Small Forwarding Tables for Fast Routing Lookups

Mikael Degermark,[2] Andrej Brodnik,[3] Svante Carlsson,[2] and Stephen Pink [2 4]

micke@cdt.luth.se, Andrej.Brodnik@IMFM.Uni-Lj.SI, svante@sm.luth.se, steve@sics.se

Department of Computer Science and Electrical Engineering
Luleå University of Technology
S-971 87 Luleå, Sweden

Abstract

For some time, the networking community has assumed that it is impossible to do IP routing lookups in software fast enough to support gigabit speeds. IP routing lookups must find the routing entry with the *longest matching prefix*, a task that has been thought to require hardware support at lookup frequencies of millions per second.

We present a forwarding table data structure designed for quick routing lookups. Forwarding tables are small enough to fit in the cache of a conventional general purpose processor. With the table in cache, a 200 MHz Pentium Pro or a 333 MHz Alpha 21164 can perform a few million lookups per second. This means that it is feasible to do a full routing lookup for each IP packet at gigabit speeds without special hardware.

The forwarding tables are very small, a large routing table with 40,000 routing entries can be compacted to a forwarding table of 150–160 Kbytes. A lookup typically requires less than 100 instructions on an Alpha, using eight memory references accessing a total of 14 bytes.

1 Introduction

For some time, the networking community has assumed that it is impossible to do full IP routing lookups in software running on general purpose microprocessors fast enough to support routing at gigabit speeds. In fact, some believe that IP routing lookups cannot be done quickly at low cost in hardware [23].

We present a forwarding table that allows fast IP routing lookups in software. Pessimistic calculations based on experimental data show that Pentium Pro and Alpha 21164 processors can do at least two million full IP routing lookups per second. No traffic locality is assumed.

IP routers do a *routing lookup* in a *routing table* to determine where IP datagrams are to be forwarded. The result of the operation is the *next hop* on the path towards the destination. An entry in a routing table is conceptually an arbitrary length *prefix* with associated next-hop information. Routing lookups must find the routing entry with the *longest matching prefix*.

The belief that IP routing lookups are inherently slow and complex operations has lead to a proliferation of techniques to avoid doing them. Various link layer switching technologies below IP, IP layer bypass methods [15, 19, 20] and the development of alternative network layers based on virtual circuit technologies such as ATM, are, to some degree, results of a wish to avoid IP routing lookups.

The use of switching link layers and flow or tag switching architectures below the IP level adds complexity and redundancy to the network. Link layer switching and IP layer routing perform the same functions, so it would be simpler to have only one of these in the network.

Most current IP router designs use caching techniques where the routing entries of the most recently used destination addresses are kept in a cache. The technique relies on there being enough locality in the traffic so that the cache hit rate is sufficiently high and the cost of a routing lookup is amortized over several packets. These caching methods have worked well in the past. However, as the current rapid growth of the Internet increases the required size of address caches, hardware caches might become uneconomical.

---

[2] With the Centre for Distance-spanning Technology (CDT), Luleå, Sweden.

[3] Also at the Department of Theoretical Computer Science, Institute of Mathematics, Physics, and Mechanics, Jadranska 19, 1111 Ljubljana, Slovenia.

[4] Also at the Swedish Institute of Computer Science, PO box 1263, S-164 28 Kista, Sweden.

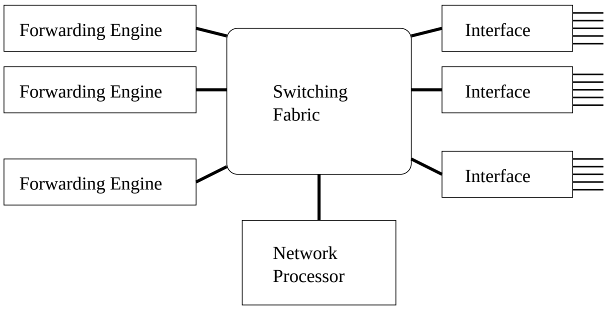

Figure 1: Router design with forwarding engines

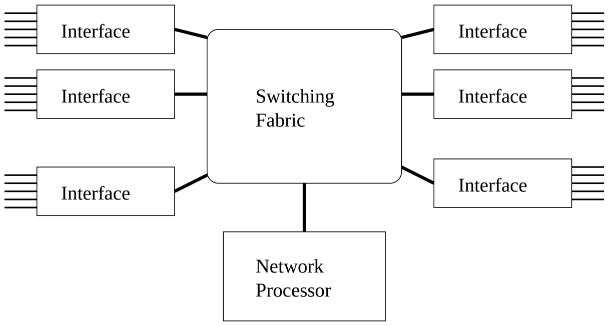

Figure 2: Router design with processing power on interfaces

Traditional implementations of routing tables use a version of Patricia trees [13], a data structure invented almost thirty years ago, with modifications for longest prefix matching. By applying modern results in algorithm theory, routing lookup performance can be improved by orders of magnitude compared to Patricia trees.

A straightforward implementation of Patricia trees for routing lookup purposes, for example in the NetBSD 1.2 implementation, uses 24 bytes for leaves and internal nodes. With 40,000 entries, the tree structure alone is almost 2 megabytes, and in a perfectly balanced tree 15 or 16 nodes must be traversed to find a routing entry. In some cases, due to the longest matching prefix rule, additional nodes need to be traversed to find the proper routing information as it is not guaranteed that the initial search will find the proper leaf. There are optimizations that can reduce the size of a Patricia tree and improve lookup speeds. Nevertheless, the data structure is large and too many expensive memory references are needed to search it. In short, Internet routing tables were too large to fit into on-chip caches and off-chip memory references onto DRAMs are too slow to support gigabit routing speeds.

In the rest of this paper we present a data structure that can represent large routing tables in a very compact form and can be searched quickly using few memory references. For the largest routing tables we have found at key interconnection points in the Internet [21, 22], the data structure is 150 – 160 Kbytes. That is small enough to fit entirely in the secondary cache of Pentium Pro processors, and to almost fit in the secondary cache of Alpha 21164 processors. A lookup with an Alpha processor typically requires less than 100 instructions, uses eight memory references, and accesses a total of 14 bytes. In the worst case, where the prefix is longer than 28 bits (very rare), an additional 50 instructions, four memory references, and 7 bytes are needed. With the data structure in secondary cache, both Alpha and Pentium Pro processors can do more than two million routing lookups per second. With a packet size of 1000 bits (125 bytes), that is equivalent to more than 2 Gbit/s.

