means internal/end node */
5.   skip_length       : 2; /* Number of levels skipped */
6.   reserved          : 2; /* A 1 indicates there are internal prefixes */
7. }

```

Figure 5-6 Internal/End Node Data Structure

Figure 5-7 shows the result pointer. Up to three next hop pointers can be stored in a single result node. If a result array has a non modulo-3 number of results then there will be wasted space since one result node will not be completely filled.

```

1. struct result_node {
2.   next_hop_pointer_a : 12;
3.   next_hop_pointer_b : 12;
4.   next_hop_pointer_c : 12;
5.   reserved           : 2;
6. }

```

Figure 5-7 Result Node Data Structure

Figure 5-8 below shows the allocation header placed at the front of every allocation block. This header is used for both memory management and for checking skip bits after a search completes of a compressed path. The type bit in allocation headers indicates if array is the child array of a trie node or a result array.

```

1. struct allocation_header {
2.   allocation_address : 29; /* the address up to this block */
3.   allocation_level   : 3; /* What level in this trie is this block represent ? */
4.   type              : 1; /* What type of allocation block ? */
5.   reserved          : 5;
6. }

```

Figure 5-8 Allocation Block Header

## 5.2.4. Node Access Patterns

Figure 5-9 illustrates a possible node access pattern for the reference design. In this figure, the first 9 bits of the search address (bits 0 through 8) are used to index into the special initial array memory. At the indexed location is stored a node (in this illustration it is a trie node) which represents bits 9-12 of the search address. After the first trie node is fetched from the initial array, 4 trie nodes and then an end node are fetched from the main memory. The end node points at a result node which contains the next hop pointer for this example search.

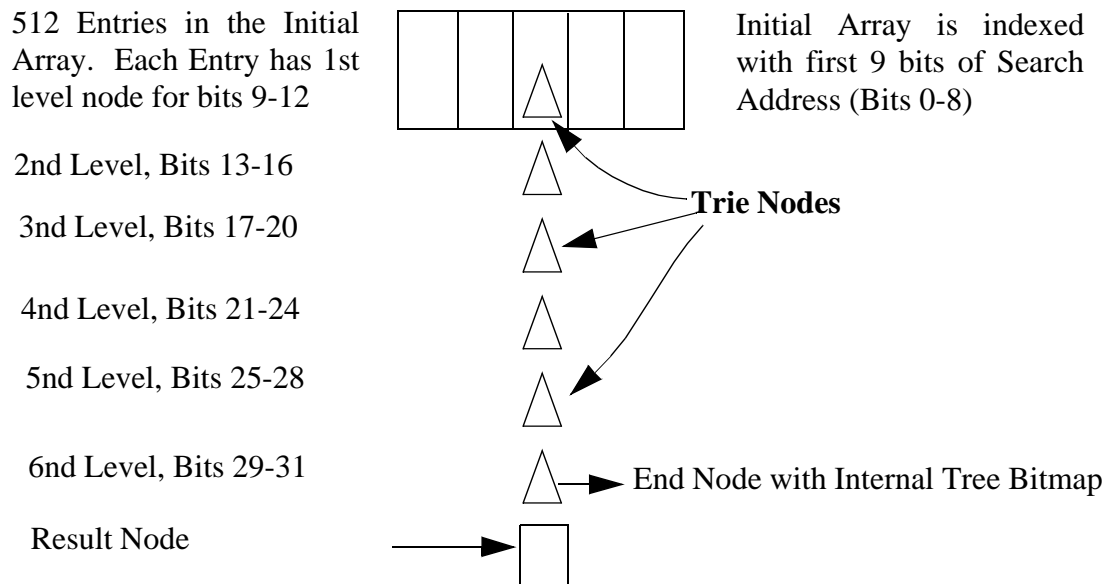


Figure 5-9 Node Sequence For a full 32 bit search

To study a variety of node access patterns, Table 5-1 presents a condensed format for representing the patterns. The first entry of Table 5-1 is the same node pattern as shown in Figure 5-9. For any search there is a fetch from the initial array and then up to 7 memory accesses from the main memory. The node types are Allocation Block Header (A), trie node (T), end node (E), CAM node (C), internal node (I), or result node (R). Any node that contains a prefix that really matches the example search is noted with an underline under the node type (nodes after a skip that seem to match during the search, but

really don't will not be underlined). Along with each node type is the level that the node represents. For entry number 1 of the figure, there are several nodes that contain matching prefixes. However, at the end of the search the final node contains a match which makes it the longest matching prefix. The final memory access in the first entry is to fetch a result node. For this example even though there were matches in the T2 and T4 nodes these were not the longest matches and so the internal and result nodes for these matches are never fetched.

**Table 5-1 Representative Node Access Patterns**

#	Initial Array	Main Memory Accesses						
		Access 1	Access 2	Access 3	Access 4	Access 5	Access 6	Access 7
1	T1	<u>T2</u>	T3	<u>T4</u>	T5	<u>E6</u>	R6	--
2	T1	T2	<u>T3</u>	T4	T5	E6	I3	R3
3	<u>T1</u>	T2	T3	T4	T5	C6	I1	R1
4	T1	<u>T2</u>	T4	T5	E6	A6	I2	R2
5	T1	<u>T3</u>	T4	E6	A6	I3	R3	--
7	T1	C4	A4	--	--	--	--	--

The second entry of Table 5-1 shows an example in which the search ends with an end node but no match is found in the end node. As the search engine traversed the trie, the field `parent_has_match` bit was used to maintain state on the last trie node that contained a matching prefix. For this example stored state indicates that T3 contained the longest matching prefix. So after the end node, the internal node for level 3 was fetched and then the appropriate result node is fetched. The third example ends with a CAM node instead of an end node. However, there is no match in the CAM node and so the search logic conceptually backtracks like the previous example.

The fourth entry introduces a skip in the search pattern. After the 2nd level trie node (T2) the next node fetched is a 4th level node. The `skip_length` field in the T4 node will be set to 1 to indicate 1 level of the search was skipped. In this example the last node containing a real matching prefix is T2. If we suppose that the bits for the skipped level

did not match then it is possible that any of the nodes T4, T5 and T6 can seem to have matching prefixes even though they are not really matches. When no extending paths are left the allocation block header of the last node (A6) is checked. In this example we find that between the 2nd and 4th levels the search address deviated from the address of the 4th level node. Using stored state and the fact we can now calculate the last node that was on the valid search path, we can conceptually backtrack to the node containing the real longest matching prefix. In this case the 2nd level contains the prefix so after the allocation header we fetch the internal node and then the result node for this prefix. So even in this worst case access pattern with skips, the total number of memory accesses is 7.

