Tree Bitmap : Hardware/Software IP Lookups with Incremental Updates

Will Eatherton {will@cisco.com}, Cisco Systems, Inc.
George Varghese {varghese@cs.ucsd.edu}, UCSD
Zubin Dittia {zubin@jibe.biz}, Jibe Networks

Abstract

Even with the significant focus on IP address lookup in the published literature as well as focus on this market by commercial semiconductor vendors, there is still a challenge for router architects to find solutions that simultaneously meet 3 criteria: scaling in terms of lookup speeds as well as table sizes, the ability to perform high speed updates, and the ability to fit into the overall memory architecture of an Level 3 forwarding engine or packet processor with low systems cost overhead. In this paper, we describe a scheme that meets all three criteria. By contrast, published and commercial semiconductor solutions meet some but not all of these three criteria.

For example, many approaches that provide dense tables have poor update times; others require large amounts of expensive high speed memory dedicated to this application. Many IP address lookup approaches do not take into account the flexibility of ASICs or the structure of modern high speed memory technologies such as RLDRAM[1] and FCRAM[2]. In this paper, we present a family of IP lookup schemes using a data structure that compactly encodes large prefix tables in order to address the criteria listed above. We also present a series of optimizations to the core algorithm that allows the memory access width of the algorithm to be reduced at the cost of memory references or allocated memory. Such flexibility in performance versus density is an important feature for the lookup engine of routers that may be deployed in different networks with varying requirements on address lookup length and table density (e.g. global IPv4 networks, global v6, VPN based v4 networks, MPLS, and IP tunneling encapsulation points).

1 Introduction:

The area of longest prefix match searches with particular focus on IP destination address lookups have received a great deal of attention from the networking community over the past several years [6][7][12][17][19][20][22]. The reason is that for common benchmarks tests of the datapath performance of routers, destination IP lookup has tended to be a performance bottleneck and has required significant amounts of memory. However, router architecture requirements are evolving as well as memory technology. Thus published solutions for lookups do not meet the requirements of the next generation of routers. Most algorithmic approaches to lookups are not suitable for use in future mid- to high-end routers due to

one or more of the following reasons:

o They require significant overhead for computing updates, significant bandwidth within internal router datapaths to download updates, and/or forwarding is interrupted during updates

o The algorithm is based on the assumption that the result is simply an output port id (so result is 8-10 bits) when actually due to per prefix statistics, load balancing, or different L2 header rewrites there should be a leaf (e.g. 16 byte of associated data) for every prefix in the table.

o Due to focus on typical performance or table density, the worst case performance or table density of the algorithm is unacceptable

o While the order of the complexities are not bad, the constants are significantly high.

The principal difficulty in IP lookups is that it requires a longest matching prefix computation at wire speeds. We will denote each prefix by a bit string (e.g., 01) followed by a ``*''. The prefix database is built by routing protocols such as BGP and OSPF; each prefix entry consists of a prefix and a next hop value. For example, suppose the database consists of only two prefix entries (01*--> P1; 0100* --> P2) If the router receives a packet with destination address that starts with 01000, the address matches both the first prefix (01*) and the second prefix (0100*). Since the second prefix is the longest match, the packet should be sent to next hop P2. On the other hand, a packet with destination address that starts with 01010 should be sent to next hop P1.

The next hop information will typically specify an adjacency (e.g. what output port of the router to use, and a MAC rewrite) or could point into a hierarchical load balancing data structure for L2/L3 load balancing. Note that throughout this paper we will use the common abbreviation L2 to denote Layer 2 (Data Link) and L3 to denote Layer 3 (Routing). This paper will go through the requirements of next generation routers as they pertain to lookups, look at the memory technology options available today and trends into the future, and then extract the requirements for the lookup engine/algorithms itself from a systems perspective. We then present an algorithm (Tree Bitmap) that satisfies these requirements and has sufficient flexibility that makes it reasonable to expect it to adapt to the next several generations of memory technology. We pay special attention to the update properties of Tree Bitmap and how they play into a real router system.

We note that our core algorithm differs from the Lulea [6] algorithm (which is the only existing algorithm to encode prefix tables as compactly as we do) in several key ways. First, we use a completely different encoding scheme that relies on two bitmaps per node. Second, we use only one memory reference per trie node as opposed to two or three per trie node in Lulea. Third, we have guaranteed fast update times; in the Lulea scheme, a single update can cause almost the entire table to be rewritten. Fourth, unlike Lulea, our core algorithm can be instantiated to leverage off the structure of modern memories.

The outline of the rest of the paper is as follows. In Section 2, we present the requirements of next generation routers and lookup engines as well as models of both and the memory technology options. In Section 3, we briefly review previous work and concentrate on the schemes that are most closely related to ours: the Lulea trie compression scheme [6] and the simple expanded trie schemes of [19]. In Section 4, we present the core lookup scheme called Tree Bitmap and contrast it carefully with the Lulea and expanded trie schemes. In Section 5, we describe a series of optimizations to the core algorithm that reduce the burst size required for a given stride size, and provide deterministic size requirements. In Section 6, we describe reference software implementations. In Section 7, we describe a reference hardware implementation. We emphasize that Sections 6 and 7 are based on the models in Section 2 and the core ideas and optimizations of Sections 4 and 5. We conclude in Section 8.

2 Requirements

2.1 Router Requirements and Model

Frequently lookup algorithms are proposed in context of very high end backbone routers with assumptions about a narrow feature set. Such papers also assume the willingness to dedicate resources to a particular function like lookups. While there are limited markets for routers of this type, the major focus for routers is middle to high platforms which (essentially) need to act as "swiss-army knives", able to be deployed in a wide variety of applications, thereby reducing the need for specialty boxes for applications like accounting, monitoring, intrusion detection, NAT, etc.. Table 1 illustrates a potential array of market applications and distinguishing characteristics a single router architecture might have to satisfy. The key attribute of a router architecture under these kinds of demands is flexibility and programability.

Table 1: Potential market applications for a single router architecture

Broadband (e.g. DSL)	100k's of sessions, queuing per session, fancy qos, potentially features on per session basis
Leased Line	Thousands of sessions (e.g. T1), Frame, Voice, Virtual Private Networking per session (and potentially large # of VPN routes)
Large Enterprise Router	Fancy qos, crypto, hundreds of interfaces, typically not very demanding on total # of prefixes, but can be demanding on performance
Service Provider Edge	Hundreds to thousands of interfaces, v4/v6/mpls/multicast, per interface VPN tables, lots of per interfaces, can be very demanding on both total # of prefixes and performance
Service Provider Core	Hundreds of interfaces, v4/v6/mpls global tables, very high and deterministic performance (but not as much pressure on total route table as edge)

Relative to this paper, a key component of flexibility is the ability to allocate memory bits and memory bandwidth across a wide range of features. For example, in a large enterprise there might be several hundred thousand prefixes; in a service provider core there might be 1-2 million prefixes for global v4/v6/multicast tables; in a service provider edge there might be several million prefixes for a small number of large VPN (Virtual Private Network) tables; and finally, a broadband network might have millions of prefixes but with a very large number of VPN/tunnel tables, and a small number of prefixes per table.

Statically allocating memory and memory bandwidth for the worst case scenarios is not acceptable due to limitations in power, cost, board space, and pin counts.

A router model with focus on lookups that can address the range of markets and the flexibility required is shown in Figure 1. This model focuses on a centralized forwarding engine, though the trade-offs and breakpoints between centralized and distributed is not a focus of this paper. In this model the external interfaces are aggregated and fed into a packet processor which has an array of n processors that packets are distributed to. A key point of this model is that there are *m* general purpose memory controllers accessible by the processors as well as a lookup engine that is a HW block that can facilitate the rapid lookup of prefixes in the external memory. While there are subtle variations this is a common network processor architecture at macro level[28]. .

As part of the router model it should be noted that while the number of physical plug holes on a router limits the number of physical interfaces to say dozens, the number of logical interfaces (taking into account ATM VC's and Ethernet VLANs) can be in the 10's of thousands. For purpose of this paper we will assume a router model with 64k logical interfaces. This is important since some prefix lookup schemes are