2 Routing and forwarding tables

A router design is schematically shown in Figure 1. A number of *network interfaces*, *forwarding engines*, and a *network processor* are interconnected with a *switching fabric*. Inbound interfaces send packet headers to the forwarding engines through the switching fabric. The forwarding engines in turn determine which outgoing interface the packet should be sent to. This information is sent back to the inbound interface, which forwards the packet to the outbound interface. The only task of a forwarding engine is to process packet headers. All other tasks such as participating in routing protocols, resource reservation, handling packets that need extra attention, and other administrative duties, are handled by the network processor. The BBN Multigigabit router [17] is an example of this design.

Another router design is shown in Figure 2. Here, processing elements in the inbound interface decide to which outbound interface packets should be sent. The GRF routers from Ascend communications, for instance, use this design.

The forwarding engines in Figure 1 and the processing elements in Figure 2 uses a local version of the routing table, a *forwarding table*, downloaded from the network processor to make their routing decisions. It is not necessary to download a new forwarding table for each routing update. Routing updates can be frequent but since routing protocols need time in the order of minutes to converge, forwarding tables can grow a little stale and need not change more than at most once per second [6].

The network processor needs a dynamic routing table designed for fast updates and fast generation of forwarding tables. The forwarding tables, on the other hand, can be optimized for lookup speed and need not be dynamic.

3 Design goals and parameters

When designing the data structure used in the forwarding table, the primary goal was to minimize lookup time. To reach that goal, we simultaneously minimize two parameters;

- the number of *memory accesses* required during lookup, and
- the *size* of the data structure.

Reducing the number of memory accesses required during a lookup is important because memory accesses are relatively slow and usually the bottleneck of lookup procedures. If the data structure can be made small enough, it can fit entirely in the cache of a conventional microprocessor. This means that memory accesses will be orders of magnitude faster than if the data structure needs to reside in memory consisting of relatively slow DRAM, as is the case for Patricia trees.

If the forwarding table does not fit entirely in the cache, it is still beneficial if a large fraction of the table can reside in cache. Locality in traffic patterns will keep the most frequently used pieces of the data structure in cache, so that most lookups will be fast. Moreover, it becomes feasible to use fast SRAM for the small amount of needed external memory. SRAM is expensive, and more expensive the faster it is. For a given cost, the SRAM can be faster if less is needed.

As secondary design goals, the data structure should

- need *few instructions* during lookup, and
- keep the entities naturally aligned as much as possible to avoid expensive instructions and cumbersome bit-extraction operations.

These goals have a second-order effect on the performance of the data structure.

To determine quantitative design parameters for the data structure, we have investigated a number of large routing tables (see section 5). In these tables there are fairly few distinct next-hops, less than 60 distinct next-hops in tables consisting of up to 40,000 routing entries. If next-hops are identical, the rest of the routing information is also the same, and thus all routing entries specifying the same next-hop can share routing information. The number of distinct next-hops in the routing table of a router is limited by the number of other routers or hosts that can be reached in one hop, so it is not surprising that these numbers can be small even for large backbone routers. However, if a router is connected to, for instance, a large ATM network, the number of next-hops can be much higher.

The forwarding table data structure is designed to accommodate 2^{14} or 16K different next-hops. This should be sufficient for most cases. If there are fewer than 256 distinct next-hops, so that an index into the next-hop table can be stored in a single byte, the forwarding tables described here can be modified to occupy considerably less space.

4 The data structure

The forwarding table is essentially a tree with three levels. Searching one level requires one to four memory accesses. Consequently, the maximum number of memory accesses is twelve. However, with the routing tables we have tried, the vast majority of lookups requires searching one or two levels only, so the most likely number of memory accesses is eight or less.

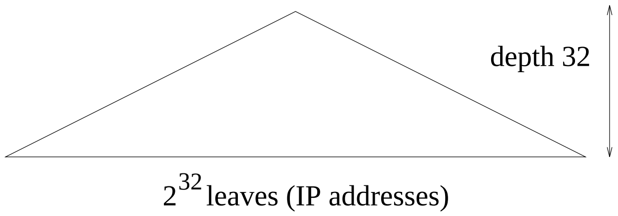

Figure 3: Binary tree spanning the entire IP address space.

For the purpose of understanding the data structure, imagine a binary tree that spans the entire IP address space (Figure 3). Its height is 32, and the number of leaves is 2^{32}, one for each possible IP address. The prefix of a routing table entry defines a path in the tree ending in some node. All IP addresses (leaves) in the subtree rooted at that node should be routed according to that routing entry. In this manner each routing table entry defines a range of IP addresses with identical routing information.

If several routing entries cover the same IP address, the rule of the *longest match* is applied; it states that for a given IP address, the routing entry with the longest matching prefix should be used. This situation is illustrated in Figure 4; the routing entry $e1$ is hidden by $e2$ for addresses in the range r.

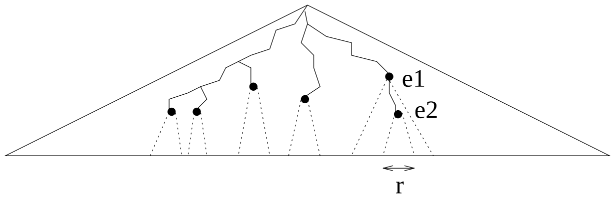

Figure 4: Routing entries defining ranges of IP addresses.

The forwarding table is a representation of the binary tree spanned by all routing entries. This is called the *prefix tree*. We require that the prefix tree is *complete*, i.e., that each node in the tree has either two or no children. Nodes with a single child must be expanded to have two children; the children added in this way are always leaves, and their next-hop information is the same as the next-hop of the closest ancestor with next-hop information, or the “undefined” next-hop if no such ancestor exists.

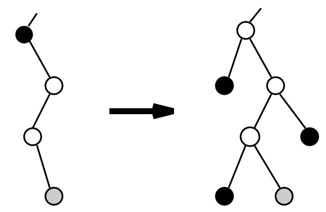

Figure 5: Expanding the prefix tree to be complete.

This procedure, illustrated in Figure 5, increases the number of nodes in the prefix tree, but allows building a small forwarding table. Note that it is not needed to actually build the prefix tree to build the forwarding table. We use the prefix tree to simplify our explanation. The forwarding table can be built during a single pass over all routing entries.

