Fast Incremental Updates for Pipelined Forwarding Engines

Anindya Basu and Girija Narlikar, *Member, IEEE*

***Abstract*— Pipelined ASIC architectures are increasingly being used in forwarding engines for high speed IP routers. We explore optimization issues in the design of memory-efficient data structures that support fast incremental updates in such forwarding engines. Our solution aims to balance the memory utilization across the multiple pipeline stages. We also propose a series of optimizations that minimize the disruption to the forwarding process caused by route updates. These optimizations reduce the update overheads by over a factor of two for a variety of different core routing tables and update traces.**

***Index Terms*— Packet forwarding, pipelined IP lookup, core routers, route updates.**

I. Introduction

Recent advances in optical networking technology have pushed linecard data transfer rates in high speed IP routers to 40Gbits/s, and even higher data rates are expected in the near term. Given such high data rates, packet forwarding in high speed IP routers must be done in hardware. Current hardware-based solutions for high speed packet forwarding fall into two main categories, namely, ASIC-based solutions and ternary CAM or TCAM-based solutions.[1]

In this paper, we focus on ASIC-based packet forwarding engines that utilize pipelining. ASIC-based architectures usually implement a routing trie data structure using some sort of high speed memory such as static RAMs (SRAMs) [1]. If a single SRAM memory block is used to store the entire routing trie, multiple accesses (one per routing trie level) are required to forward a single packet. This can slow down lookups considerably, and the forwarding engine may not be able to process incoming packets at the line rate. A number of researchers have pointed out that forwarding speeds can be significantly increased if pipelining is used in ASIC-based forwarding engines [2–4] — with multiple stages in the pipeline (e.g., 1 stage per trie level), one packet can be forwarded during every memory access time period.

In addition, pipelined ASICs that implement forwarding tries provide a general and flexible architecture for a wide variety of forwarding and classification tasks. This flexibility is a major advantage in today's high-end routers which have to provide IPv6 and multicast routing in addition to IPv4 routing, as well as packet classification or filtering. Since longest prefix matching is applicable to all these tasks, the same pipelined hardware can be used to perform them efficiently, thereby producing significant savings in cost, complexity and space. We also note that individual solutions tailored for a particular forwarding/filtering task can be designed. For example, the solution discussed in [2] leverages the distribution of IPv4 routing prefixes to provide fast hardware lookups at low cost. However, to our knowledge, a general hardware architecture that can simultaneously accommodate various routing and forwarding tasks as efficiently as a trie-based pipelined ASIC is difficult to design.

Despite the various advantages of pipelined ASIC architectures, managing routing tries during route updates in such architectures is difficult. One way to simplify management is to use double buffering, that is, to create a duplicate copy of the lookup trie and use one for lookups and the other for updates. However, the memory required becomes twice as much as in the normal case. Therefore, we concentrate here on providing efficient incremental updates. When updates are applied incrementally, however, packet forwarding operations can be disrupted by trie update operations—hence, the cost of update operations needs to be minimized.

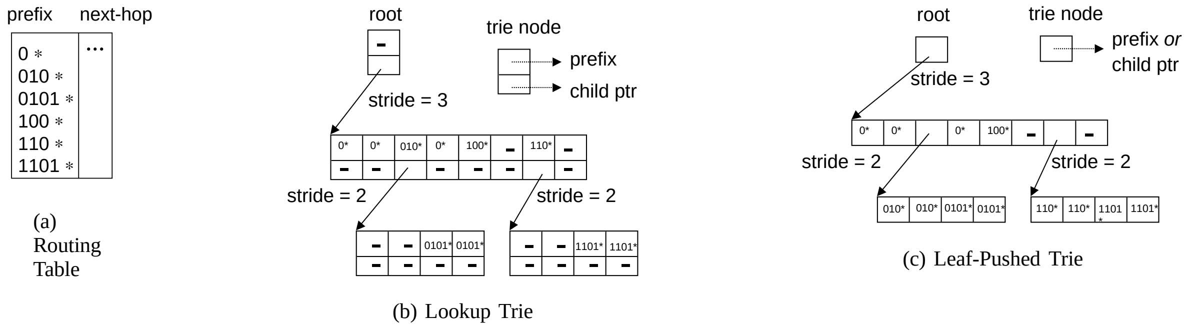
There are two main issues that affect the number of trie update operations when incremental updates are applied. First, the memory allocated to the trie should be evenly balanced across the multiple pipeline stages (in addition to minimizing the total memory use). Otherwise, the more heavily utilized stages may overflow on frequent insertions, and the entire trie will have to be reconstructed. This can create a heavy update load that will cause significant disruption to packet forwarding operations. Second, the (possibly multiple) memory locations that are modified due to route update operations must be limited in number and evenly balanced across the multiple pipeline stages, ensuring that no particular stage becomes a bottleneck. This will also allow modifications to different memory locations (up to one per pipeline stage) due to multiple route updates to be combined into a single pipeline update operation.