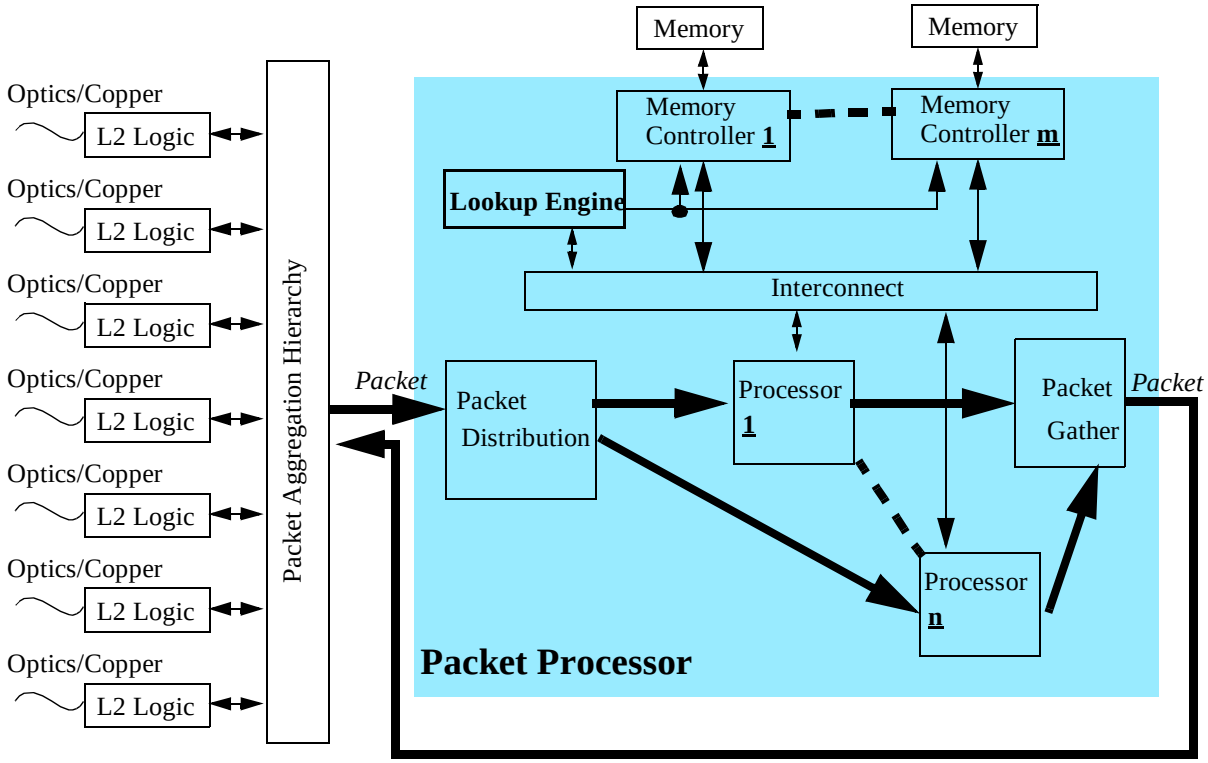


Figure 1: Router Model

based on the fact that there a very few results from a lookup scheme and that a transformation to say an 8-bit result or 256 logical interfaces is useful. With 64k logical interfaces it can be seen this is not a valid assumption.

The function of resolving the output interface and any encapsulation information for a packet (which together we will refer to as a packet's *adjacency* information) can become quite complex as illustrated in Figure 2 below. Figure 2 illustrates the flow of data structures that might be involved with IP forwarding on a router with features like policy based routing (PBR) and load balancing enabled.

The lookup engine leaf can have a statistics pointer (or embedded statistics), flags, and a pointer shown here to a L3 load balancing information data structure. Note that for features that require per prefix or per BGP AS statistics, this results in a requirement for the # of unique leaves for the lookup database to be as large as the # of routes. In parallel with the lookup, there could be a TCAM classification taking place based on multiple fields in the IP/TCP header that could resolve in an over-ride of the lookup engine result (policy over-ride).

There are alternate possible implementations of load balancing, but in this example there is an initial data structure (L3 info) that indicates how many elements are in the load balancing array and a pointer to the location of the array. Then a hash of select packet header fields is used to index into the array (step and repeat for L2 load balancing). The final result after L2/L3 load balancing is the actual adjacency of the packet.

For future routers the number of leaves of the lookup tables may be in the millions (as discussed more below), and the number of L2 adjacencies might be as high as in the several hundreds of thousands. The key point of Figure 2 is to show that the lookup (longest prefix match) function is just one part of the forwarding decision and should share memory bandwidth and memory bits with other features.

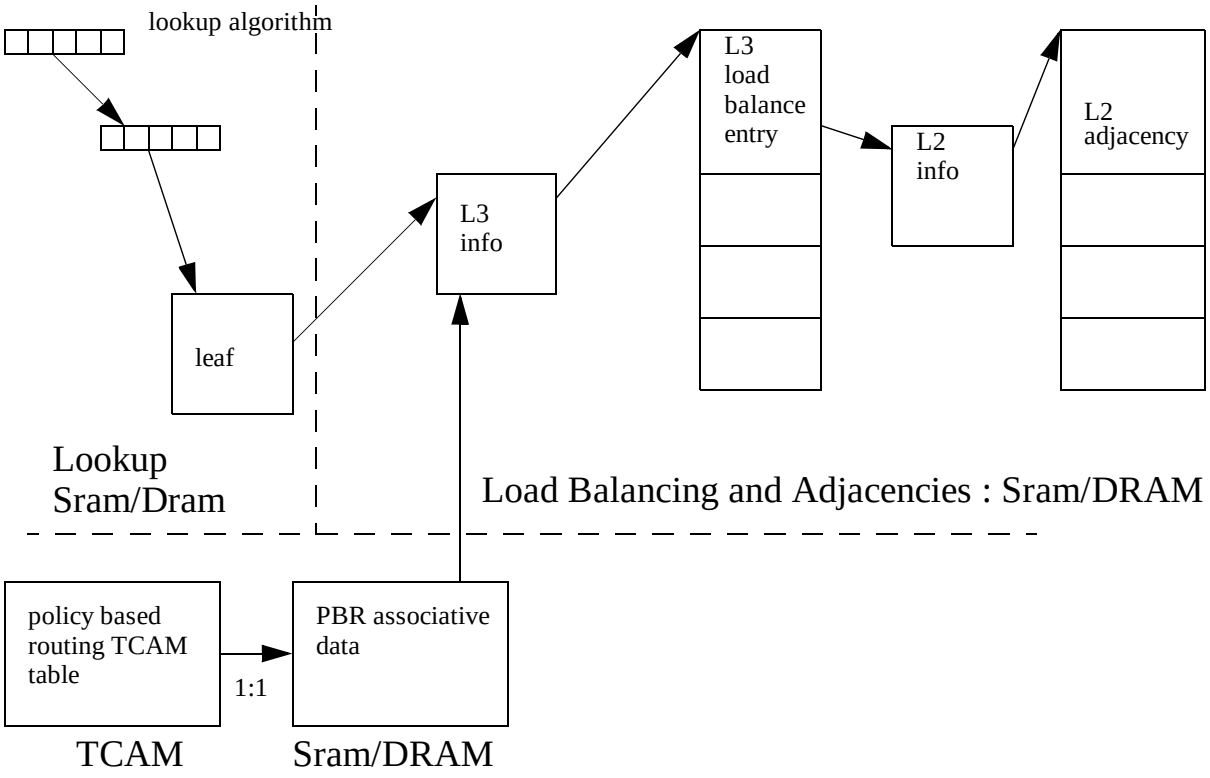


Figure 2: Forwarding Data Structures

2.2 Lookup Table Depth/Performance Requirements

Given the router model, presented in the prior section, it is useful to detail the range of requirements that such a router might have in the future (2005+ time frame) relative to lookup tables. The key parameters are the types of tables stored in the lookup tables, the search lengths per packet for each table type, and the size of each table. For IPv4 unicast there is at minimum a global table, which due to inefficiencies in route distribution may require a million or more prefixes. One of the major reasons for this increase is due to the increasing allocation of long prefixes for multi-homing as described in [29]. A popular feature in addition to normal destination address forwarding is called the reverse path forwarding check (RPF). This feature uses the source address to check if the packet arrived on a "proper" input interface to the router, and can normally use the same tree as the destination with increased memory bandwidth to the forwarding table required.

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 2 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 this example. 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. For the third entry(DVMRP), 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 then for source prefixes. The right most column of Table 2 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 on twice the address size..

Table 2: 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 MPLS applications, given that the goal of label allocation in general is to densely pack the label space, a common approach is to use direct index tables. However there are several reasons why there are benefits to simply considering MPLS label lookup to be part of the longest prefix tree of unicast/multicast IP. Some of the reasons for merging MPLS label lookup into a unified prefix tree with IP:

o VPNs: with VPN deaggregation some labels will point at IP forwarding tables. The concatenation of labels and prefixes for insertion into a longest match table is a natural approach even though it is feasible to put the labels in a direct access memory and point to separate IP tables.

o Non-contiguous label assignment: if there do arise cases in which label assignment is non-contiguous then there could be very large gains to use a longest prefix match table which can effectively condense the storage

o Hierarchical label spaces: if label spaces of a 2nd level label become based on the top label (hierarchical) then the benefits of a tree based structure for storage are obvious.

Table 3 shows a summary of the kinds of performance and database size requirements a general purpose router might have. Due to the number of different tables a 4-bit "table type" is assumed to be prepended to each table type to differentiate them. In this way the lookup engine does not need to know anything about different types of tables, the prefix with pre-pended table type is simply submitted to the lookup engine. Note that a key point here is that if memory were dedicated for all potential applications, the sum total would be extremely large (10M-20M prefixes). However in practice a given router deployment may use a very small number of prefixes (<1M) and the memory can be used for other applications besides lookups.

Feature	**Feature Performance Requirement**	**Database Size**	**Notes**
Global IPv4 Lookups	4+32=36 Bits	1M	
Global IPv4 Lookups + RPF check for IPV4	Dual 4+32=36 Bit searches	1M	RPF is reverse path forwarding check
IPv4 VPN databases per Router interface	4-bit table type + 16-bits+ 32 bits=52 bits	5M-10M	Note that these types of #'s could be due to say 1k VLAN's * 10K prefixes each = 10M
Global IPv4 Multicast	4+64=68 Bits	1M	

Table 3: Example Performance/Density Requirements by Feature

Feature	Feature Performance Requirement	Database Size	Notes
MPLS Label Lookup	4-bit table type + 21-bits label=25 bits	1M	
MPLS Label space per sub-port	4-bit table type + 16 port+21 label=41 bits	several Million ?	Unclear what future service provider requirements here may be
VPN Deaggregation (i.e. terminating packets out of an MPLS tunnel and forwarding the underlying IP packet)	4-bit table type + 21-bits label+ 32- bits IP DA = 57 bit search	1M	
VPN Deaggregation with separate label spaces per sub-port	4+16+21+32=73 bits	??	Unclear if this is really needed, but it shows the kind of convoluted future lookups that might be needed
IPv6 unicast	Typically <64 bits in practice, but market pressure to resolve 128 bits at rate	1M	
IPv6 multicast	4+256 bits=260	512k-1M	Requirements and implications of v6 multicast are still in flux

Table 3: Example Performance/Density Requirements by Feature

2.3 Memory Technology

The constraints/features of memory technology have a major impact on the trade-offs among various IP lookup approaches. The key parameters to consider for differing lookup approaches coupled with the various memory technologies is the cost, raw interface BW, random access rate, and the board space used (i.e. # of components and density of components). Pricing comparisons in this section are based in the 2005+ time frame, and normalized to each other based on quotes given to a major networking systems vendor (and should be taken as approximate numbers).