A set of routing entries partitions the IP address space into sets of IP addresses. The problem of finding the proper routing information is similar to the more general *interval set membership* problem [12]. However, in our case the intervals are defined by nodes in the complete prefix tree and, therefore, has properties that we can use to obtain an even smaller data structure. For instance, each range of IP addresses has a length that is a power of two.

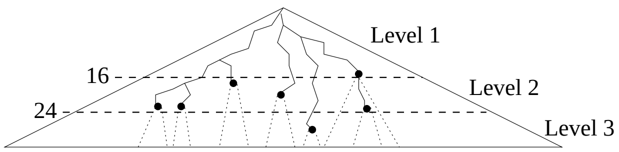

Figure 6: The three levels of the data structure.

As shown in Figure 6, level one of the data structure covers the prefix tree down to depth 16, level two covers depths 17 to 24, and level three depths 25 to 32. Wherever a part of the prefix tree extends below level 16, a level two *chunk* describes that part of the tree. Similarly, chunks at level three describe parts of the prefix tree that are deeper than 24. The result of searching one level of the data structure is either an index into the next-hop table or an index into an array of chunks for the next level.

4.1 Core result

Our core result is that we can represent a complete binary tree of height h using only one bit per possible leaf at depth h, plus one base index per 64 possible leafs, plus the information stored in the leaves. For $h > 6$, the size in bytes of a tree with l leaves holding information of size d is

$$2^{h-3} + b \times 2^{h-6} + l \times d \qquad (1)$$

where b is the size of a base index. With two-byte base indices, a tree of height 8 (a chunk) requires 40 bytes plus leaf information, and a tree of height 16 requires 10 Kbytes plus leaf information.

To achieve these small sizes, an additional 5408 byte table is needed. The table can be made smaller, 1352 bytes, but then typical processors will need more instructions when using it.

4.2 Level 1 of the data structure

The first level is essentially a tree node with 1 – 64K children. It covers the prefix tree down to depth 16.

Imagine a cut through the prefix tree at depth 16. The cut is represented by a bit-vector, with one bit per possible node at depth 16. 2^{16} bits = 64Kbits = 8 Kbytes are required for this. To find the bit corresponding to the initial part of an IP address, the upper 16 bits of the address is used as an index into the bit-vector.

Heads. When there is a node in the prefix tree at depth 16, the corresponding bit in the vector is set. Also, when the tree has a leaf at a depth less than 16, the lowest bit in the interval covered by that leaf is set. All other bits are zero. A bit in the bit vector can thus be

- a one representing that the prefix tree continues below the cut; a *root head* (bits 6, 12 and 13 in Figure 7), or
- a one representing a leaf at depth 16 or less; a *genuine head* (bits 0, 4, 7, 8, 14 and 15 in Figure 7), or
- zero, which means that this value is a *member* of a range covered by a leaf at a depth less than 16 (bits 1, 2, 3, 5, 9, 10 and 11 in Figure 7). Members have the same next-hop as the largest head smaller than the member.

The bit-vector is divided into bit-masks of length 16. There are $2^{12} = 4096$ of those.

Head information. For genuine heads we need to store an index into the next-hop table. Members will use the same next-hop as the largest head smaller than the member. For root heads, we need to store an index

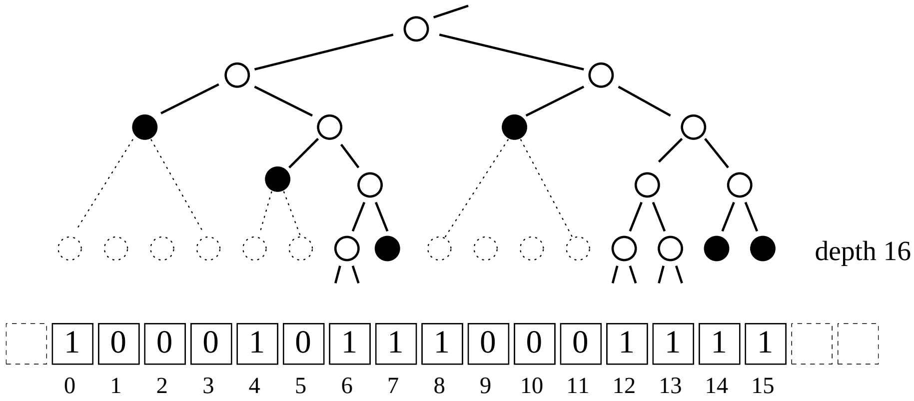

Figure 7: Part of cut with corresponding bit-vector

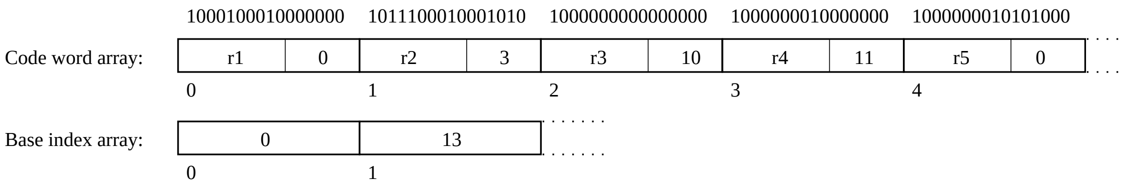

Figure 8: Bit-masks vs code words and base indices.

to the level two chunk that represents the corresponding subtree.

The head information is encoded in 16-bit *pointers* stored consecutively in an array. Two bits of each pointer encode what kind of pointer it is, and the 14 remaining bits either form an index into the next-hop table or an index into an array containing level two chunks. Note that there are as many pointers associated with a bit-mask as its number of set bits.

Finding pointer groups. Figure 8 is an illustration of how the data structure for finding pointers corresponds to the bit-masks. The data structure consists of an array of *code words*, as many as there are bit-masks, plus an array of base indices, one per four code words. The code words consists of a 10-bit value $(r1, r2, \ldots)$ and a 6-bit offset $(0, 3, 10, 11, \ldots)$.

The first bit-mask in Figure 8 has three set bits. The second code word thus has an offset of three because three pointers must be skipped over to find the first pointer associated with that bit-mask. The second bit-mask has 7 set bits and consequently the offset in the third code word is $3 + 7 = 10$.

After four code words, the offset value might be too large to represent with 6 bits. Therefore, a base index is used together with the offset to find a group of pointers. There can be at most 64K pointers in level 1 of the data structure, so the base indices need to be at most 16 bits ($2^{16} = 64$K). In Figure 8, the second base index is 13 because there are 13 set bits in the first four bit-masks.