The main contribution of this paper is to address both these issues. First, we present an algorithm to build a memory-efficient trie while balancing the memory utilization over the different pipeline stages. The balancing problem was first mentioned in [3] where the authors pointed out that this is a difficult problem, and left it as future work. Our algorithm balances the memory requirements across the pipeline stages by finding the trie for which the size of the largest pipeline stage is *minimized*. We find that the balance of memory allocations for the trie (based on the initial prefix database)

This work was done while Anindya Basu was at Bell Labs, Lucent Technologies, Murray Hill, NJ 07974 (email: anindya@gmail.com).

Girija Narlikar is with Bell Labs, Lucent Technologies, Murray Hill, NJ 07974 (email: narlikar@lucent.com).

[1] Network processor-based solutions are also increasingly being considered for high speed packet forwarding, though network processors that can handle 40Gbits/s wire speeds are not yet available (to our knowledge).



(a) Routing Table

(b) Lookup Trie

(c) Leaf-Pushed Trie

Fig. 1. (a) A routing table —the * in the prefixes represent the don't care bits, (b) the corresponding forwarding trie, and (c) the corresponding trie with leaf pushing. A '–' represents a null pointer. In the trie without leaf pushing, each node has 2 fields: a prefix, and a pointer to an array of child nodes. In the leaf-pushed trie, each node has only one field, a prefix, *or* a pointer to an array of child nodes.

does not change significantly even when large, real-life update traces are applied. We also present upper bounds for the memory consumed per stage by our algorithm—hardware designers can utilize these bounds to design for the worst-case scenarios. Second, we develop multiple optimizations that leverage certain characteristics of the IPv4 address allocation process and the nature of the BGP routing protocol. Our optimizations are aimed at reducing as well as balancing the number of memory locations that are modified in each pipelined stage due to route updates.

The combined effect of our optimizations typically produces over a 2-fold improvement in the trie update overheads to the pipeline when tested over several core router tables and routing update traces. Even though there has been previous work on optimizing memory usage in routing tries [3–12], it has mostly focused on non-pipelined architectures. To our knowledge, this is the first attempt to develop routing trie construction and management techniques for pipelined ASIC architectures.

While route update rates can be around two orders of magnitude lower than packet forwarding rates, we have noticed in our analysis (using real traces) that a single route update (route add or withdrawal) can modify over a thousand memory locations in the pipelined routing trie. Therefore, unless route update operations are managed carefully, the resulting pipeline writes may prevent packet forwarding operations from keeping up with wire speeds. However, given that the *route update* rates are small enough, it is still possible to process each individual update in software and generate a sequence of optimized pipeline write operations for each update. The advantage of such a software-based scheme is that it is flexible and it keeps the forwarding hardware simple, thereby making it cost effective.

The rest of the paper is organized as follows. In the next section, we provide an overview of trie-based lookups, along with issues that arise when such lookups are pipelined. In Section III, we describe the experimental setup and the assumptions that we make about the hardware to abstract out the problem. Next, in Section IV, we list the characteristics of the IPv4 address allocation process and the BGP routing protocol that drive the optimizations presented in Sections V and VI. Finally, we summarize related work in Section VII and conclude in Section VIII.

II. Trie-Based IP Lookups in Pipelined Forwarding Engines

In this section, we describe some terminology for trie-based data structures and explain how routing tries can be implemented in pipelined forwarding engines.

A *trie* is essentially a tree that is used to store a set of routing prefixes for longest prefix matching. Every prefix in the set is represented by a node in the trie. Each trie node contains two fields: an IP prefix represented by the node (null if this prefix is not in the set) and a pointer to an array of child nodes (null if none). The packet lookup process starts from the root and works as follows. At each trie node v, b consecutive bits (called the *stride* of node v) from the destination IP address are used as an index to select which of the 2^b child nodes to traverse next. When a leaf node is reached, the last prefix seen along the path to the leaf is the longest matching prefix for the packet. Note that this implies that the prefix represented by a node n is determined by the path from the root node to n. Figure 1(b) shows a sample trie. In general, the stride at each trie node can be selected independently. A trie that uses the same stride for all the nodes in one level is called a *fixed-stride trie*; otherwise, it is a *variable-stride trie*. The trie in Figure 1(b) is a fixed-stride trie.