2.3.1 On-Chip ASIC Memory

Recall that an ASIC is a special IC for (say) packet processing or a special ASIC for IP lookups. Recall also that SRAM stands for Static RAM, which is the fast memory found in say caches and often used in high speed lookup applications. In 90nm (a common process feature size in the 2005+ time frame), 1.5sqmm per 1Mbit of SRAM is a target density in large configurations (out of an automated generation tool). These generated SRAMs will have "modest" access rates (e.g. 300-400 Mhz). Higher performance memories are certainly possible in 90nm (over 1Ghz in 90nm is very feasible) but the density of such memories is normally not as high, and the amount of custom work required is very high (which reduces the number of different memory variations possible).

Devoting half a 12mm per side die for SRAM and assuming 10% top level loss due to floor planning it would seem that 50Mbits could reasonably be placed on a 90nm ASIC (in large instances). The typical uses that engineers think of for on-chip SRAM relative to lookup engines is to consider either specialized lookup ASICs, putting the memory for a lookup engine directly on a packet processor, or putting a portion of a lookup table on-chip (for instance in a tree based lookup algorithm, the top of the tree can be safely put

on-chip without restricting flexibility). Given the requirements for future routers/lookup tables already explored, an obvious drawback to specialized lookup chips is that the use of dedicated chips for 1 function (lookups); also, the number of chips needed has to be dimensioned for the worst case requirements and can't be "statistically multiplexed" with other features.

In general another negative of on-chip SRAM compared to say discrete DRAM is cost. A rough cost comparison (normalized) of the two memory technologies by cost per bit can be as much as ~25x more for on-chip SRAM compared to networking DRAM (category of DRAM technology optimized for networking and discussed in Section 2.3.2) and 50x+ compared to commodity DRAM.

Another memory technology often talked about is embedded DRAM. The density of embedded DRAM in the 90nm generations is expted to be around 3x the density of SRAM (both in large configurations). However, embedded DRAM continues to have drawbacks due to not being as standardized as SRAM and therefore not generally available in all foundry processes. Additionally, embedded DRAM typically has more mask steps than normal processes and therefore has additional cost reducing the cost benefit. Also, due to the special handling of embedded DRAM based ASICs, it is typically qualified later in a process time line than normal ASIC flows, and the time from tape-out to getting chips available is typically longer. There are certainly applications where embedded DRAM is a benefit but it is not quite the panacea that some vendors project it to be.

2.3.2 Discrete Memory Technologies

A common technology considered for use in lookup tables is SRAM, recent example specifications for networking focused sram technology are QDR2/DDR2[1] (the 'Q' stands for quad data rate and indicates that in addition to clocking data on the rising and falling edge there are separate read and write data busses). However, with the introduction of networking DRAM technologies (two examples are RLDRAM[2]/ FCRAM[3]), as well as a gulf in price between the two technologies, discrete SRAM technology for lookup tables are no longer as attractive. The rough normalized per bit price ratio between networking sram and networking dram with same total BW for each can be as high as 36:1 (in dram's favor).

To see how dram can be efficiently used for lookup engines despite back conflicts as well the use of multiple channels, consider a lookup engine with a FIFO of requests on it's input. The FIFO is so that we can allow some limited pipelining of destination address lookups to fully utilize the memory bandwidth. For example, assume that lookups require 4 accesses to a table in memory, 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 channels of dram and each channel has two banks of memories, we can place the first two levels of data structure in the first channel (one level in each bank) and the third and fourth in the second channel (once again one level per bank). If we can simultaneously work on the lookup for two IP addresses (say D1 and D2), then while we are looking up the third and then fourth level for D1 in channel 2, we can lookup the first and then second levels for D2 in channel 1. Within each channel, the accesses ping pong between banks. If it takes 10 nsec per memory access and each bank can be accessed every 20ns, it still takes 4 * 10 = 40 nsec to lookup D1. However, we have a higher throughput of an IP lookup every 20nsec.

Pipelining may appear complex but it is not because the control of the multiple memory banks is done across the ASIC interface pins by HW. The user of the lookup algorithm (e.g., the forwarding code) also has a simple interface: an IP lookup answer can be read in a fixed amount of time (40 nsec in the above example) after the request is made. The results (e.g., adjacency information) are posted to a FIFO buffer which are read by the forwarding engine at the appropriate time.

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. Because interface pins for an ASIC are a precious resource, it is important 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 (banks), and the optimal burst size for a given bus width and random access rate that fully utilizes the memory.

Consider an example of why DRAM can be used in even high end routers for lookup tables. With RLDRAM2 the t_{RC} parameter is 20ns which means at 400 Mhz DDR (and burst length 4) an access can be made to each of 4 logical banks every 5ns. A simple observation is that a database that is mostly read-only could be duplicated 4 times in the DRAM technology, the random access read rate and raw memory bandwidth would be the same for DRAM/SRAM, and the DRAM would still be around 9x cheaper per bit even with 4-way duplication taken into account. An additional issue with SRAM is that the densities in a given process feature size tend to be on the order of 8x between DRAM/SRAM (e.g. 288 Mbit DRAM Vs 36 Mbit SRAM). Strictly SRAM based solutions are not explicitly considered further in this paper.

Table 4 shows a table comparing the more quantifiable differences between the major discrete DRAM memory technology choices. While somewhat arbitrary, a typical bus width for comparison of all technologies used is about 32-36 data pins (depending on the memory component width options). If more memory bandwidth is required beyond that of a single channel, then more then 1 channel can be instantiated. A very important point is that from this example the SRAM with burst length of 2 example(first row) has a transfer size of only 9 bytes which means with ECC protection the usable memory is actually less then 64-bits, which can be problematic for SW memory usage (in other words this is a bad configuration). Similarly DDR2 components are power-of-2 in data bus width and therefore do not naturally allow for ECC support (without addition of addtional components). From an error detection standpoint (which is very important in networking equipment due to high availabilty requirements of service providers today) the only real options in this table are the RLDRAM2 (BL4) and SRAM (BL4).

Memory Technology with Data path Width	Typical Bus width	ASIC pins for typical bus width	Max Data Rate per pin (Mbps)	Logical # of Banks	t_{RC} max (random access rate per bank)	Block Size (bytes) for max access rate	Normalized Cost per bit
QDR3/DDR3 SRAM (BL2)	36	~75	800	1	2.5ns	9 Bytes	72-108
QDR3/DDR3 SRAM (BL4)	36	~75	800	1	5ns	18 Bytes	72-108
DDR2 SDRAM	32	~60	800	4	60ns	32 Bytes	1
RLDRAM2 (BL4)	36	~60	800	8	20ns	18 Bytes	2-3

Table 4: Discrete Memory Technology

What is the future direction of networking DRAM type memories? In general the vendors indicate that the t_{RC} parameter will not be greatly reducing in each future generation.

2.3.3 Ternary Content Address Memories (TCAMs)

Ternary Cams are at this point a well known memory technology for doing general packet classification in which where there is an ordered database of compare and mask patterns. The database is compared to a search key in parallel and the address of the first entry in the database that matches the search key is returned. TCAMs have been used to address the problem of general packet classification for features like Access Control Lists (e.g. matching 5-7 fields in the IP header against service provider programmed databases to block certain types of traffic). In addition, TCAMs have been used in the lookup application. The density/performance of TCAMs has improved quite dramatically in the past several years as more focus and circuit design expertise from the semiconductor field have converged on the technology.

So the obvious question is whether TCAMs as a memory technology could be used for the lookup tables and meet the router requirements as far as the router model, as well as the requirements for lookup tables in terms of performance/density. Today in.13um feature technology there are 18Mbit (that is 256k by

72-bit entries of data and mask) components but the die sizes are on the large size. Going forward the best case will be roughly doubling the density of TCAMs with each step down in feature size (e.g. from.13um to 90nm), but even that may be difficult to maintain at reasonable cost (and in fact the author is not aware of any plans by TCAM vendors to do 36Mbit or larger densities in 90nm). The performance of TCAMs is more then adequate for the lookup application, but the power and board space required is definitely an area of concern.

To consider the issues of using TCAM technology for lookups consider a scenario where is a desire to support say 10 Million entries of 52-bits (VPN deaggregation width from Table 3). WIth 18Mbit TCAMs this would require 20 TCAM components. The cost could easily exceed $4000, the board space becomes very large, and power could easily be >200W if techniques are not employed to limit the components and banks of CAMs that are searched (but this increases complexity of interfacing which was one of the benefits of TCAMs, see below). The benefits of using TCAMs are as follows:

- Easy to interface to from packet processor perspective
- Very high performance (more then adequate for mid to high end routers)
- Can be shared with classification functions (.e.g ACL, QoS classification, etc,.)