This explains how a group of pointers is located. The first 12 bits of the IP address are an index to the proper code word, and the first 10 bits are an index to the array of base indices.

Maptable. It remains to explain how to find the correct pointer in the group of pointers. This is what the 10-bit value is for ($r1, r2,\ldots$in Figure 8). The value is an index into a table that maps bit-numbers in the IP address to pointer offsets. Since the bit-masks are 16 bits long, one might think that the table needs 64K entries. However, bit-masks are generated from a complete prefix tree, so not all combinations of the 16 bits are possible.

A non-zero bit-mask of length $2n$ can be any combination of two bit-masks of length n or the bit-mask with value 1. Let $a(n)$ be the number of possible non-zero bit-masks of length 2^n. $a(n)$ is defined by the recurrence

$$a(0) = 1, \quad a(n) = 1 + a(n-1)^2 \qquad (2)$$

The number of possible bit-masks with length 16 are thus $a(4) + 1 = 678$, the additional one is because the bit-mask can be zero. An index into a table with an entry for each bit-mask thus only needs 10 bits.

We keep such a table, `maptable`, to map bit numbers within a bit-mask to 4-bit offsets. The offset specifies how many pointers to skip over to find the wanted one, so it is equal to the number of set bits smaller than the bit index. These offsets are the same for all forwarding tables, regardless of what values the pointers happen to have. `Maptable` is constant, it is generated once and for all.

Searching. The steps in Figure 10 are required to search the first level of the data structure; the array of

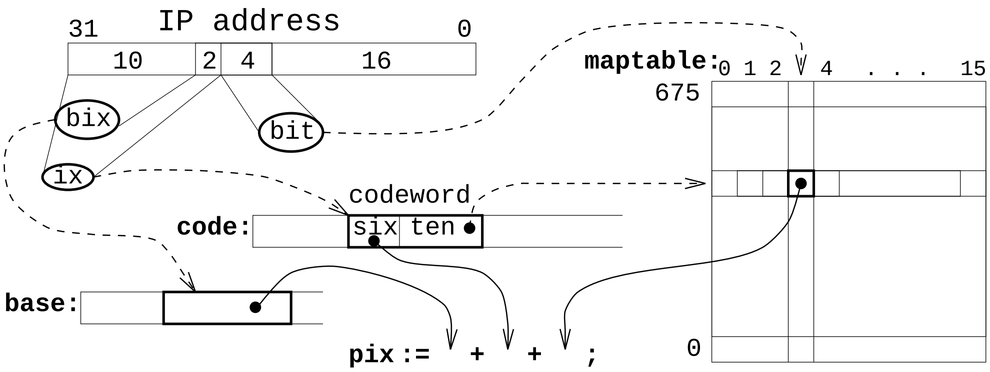

Figure 9: Finding the pointer index.

code words is called `code`, and the array of base indices is called `base`. Figure 9 illustrates the procedure. The

```
ix  := high 12 bits of IP address
bix := high 10 bits of IP address
bit := low 4 of high 16 bits of IP address
codeword := code[ix]
ten := ten bits from codeword
six := six bits from codeword
pix := base[bix] + six + maptable[ten][bit]
pointer := level1_pointers[pix]
```

Figure 10: Steps to search the first level

index of the code word, `ix`, the index of the base index, `bix`, and the bit number, `bit`, are first extracted from the IP address. Then the code word is retrieved and its two parts are extracted into `ten` and `six`. The pointer index, `pix`, is then obtained by adding the base index, the 6-bit offset `six`, and the pointer offset obtained by retrieving column `bit` from row `ten` of `maptable`. After the pointer is retrieved from the pointer array, it will be examined to determine if the next-hop has been found or if the search should continue on the next level.

The code is extremely simple. A few bit extractions, array references, and additions is all that is needed. No multiplication or division instructions are required except for the implicit multiplications when indexing an array.

A total of 7 bytes needs to be accessed to search the first level: a two byte code word, a two byte base address, one byte (4 bits, really) in `maptable`, and finally a two byte pointer. The size of the first level is 8K bytes for the code word array, 2K bytes for the array of base indices, plus a number of pointers. The 5.3 Kbytes required by `maptable` are shared among all three levels.

4.2.1 Optimizations at level 1

When the bit-mask is zero or has a single bit set, the pointer must be an index into the next-hop table. Such pointers can be encoded directly into the code word and thus `maptable` need not contain entries for bit-masks one and zero. The number of `maptable` entries is thus reduced to 676 (indices 0 through 675). When the ten bits in the code word (`ten` above) are larger than 675, the code word represents a direct index into the next-hop table. The six bits from the code word are used as the lowest 6 bits in the index, and (`ten-676`) are the upper bits of the index. This encoding allows at most $(1024 - 676) \times 2^6 = 22272$ next-hop indices, which is more than the 16K we are designing for. The optimization eliminates three memory references when a routing entry is located at depth 12 or higher, and reduces the number of pointers in the pointer array considerably. The cost is a comparison and a conditional branch.

4.3 Levels 2 and 3 of the data structure

Levels two and three of the data structure consist of *chunks*. A chunk covers a subtree of height 8 and can contain at most $2^8 = 256$ heads. A root head in level $n-1$ points to a chunk in level n.

There are three varieties of chunks depending on how many heads the imaginary bit-vector contains. When there are

- 1–8 heads, the chunk is *sparse* and is represented by an array of the 8-bit indices of the heads, plus eight 16-bit pointers; a total of 24 bytes.
- 9–64 heads, the chunk is *dense*. It is represented analogously with level 1, except for the number of base indices. The difference is that only one base index is needed for all 16 code words, because 6-bit offsets can cover all 64 pointers. A total of 34 bytes are needed, plus 18 to 128 bytes for pointers.
- 65–256 heads, the chunk is *very dense*. It is represented analogously with level 1. 16 code words and

4 base indices give a total of 40 bytes. In addition the 65 to 256 pointers require 130 to 512 bytes.

Dense and very dense chunks are searched analogously with the first level. For sparse chunks, the 1 to 8 values are placed in decreasing order. To avoid a bad worst-case when searching, the fourth value is examined to determine if the desired element is among the first four or last four elements. After that, a linear scan determines the index of the desired element, and the pointer with that index can be extracted. The first element less than or equal to the search key is the desired element. At most 7 bytes need to be accessed to search a sparse chunk.