A. Pipelined Lookups Using Tries

Tries are a natural candidate for pipelined lookups; each trie level can be stored in a different pipeline stage. In a pipelined hardware architecture, each stage of the pipeline consists of its own fast memory (typically SRAMs) and some hardware to extract the appropriate bits from a packet's destination address (see Figure 2). These bits are concatenated with the lookup result from the previous stage to form an index into the memory for the current stage. A different packet can be processed independently in each stage of the pipeline. It is easy to see that if each packet traverses the pipeline once, the forwarding result for one packet can be output every memory access cycle.

Using an optimization called *leaf-pushing* [4], the trie memory as well as the bandwidth required between the SRAM and the logic can be halved. Here, prefixes at non-leaf nodes are pushed down to all the leaf nodes under it that do not already contain a more specific prefix. In this manner, each

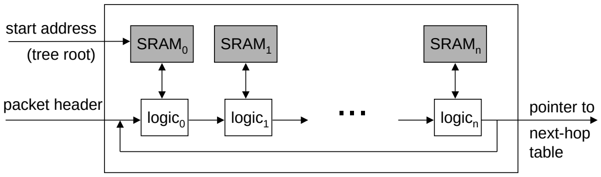


Fig. 2. A typical n-stage forwarding pipeline. In general, a packet header can travel through the pipeline multiple times.

node need only contain one field—a prefix pointer *or* a pointer to an array of child nodes. Thus each trie node can now fit into one word instead of two. In a leaf-pushed trie, the longest matching prefix is always found in the leaf at the end of the traversed path (see Figure 1(c)). For the rest of this paper, we only consider leaf-pushed tries.

B. *Issues in Pipelined Architectures*

Since the pipeline is typically used for both forwarding and classification, its memories are shared by multiple tables (such as IPv4 and IPv6 routing tables, as well as packet filtering tables and multicast routing tables for each input interface). Therefore, evenly distributing the memory requirement of each table across the pipeline memories simplifies the task of memory allocation to the different tables. It also reduces the likelihood of any one memory stage overflowing due to route/filter additions.

Updates to the forwarding table go through the same pipeline as the lookups. A single route update can cause several write messages to be sent through the pipeline. For example, the insertion of the route 1001* in Figure 1(c) will cause one write in the level 2 node (linking the new node to the trie), and 4 writes in the level 3 node (2 writes for 1001* and 2 writes for pushing down 100*).

These software-controlled write messages from one or more route updates are packed into special write packets and sent down the pipeline, similar to the reads performed during a lookup. We call each write packet a ***bubble***—each bubble consists of a sequence of (*stage*, *location*, *value*) triples, with at most 1 triple for each stage. The pipeline logic at each stage issues the appropriate write command to its associated memory. Minimizing the number of write bubbles introduced by route updates reduces the disruption to the lookup process. Finally, care should be taken to keep the trie in a consistent state between consecutive write bubbles, as route lookups may be interleaved with write bubbles.

III. Solving the Pipelined Architecture Problem

In order to address the issues described in the previous section, we developed a series of optimizations using a combination of simulation and analysis. To this end, we first developed a simulation model of the packet forwarding components in a typical (high-end) router linecard. We then used certain well-known characteristics of the IPv4 address allocation process and the BGP routing protocol to develop the optimizations for balancing the memory allocations and reducing the occurrence of “write bubbles” in the pipeline. Finally, we validated our findings by incorporating these optimizations in our simulator and running a set of route update traces through the simulator. We now briefly describe the simulator for the packet forwarding engine and the assumptions that we made.

A. *Forwarding Engine Model*

The distributed router architecture described in this section is similar to that of commercially available core routers [13], [14]: the router has a central processor that processes BGP route updates from neighboring routers and communicates the resulting forwarding table changes to the individual linecards. Each linecard has a local processor that controls the pipelined forwarding engine on the linecard. Using a software shadow of the pipelined routing trie, the local processor computes the changes to be made to each stage of the forwarding pipeline for each route update. It also typically performs all the memory management for the pipeline memories. In this work, we focus on optimizations that will minimize the cost of route updates, and not on memory management issues.[2]

Our simulation model consists of three components—first, we have the trie component that constructs and updates the routing trie (see Figure 3). It processes one route update at a time and generates the corresponding writes to the pipeline memories. The second component is the packing component that packs writes from a batch of consecutive route updates into write bubbles that are sent down the pipeline. When a new subtree is added to the trie (due to a route add), the pipeline write that adds the root of the subtree is tagged by the trie component. The packing component ensures that this write is not packed into a write bubble before any of the writes to the new subtree (to prevent any dangling pointers in the pipelined trie). Finally, we have a pipeline component that actually simulates the traversal of these write bubbles through a multi-stage pipeline.