The drawbacks for using TCAMs for the lookup application (even if a router database is not as high as 10 Million) are:

- Granular sizing of entries. Even if every 288-bit entry can be configured as either a single 288-bit entry, 2 by 144-bit entries, 4 by 72-bit entries, or 8 by 36-bit entries, if a given feature requirements go slightly over a boundary, the table depth halves.
- While there can be some sharing between the TCAMs for classification features as well as lookups, the TCAM memory is not general usable (and too expensive to use) for as wide a variety of other databases/features as DRAM.
- Cost is extremely prohibitive given cost of goods (COGs) targets of routers today
- Board space can be much higher then DRAM
- Power can be extremely high

2.4 Lookup Engine Requirements

Based on the router model presented as well as the summary of requirements for a lookup engine are as follows:

1. Optimizing for cost of the final forwarding engine subsystem is very important.

2. Flexible sharing of memory across lookup table types (VPNs, MPLS, unicast/multicast, IP v4/v6). This requires efficient support for any size lookup string.

3. Flexible trade-off of density versus performance within the lookup engine at the same time (e.g., different tables in the lookup engine may have different performance/density targets; additionally as table sizes grow to maximum capacity it may be desirable to have SW updates to the table gracefully trade-off performance for more density.)

4. Flexible sharing of memory between lookup tables and other applications (statistics, deep packet classification, flow caches, etc,.)

5. Incremental and "in-place" updates - Update rate requirements are high, and memory usage is too high to consider shadowing. Ideally the amount of memory/datapath bandwidth required for updates should be deterministic and modest.

6. To facilitate control plane management, ideally the lookup algorithm should be well suited for data plane as well as control plane implementation with a way to cohesively connect them

7. While routers can be designed and sold based on "typical" database sizing, there is a definite requirement to have a reasonable worst case (including pathological cases).

8. Performance needs to be deterministic and designed to worst case for benchmarking purposes

3 Related Work

A lookup scheme based on caching is the Stony Brook scheme [21]. 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.

With new CAM technologies, CAMs in the future may be a fine solution for enterprise and access routers that have less than 32000 prefixes[16]. Thus algorithmic solutions like ours can support much larger prefixes for backbone routers; the flip side is that they need to exhibit some of the determinism in update and lookup times that CAMs provide. The multi-column lookup scheme [12] uses a B-tree like approach which has reasonably fast lookup times and reasonable storage but poor update times. The binary search on hash tables approach [22] scales very well to IPv6 128 bit addresses but has poor update times and possible non-determinism because of hashed lookups.

The Stanford scheme [7] uses a multibit trie with an initial stride of 24 bits, and then two levels of -4 bit tries. Once again updates can be slow in the worst case, despite a set of clever optimizations. Both the LC-Trie[17] and Expanded Trie schemes [19] allow multibit tries that can have variable strides. In other words, the pointer to a trie code can encode the number of bits sampled at the trie node.

The Expanded trie [19] allows memory to be traded off for lookup speed but requires a fairly high amount of memory (factor of two worse than those of our scheme and the Lulea scheme) despite the use of dynamic programming optimizations. The optimizations can also slow down worst case update times; without considering the time for the optimization program to run, the update times are around 5 msec. The LC-trie fundamentally appears to have slow update times though none are reported.

In the last five years since our scheme was first described in [24], some new schemes have been described. In [23], a new compression scheme is described that is based on lookup data characteristics but has no worst-case memory guarantees (as we argue is needed). In [25], the case is made for new memory allocation schemes to support compressed IP lookup schemes that use variable length nodes (as ours do). We use entirely different memory allocators from those in [24] but ours work equally well; besides, memory allocation is only a small part of this paper's contributions. In [26], a hardware implementation of our original idea [24] is described; the current paper is the journal version of our original idea [24] for which implementation results are described in [26]. Finally, [27] presents some ideas for reducing power consumption in TCAMs but these ideas do not appear to be reflected in practice.

The Lulea scheme is closest in spirit to ours in that both use multibit tries and bitmaps to compress wasted storage in trie nodes. There are fundamental differences, however, which allow our scheme more tunability and faster update times. To show these differences in detail we spend the rest of the section describing three detailed examples of the unibit trie algorithm, the expanded trie algorithm, and the Lulea scheme on a sample prefix database. Since we will use the same sample database to describe our scheme, this should help make the differences clearer.

3.1 Unibit Tries

For updates the one bit at a time trie (unibit trie) is an excellent data structure. The unibit trie for the

sample database is shown in Figure 3. With the use of a technique to compress out all one way branches, both the update times and storage needs can be excellent. Its only disadvantage is its speed because it needs 32 random READS in the worst case. The resulting unibit trie structure is somewhat different from Patricia tries which uses a skip count compression method that requires backtracking.

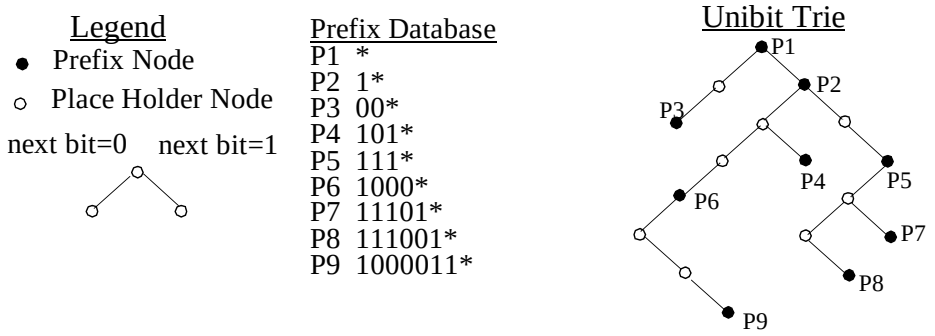


Figure 3: Sample Database with Unibit Trie Representation

3.2 Expanded Tries

In Expanded tries [19], the main idea is to use tries that go several bits at a time (for speed) rather than just 1 bit at a time as in the unibit trie. Suppose we want to do the database in Figure 3 three bits at a time. We will call the number of bits we traverse at each level of the search the "stride length".

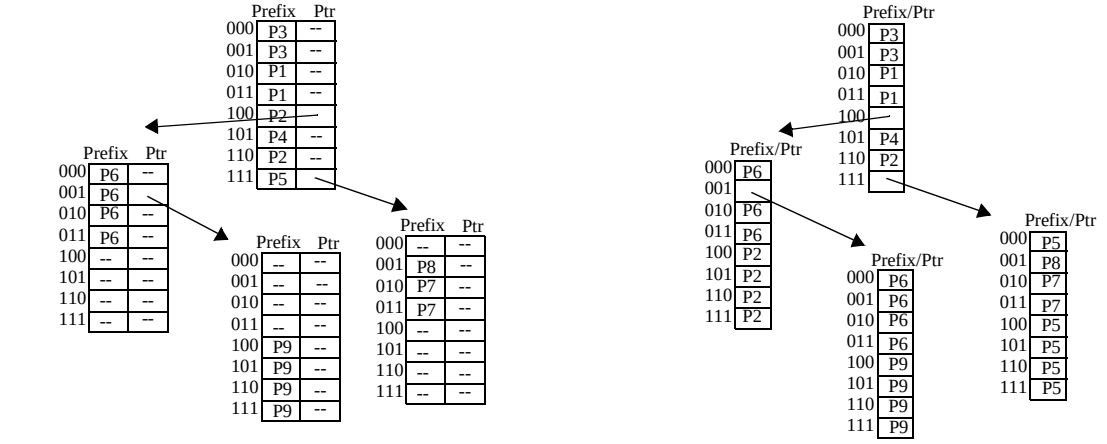
A problem arises with prefixes like P2 = 1* that do fit evenly in multiples of 3 bits. The main trick is to expand a prefix like 1* into all possible 3 bit extensions (100, 101, 110, and 111) and represent P1 by four prefixes. However, since there are already prefixes like P4 = 101 and P5 = 111, clearly 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 is does 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.

Part A of Figure 4 shows the expanded trie for the same database as Figure 3 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 is a record that 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.)

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 with expansions of P1. In general, using larger strides requires the use of larger trie nodes with more wasted space. Reference [19] does show how to use variable stride tries (where the strides are picked to reduce memory) while keeping good search times. However, the best case storage requirements for the Mae East database (after the most sophisticated optimizations) are around 500 Kbytes.