The fifth entry of Table 5-1 illustrates that there can be multiple skips in a single search (between T1-T3 and T4-E6), and that node besides trie node (in this case an end node) contain a skip length. For the sixth entry the 9 bits indexing into the initial array only has a single prefix encompassing them, this prefix is in the 4th level in a CAM node. However in this case the CAM node can not be placed directly in the initial array since there would be no corresponding allocation block header with the exact search to the 4th level CAM node. Therefore a trie node must be placed in the initial array with a single extending path to the CAM node. After the CAM node is fetched the corresponding allocation header must be fetched. The simple rule that will prevent errors is that the initial array nodes must always be first level nodes for the search address bits 9-12.

With a target of 25 million lookups per second, that allows 40 ns per lookup. At 200 Mhz operation, there are 8 memory accesses to the main memory possible per lookup. As was seen in this section we need to dedicate 7 memory accesses per lookup slot for search accesses. This leaves 1 memory access every 40 ns for a control operation. This control operation is for update and memory management operations. Normally on-chip SRAM memories have separate input and output data busses so there is no cost for interleaving read and write accesses. This means the control memory accesses can be either read or write and can be perfectly interleaved with the search memory accesses. It should be noted that this is not normally possible with off-chip memory implementations.

## 5.3. Memory Management

A very important part of an IP lookup engine design is the handling of memory management. A complex memory management algorithm requires processor management of updates which is expensive and hard to maintain. Hardware memory management with poor memory utilization can negate any gains made by optimizations of the lookup algorithm itself for space savings. Memory management is therefore a very important part of a lookup engine and requires careful consideration.

Controlled prefix expansion has fixed size nodes (if you consider the simple description of CPE without involving dynamic programming to minimize data structure memory utilization) which makes memory management as simple as a free list implementation. Tree Bitmap requires variable length allocation of memory and therefore is a more difficult problem. However, hardware implementation of the memory management is still desirable so we will carefully look at the memory management options.

### 5.3.1. Fixed allocation block memory management

For memory management with fixed sized allocation, the most efficient method of memory management is list based. The worst case memory utilization of this scheme is 100% and the enqueue and dequeue of allocation blocks from the free list is trivial allowing very high update rates. The only way that a simple free list implementation could be applied to Tree Bitmap would be to always allocate the maximum size child array. In this case you no longer need an extending bitmap since you can always directly index into each child array. Also rather than having an internal bitmap and a pointer to an array of results, we can simply put any internal results with the appropriate child. At this point the controlled prefix expansion has been re-invented and all space saving advantages of Tree Bitmap have been lost.

### 5.3.2. Memory Space Segmentation for Memory Management

For variable length allocation blocks the memory management problem is more difficult than for fixed sized allocation blocks. The simplest method of handling a small fixed number of possible lengths, is to segment memory such that there is a separate memory space for each allocation block size. Within each memory space a simple list based memory management technique is used. The problem with this approach is that it is possible that one memory space will fill while another memory space goes very under-utilized. This means that the critical point at which a filter insertion fails can happen with total memory utilization being very low. Since the final worst case memory utilization calculations must include the effects of memory management, this is an unacceptable characteristic. For example, with 17 memory spaces memory utilization could be approximately 6% at the point of insertion failure.

A possible way to avoid the under utilization possible with this technique is to employ programmable pointers that divide the different memory spaces. A requirement of this technique is an associated compaction process that keeps nodes for each bin packed to allow pointer movement. With perfect compaction, the result is perfect memory management in that a filter insertion will only fail when memory is 100% utilized. This is the general approach taken by the reference design for memory management.

### 5.3.3. Reference Design Memory Management

Compaction without strict rules is difficult to analyze, difficult to implement, and does not provide any guarantees with regards to update rates or memory utilization at the time of route insertion failure. What is needed is compaction operations that are tied to each update such that guarantees can be made. Presented here is a simple memory management algorithm that uses programmable pointers, and *reactive compaction*.

For the reference design, there are 17 different possible allocation block sizes. The minimum allocation block is 2 nodes since each allocation block must have an allocation

block header. The maximum allocation block size is 18 nodes. This would occur when you have a 16 node child array, the required allocation header node, and an internal node for a parent that is a trie node. The memory management algorithm presented here first requires that all blocks of each allocation block size be kept in a contiguous array in memory. A begin and end pointer bounds the 2 edges of a given memory space. Memory spaces for different block sizes never inter-mingle. Therefore, memory as a whole will contain 17 memory spaces, normally with free space between the memory spaces (it is possible for memory spaces to abut).

To fully utilize memory we need to keep the memory space for each allocation block size tightly compacted with no holes. For the reference design there are 34 pointers total, 17  $\alpha$  pointers representing the ‘top’ of a memory space and 17  $\beta$  pointers representing the ‘bottom’ of a memory space. For the reference design these 34 pointers (each 17 bits in length to allow addressing across the maximum of 128k nodes) are assumed to be in registers though they could also be implemented in a small memory (17 bits wide and 34 entries deep).

Figure 5-10 shows an example of a portion of memory space near the top of main memory with memory spaces for allocation block sizes of 2, 3, and 4 nodes. For Figure 5-10 note that the memory spaces for 3 node allocation blocks and 4 node allocation blocks are abutted. Also the 4 node allocation area abuts with the top of the 5 node space (not labeled).

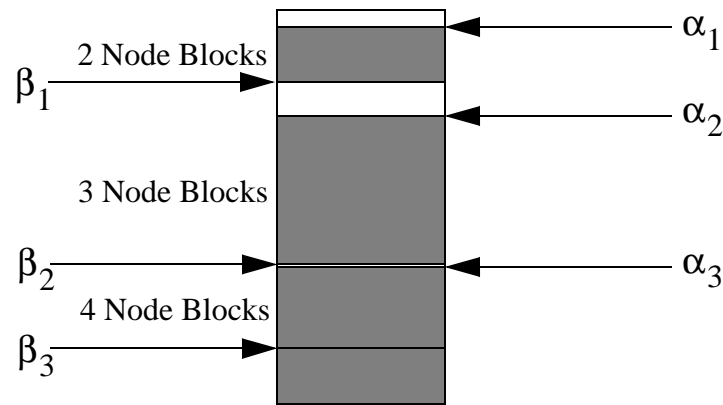


Figure 5-10 Programmable Pointer Based Memory Management

The rules for allocating a new free block are:

1) When an allocation block of a given size is needed, check if there is room to extend the size of the memory space for that block size either up or down. Note the memory spaces are kept compacted so there is no need for a free list within the memory space. After allocating the new node, adjust the appropriate end pointer (either  $\alpha$  or  $\beta$ ) for the memory space.

2) If the memory space cannot be extended either up or down, a linear search is done in both directions until a gap is found between two memory spaces large enough for the desired allocation size. For whatever distance from the free space to the memory space needing an allocation, the free space will have to be passed by copying the necessary number of allocation block in each memory space from one end to the other. For each memory space the free space must be passed through, this might require copying one or more allocation blocks from one end to the other and adjusting its pointers. Additionally, for each allocation block moved, the parent of the block must have its child address pointer adjusted to reflect the block's new location. The mechanism for finding the parent of an allocation block is not a simple one and is described in more detail at the end of this section.