B. *Assumptions*

We now describe the assumptions about the forwarding engine setup that we made when designing the simulation model (which therefore affect the nature of the optimizations that we have developed).

- The initial trie construction takes as input a snapshot of the entire table. After this construction phase, it is only updated *incrementally* for the several million updates in our datasets.
- Bubbles are processed by the pipeline in the same order as they are generated by the packing component. A bubble is not interrupted in its progression from the first to the last stage of the pipeline. However, consecutive bubbles may be interspersed with lookups. Therefore, the trie structure should always be *consistent between consecutive write bubbles*.
- Only tries with *fixed strides* are considered. Although variable-stride tries could be more memory efficient, they are difficult to maintain during incremental updates. Since

[2] In particular, we do not address how memory blocks are allocated for trie construction and how the free lists are maintained.

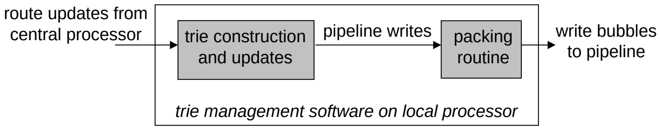

route updates from central processor
trie construction and updates
pipeline writes
packing routine
write bubbles to pipeline
trie management software on local processor

Fig. 3. The software system on the linecard that constructs and maintains the forwarding trie.

site	date	table size	#updates	Avg upd	Max upd
rrc03	4/1/01	102,424	3.00M	19.20	8,451
rrc04	5/2/01	104,181	2.74M	17.75	52,012

TABLE I

The two datasets used to corroborate the routing table/route updates characteristics. rrc03 is a peering point in Amsterdam, and rrc04 is a peering point in Geneva. “Avg upd” (or “Max upd”) are the average (or maximum) number of updates that have the same timestamp (granularity—1 second).

each node can have a different stride, there is no straightforward way to determine the strides for trie nodes that are created when new prefixes are added.

- We focus on *leaf-pushed tries*. Although updates to leaf-pushed tries can result in more pipeline writes, leaf pushing allows for a higher pipeline throughput and more efficient use of the pipeline memory. All our optimizations, however, are also *applicable to non-leaf pushed tries*.
- Writes to different pipeline stages can be combined into a single write bubble — each bubble can contain at most one write to each stage of the pipeline. We use the *number of bubbles* created as a measure of how disruptive incremental updates are to the search process. Note that it is possible to have more than one write to each pipeline stage in a single bubble. For example, this would be the case when the pipeline memory can service multiple back-to-back write requests. Our assumptions (of at most one write per stage) therefore provide an upper bound on the number of bubbles generated (worst-case).
- The packing component is permitted to pack pipeline writes from *multiple route updates* into a single write bubble, as long as the route updates arrive with the same timestamp (granularity—one second). To further limit the delay in updating the pipelined trie, we combine writes from batches of at most 100 such concurrent route updates.
- Our experiments focus on *IPv4 lookups*, since no extensive data is currently available for IPv6 tables or updates. We focus on a pipeline with 8 stages, and assume that only one pass is required through the pipeline to look up a IPv4 packet. Section VI also shows results for pipelines with fewer stages.
- The next hop information is stored in a separate Next Hop table that is distinct from the pipelined trie.
- Memory management is done by a separate component. In particular, we assume the existence of `malloc` and `free`-like primitives (see, for example, [3] for work in this area).

IV. Routing Table and Route Update Characteristics

In this section, we describe certain characteristics of the IPv4 address allocation process and the mechanics of the BGP routing protocol. These characteristics affect the structure of the routing tables in the Internet core as well as the route update process, and have been described elsewhere in the literature. We use these well-known characteristics (which we also corroborated using the sample traces in Table I) to drive the optimizations for pipelined lookup architectures.