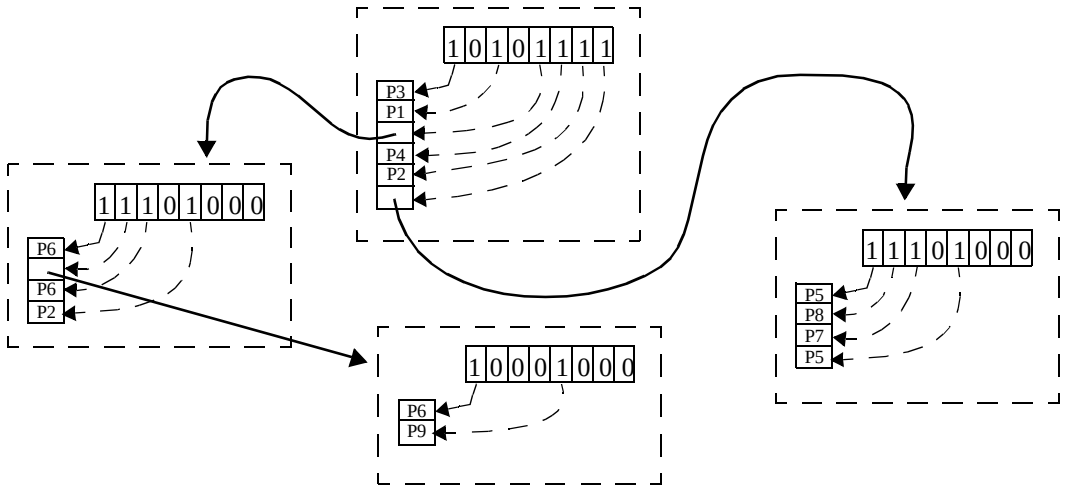
3.3 Lulea

The Lulea scheme [6] does much better then expanded tries in terms of storage, using only 200 Kbytes of memory to store the current Mae East database. One way to describe the scheme is to first 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 Part A of Figure 4 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 [19]. This is shown in Part B of Figure 4. Notice that the prefix P2 associated with



A) Controlled Prefix Expansion w/out Leaf Pushing

B) Controlled Prefix Expansion with Leaf Pushing



C) Lulea

Figure 4: Related Work

the 100 root entry in Part A has been pushed down to several leaves in the node containing P6.

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 bitmap with 0's corresponding to the empty positions is associated with each compressed node.

For example consider the root node in Figure 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 bitmap 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 Part C of the figure. This transformation is also known as run length encoding.

The search of a trie node now consists of using a number 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 bitmap. 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 eight 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 Part A of Figure 4, 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 bitmaps, one for internal entries and one for external entries. This is indeed one of the ideas behind the tree bitmap scheme we now describe.

4 Tree Bitmap Algorithm

In this section, we describe the core idea behind the algorithm that we call Tree Bitmap. Tree Bitmap is a multibit trie algorithm that allows fast searches 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 of stride of no more than 8, this requires only 256 + C node copy operations in the worst case where C is small). Its also has memory storage requirements comparable (and sometimes slightly better) when compared to the Lulea scheme.

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

* A multibit node (representing multiple levels of unibit nodes) has two functions: to point at children multibit nodes, and to produce the next hop pointer for searches in which the longest matching prefix exists within the multibit 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 very large (e.g, 32 bytes for SDRAM, see Section 2.2). This is because while the random access rate for core DRAMs have improved very slowly, high speed synchronous interfaces have evolved to make the most of each random access. (See Section 2.2). 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 several in Lulea).

* To ensure that a single node is always retrieved 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 the 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 multibit 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 larger and variable sized allocation chunks. Using this idea, the same 3-bit stride trie of Figure 3 is redrawn as Figure 5.

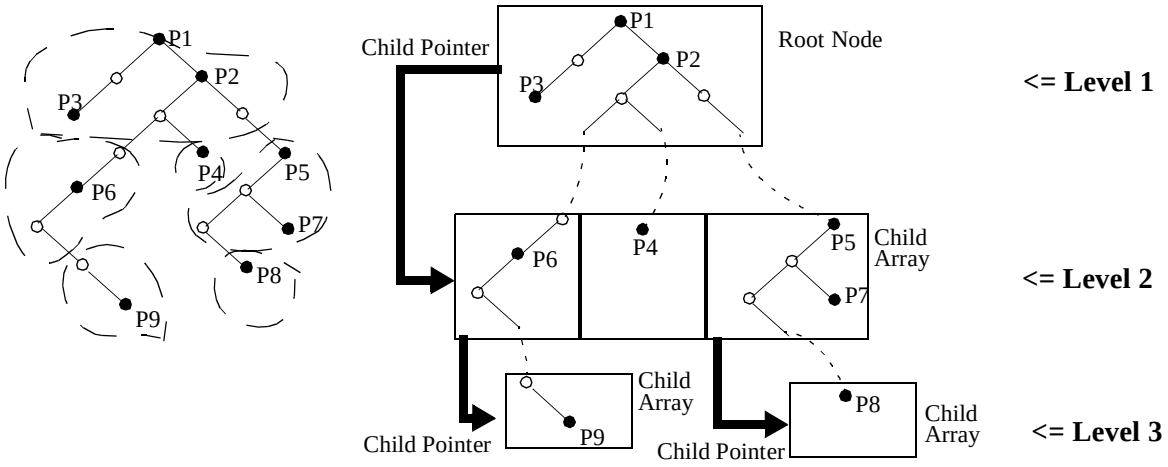


Figure 5: Sample Database with Tree Bitmap

The second idea is that there are two bitmaps per trie node, one for all the internally stored prefixes and one for the external pointers. See Figure 6 for an example of the internal and external bitmaps for the root node. The use of two bitmaps allows us to avoid leaf pushing. The internal bitmap is very different from the Lulea encoding, and has a 1 bit set for every prefixes 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 bitmap.

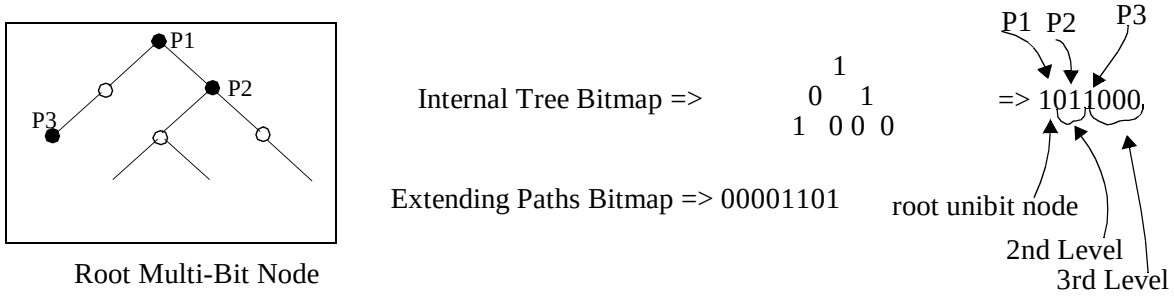


Figure 6: Multibit Node Compression with Tree Bitmap

For the root trie node of Figure 5, we see that we have three internally stored prefixes: P1 = *, P2 = 1*, and P3 = 00 *. Suppose our internal bitmap 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 bitmap becomes 1-01-1000. The first 1 corresponds to P1, the second to P2, the third to P3. This is shown in

Figure 6.

The external bitmap contains a bit for all possible 2^r child pointers. Thus in Figure 5, we have 8 possible leaves of the 3 bit subtrie. Only the fifth, sixth, and eight leaves have pointers to children. Thus the extending paths (or external) bitmap shown in Figure 6 is 00011001 As in the case of Lulea, we need to handle the case where the original pointer position contains a pointer and a stored prefix (e.g., location 111 which corresponds to P5 and also needs a pointer to prefixes like P7 and P8. The trick we use is to push all length 3 prefixes to be stored along with the zero length prefixes in the next node down. For example, in Figure 5, we push P5 to be in the right most trie node in Level 2. In the case of P4 (which was stored in the root subtrie in the expanded trie of Figure 4a) we actually have to create a trie node just to store this single zero length prefix.

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 a trie node is of fixed size and only contains an external pointer bitmap, an internal next hop info bitmap, 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...

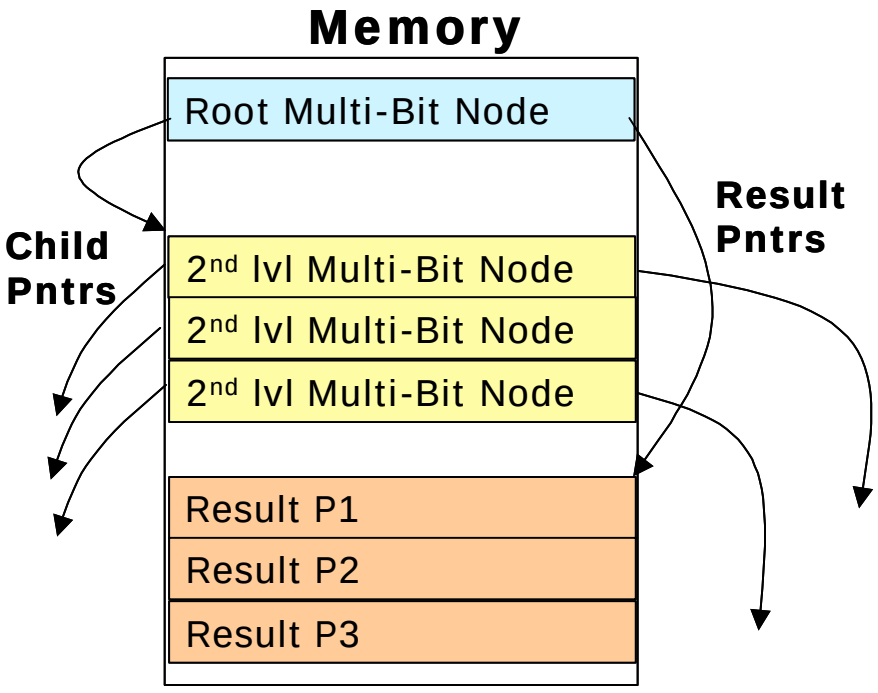


Figure 7: Memory Map of Example Tree

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 external 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 H) and the size of each trie node (say S), we can easily compute the pointer to the child node as H + (I * S).