For the example shown in Figure 5-10, if an allocation is needed of a block that is 4 nodes in size, it will need the free space between  $\alpha_2$  and  $\beta_1$  (we will assume the free space is at least 6 nodes in size). To do this allocation, two blocks from the 3 node memory space will be moved from the bottom of the 3 node memory space to the top of that space. Then  $\alpha_2$  and  $\beta_2$  will be shifted UP 6 memory locations and  $\alpha_3$  will be shifted UP 4 memory locations making room for the newly allocated block. Note that 2 nodes are left between the memory space for 3 and 4 nodes.

For the reference design, the worst case allocation scenario is that the 18 node memory space runs out of adjacent free space, and all the free space remaining is at the other end of memory between the 2 node memory space and the “top” of memory. With the assumption that memory spaces are organized in order {allocation block size of 2 nodes, 3 nodes, ..., 18 nodes} then for this worst case scenario 34 nodes of free space must be passed from the far side of the 2 node memory space to the 18 node memory space. The reason that 34 free nodes must be passed from end to end rather than the exact 18 nodes needed is that as the free space propagates through memory, the larger nodes (9-17) must still copy 2 nodes from one end of their memory space to the other to pass at least 18 nodes of free space. For the 17 node memory space, copying of 2 nodes means that we actually need to be passing 34 nodes of free space. A very simple upper bounds (overtly conservative but easy to calculate) of the number of memory accesses to allocate a block can be found by counting the number of memory accesses to move 34 nodes across all of memory and adding six memory accesses for every block moved to account for the search and modify operation for the parent of each block moved (explained more later). This would be  $\{34 \times 17 \times 2\}$  or 1156 memory accesses to move all the blocks and  $\{6 \times (2 \times 9 + 3 \times 3 + 5 + 6 + 7 + 8 + 9)\}$  or 696 memory accesses to update the parent of every allocation block moved. With 1 control memory access allowed every 40 ns, the worst case memory allocation time is 74 microseconds. For later use we also note that the worst case allocation time for a node of size 2 is significantly less than 74 microseconds. This is because in the worst case a single allocation block of each memory space must be

propagated through memory. This indicates a worst case of  $2 \times \sum_{i=2}^{17} i$  or 304 memory

accesses for the movement of the allocation blocks and  $\{16 \times 6\}$  or 96 memory accesses for the updating of the parents of the allocation blocks being moved. The final result is 400 memory accesses or 16 microseconds to allocate an allocation block of 2 nodes in size.

Deallocation of blocks is very simple. For the majority of the cases the deallocated block will be inside the memory space and not on the edge. For this typical case, simply copy one of the blocks at the edge of the memory space into the now vacant block. Then adjust the appropriate end pointer to reflect the compressed memory space. If the deallocated block is on the edge of the memory space then and all that needs to be done is pointer manipulation. The worst case for deallocation is the copy of a single allocation block, or up to 36 memory accesses.

One of the problems with compaction mentioned above is that when an allocation block is moved in memory, the parent of the allocation block must have its pointer modified to reflect the new position. The problem with this requirement is that the normal data structures for search do not include any pointers UP the tree. There are two ways around this problem, one way is to simply include a parent pointer for every node or for every allocation block. Since there is a 17 bit address space for the reference design this pointer would be 17 bits. The problem with this approach is that when a given allocation block is moved all children arrays of the block must change their parent pointer. For the reference design with a maximum of 17 children nodes (recall the 18th node of a maximum size allocation block is the allocation header), this can significantly increase the worst case allocation time and would be worse with a longer stride. Another method for finding a parent is to put with every allocation block the search value that would be used to get to this node. To find the parent of a block, a search is performed until the parent of node being moved is found and modified. For the reference design we will use this second method and require a header for every allocation block that includes the search address used to get to this allocation block. A secondary use of this header block as mentioned previously is for the efficient implementation of path compression. There can be an arbitrary skip

between parent and child since at the bottom of the search, a complete address for the block containing the final node is checked.

## 5.4. Updates

Forwarding-table updates describe any changes to the table. Types of updates might be *insertion* of a new prefix, *deletion* of a prefix, or a *change* of a next hop pointer. The classical approach to forwarding-table organization has the search structures ending in a next hop pointer that can be used to simply index into a table with output port, Qos, and more. This means that the most common routing operation which is the change of next hop for a given prefix, will result in the simple change of a next hop pointer. The less common operations of insertion and deletion only occur when a new prefix is added or deleted from the routing database. However, there is a growing desire to support high update rates of all types. With the number of ports on a typical router increasing from a dozen to several hundred it is unclear to what degree routing update frequency will increase, but it will increase. Therefore the following update characteristics are desirable:

- \* Incremental updates that can be done without affecting lookup rate and without regenerating the forwarding table

- \* High speed updates

- \* An additional requirement which is more related to implementation, every lookup must be correct. This means that a lookup can get the old or new entry during the update, but there should never be an error lookup under any condition

### 5.4.1. Update Procedures

Out of the three update procedures, prefix insertion is the most complex and is detailed here for the reference design. When an update arrives with prefix, prefix length, and next hop pointer, the following steps are taken:

1) Traverse trie to find where the new prefix either deviates from the established multi-bit nodes or to find the existing node in which the new prefix should be placed.

2) If the new prefix belongs within an existing multi-bit node then:

a) If the existing node is a trie node with a mirror internal node of the CAM type or an end node of the CAM type, and there is room for another prefix in the node then insert the new prefix and the update is done.

b) If the existing node is a trie node with a mirror internal node of the CAM type or an end node of the CAM type, and there is NO room for another prefix, then the existing node must be converted to a bitmap type node and a new block must be allocated as size 2 with an allocation header and a single result node with 3 prefixes.

c) If the existing node is a trie node with a mirror internal node of the bit-map type, or an end node of the bitmap type, then modify the bitmap to include the new prefix. Next the result array must be modified to insert the new prefix, this may involve the addition of a new result node which means a new allocation block of a size one larger than the old result array must be allocated, the modified array copied in, and the old allocation block deallocated.

3) If the prefix does not belong within an existing multi-bit node and the longest matching node is of the type trie, then the trie node must be updated to include an extending path to a new CAM-end node containing the new prefix (since the new node has 1 prefix and no extending paths it is of the CAM-end type). The details of this action are:

a) Allocate a block 1 size larger than the previous child array of the parent that is the longest matching node.

b) Rebuild the child array in the new allocation block including the new node of type CAM-end. Set the skip\_length field to the appropriate number of levels skipped between the last node and the new node.

c) Modify the parent to include a pointer-bit to the new node by setting the appropriate bit in the extending bitmap.

d) Deallocate the old allocation block.

4) If the prefix does not belong within an existing multi-bit node and the longest matching node is of the end type, then the end node must be converted to a trie node and a new child allocation block created with a CAM-end node. The skip\_length field of the new node should once again be set to indicate the number of levels skipped between the node previously of the end type and the new CAM-end node.