4.3.1 Optimizations at levels two and three

Dense and very dense chunks are optimized analogously with level 1 as described in section 4.2.1. In sparse chunks, consecutive heads can be merged and represented by the smallest if their next-hops are identical. When deciding whether a chunk is sparse or dense, this merging is taken into account so that the chunk is deemed sparse when the number of *merged* heads is 8 or less. Many of the leaves that were added to make the tree complete will occur in order and have identical next-hops. Heads corresponding to such leaves will be merged in sparse chunks.

This optimization shifts the chunk distribution from the larger dense chunks towards the smaller sparse chunks. For large tables, the size of the forwarding table is typically decreased by 5 to 15 per cent.

4.4 Growth limitations in the current design

The data structure can accommodate considerable growth in the number of routing entries. There are three limits in the current design.

1. The number of chunks of each kind is limited to $2^{14} = 16384$ per level.

   Table 1 shows that this is about 16 times more than is currently used. If the limit is ever exceeded, the data structure can be modified so that pointers are encoded differently to give more room for indices, or so that the pointer size is increased.

2. The number of pointers in levels two and three is limited by the size of the base indices.

   The current implementation uses 16-bit base indices and can accommodate a growth factor of 3 to 5. If the limit is exceeded it is straightforward to increase the size of base pointers to three bytes. The chunk size is then increased by 3 per cent for dense chunks and 10 per cent for very dense chunks. Sparse chunks are not affected.

3. The number of distinct next-hops is limited to $2^{14} = 16384$.

   If this limit is exceeded all next-hop indices cannot be encoded directly into code words, as explained in section 4.2.1. It is possible to avoid storing a pointer when the bit-mask is zero. When the bit-mask has one head, however, it is necessary that a pointer is stored. Consequently the size of the data structure will increase because there needs to be one pointer per interval and pointers are larger.

To conclude, with small modifications the data structure can accommodate a large increase in the number of routing entries.

5 Performance measurements

To investigate the performance of the forwarding tables, a number of IP routing tables were collected. Internet routing tables are currently available at the web site for the Internet Performance Measurement and Analysis (IPMA) project [22], and were previously made available by the now terminated Routing Arbiter project [21]. The collected routing tables are daily snapshots of the routing tables used at various large Internet interconnection points. Some of the routing entries in these tables contain multiple next-hops. In that case, one of them was randomly selected as the next-hop to use in the forwarding table.

5.1 Size of forwarding table

Table 1 shows data on forwarding tables constructed from various routing tables. For each site, it shows data and results for the routing table that generated the largest forwarding table. *Routing entries* is the number of routing entries in the routing table, and *Next-hops* is the number of distinct next-hops found in the table. *Leaves* is the number of leaves in the prefix tree after leaves have been added to make it complete.

Build time in Table 1 is the time required to generate the forwarding table from an in-memory binary tree representation of the routing table. Times were measured on a 333 MHz Alpha 21164 running DEC OSF1. Subsequent columns show the total number of sparse, dense, and very dense chunks in the generated table followed by the number of chunks in the lowest level of the data structure.

It is clear from Table 1 that new forwarding tables can be generated quickly. At a regeneration frequency of one Hz, less than one tenth of the Alpha's capacity is consumed. As discussed in section 2, higher regeneration frequencies than 1 Hz are not required.

The larger tables in Table 1 do not fit entirely in the 96 Kbyte secondary cache of the Alpha. It is feasible,

Site	Date	Year	Routing entries	Leaves	next-hops	Size (Kb)	Build time	sparse chunks	dense chunks	dense+ chunks	level 3 chunks
Mae East	Jan 9	'97	32732	58714	56	160	99 ms	1199	587	186	2
Mae East	Oct 21	'96	38141	36607	50	148	91 ms	1060	593	149	4
Sprint	Jan 1	'97	21797	43513	17	123	72 ms	988	483	98	3
PacBell	Jan 28	'97	18308	33250	2	99	49 ms	873	357	67	0
Mae West	Jan 1	'97	12049	28273	51	86	46 ms	775	312	42	3
AADS	Jan 4	'97	1109	5670	12	28	11 ms	320	38	0	2

Table 1: Forwarding table generation data

Processor	Clock cycle	Primary Cache Size	Primary Cache Latency	Secondary Cache Size	Secondary Cache Latency	Tertiary Cache Size	Tertiary Cache Latency
Alpha 21164	3 ns	8 Kbyte	6 ns	96 Kbyte	24 ns	2 Mbyte	72 ns
Pentium Pro	5 ns	8 Kbyte	10 ns	256 Kbyte	30 ns		

Table 2: Processor and cache data

however, to have a small amount of very fast SRAM in the third level cache for the pieces that do not fit in the secondary cache, and thus reduce the cost of a miss in the secondary cache. With locality in traffic patterns, most memory references would be to the secondary cache.

An interesting observation is that the size of these tables are comparable to what it would take to just store all prefixes in an array. For the larger tables, no more than 5.6 bytes per prefix is needed. More than half of these bytes are consumed by pointers. In the Sprint table there are 33469 pointers that require over 65 Kbytes of storage. It is clear that further reductions of the forwarding table size could be accomplished by reducing the number of pointers.

5.2 Lookup performance

Our measurements of lookup speed are done on a C function compiled with the GNU C-compiler `gcc`. Reported times do not include the function call or the memory access to the next-hop table. `gcc` generates code that uses approximately 50 Alpha instructions to search one level of the data structure in the worst case. On a Pentium Pro, `gcc` generates code that uses 35 to 45 instructions per level in the worst case. It is conceivable that better code can be obtained by hand-coding the lookup routine in assembler; we have not tried this.

It is possible to read the current value of the clock cycle counter on Alphas and Pentium Pros. We have used this facility to measure lookup times with high precision: one clock tick is 5 nanoseconds at 200 MHz and 3 nanoseconds at 333 MHz.

Ideally, we would like to place the entire forwarding table in cache so lookups would be performed with an undisturbed cache. That would emulate the cache behavior of a dedicated forwarding engine. However, we have access to conventional general-purpose workstations only and it is difficult to control the cache contents on such systems. The cache is disturbed whenever I/O is performed, an interrupt occurs, or another process gets to run. It is not even possible to print out measurement data or read a new IP address from a file without disturbing the cache.