Before we move on to the child, we also check the internal bitmap to see if there is a stored prefix corresponding to position P. This requires a completely different calculation from the Lulea style bit calculation. To do so, we can imagine that we successively remove bits of P starting from the right and index into the corresponding position of the internal bitmap looking for the first 1 encountered. For example, suppose P is 101 and we are using a 3 bit stride at the root node bitmap of Figure 6. We first remove the rightmost bit, which results in the prefix 10*. Since 10* corresponds to the sixth bit position in the internal bitmap, we check if there is a 1 in that position (there is not in Figure 6). If not, we need to remove the rightmost two bits (resulting in the prefix 1*). Since 1* corresponds to the third position in the internal bitmap, we check for a 1 there. In the example of Figure 6, there is a 1 in this position due to P1, so our search ends. (If we did not find a 1, however, we simply remove the first 3 bits and search for the entry corresponding to * in the first entry of the internal bitmap which would match P1.).

This search algorithm appears to require a number of iterations proportional to the logarithm of the internal bitmap length. However, in hardware for bitmaps 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). In software this can be implemented using table lookup. (A further trick using a stored bit in the children further avoids doing this processing more than once for software; see software reference implementation in Section 6.) Thus while this processing appears more complex than the Lulea bitmap processing it is actually not an issue in practice.

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 (more combinational logic!) to indicate its position in the result array. Accessing the result array 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 external 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.

Figure 8 gives pseudocode for full tree bitmap search. It assumes a function treeFunction that can find the position of the longest matching prefix, if any, within a given node by consulting the internal bitmap (see description above). "LongestMatch" keeps track of a pointer to the longest match seen so far. The loop terminates when there is no child pointer (i.e., no bit set in External 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. We assume that the address being searched is already broken into strides and stride[i] contains the bits corresponding to the i'th stride.

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 what follows, we describe some optimizations that reduce the trie node size even further.

```
1.    node:= root;  (* node is the current trie node being examined; so we start with root as the first trie node *)
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.internalBitmap,stride[i]) is not equal to null) then
5                 (* there is a longest matching prefix, update pointer *)
6         LongestMatch:= node.ResultsPointer + CountOnes(node.internalBitmap,
7               treeFunction(node.internalBitmap, stride[i]));
8       if (externalBitmap[stride[i]] = 0) then    (* no extending path through this trie node for this search *)
9           NextHop:= Result[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.childPointer + CountOnes(node.externalBitmap, stride[i]);
13          i=i+1; (* move on to next stride *)
14    end do;
```

Figure 8: Tree Bitmap Search Algorithm for Destination Address whose bits are in an array called stride

5 Optimizations

This section presentations optimizations for increased performance, improved memory density, or for controlling the size of nodes.

For a stride of 8, the data structure for the core algorithm would require 255 bits for the Internal Bitmap, 256 bits for the External Bitmap, 20 bits for a pointer to children, 20 bits for a result pointer which is 551. Thus the next larger power of 2 node size is 1024 bits or 128-bytes. From our table in Section 2.3. we can see that for some technologies this would require several memory interfaces and so drive up the cost and pin count. Thus we seek optimizations that can reduce the size of nodes.

5.1 Initial Array Optimization

Almost every IP lookup algorithm can be speeded up by using an initial array (e.g., [6],[14],[19],[20]). Array sizes of 13 bits or higher could, however, 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. Then 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 8-bytes in size which results in a dedicated initial array of 2k bytes. This is a reasonable price in bytes to pay for the savings in memory accesses. In a hardware implementation, this initial array can be placed in on chip memory.

5.2 End Node Optimization

We have already seen an irritating feature of the basic algorithm in Figure 5. Prefixes like P4 will require a separate trie node to be created (with bitmaps that are almost completely unused). Let us call such nodes as "null nodes". While this cannot be avoided in general, it can be mitigated by picking strides carefully. In a special case, we can also avoid this waste completely. Suppose we have a trie node that only has external pointers that point to "null nodes". Consider Figure 9 as an example that has only one child which is a "null node". In that case, we can simply make a special type of trie node called an endnode (the type of node can be encoded with a single extra bit per node) in which the external bitmap is eliminated and substituted with an internal bitmap of twice the length. The endnode now has room in its internal bitmap to indicate prefixes that were previously stored in "null nodes". The "null nodes" can then be eliminated and we store the prefixes corresponding to the null node in the endnode itself. Thus in Figure 9, P8 is moved up to be stored in the upper trie node which has been modified to be an endnode with a larger bitmap. In addi-

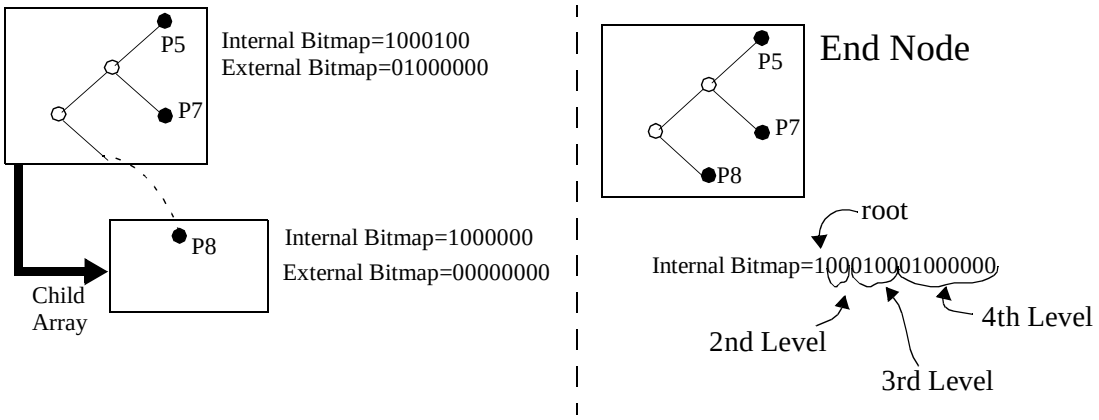


Figure 9: End Node Optimization

tion to modest memory savings (see [24] for more details of savings in common databases) another advantage of end node optimization is that for a given number of memory accesses an extra bit is added to the length of the search since the bottom of the tree will all be end nodes (since there is no extending paths).

5.3 Split Tree Bitmaps

Keeping the stride constant, one method of reducing the size of each random access is to split the internal and external bitmaps into separate nodes (separate memory accesses). This is done by placing only the external bitmap in each "Trie" node and the internal bitmap in a new “Internal Node”. If there is no separation of levels of the tree into separate banks/channels of memory, the children and the internal nodes from the same parent trie node can be placed contiguously in memory. If memory is segmented into banks/channels, it is a bad design to have the internal nodes scattered one option is to have the trie node point at the internal node, and the internal node point at the results array. Or the trie node can have three pointers to the child array, the internal node, and to the results array. Figure 10 shows both options for pointing to the results array and implementing split tree bitmap..

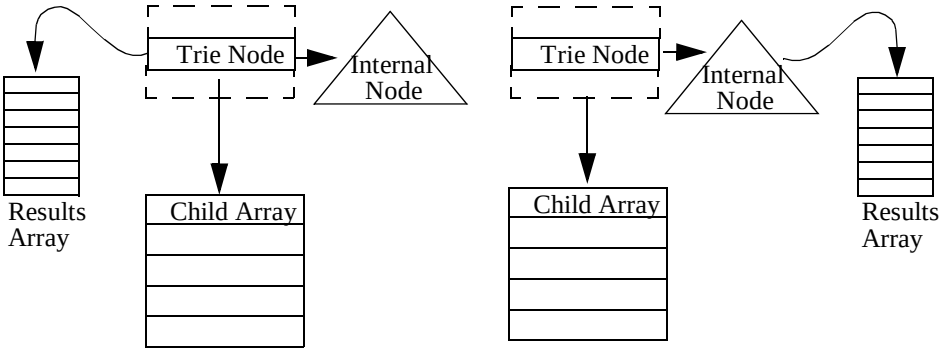


Figure 10: Split Tree Bitmap Optimization

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. If there was a prefix in the path, the lookup engine records the location of the internal node (calculated from the data structure of the last node) as containing the longest matching prefix thus far. Then when the search terminates, we first have to access the corresponding internal node and then the results node corresponding to the internal node. Notice that the core algorithm accesses the next hop information lazily; the split tree algorithm accesses even the internal bitmap lazily. What makes

this work is that any time a prefix P is stored in a node X, all children of X that match P can store a bit saying that the parent has a stored prefix. The software reference implementation uses this optimization to save internal bitmap processing; the hardware implementations use it only to reduce the access width size (because bitmap processing is not an issue in hardware).

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

5.4 Segmented Bitmaps

After splitting the internal and external bitmaps into separate nodes, the size of the nodes may still be too large for the optimal burst size (see 2.3). The next step for reducing the node sizes is to segment the multi-bit trie represented by a multi-bit node. The goal behind segmented bitmaps is to maintain the desired stride, keep the storage space per node constant, but reduce the fetch size per node. The simplest case of segmented bitmaps is shown in Figure 11 with a total initial stride of 3. The subtrie corresponding to the trie node is split into its 2 child subtries, and the initial root node duplicated in both child subtries. Notice that the bitmaps for each child subtrie is half the length (with one more bit for the duplicated root). Each child subtrie is also given a separate child pointer as well as a bitmap and is stored. as a separate "segment". Thus each trie node contains two contiguously stored segments.