For all of the above update scenarios, the allocation time will dominate. In the worst case a block of indeterminate number of nodes and a block with 2 nodes need to be allocated. Section 5.3.3. on page 71 presents 74 microseconds and 16 microseconds respectively for the above allocations. Given the small number of memory accesses for the update itself, 100 microseconds is a conservative upper bound for a worst case prefix insertion. Since prefix deletions are similar to insertions 100 microseconds is a good upper bound for all updates. A smaller upper bound than this could be found with a more precise counting of clock ticks for memory management operations as well as update operations.

## **5.5. Design Details**

### **5.5.1. Memory Access Patterns**

One design detail glossed over up to this point is that for a real implementation, a given lookup can not be done sequentially due to the latency involved with each memory

access. Figure 5-11 below illustrates a conservative delay picture for a high performance ASIC design in which the address is registered before going to the Synchronous SRAM and the resulting data is registered again after exiting the SRAM. The register shown inside the SRAM signifies that the SRAM is synchronous and induces a single cycle latency itself. For the IBM SA-12 process being targeted by this design, this access is known as a mid-cycle access[14]. In high speed designs the SRAMs in the SA-12 need an additional cycle of latency, in this case the register for data from the SRAM to the search logic could possibly be removed. The result is a 3 cycle latency from the point the search logic generates an address to the point at which the search logic has the result of the memory access.

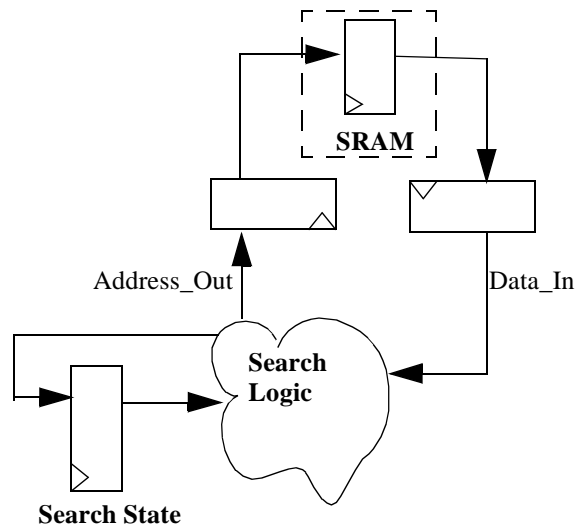


Figure 5-11 RTL View of Search Engine

Given a 3 cycle latency for memory accesses, if lookups were done sequentially it would take 3 times longer than previous calculations suggest. However, if lookups for three separate searches are interleaved then the throughput of lookups can be maximized with a cost of 3 times the latency. Since 3 times the latency of one lookup is 120 ns for the reference design this is still acceptable.

There are many different ways the low level memory access schedule could be worked out for the lookups of three independent addresses at the same time. In each 120 ns period there will be 7 memory accesses for each of the 3 searches and 3 control

accesses for updates and memory management. Similar to address searches, a desired property of the schedule is that control accesses are spaced in order to allow the control logic to be effective with each memory access.

Figure 5-12 shows a possible memory access pattern which would meet all the requirements set forth. The signals defined in the timing diagram are Address\_Out and Data\_In, both of which match the signals of the same name in Figure 5-11. In clock cycle 0, Address\_Out is receiving the first address of a new search in the 'A' pipeline. During the same cycle the data for the final access of the 'B' pipeline is presented to the search logic by the Data\_In signal. In cycle 1 the first memory access of a search is launched for the 'B' pipeline while data for the final memory access of a search for the 'C' pipeline arrives for the search logic to evaluate. In cycle 3 the first memory access of a new search in the 'C' pipeline is launched and the results of a control operation (denoted with \*\*) is received by the search logic (or in this case is really consumed by the control logic). The basis of the schedule is a simple A-B-C pattern with a control operation injected every 8 cycles. A property of this memory access schedule is that all three searches in the pipeline begin and end close together in time.

	0	1	2	3	4	5	6	7
Address_Out	A1	B1	C1	A2	B2	C2	A3	**
Data_In	B7'	C7'	**'	A1	B1	C1	A2	B2
	8	9	10	11	12	13	14	15
Address_Out	B3	C3	A4	B4	C4	A5	B5	**
Data_In	C2	A3	**	B3	C3	A4	B4	C4
	16	17	18	19	20	21	22	23
Address_Out	C5	A6	B6	C6	A7	B7	C7	**
Data_In	A5	B5	**	C5	A6	B6	C6	A7

Figure 5-12 Memory Access Patterns

## 5.5.2. External Interfaces

For the reference design, the goal of the external interface is to be as simplistic as possible since the details of communication within the rest of the IP Forwarding ASIC chip are orthogonal to most of the complexities of IP address lookup addressed in this paper. Figure 5-13 shows all the external signals of the reference design.

The memory access schedule presented in the last section requires that lookups are done three at a time, this requires a queue of search addresses waiting to enter the search engine and a queue upon exit from the search engine since latency is stated to be fixed through the lookup engine as a whole. The suggested design is the simplest which involves room for 3 cells in the queue before the search engine and a queue of 3 cells upon exit of the search engine. The cells exiting from the search are delayed if necessary so that all cells have a latency through the lookup engine of exactly 360 ns or 9 search times.

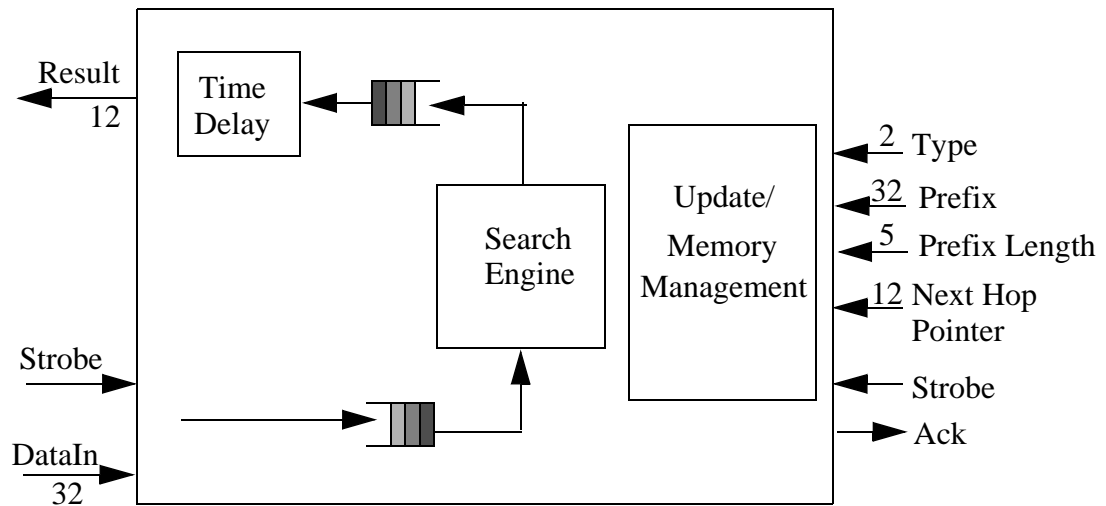


Figure 5-13 Reference Design External Interface

## 6. Reference Design Analysis

Since the reference design had deterministic lookup rate as a design parameter, the lookup performance either empirically or analytically is not something that needs review. The areas of non-determinism that need analysis are the memory usage, and the implementation cost. For memory usage both an analytic and empirical approach can be taken. For implementation costs, the size of the design in an ASIC implementation is used as a cost metric when considering the trade-offs of varying sizes for the main memory.