The IPv4 address allocation process and its consequences on BGP routing tables were described in a recent study [15]. The growth in the routing table sizes in the pre-CIDR era was mainly due to the growth in the Class C addresses (i.e., 24 bit prefixes), while very few Class A addresses (i.e., 8 bit prefixes) were allocated. Even though the introduction of CIDR caused more addresses in the 19-20 bit range to be allocated, the 24 bit prefixes still continue to dominate the BGP routing tables of today. This kind of uneven growth has resulted in the following characteristics of the routing tables.

O–I: A majority of the prefixes in the routing tables of today are 24 bit prefixes—consequently most routing updates affect 24 bit prefixes. For example, 57.7% of the rrc03 prefixes and 58.8% of the rrc04 prefixes were 24-bit prefixes. 64.2% of rrc03 route updates and 61.6% of rrc04 route updates were to 24-bit prefixes.

O–II: The number of very small ($\leq$ 8 bit) prefixes is very low, and very few updates affect them. For example, about 0.02% of the prefixes in the rrc03 and rrc04 prefixes were less then 8 bits long, and 0.04% of the rrc03 updates and 0.03% of the rrc04 updates were to such prefixes. However, since a short route is typically replicated a number of times in a trie[3], each update to it may result in modifications to a large number of memory locations.

Route aggregation considerations have also influenced the nature of routing updates. Historically, IPv4 addresses have been allocated in a hierarchical fashion such that routing advertisements can be aggregated. In particular, the address blocks allocated to an ISP customer are sub-blocks of the address block allocated to the ISP. This has two important consequences.

O–III: Prefixes corresponding to the customers of a given ISP are typically neighboring 24-bit prefixes. Hence prefixes close together and differing in only a few low order bits (but with possibly different next hops) often fully populate a range covered by a single, shorter prefix.

O–IV: A link failure (recovery) in an ISP network disconnects (reconnects) some or all of its customer networks (represented by neighboring prefixes in the routing trie). In turn, this can cause updates to the corresponding neighboring routes to occur simultaneously. For example, in the rrc03 and rrc04 traces, we observed multiple instances of routes in an entire subtree getting added or withdrawn at the same time (in the same second).

[3] This is because tries are often designed to have a stride of 12-16 bits in the very first level.

label	site	location	start date	start time	table size	hops	# updates	Avg upd	Max upd
rrc01	rrc01	London	Oct 1, 2001	2:30am	103,555	37	4,719,521	12.52	30,060
rrc03-a	rrc03	Amsterdam	Jan 31, 2001	7:00pm	98,974	74	3,769,903	16.70	8,867
rrc03-b	rrc03	Amsterdam	Oct 1, 2001	12:00am	108,267	80	4,722,493	12.45	30,060
rrc04	rrc04	Geneva	Nov 1, 2001	3:30am	109,600	29	3,555,839	13.80	36,326
me-a	mae-east	Virginia	Oct 29, 1999	12:00am	50,374	59	3,685,469	10.63	8,509
me-b	mae-east	Virginia	July 1, 2000	12:00am	46,827	59	4,350,898	09.66	10,301
mw	mae-west	San Jose	Aug 16, 2000	12:00am	46,732	53	2,856,116	10.71	14,024

TABLE II

THE 7 DATASETS USED IN ALL THE EXPERIMENTS IN SECTIONS V AND VI. "HOPS" IS THE NUMBER OF UNIQUE NEXT HOPS IN THE TABLE; THIS IS REPRESENTATIVE OF THE NUMBER OF PEERS AT THE PEERING POINT. "AVG UPD" (OR "MAX UPD") ARE THE AVERAGE (OR MAXIMUM) NUMBER OF UPDATES THAT HAVE THE SAME TIMESTAMP (GRANULARITY—1SECOND).

O–V: Finally, recent studies of BGP dynamics [16], [17] indicate the following. First, the proportion of route updates corresponding to network failure and recovery is fairly high: about 40% of all updates are route withdrawals and additions in response to network failure and recovery [17]. Second, once a network failure occurs, the mean time to repair is of the order of several minutes [16]. Thus, a large proportion of routes that are withdrawn get added back a few minutes later. For example, our analysis of the rrc03 and rrc04 traces indicate that about 80% of the routes that are withdrawn get added back within 20 minutes.