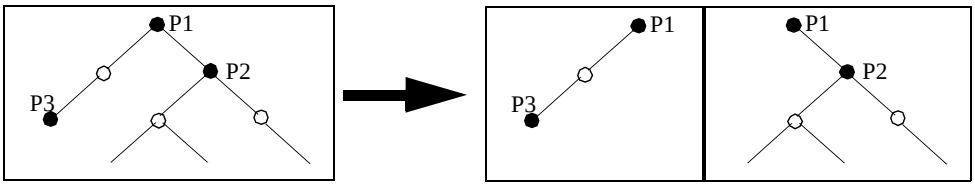


Figure 11: Segmented Tree Bitmap

Because each segment of a trie node has its pointers, the children and result pointers of other segmented nodes are independent. While the segments are stored contiguously, we can use the high order bits of the bits that would normally have been used to access the trie node to access only the required segment. Thus we need to roughly access only half the bitmap size. For example, using 8 bit strides, this could reduce the bitmap accessed from 256 bits to 128 bits.

When this simple approach is extended to multiple segmentations, the complexity introduced by the segmented bitmap optimization is that if k is the initial stride, and 2^j is the number of final segmented nodes, the internal prefixes for the top k-j rows of prefixes are shared across multiple segmented nodes. The simplest answer is to simply do controlled prefix expansion and push down to each segmented node the longest matching result from the top k-j rows.

5.5 CAM Nodes

Empirical results show that a significant number of multi-bit nodes have only a few internal prefixes. A common occurrence is a multi-bit trie node with no extending bits set, and only a single bit set in the internal bitmap. In these cases the space normally occupied by internal bitmaps and a pointer to a results arrays can be replaced by a simple CAM type entry that has match bits, match length, and next hop pointers. The gain is that the next hop pointers are in the CAM nodes and not in a separate results array taking

up additional space. Even single entry CAM nodes (as opposed to packing multiple cam entries in a single node) was found to typically result in over half of the next hop pointers moving from results arrays, to inside CAM end nodes. There are quickly diminishing returns as more then 1 cam entry is packed per node.

6 Software Reference Design

Although, the main benefit of our algorithm is its considerable flexibility, performance, and scalability in terms of hardware implementation, we note here that our software reference implementation of the Tree Bitmap algorithm actually works quite well. Earlier tests with databases around 5 years ago [23], showed that Tree BitMap outperformed other state-of-the-art IP address lookup algorithms in some dimension of interest: these include lookup speed, or insertion/deletion speed, and total size.

Given that hardware is the main focus of this paper, we will limit ourselves in this section to show that the software implementation of the algorithm is very simple. Efficient software algorithms for lookups can be important both for low end router platforms, as well as for control plane software that has to maintain large databases with associative data independent of a packet processor such as described in Section 2.1. By unifying on a Tree Bitmap in both control plane and dataplane the ability to do synchronized incremental updates can be very important in achieving high update rates. High update rates for both control and data plane forwarding tables is useful in relation to fast restart/boot-up scenarios, certain IP/MPLS tunnel movement scenarios, as well as for fast routing table convergence.

For software, we chose a stride of 4; by keeping this small, we are able to fit the entire node into 8 bytes, which is the size of an internal cache line on many processors. Also, with a small stride, the bitmaps are small: the internal tree bitmap is 15 bits, and the external paths bitmap is 16 bits. With such small bitmaps, we can use a byte lookup table to efficiently compute the number of set bits in a bitmap. We also eliminate the pointer to a results array by allocating children nodes as well as the results in a contiguous buffer, so that only a single pointer can be used to access both arrays with children to the right and results to the left (so the number of entries for either array can still be variable), Thus either array can be accessed directly by adding or subtracting relative to the single point, and without having to know the total number of entries in the other array.

We also add a "parent_has_match" bit to each node. This bit allows us to avoid searching for a matching prefix in all but two of the internal bitmaps of encountered node. During trie traversal we use this bit to determine the last node that had such a valid prefix and lazily search the prefix bitmap of that node after the search terminates. Our reference implementation uses an initial 8K entry lookup table and a 4 bit stride to create a 13,4,4,4,4,3 trie. Since there is a large concentration of prefixes at lengths 16 and 24, we ensure that for the vast majority of prefixes the end bits line up at the bottom of a trie node, with less wasted space.

This explains the 13,4,4,... ordering: 13+4 is 17, which means the 16 bit prefixes will line up on the lowest level of the tree_bitmap in the nodes retrieved from the initial table lookup. Similarly, 13+4+4+4 is 25, so once again the pole at 24 will line up with the lowest level of the tree_bitmap in the third level nodes of the trie.

More details of early experimental comparisons with other software schemes can be found in [23].

7 Hardware Reference Design

In this section we investigate the design of a hardware lookup engine based on the Tree Bitmap algorithm at a level below what was presented in the router model of Section 2.1. We first present a simple modification that allows the storage needs of our trie scheme to be deterministically bounded.

7.1 Skip Nodes and Worst case Table Size Bounds

A desire by many commercial router customers is to get some sort of guarantee about the number of prefixes that can be supported with a given amount of memory. This is one area that hardware CAMs have an advantage over algorithmic solutions to IP lookups [15]. To reduce the upper bounds on maximum table size, we notice for the worst case bound to occur, each path (below the point were we assume every node exists) will be a long non-branching, prefixless path that terminates at the maximum address length.

A simple optimization to counter this worst case analysis is the introduction of a new type of node, the skip node. There are two types of skip nodes possible: path skip nodes (that simply compress long non-branching, prefixless paths but that point at a child that branches or that contains a prefix and continues), and end skip nodes (that compress long non-branching, prefixless paths that terminate with a single prefix). Path skip nodes contain a skip length to enable the skipping of multiple levels, match bits which indicate if the skip is appropriate or if the search should terminate with the longest match thus far, and a pointer to a single child node.

Using both path skip nodes, and end skip nodes, the result is that in the worst case the multi-bit tree has a similar property to binary trees. Every node will have either have an internal prefix (with zero or more extending paths), or at least two children (with possibly no internal prefixes). With these properties, we can state that the worst case bound is that the number of nodes will always be at most 2n (not counting the results arrays). So if node sizes are 8-bytes, the worst case bounds table size is 16-bytes per prefix.

Skip nodes do not result in much savings in practice for the Tree Bitmap scheme with an initial array for any of the current databases. The reason is that there are relatively few long non-branching, prefixless paths. We expect that IPv6 databases as well as VPN deaggregation will benefit significantly from skip nodes in practice.

7.2 Reference HW Lookup Engine

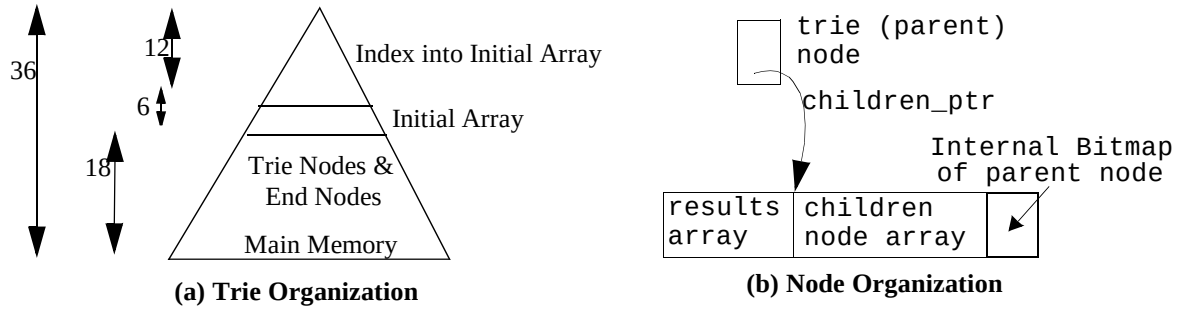
For our main hardware reference design, we tailor the Tree Bitmap algorithm towards an implementation set within Section 2.1 that can use up to four RLDRAM2 channels (so from Figure 1, m=3). This design is targeted to support up to 25 million worst case lookups per second. Therefore the physical parameters are 4 channels of 36 data bits each (2-bits of each channel used for ECC error detection/correction, so each channel is effectively 32-bits), 800Mb/s per pin, and on-chip memory available for the root of the tree. A burst length of 4 is used so each node is 16 bytes. For the purposes of this reference engine, each channel of memory has 4 copies of it's data stored in that channel in different banks so that random access time for any 16-byte node in a channel is 5ns (or 200M access per second). To avoid replication within a channel, care can be taken to map levels of the tree to different banks. Note that it is not intended there is any replication of state between channels.

With 4 channels the maximum access rate is 800M 16-byte nodes per second (note that there is refresh of the DRAM required and update memory bandwidth for maintenance of the tables but these are a relatively small (<2%) percentage of total bandwidth and ignored here). The algorithm parameters are: 6-bit stride, a 12-bit initial stride (in on-chip sram), CAM optimization for end nodes and internal nodes, skip nodes, and the split tree bitmap optimization.

The trie data structure used is shown below, but the data structures of the other node types are not presented due to space constraints. It is important to note that all addressing is done relative to the node size of 128-bits. Since the child address is relative to 16 byte nodes, this data structure can address 4GB of physical memory across the 4 channels.

```
1.  struct trie_node {
2.    extending Bitmap      : 64;   /* Simply the 16-bit bitmap for the extending paths */
3.    child_address         : 28;   /* Pointer to children node array */
4.    parent_has_match      : 1;    /* a 1 indicates the parent has a match
5.    type                  : 1;    /* type=1 means trie node, 0 indicates an'other' type */
6.    is_internal_node      : 1;    /* A 1 indicates there are internal prefixes */
7.    reserved              : 33;   /* in a commercial implementation there are other features these bits would be used for */
8.  }
```

Figure 12a shows the basic 36-bit trie for IPv4 lookups (4-bit table type + 32-bits v4 address) broken down for this reference design. The first 12 address bits are simply used to index into an initial array. The initial array is a separate dedicated memory that contains 4k entries. Each initial array entry contains a standard 128-bit node which can either be an empty node, trie node, or an end node. Figure 12b shows the parent-child relationship with the internal bitmap for each trie node stored at the end of the children node array.