An additional area of non-determinism in the design is the update rate. In Chapter 5 a simple upper bounds of 100 us was found analytically. This yields a possible update rate of 10k per second which is very aggressive compared to today's router implementations.

### 6.1. Empirical Memory Usage Results

This section explores the empirical results of mapping five databases from the IPMA onto the reference design. This was done by creating a C program that models the update and memory management algorithms that would be implemented in hardware.

Table 6-1 presents the basic results of the simulation. These results do not include nodes stored in the initial array, additionally trie nodes skipped due to path compression are not counted. The first row presents the number of prefixes in the database which ranges from about 3 thousand to over 40 thousand. Rows 2 through 6 present the node counts for the 5 major "search" nodes. These nodes are one of 3 basic types: trie nodes,

internal nodes (nodes mirroring trie nodes with at least 1 prefix), and end nodes (replacing trie nodes with no extending paths). For both internal and end nodes with less than 2 prefixes, the node is of CAM type with the next hop pointers embedded in the node, otherwise an internal bitmap is used. Internal bitmaps point at one or more result nodes that each contain up to 3 prefixes. The number of result nodes are found in row 7 of the table. For every allocation block an allocation header node is needed and row 8 presents the number of nodes of this type. The second to last row presents the total number of nodes needed to represent the database.

**Table 6-1 Reference Design Empirical Size Results**

	Database Name				
	Mae East	Mae West	Pac-Bell	AADS	PAIX
Number of Prefixes	40902	19229	22246	23373	2994
Trie Nodes	2411	1588	1714	1775	521
CAM Internal Nodes	1117	620	632	673	55
Bitmap Internal Node	270	520	58	73	48
Bitmap End Nodes	4765	1448	2546	2709	183
CAM End Nodes	9779	6145	6832	7042	1805
Result Nodes	10568	4179	5240	5306	278
Allocation Nodes	7526	3975	2673	4221	768
Total Nodes	36436	18475	19695	21799	3658
Nodes Per Prefix	.89	.96	.89	.93	1.22

Looking at the results of the simulation in Table 6-1, one simple observation is that the total number of nodes is approximately equal to the number of prefixes within about 15%. The final row presents the number of nodes per prefix for each database. Given the five databases the ratio of nodes per prefix seems to decrease as the database gets larger. For future databases larger than Mae East, it could be extrapolated that the ratio of 0.89 Nodes for every prefix is a conservative estimate. For the reference design with possible main memory sizes of 16k, 32k, 64k, 98k, and 128k nodes, we can conservatively estimate the node to prefix ratios will be: {1.22, 0.96, 0.89, 0.89, 0.89}. This suggests that actual

database sizes supported for the varying memory sizes might be {13k, 33k, 72k, 110k, 143k}.

Table 6-2 presents a more detailed analysis of the data presented in Table 6-1. The first row in Table 6-2 repeats the prefix count for each database, the second row gives the total size of the table for each database. The third row gives the total number of bits stored for every bit of next hop pointer. This number can be useful for comparing storage requirements with other lookup schemes since the size of the next hop pointer varies in the published literature for other schemes. Notice that the bit ratio generally decreases as the number of prefixes decreases. This implies that larger databases are more efficient with less bits per prefix bit. The third row summarizes the percent overhead of the allocation headers used for memory management (MM). For 4 out of 5 databases the memory management overhead is around 20% and does not seem correlated with database size. The next two rows of Table 6-2 are the table size and ‘bits per next hop pointer bit’ without the allocation headers counted. The reason the table size is recalculated without the memory management overhead, is that results for other IP address lookup schemes (like the results in Table 2-2) traditionally do not contain any memory management overhead.

**Table 6-2 Analysis of Empirical Results**

	Database Name				
	Mae East	Mae West	Pac-Bell	AADS	PAIX
Number of Prefixes	40902	19229	22246	23373	2994
Total Size	1.4 Mbits	702 Kbits	748 Kbits	828 Kbits	139 Kbits
Storage per Next Hop Pointer bit	2.85	3.04	2.8	2.95	3.87
MM Overhead	20%	21%	13.5%	19.4%	21%
Total Size without MM	1.1 Mbits	551 Kbits	636 Kbits	668 Kbits	109 Kbits
Storage per Next Hop Pointer bit (Without MM) {bits}	2.24	2.38	2.38	2.38	3.03
Path Compression Savings	16%	24%	23%	21%	35%

The total storage per next hop pointer bit without memory management for Mae East is 2.24 bits per prefix. For Lulea[6] with the Mae-East database (dated January 1997

with 32k prefixes) 160 KB are required which indicates 39 bytes per prefix or 2.8 bits per next hop pointer bit (they use 14-bit next hop pointers). This analysis suggests that the reference design and of Tree Bitmap is similar in storage to Lulea but without requiring a complete table compression to achieve the results. The final row of Table 6-2 gives the trie node savings by the use of path compression. On average 1 out of every 5 original trie nodes had only 1 extending path and no internal prefixes allowing it to be optimized out with path compression. While the main reason path compression was introduced was to improve worst case bounds on memory storage, the empirical results indicate a real savings in practice by the optimization as well.

## 6.2. Projected Worst Case Memory Usage Analysis

As was explored in Section 5.2.2. on page 63 without path compression the worst case scenario is when every prefix is maximum length and therefore requires a node for every level of the trie, (two nodes with allocation headers taken into account). With the path compression approach, this is no longer the worst case. In fact when path compression, CAM nodes, internal nodes, result nodes, and allocation headers are all taken into account, the analysis of the worst case memory usage is quite difficult. The approach taken here is to explore multiple ‘bad scenarios’ that exploit various weaknesses of the reference design in order to project a plausible worst case.

For the first scenario, we will try to recreate the non-branching path scenario described above by placing a prefix at each node to keep path compression from working. Imagine paths from the initial array to the bottom of the tree in which every trie node has 1 internal prefix and 1 extending path (note this scenario is not quite possible since at the top levels there would not be enough possible nodes for a single extending path from each node with any reasonably large database). For this scenario each prefix is in an allocation block with an allocation header, a trie node (from the extending path), and an internal

node (actually a CAM internal node since there is only 1 prefix). The result is for  $n$  prefixes, there will be  $3n$  nodes in the worst case. For this scenario the maximum amount of memory considered for the reference design (128k nodes) would yield support a worst case database of 42k prefixes. This scenario exposes the overhead of the allocation header, does not have any path compression, and forces an internal node for every prefix above the leaves.