The best method we could devise is to perform each lookup *twice*, measuring the lookup time for the second lookup. In this way, the first lookup is done with a disturbed cache and the second in a cache where all necessary data has been forced into the primary cache by the first lookup. After each pair of lookups measurement data is printed out and a new address is fetched, a procedure that again disturbs the cache.

The second lookup will perform better than lookups in a forwarding engine because data and instructions have moved into the primary cache closest to the processor. To get an upper limit on the lookup time, the additional time required for memory accesses to the secondary cache must be added to the measured times. To test all paths through the forwarding table, lookup time was measured for each entry in the routing table, including the entries added by the expansion to a complete tree.

Average lookup times can not be inferred from these experiments because it is not likely that a realistic traffic mix would have a uniform probability for accessing each routing entry. Moreover, locality in traffic patterns will

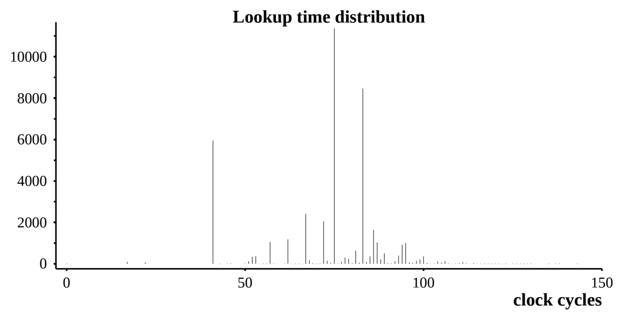

Figure 11: Lookup time distribution, Alpha 21164

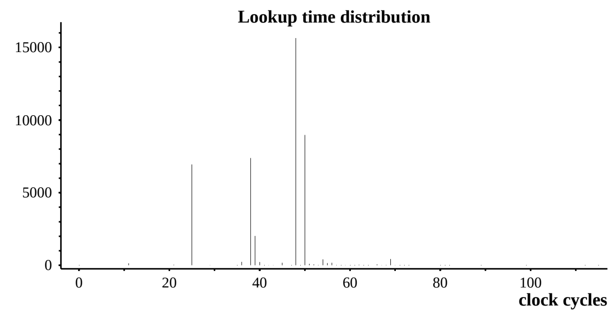

Figure 12: Lookup time distribution, Pentium Pro

keep frequently accessed parts of the data structure in the primary cache and, thus, reduce the average lookup time. The performance figures calculated below are conservative because it is assumed that all memory accesses miss in the primary cache, and that the worst case execution time will always occur. Realistic lookup speeds would be higher.

Table 1 show that there are very few chunks in level three of the data structure. That makes it likely that the vast majority of lookups need to search no more than two levels to find the next hop. Therefore, the additional time for memory accesses to the secondary cache is calculated for eight instead of the worst-case twelve memory accesses. If a significant fraction of lookups were to access those few chunks, they would migrate into the primary cache and all twelve memory accesses would become less expensive.

Lookup performance for Alpha 21164

The Alpha 21164 we experimented with has a clock frequency of 333 MHz; one cycle takes 3 nanoseconds. Accesses to the 8 Kbyte primary data cache completes in 2 cycles and accesses to the secondary 96 Kbyte cache requires 8 cycles. See Table 2.

Figure 11 shows the distribution of clock ticks elapsed during the second lookup for the Alpha on the Sprint routing table from January 1st. The fastest observed lookups require 17 clock cycles. This is the case when the code word in the first level directly encodes the index to the next-hop table. There are very few such routing entries. However, as each such routing entry covers many IP addresses, actual traffic might contain many such destination addresses. Some lookups take 22 cycles, which must be the same case as the previous. Experiments have confirmed that when the clock cycle counter is read with two consecutive instructions, the difference is sometimes 5 cycles instead of the expected 0.

The next spike in Figure 11 is at 41 clock cycles, which is the case when the pointer found in the first level is an index to the next-hop table. Traditional class B addresses fall in this category. Spikes at 52–53, 57, 62, 67, and 72 ticks correspond to finding the pointer after examining one, two, three, four, or five values in a sparse level 2 chunk. The huge spikes at 75 and 83 ticks are because that many ticks are required to search a dense and very dense chunk, respectively. A few observations above 83 correspond to pointers found after searching a sparse level 3 chunk, but we believe that most are due to variations in execution time. Cache conflicts in the secondary cache, or differences in the state of pipelines and cache system before the lookup, can cause such variations. The tail of observations above 100 clock cycles are either due to such variations or to cache misses. 300 nanoseconds should be sufficient for a lookup when all data is in the primary cache.

The difference between a data access in the primary cache and the secondary cache is $8-2 = 6$ cycles. Thus, searching two levels of the data structure in the worst case requires $8\times 6 = 48$ clock cycles more than indicated by Figure 11. That means at most $100+48 = 148$ cycles or 444 nanoseconds for the worst case lookup when 2 levels are sufficient. The Alpha should thus be able to do at least 2.2 million routing lookups per second with the forwarding table in the secondary cache.

Lookup performance for Pentium Pro

The Pentium Pro we experimented with has a clock frequency of 200 MHz; one cycle takes 5 nanoseconds. The primary 8 Kbyte data cache has a latency of 2 cycles and the secondary cache of 256 Kbytes has a latency of 6 cycles. See Table 2. The latency of the Pentium Pro caches were measured using the tool *lmbench* [10, 11] as we were unable to obtain this information otherwise.

Figure 12 shows the distribution of clock ticks elapsed during the second lookup for the Pentium Pro with the same forwarding table as in the previous section. The sequence of instructions that fetches the clock cycle counter takes 33 clock cycles. When two fetches occur immediately after each other the counter values differ by 33. For this reason, all reported times have been reduced by 33.

The fastest observed lookups are 11 clock cycles, about the same speed as for the Alpha. The spike corresponding to the case when the next-hop index is found immediately after the first level occurs at 25 clock cycles. The spikes corresponding to a sparse level 2 chunk are grouped closely together in the range 36 to 40 clock cycles. The different caching structure of the Pentium seems to deal better with linear scans than the caching structure of the Alpha.

When the second level chunks are dense and very dense, the lookup requires 48 and 50 cycles, respectively. There are some additional irregular spikes up to 69, above which there are very few observations. It is clear that 69 cycles (345 nanoseconds) is sufficient to do a lookup when all data is in the primary cache.