(a) Trie Organization **(b) Node Organization**

Figure 12: Algorithm Organization and Implementation

Since the parent trie node contains a bitmap of the children, and an indication of whether there exists an internal node (is_internal_node), the address of the internal node can be calculated. The result of the search (stored in result array) is assumed to be a 16 Byte leaf with associative data. For a given prefix width, the worst case performance is for the scenario where the entire length of the search key is consumed (so the number of access is $\left\lceil \frac{width - 12}{6} \right\rceil$ and the last access is an end-node which does not have a prefix for the search, so a fetch of an internal bitmap is needed (based on parent_has_match flag) and finally a fetch of the result.

Therefore, with j accesses to external memory the worst-case search length is 12+6*(*j*-2). Table 5 below shows for 2 packet performance rates (12 and 25 MPPS), the search key lengths that can be handled by devoting 25%, 50%, and 100% of the DRAM bandwidth available to the lookup application. There are separate rows for 1 and 2 lookups per packet (2 lookups would be used to support the reverse path forwarding check). An important note is that with 200 MAccesses per second (25% of available DRAM bandwidth out of the 4 channels from our example), the VPN deaggregation case of Table 3 can be handled (without an RPF check).

Table 5: Lookup Engine Search Lengths (worst case)

MPPS	# Lookups Per Packet	200M Access/s	400M Access/Sec	800M Access/Sec
12	1	105	207	405
12	2	57	105	207
25	1	57	105	201
25	2	33	57	105

Table 6 shows the empirical results for the mapping of real databases to the reference design. This table shows a break down for each database of how many instances of each type of node. Note these numbers do not include a leaf (result) with associative data (which for the Mae-East/West databases is actually more bytes per prefix of storage then for the search structures). Additionally, Table 6 has a row for the density of one million prefixes of random address of length 36-bits (4-bits of fixed table type).

Database	No. of prefixes in database	Total # Nodes	Total Size (Kbytes)	Bytes per Prefix
Mae-East	40,902	28216	452	11
Mae-West	19,229	15429	247	13
Random Generated	1,000,000	1,450,000	23,200	23

Table 6: Reference Design Empirical Size Results

With skip nodes (path compression) additional experiments have shown modest degradation of density as the search key gets longer for IPv6 or VPNs. Based on the empirical data above, a very rough "marketing" summary of the density of our reference engine might be to average the "Bytes per prefix" of the 3 experiments above (so about 16B per prefix of search structures), add another 16B for the leafs of each prefix, and multiply by 4 (due to the brute force table replication used in this reference design).

The result is a total of 128 Bytes of physical DRAM used per prefix. So to support a deployment requiring 10M prefixes (which would seemingly not be the common case) would require 1.6GB of DRAM; when that database size is not deployed the memory could be used for other features. While this amount of DRAM is possible it is still more then desired. Without replicating the table 4 times per channel (which requires more sophisticated memory management and layout), the requirement is 400MB. For 400MB, using 576Mbit DRAM components, a total of at least 7 DRAM components are required.

The design and design verification of a lookup engine similar to the hardware reference engine presented here required about 2 man years of effort (one year design effort and one year design verification effort). The area was only a few percent of the total die size of a proprietary packet processor design.

8 Conclusions

We have described a new family of compressed trie data structures that have small memory requirements, fast lookup and update times, and is tunable over a wide range of architectures. While the earlier Lulea algorithm [5] is superficially similar to ours in that it uses multibit tries and bitmaps, our scheme is very different from Lulea. We use two bitmaps per trie node as opposed to one, we do not use leaf pushing which allows us to bound update times, we use a completely different encoding scheme for internal bitmaps which requires a more involved search algorithm than finding the first bit set, and we push prefixes that are multiples of stride lengths to be associated with the next node in the path, even if it means creating a new node.

To do the complete processing in one memory access per trie node (as opposed to the three memory accesses required by Lulea), we keep the trie node sizes small by separating out the next hop information into a separate results node that is accessed lazily at the end of the search. We also described several further optimizations including split tree bitmaps and segmented bitmaps to further reduce trie node size. None of these techniques appear in [5] or any earlier schemes.

We have also described a model for future router and memory architectures that together with the family of Tree Bitmap variants allows us to pick the required algorithm parameters for a given architecture. This model provides us with knowledge of the optimal burst size which in turn determines the stride size. Recall that for a particular set of optimizations, the stride size defines the bitmap size and hence the burst size. The model also directly indicates the degree of pipelining required for maximum memory bandwidth utilization. Using these trie algorithms and memory models, we described competitive software and hardware implementations. By using skip nodes, we also described how we could give deterministic guarantees not just for update and lookup times but also for worst case number of prefixes supported. Together, our algorithms provide a complete suite of solutions for the IP lookup problem across the spectrum of router requirements (from low to high end).

From the time it was first described in thesis form several years ago in [23] (in which considerable emphasis was placed on SRAM implementations both on-chip and off-chip), the original idea appears to have adapted gracefully to changes in memory technology (with the current emphasis on DRAM for cost) and the considerable increases in table size demands which we could not have foreseen (e.g., due to the popularity of VPNs). The fundamental algorithm ideas have, however, remained essentially the same. In this context, it appears that our original design goals of tunability and flexibility remain essential goals for any IP lookup algorithm when facing the future.

Finally, while CAM technology will continue to improve as continued attention is devoted to decreasing cost and power consumption, reports of the death of algorithmic solutions are greatly exaggerated, especially for cost-sensitive deployments with large table sizes.

9 References

[1] QDR SRAM consortia, http://www.qdrsram.com/

[2] RLDRAM (Reduced Latency DRAM) consortia, www.rldram.com/

[3] Networking FCRAM (Fast Cycle Ram), http://www.toshiba.com/taec/main/promo/fcram/

[4] Artisan Components, Inc., http://www.artisan.com/

[5] U. Black, S. Waters, SONET & T1: Architectures for Digital Transport Networks, Prentice Hall, 1997

[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] P. Gupta, S. Lin, N. McKeown, "Routing Lookups in Hardware at Memory Access Speeds," Proc. IEEE Infocom `98.

[8] IEEE Draft P802.3z/D3.0 "Media Access Control (MAC) Paramaters, Physical Layer, Repeater and Management Paramters for 1000Mb/s operation", June 1997.

[9] S. Keshav, R. Sharma, "Issues and Trends in Router Design", in IEEE Communications Magazine , May 1998.

[10] V.P. Kumar, T.V. Lakshman, D. Stiliadis, "Beyond Best-Effort: Gigabit Routers for Tomorrow's Internet," in IEEE Communications Magazine , May 1998.

[11] C. Labovitz, G. R. Malan, F. Jahanian. "Internet Routing Instability." Proc. ACM SIGCOMM `97, p. 115-126, Cannes, France.

[12] B. Lampson, V. Srinivasan, G. Varghese, "IP Lookups using Multiway and Multicolumn Search," Proc. IEEE Infocom `98.

[13] J. Manchester, J. Anderson, B. Doshi, S. Dravida, "IP over SONET", IEEE Communications Magazine, May 1998.

[14] N. McKeown, "Fast Switched Backplane for a Gigabit Switched Router", http://www.cisco.com/warp/public/733/12000/technical.shtml

[15] Merit Inc. IPMA Statistics. http://nic.merit.edu/ipma

[16] Netlogic Microsystems, IPCAM-3 ternary CAMs. http://www.netlogicmicro.com/.

[17] 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.

[18] C. Partridge et al., "A Fifty Gigabit Per Second IP Router", IEEE/ACM Trans. on Networking, Vol. 6, No. 3, June 1998, pp. 237-248.

[19] V. Srinivasan, G. Varghese, "Faster IP Lookups using Controlled Prefix Expansion," Proc. SIGMETRICS `98.

[20] V. Srinivasan, G. Varghese, S. Suri, M. Waldvogel, "Fast Scalable Algorithms for Layer 4 Switching" Proc ACM SIGCOMM `98

[21] Tzi-cker Chiueh, Prashant Pradhan, "High Performance IP Routing Table Lookup using CPU Caching," submitted to IEEE INFOCOMM 1999 .

[22] 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).

[23] P. Crescenzi, L. Dardini, R. Grossi, "IP Address Lookup Made Fast and Simple", European Symposium on Algorithms, 1999.

[24] W. Eatherton, "Hardware-Based Internet Protocol Prefix Lookups" MS Thesis, Washington University in St. Louis, 1998. Available at http://www.arl.wustl.edu/.

[25] S. Sikka and G. Varghese, "Memory-efficient state lookups with fast updates", SIGCOMM 2001.

[26] D. Taylor, J. Lockwood, T. Sproull, J. Turner, D. Parlour, Scalable IP Lookup for Programmable Routers, Infocom 2002.

[27] F. Zane, G. Narlikar, A. Basu. "CoolCAMs: Power-Efficient TCAMs for Forwarding Engines," Infocom 2003.

[28] P. Crowley , M. Franklin, H. Hadimioglu, P. Onufryk , Network Processor Design : Issues and Practices, Volume 1 , Morgan Kaufmann, October 2002

[29] T. Bu, L. Gao, D. Towsley, "ON Charactertizing Routing Table Growth", Proceedings of the GlobalInternet 2002