For the next scenario, the above non-branching path case is altered to add more trie node overhead but keep the allocation header overhead still significant. The approach is to create a binary tree with the nodes above the leaves each having 1 internal prefix and 2 extending paths. This arrangement requires internal nodes (but of the CAM type) for every prefix attached to a trie node. In this scenario each allocation block above the leaves typically has the header, 2 trie nodes, and an internal node (with a single prefix). This means there are 4 nodes to every prefix. For the prefixes at the leaves of the binary tree, there will be an allocation header and CAM node or 2 nodes per prefix. Note the binary tree is not full since the leaves of the binary tree do not have siblings. For a full binary tree, there would be approximately  $n$  prefixes at the leaves and  $n$  prefixes above the leaves. In this case there are half the number of nodes at the leaves so there are  $1/3*n$  prefixes at the leaves of the tree and  $2/3*n$  above the leaves. If  $n$  is the number of prefixes then the equation for the required number of nodes is:

$\left\{ \frac{2}{3} \times n \times 4 + \frac{1}{3} \times n \times 2 \right\}$  or  $3.33n$ . This means

that with 128k nodes (the high end of memory space possible for the reference design) about 38k prefixes would be guaranteed storage if this scenario were worst case.

For the next scenario, the waste in the use of result nodes is exploited (as opposed to having all next hop pointer results in CAM nodes as in the above scenarios) by thinking about groups of 3 prefixes together. If prefixes existed alone or in groups of 2, the internal node would be of the CAM type and not require a separate result node. We take the binary tree scenario and substitute groups of 3 prefixes every time there was 1 before. Above the leaves for every trie node with internal prefixes there would be a child array with an allocation header, 2 trie node children, and an internal node. Additionally the internal node in

this case will point to a result array that will contain an allocation header and a single result node. The result is 6 nodes for three prefixes which is a ratio of 2 nodes per prefix. For the leaves there will be an allocation header and an end node which will point to a result array with an allocation header and a single result node. Therefore the leaves will have 3 prefixes and 4 nodes for a ratio of 1.33 nodes per prefix. The final result is

$\left\{ \frac{2}{3} \times n \times 2 + n \times \frac{4}{3} \times \frac{1}{3} \right\}$  or 1.75 nodes per prefix. Clearly this is much better than either of the previous scenarios. The summary for this example is that the overhead of result nodes and additional allocation headers is more than offset by the advantages of concentrated prefixes in one place.

In summary the worst case scenario found in this section has a ratio of 3.33 nodes per prefix which is conjectured to be the worst case for the design. To complete the analysis the explicit effects of memory management on memory usage must be included (the implicit cost of the allocation headers used for memory management have already been counted). As presented in Section 5.3.3. on page 71, 34 nodes is the explicit overhead for memory management. This is such a small number compared to the number of nodes in a typical database (10's of thousands) it will not significantly effect any numbers.

The worst case analysis of memory usage can be an iterative one with modifications to the lookup algorithm and data structures. For example the CAM node and path compression techniques are not fundamental to the correctness of the lookup algorithm but simply aid in empirical and analytical memory usage numbers. Additional optimizations to counter the above pathological scenarios could include new node types that compress a small number of internal prefixes and extending paths into a single CAM type node.

## 6.3. Implementation Analysis

### 6.3.1. Gate Count Analysis

In this section the implementation costs of the reference design presented in Section 5. on page 58 are estimated. In the early analysis of a design there are several parameters that can help characterize a design. Some of these parameters answer the following questions:

- \* How many clock domains in the design and what are the rates of each domain?
- \* What is the width of the interface to the design? For analyzing an entire ASIC this results in a pin-out analysis, for a block of an ASIC (like the reference design was focused on) the interface is internal.
- \* How many flip flop bits are there? How many gates?
- \* How many SRAMs are there of types single and dual port? How many total bits of SRAM for each type?

Some of these questions can be answered based on the details presented so far. The reference design is a synchronous design running at 200 Mhz throughout. The external interface is 102 bits wide and also operates at 200 Mhz. To estimate the number of register bits in the design it is necessary to review the various blocks in the design and call out the storage necessary for various functions as is done in Table 6-3.

**Table 6-3 Flip Flop Count in Reference Design**

Block	Function	Flip Flop Count
Main Memory SRAM Interface	For every Address and Data bit interfacing to the ram, the signals need to be registered in this block $\{38*2 + 17\}$	93

**Table 6-3 Flip Flop Count in Reference Design**

Block	Function	Flip Flop Count
Initial Array SRAM Interface	For every Address and Data bit interfacing to the ram, the signal needs to be registered in this block $\{56*2 + 9\}$	117
Search Engine	A complete node needs to be registered for each fetch for each of the three lookups in parallel $\{3*(38*7)\}$	798
	Every external signal needs to be registered	45
	All search addresses within the Lookup Engine at the same time must be stored $\{9*32\}$	288
Memory Management	For each of the 17 memory spaces, a begin and end pointer needs to be stored $\{17*17*2\}$	578
	When copying block a node at a time, the current node and dual address pointers are needed $\{17*2 + 38\}$	72
Updates	Every external signal needs to be registered	57
	The update logic performs searches on the prefix to be inserted and must be able to store all nodes in search $\{38*7\}$	266
	When adding a child to a node the old and new nodes must be stored in registers for manipulation $\{38*2\}$ .	76
Miscellaneous	To attempt and account for the flip flops that have been overlooked (like state machine storage, additional pipelining storage, etc.), an additional 250 flip flops are added to the tally.	250
Total		2640

With a total of 2640 flip flops, an estimate can be made of the gate count by using an approximation of 7 gates per flop for a total of 18480 gates. This approximation includes a built-in scan path which is included with many high end ASICs produced commercially for testing. Since the target process for this reference design is the IBM SA-12 process, the gate count per flop would ideally come from the data book, but that number is not supplied. For the estimation of combinational logic the normal approach is to first estimate the size of major logic structures like comparators, multiplexors, and adders. The IP Lookup engine while obviously having significant control logic for search, updates, and memory management does not have any major logic structures. Based on prior design experience, the combinational gate count is estimated to be 50k-75k gates. With the gate

count for the flip flops added in the total gate count ranges from 68.4k to 93.4k. Using the rule of thumb that in a quarter micron process (drawn) the ratio of gates to area is 40K gates/mm<sup>2</sup>, we can estimate gate area as ranging from 1.71 mm<sup>2</sup> to 2.33 mm<sup>2</sup>.