The difference in access time between the primary and secondary cache is 20 nanoseconds (4 cycles). The lookup time for on the Pentium Pro when two levels need to be examined is then at worst $69 + 8 \times 4 = 101$ cycles or 505 nanoseconds. The Pentium Pro can do at least 2.0 million routing lookups per second with the forwarding table in secondary cache.

6 Scaling

The number of instructions used for lookup is independent of the size of the forwarding table. Thus, the number of routing entries does not affect lookup performance as long as the forwarding table fits in cache. If it does not fit entirely in cache, some lookups will access slower memory. Traffic locality will then determine how much average lookup performance decreases.

Forwarding table size

The table size is at least 15.3 Kbyte, because that is what is needed for `maptable` and the first level (excluding pointers). The rest of the table size is at most linear in the number of leaves in the prefix tree.

The relation between the number of prefixes and the number of leaves in the prefix tree is not simple. It will depend on how prefixes are spread out over the address space. It is easy to construct a prefix set that maximizes the table size by maximizing the number of sparse chunks. This is the worst case for the table size. If such prefix distributions were to become common, however, it would be simple to introduce a new kind of chunk that dealt better with this situation.

Because the address space is limited, it is not appropriate to use $\mathcal{O}()$-notation to describe how forwarding tables grow with the number of routing entries. $\mathcal{O}()$-notation is defined to capture the assymptotical growth. However, when the number of prefixes approaches the number of possible addresses, the number of leaves in the prefix tree will approach the number of prefixes assymptotically. Assymptotical growth is a bad indication of how the table size increases with the number of routing entries for the table sizes we worry about.

Our experimental data indicate that, for larger tables, the table size is around 4-5 bytes per prefix plus the fixed cost of 15.3 Kbytes. However, the forwarding table includes the table of next-hop information, which increases linearly with the number of distinct next-hops. The cost per prefix will grow significantly if most routing entries have distinct next-hops.

Table building time

The reported table building times (Table 1) are for building the forwarding table from an in-memory prefix tree. We have devised a way to build the table that is linear in the number of routing entries and in the size of the resulting forwarding table. The table is built during a single pass over all routing entries.

Larger addresses

With the coming of IPv6 [4, 8] it is desirable to do fast lookups for 128-bit IPv6 addresses as well. With such large addresses, there is a danger of inflating the table size if the address space is sparsely utilized everywhere. However, there are techniques to adapt depths of chunks to the density and sparsity of the prefix tree so that a small size can be guaranteed. To tune the data structure to the actual properties of the IPv6 address space, a number of representative IPv6 routing tables would have to be examined, but such tables do not yet exist. We strongly believe that small forwarding tables and fast routing lookups are possible for IPv6 as well as IPv4.

7 Related work

We are not aware of any substantial improvements in the performance of software for full IP routing lookups in recent years. However, [5] shows how to extend Patricia trees to deal better with longest matching prefix searches, insertions and deletions. The resulting data structure is called *dynamic prefix tries*. There are at least as many nodes in a dynamic prefix trie as in the corresponding Patricia tree. Nodes contain five pointers, a bit-index, and a prefix, so the resulting data structure is fairly large. Consequently, the lookup time is reported to be between 6 and 13 microseconds when the data structure holds 40 000 entries. However, the insertion and deletion operations appear efficient, so dynamic prefix tries might be a good candidate for the routing table maintained by the network processor.

An early work on improving IP routing performance by *avoiding* full routing lookups [7] found that a small destination address cache can improve routing lookup performance by at least 65 per cent. Less than 10 slots was needed to get a hit rate over 90 per cent. Much larger destination address caches are needed with the larger traffic intensities and number of hosts in today's Internet; several thousand slots are necessary.

ATM avoids doing routing lookups by having a signaling protocol that passes addresses to the network during connection setup. Forwarding state, accessed by a virtual circuit identifier (VCI), is installed in switches along the path of the connection during setup. ATM cells are labeled with the VCI which can then be used as a direct index into a table with forwarding state or as the key to a hash function. The routing decision is simpler for ATM. However, when packet sizes are larger than 48 bytes, more ATM routing decisions need to be made. When packets are large, it can be more efficient to make a few IP routing lookups instead of a large number of ATM VCI lookups. If network traffic consists mostly of large packets in the future, IP will be more efficient.

Tag switching and flow switching [15] are two IP bypass methods that were originally meant to be operated over ATM. The general idea is to let IP control link-level ATM hardware that performs actual data forwarding. Special purpose protocols [14] are needed between routers to agree on what ATM virtual circuit identifiers to use and which packet should use which VCI. If IP processing was fast enough, that extra machinery would not be needed.

Another approach with the same goal of avoiding IP processing is taken in the IP/ATM architecture [19, 20], where an ATM backplane connects a number of line cards and routing cards. IP processing elements located in the routing cards process IP headers. When a packet stream arrives, only the first IP header is examined and the later packets are routed the same way as the first one. The main purpose of these shortcuts seems to be to amortize the cost of IP processing over many packets. Again, that would not be necessary if IP processing was fast enough.

IP router designs can use special-purpose hardware to do IP processing, as in the IBM router [1]. This can be an inflexible solution. Any changes in the IP format or protocol could invalidate such designs. The flexibility of software and the rapid performance increase of general purpose processors makes such solutions preferable. Another hardware approach is to use CAMs to do routing lookups [9]. This is a fast but expensive solution.

BBN is currently building a pair of multi-gigabit routers that use general purpose processors as forwarding engines [17]. Little information has been published so far. The idea, however, seems to be to use Alpha processors as forwarding engines and do all IP processing in software. [18] shows that it is possible to do IP processing in no more than 200 instructions, assuming a hit in a route cache. Less than 100 instructions are necessary according to [17]. The secondary cache of the Alpha is used as a large LRU cache of destination addresses. The scheme presumes locality in traffic patterns. With low locality the cache hit rate could become too low and performance would suffer.

8 Discussion and further work

A processor in a router or forwarding engine would presumably do other IP processing than routing lookups. However, assuming that other IP processing requires 100 instructions in the common path, Pentium Pros and Alphas are still powerful enough to process a million IP packets per second.

Our analysis of the performance of forwarding table lookups is conservative. The longest observed lookup time was used and it was assumed that memory accesses always missed in the primary cache. Even if the access probability for routing entries was uniformly distributed, there would be several hits in the primary cache. With locality, many accesses would be to the primary cache and performance would increase even further. A natural way to continue this work is to study cache behavior using realistic packet traces. Significantly lower average lookup times are expected.