### 6.3.2. Design Summary

For SRAM, the reference design has assumed a range of possible main memory sizes: 16k, 32k, 64k, 96k, and 128k nodes. Table 6-4 shows these 5 different memory sizes and for each presents the total table size, total design area (for SA-12 process), and the percent of an ASIC (144 mm<sup>2</sup>) the design occupies. The final two columns first report the worst case number of prefixes supported with that table size based on the calculated ratio of 3.33 nodes per prefix (Section 6.2. on page 85), and the empirical prediction for table size (Section 6.1. on page 82).

**Table 6-4 SRAM Quantity effect on Die Area**

Size of Main Lookup Table (Number of Nodes)	Size of Main Lookup Table in KBits <sup>a</sup>	Total Area <sup>b</sup> (Square mm)	Percent of ASIC for Main Memory <sup>c</sup>	Worst Case Number of Prefixes	Empirical Prediction for Number of Prefixes
16k	608	12.63	8.8%	4800	13k
32k	1216	22.93	15.9%	9600	33k
64k	2432	43.53	30.2%	19200	72k
96k	3648	64.13	44.5%	28800	110k
128k	4864	84.73	58.8%	38400	143k

a. Given 38 bits per node

b. With 59k bits per square mm in quarter micron IBM process (SA-12)

c. With a 1.2 cm per side of die

### 6.3.3. Design Scalability

Given that ASIC solutions can be iterated less slowly than say software solutions, the length of time a single implementation (like the reference design presented in this thesis) will be effective is an important consideration. Additionally, the long term application of the Tree Bitmap Algorithm (with future revisions of the lookup chip) to the problem of IP Lookups is an interesting question.

The main area of growth that we consider here is in database size. To project future database sizes a history of total BGP prefixes from January 1, 1995 to January 1, 1999 was studied[45]. The growth in a 4 year time period was approximately linear and had a slope of 9.25 thousand prefixes per year. In January 1999, the report shows approximately 61k BGP prefixes in existence. If a lookup engine was required to hold all BGP network prefixes, and if the prefix growth rate continues to be linear then the reference design with 128k entries (projected typical support of up to 143k prefixes) could potentially be large enough until the year 2008. In reality several reasons would result in not using the same ASIC design for that length of time; Reasons includes: database change in distribution causing worse prefix to memory ratio, increase in line rates causing OC-192c rate to become obsolete, change in Internet standards (like increase of IP address length).

For future IP Lookup Engine designs implementing the Tree Bitmap Algorithm, an estimate can be made of the database sizes that could be supported. A recent empirical version of Moore's Law quantifies the growth of large commercial ASICs to be an order of magnitude in complexity every 5 years[42]. This suggests that if today a 5 Mbit lookup table is feasible, in 5 years we could see a 50 Mbit SRAM table. As the table size increases, the address pointers will have to increase also. With a 24 bit address pointer, up to 16 million table entries could be handled. If table entries are increased by 7-bits to handle the increased addresses, a 1 million entry lookup table could be constructed in a high end ASIC in 5 years. If the database distributions continued to resemble the Empirical results collected in this thesis, this would indicate that a typical database size of over 1 million prefixes could be fit in the table. Additionally, guarantees could be made of support

for hundreds of thousands of prefixes. During this same 5 year time period, if the growth of prefix count in forwarding databases follows historical trends (discussed in the last paragraph) they would grow to a size of approximately 107k prefixes. If prefix databases continue to grow linearly, and ASIC complexity exponentially, the Tree Bitmap Algorithm should continue to be a reasonable choice for prefix address resolution. With increases in link rates, however, the goal will be to handle growing database sizes while also increasing the lookup rate. Some approaches to increasing lookup rate involve parallelism and pipelining with segmented tables. Both of these approaches to increasing the lookup rate will reduce the actual number of prefixes supported.

## 7. CONCLUSIONS

The goal of this thesis is to detail a new algorithm for IP Lookups called Tree Bitmap and then illustrate that approach with a reference design.

The research contributions of the thesis are listed below:

### **Taxonomy and evaluation of existing IP Lookup schemes**

There have been numerous approaches to IP Lookups in the past two years, but this is the first attempt to evaluate all known approaches for hardware implementation. The details of these approaches are explained and then a summary of the strengths and weakness of each of the algorithms are presented.

### **Tree Bitmap algorithm for IP lookups**

A new algorithm for IP Lookups is presented. This Algorithm is optimized for hardware implementation and balances high insertion rates, fast lookup time, and small memory size.

### **Reference implementation of an IP Lookup engine**

A reference implementation is presented which illustrates the tree bitmap algorithm in a real system. The reference design operates at 25 million lookups per second which is able to support the maximum packet rate for OC-192c links. The memory usage for the reference design is better than 1 table entry (38 bits) per

prefix across several sample databases. For Mae East (a popular 41k prefix table) a total of 36k memory entries were required for a total storage of 1.4 Mbits.

### **New memory management algorithm**

A new Memory Management Algorithm was presented that is optimized for the tree bitmap algorithm. For several simulations across a variety of databases, the total overhead for memory management was under 21 percent. Conventional memory management algorithms do not provide the performance or low overhead of the scheme presented in this thesis.

There are several questions that need to be further explored.

First, memory management and updates were explored only for the entirely on-chip reference design. With off-chip implementations the amount of memory is potentially orders of magnitude greater, random access rates are normally slower, the memory must often be segmented, and the multiplexing of read and write accesses are problematic due to a single multiplexed data bus. Clearly table management with off-chip implementations needs to be further explored.

Second, worst case update bounds were not aggressively analyzed in this thesis. For a real ASIC implementation, especially a commercial one, detailed analysis must be done down to the clock tick to provide worst case update times giving the highest possible update rate. Improvements to the search or memory management algorithms may also improve worst case memory usage bounds.

Finally, the problem of hardware design for more general packet classification in which there are prefixes in both the source IP address and the destination IP address is an area that needs more exploration.

## REFERENCES

- [1] Artisan Components, Inc., <http://www.artisan.com/>
- [2] F. Baker, “Requirements for IP Version 4 Routers”, rfc1812. June 1995
- [3] Berkley Networks, <http://http://www.berkeley.net.com/>
- [4] U. Black, S. Waters, *SONET & T1: Architectures for Digital Transport Networks*, Prentice Hall, 1997
- [5] Cisco, “Cisco 12000 Gigabit Switch Router“ , [http://www.cisco.com/warp/public/733/12000/gsrfs\\_ds.htm](http://www.cisco.com/warp/public/733/12000/gsrfs_ds.htm)
- [6] M. Degermark, A. Brodnik, S. Carlsson, S. Pink, “Small Forwarding Tables for Fast Routing Lookups” Proc. ACM SIGCOMM ‘97, , Cannes (14 - 18 September 1997).
- [7] V. Fuller, et al. “Classless Inter-Domain Routing (CIDR) : an address assignment and aggregation strategy (RFC1519). <ftp://ds.internic.net/rfc/rfc1519.txt>, 1993.
- [8] E. Gray, Z. Wang, G. Armitage, “Generic Label Distribution Protocol Specification”, <draft-gray-mpls-generic-ldp-spec-00.txt> , May 1998.
- [9] P. Gupta, S. Lin, N. McKeown, “Routing Lookups in Hardware at Memory Access Speeds,” Proc. IEEE Infocom ‘98.

- [10] IBM, 64Mb Direct Rambus DRAM Advance Datasheet, December 1997.
- [11] IBM, ASIC SA-12 Databook, September 1998.
- [12] IBM, Direct Rambus Memory System Overview Application Note, March 30, 1998.
- [13] IBM, Double Data Rate SDRAM, IBM0664404ET3A, 128k x 8 x 4 banks, August 1998
- [14] IBM, Rams and RAs in 5SE and SA-12 ASICs, [http://www.chips.ibm.com/products/asics/document/appnote/221500\\_2.pdf](http://www.chips.ibm.com/products/asics/document/appnote/221500_2.pdf)
- [15] IBM, Synchronous DRAM Datasheet, IBM031609CT3-322, 1- Meg x 8 x 2 banks, January 1998
- [16] IEEE Draft P802.3z/D3.0 "Media Access Control (MAC) Parameters, Physical Layer, Repeater and Management Parameters for 1000Mb/s operation", June 1997.
- [17] Intel, *PC SDRAM Specification*, Version 1.51 , (November, 1997)
- [18] Inventra, Single Port Static RAM - Performance Data Sheet - TSMC 0.25um, <http://www.mentorg.com/inventra/physical/datasheets/TSMC25spra698.pdf>
- [19] S. Keshav, R. Sharma, "Issues and Trends in Router Design", in IEEE Communications Magazine , May 1998.
- [20] D. Kosiur, *IP Multicasting : The complete guide to Interactive Corporate Networks*, John Wiley & Sons, 1998.
- [21] V.P. Kumar, T.V. Lakshman, D. Stiliadis, "Beyond Best-Effort: Gigabit Routers for Tomorrow's Internet," in IEEE Communications Magazine , May 1998.

- [22] C. Labovitz, G. R. Malan, F. Jahanian. "Internet Routing Instability." Proc. ACM SIGCOMM '97, p. 115-126, Cannes, France.
- [23] T.V. Lakshman, D. Stiliadis, "High-Speed Policy-based Packet Forwarding Using Efficient Multi-dimensional Range Matching", Proc. ACM SIGCOMM '98, Vancouver, Canada.
- [24] B. Lampson, V. Srinivasan, G. Varghese, "IP Lookups using Multiway and Multicolumn Search," Proc. IEEE Infocom '98.
- [25] J. Manchester, J. Anderson, B. Doshi, S. Dravida, "IP over SONET", IEEE Communications Magazine, May 1998.
- [26] N. McKeown, "Fast Switched Backplane for a Gigabit Switched Router", <http://www.cisco.com/warp/public/733/12000/technical.shtml>
- [27] Merit Inc. IPMA Statistics. <http://nic.merit.edu/ipma>
- [28] Micron, Double Data Rate SDRAM, MT46LC8M8 - 2 Meg x 8 x 4 banks, May 1998
- [29] Micron, Next -Generation DRAM Product Guide, September 1998.
- [30] Micron, SLDRAM Datasheet, MT49V4M18 - 4 Meg x 18, May 1998
- [31] Micron, Synchronous DRAM Datasheet, MT48LC4M8A1/A2 - 1 Meg x 8 x 2 banks, October 1997
- [32] Micron, Syncburst SRAM Advance Datasheet, T58LC256K18G1, MT58LC128K32G1, MT58LC128K36G1, February 1998.
- [33] Music Semiconductor, A Method For Fast IPv4 CIDR Address Translation and

Filtering Using the MUSIC WidePort LANCAM Family, Application Note AN-N22, <http://www.music-ic.com/>

- [34] MUSIC Semiconductors, MUAC Routing Coprocessor Family, <http://www.music-ic.com/>
- [35] MUSIC Semiconductors, What Is A CAM (Content-Addressable Memory)? Application Brief AB-N6, <http://www.music-ic.com/>
- [36] National Laboratory for Applied Network Research, “WAN Packet Size Distribution”, <http://www.nlanr.net/NA/Learn/packetsizes.html>.
- [37] Netlogic Microsystems, Ternary Synchronous Content Addressable Memory (IPCAM) , <http://www.netlogicmicro.com/>
- [38] Network Wizards, Internet Domain Survey, <http://www.nw.com/zone/WWW/top.html>
- [39] S. Nilsson and G. Karlsson, "Fast address lookup for Internet routers"., Proceedings of IFIP International Conference of Broadband Communications (BC'98), Stuttgart, Germany, April 1998.
- [40] C. Partridge et al., “A Fifty Gigabit Per Second IP Router”, IEEE Transactions on computing
- [41] SLDRAM Inc., 4Mx18 SLDRAM Advance Datasheet, SLD4M18R400, February 2, 1998.
- [42] M. J. S. Smith, *Application Specific Integrated Circuits*, Addison-Wesley Publishing Company, 1997.
- [43] V. Srinivasan, G. Varghese, “Faster IP Lookups using Controlled Prefix Expansion,” Proc. SIGMETRICS ‘98.

- [44] V. Srinivasan, G. Varghese, S. Suri, M. Waldvogel, "Fast Scalable Algorithms for Layer 4 Switching" Proc ACM SIGCOMM '98 Vancouver, Canada.
  
- [45] Telstra, "Telstra Internet Network Performance Reports", <http://www.telstra.net/ops/bgptable.html>
  
- [46] Tzi-cker Chiueh, Prashant Pradhan, "High Performance IP Routing Table Lookup using CPU Caching," submitted to IEEE INFOCOMM 1999 .
  
- [47] A. Viswanathan, N. Feldman, Z. Wang, R. Callon, "Evolution of Multiprotocol label Switching," IEEE Communications Magazine, May 1998.
  
- [48] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, "Scalable High Speed IP Routing Lookups," *Proc. ACM SIGCOMM '97*, pp. 25-37, Cannes (14-18 September 1997).

## VITA

Bibliographical items on the author of the thesis, William Eatherton.

### **Education**

- BS in Electrical Engineering from University of Missouri-Rolla, May 1995.
- MS in Electrical Engineering from Washington University, May 1999.

### **Honors and Rewards**

- Received research assistantship from the Applied Research Lab, 1996 to 1997.
- Received departmental fellowship from Dept. of Electrical Engineering, 1995-1996

### **Work Experience**

- ASIC Engineer for the Applied Research Lab of Washington University, January 1998 to present

### **Publications**

- "An FPGA Based Reconfigurable Coprocessor Board Utilizing a Mathematics of Arrays" William Eatherton, et. al., IEEE International Circuits and Systems Symposium 1995

May 1999