Increased routing lookup speeds will make caching of destination addresses less sensitive to low locality in traffic patterns. Designs using route caching will be sound for much lower cache hit rates when routing lookups are fast.

The current forwarding table sizes of around 150–160 Kbytes for the largest routing tables are still not small enough to fit entirely in the second level cache of Alpha 21164s. The current small size and fast lookups has been realized by applying recent work in algorithm theory [2, 3, 16] and by careful tuning of the data structure. There is reason to believe that this field of algorithm theory will develop even further. Moreover, a number of techniques to reduce the table size even further are still untried. There is still hope of making forwarding tables small enough to fit entirely in the secondary 96 Kbyte cache of Alpha 21164 processors.

This paper is focused on performing routing lookups with general-purpose processors. It is also possible to do routing lookups with special-purpose hardware. It should be straightforward to implement the lookup algorithm presented here in hardware. The small memory consumption is beneficial for hardware implementations as well as software implementations. It is easy to see that two of the memory accesses in Figure 7 can be done in parallel. With pipelining, the lookup time can be reduced to what it takes to search one level of the data structure.

9 Conclusion

We have shown the feasibility of doing full IP routing lookups per packet at gigabit speeds. The technique involves generating a compact forwarding table that can be searched quickly to find the longest matching prefix. No special hardware is required. Pessimistic calculations based on experimental data show that general purpose processors are capable of performing several million full IP routing lookups per second. With locality in traffic, lookup speeds will be even higher.

These forwarding tables can scale to accommodate arbitrary growth in the size of routing tables. With small modifications, there is practically no limit. The solution is general. Similar techniques can be applied to the larger addresses of IPv6.

References

[1] Abhaya Asthana, Catherine Delph, H. V. Jagadish, and Paul Krzyzanowski. Towards a gigabit IP router. *Journal of High Speed Networks*, 1(4):281–288, 1993.

[2] A. Brodnik and J.I. Munro. Membership in a constant time and a minimum space. In *Proceedings 2^{nd} European Symposium on Algorithms*, volume 855 of *Lecture Notes in Computer Science*, pages 72–81. Springer-Verlag, 1994.

[3] A. Brodnik and J.I. Munro. Neighbours on a grid. In *Proceedings 5^{th} Scandinavian Workshop on Algorithm Theory*, volume 1097 of *Lecture Notes in Computer Science*, pages 307–320. Springer-Verlag, 1996.

[4] S. Deering and R. Hinden. Internet Protocol, Version 6 (IPv6) Specification. Request for Comments (Proposed Standard) RFC 1883, Internet Engineering Task Force, January 1996.

[5] Willibald Doeringer, Günter Karjoth, and Mehdi Nassehi. Routing on longest-matching prefixes. *IEEE/ACM Transactions on Networking*, 4(1):86–97, February 1996.

[6] Stanford University Workshop on Fast Routing and Switching, December 1996.
`http://tiny-tera.stanford.edu/Workshop_Dec96/` .

[7] David C. Feldmeier. Improving gateway performance with a routing-table cache. In *Proceedings of the Conference on Computer Communications (IEEE Infocom)*, New Orleans, Louisiana, March 1988. IEEE.

[8] Robert Hinden. IP Next Generation Home Page.
`http://playground.sun.com/pub/ipng/html/ipng-main.html` .

[9] A. J. McAuley and P. Francis. Fast routing table lookup using CAMs. In *Proceedings of the Conference on Computer Communications (IEEE Infocom)*, volume 3, pages 1382–1391, San Francisco, 1993.

[10] Larry McVoy. lmbench home page.
`http://reality.sgi.com/lm/lmbench/lmbench.html` .

[11] Larry McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. In *USENIX Winter Conference*, January 1996. Available at
`http://reality.sgi.com/lm/lmbench/lmbench-usenix.ps` .

[12] K. Mehlhorn, S. Näher, and H. Alt. A lower bound on the complexity of the union-split-find problem. *SIAM Journal on Computing*, 17(1):1093–1102, December 1988.

[13] Donald R. Morrison. PATRICIA — Practical Algorithm to Retreive Information Coded In Alfanumeric. *Journal of the ACM*, 15(4):514–534, October 1968.

[14] P. Newman, W. L. Edwards, R. Hinden, E. Hoffman, F. Ching Liaw, T. Lyon, and G. Minshall. Ipsilon Flow Management Protocol Specification for IPv4, Version 1.0. Request For Comment RFC 1953, Internet Engineering Task Force, May 1996.

[15] Peter Newman, Tom Lyon, and Greg Minshall. Flow labeled IP: a connectionless approach to ATM. In *Proceedings of the Conference on Computer Communications (IEEE Infocom)*, San Francisco, California, March 1996.

[16] S. Nilsson. *Radix Sorting & Searching*. PhD thesis, Department of Computer Science, Lund University, 1996.

[17] C. Partridge, P. Carvey, E. Burgess, I. Castineyra, T. Clarke, L. Graham, M. Hathaway, P. Herman, A. King, S. Kohlami, T. Ma, T. Mendez, W. Milliken, R. Osterlind, R. Pettyjohn, J. Rokosz, J. Seeger, M. Sollins, S. Storch, B. Tober, G. Troxel, D. Waitzman, and S. Winterble. A fifty gigabit per second ip router. *IEEE/ACM Transactions on Networking*, To Appear.

[18] Craig Partridge. *Gigabit networking*. Addison-Wesley, Reading, Massachusetts, 1993.

[19] Guru Parulkar, Douglas C. Schmidt, and Jonathan Turner. IP/ATM: A strategy for integrating IP with ATM. *Computer Communication Review*, 25(4):49–58, October 1995. Proceedings ACM SIGCOMM '95 Conference.

[20] Gurudatta Parulkar, Douglas C. Schmidt, and Jonathan S. Turner. GIPR: a gigabit IP router. In *Proc. of Gigabit Networking Workshop*, Boston, Massachusetts, April 1995.

[21] The Routing Arbiter Project. Internet routing and network statistics.
`http://www.ra.net/statistics/` .

[22] Michigan University and Merit Network. Internet Performance Management and Analysis (IPMA) Project. Details available at
`http://nic.merit.edu/~ipma/` .

[23] Washington University Workshop on Integration of IP and ATM, November 1996. Proceedings from session 5. Available at
`http://www.arl.wustl.edu/arl/workshops/atmip/` .