Our optimizations for constructing and maintaining the pipelined lookup trie are based on the above observations. To test the effectiveness of our optimizations, we use seven datasets that are different from the two datasets described in Table I. These datasets will be referenced in the following sections by the labels listed in Table II. The rrc∗∗ datasets were collected from the Routing Information Service, while the Mae-east and Mae-west databases were obtained from the Internet Performance Measurement and Analysis (IPMA) project.[4] Note that the datasets have widely distributed spatial (location of collection) and temporal (time of collection) characteristics—this ensures that our optimizations are not specific to a particular router interface or a specific time interval.

The update traces used in our experiments were collected starting from the time of the table snapshot. Each BGP update, which has a timestamp granularity of one second, is either a route addition or a route withdrawal. Updates from several consecutive days were combined, and some were filtered out as follows. Adds to prefixes already in the table can be treated as a change of the next-hop entry, and simply involve rewriting an entry in the next-hop table (thus not affecting the forwarding trie). Such adds were filtered out — similarly, withdraws for prefixes not currently in the table were also filtered out. In our experiments, the filtered update sequence was applied incrementally to the initial routing table snapshot.

V. MEMORY OPTIMIZATIONS

We now present a trie construction algorithm that takes as input the number of stages in the pipeline and a set of routing table prefixes, and finds the trie that minimizes the size of the largest trie level (pipeline stage). If there are multiple such tries, the algorithm finds the most compact (least total size) trie among them. Such an algorithm makes it easier to pack multiple different protocol tables into the set of available pipelined memories, and to avoid memory overflows as routing tables grow. We also provide upper bounds on the worst-case performance of this algorithm—this enables hardware designers to decide how big each pipeline stage should be.

A. Designing non-pipelined tries

Tries present a trade-off between space (memory requirement) and packet lookup time (number of trie levels, assuming one lookup operation per level). Large strides reduce the number of levels (and hence, the lookup time), but may cause a large amount of replication of prefixes (for example, see Figure 4(a)).

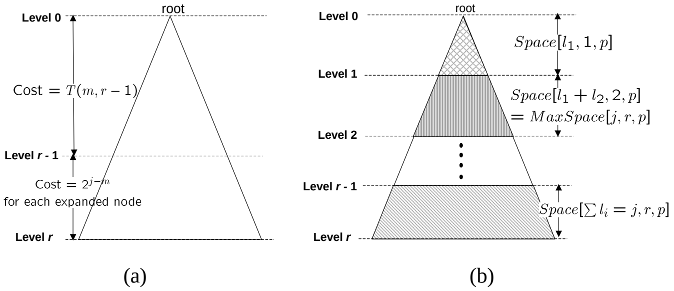


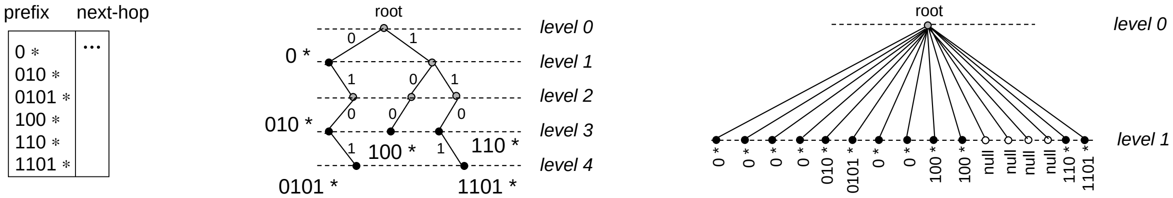
Fig. 5. Optimizing Memory (a) using the controlled prefix expansion algorithm by Srinivasan and Varghese, and (b) using our MinMax algorithm (see Section V-C). Here, the second level occupies the most memory, hence $MaxSpace[j, r, p] = Space[l_1 + l_2, 2, p]$ for the partition p shown in the figure.

To balance the space-time tradeoff in trie construction, Srinivasan and Varghese [4] use ***controlled prefix expansion*** to construct memory-efficient tries for the set of prefixes in a routing table. Given the maximum number of memory accesses allowed for looking up any IP address (i.e., the maximum number of trie levels), they use dynamic programming to find the fixed-stride trie with the minimum total memory requirement.[5]

The problem of constructing a fixed-stride trie reduces to finding the stride-size at each level, that is, finding the bit positions at which to terminate each level. The dynamic

[4] http://www.merit.edu/ipma

[5] We assume no path compression is used. There is also a dynamic programming algorithm for finding the smallest variable-stride trie presented in [4] that is not directly relevant to our work.



(a) (b) (c)

Fig. 4. (a) A sample routing table, (b) the corresponding 1-bit trie, and (c) the corresponding 4-bit trie. The dotted lines show the nodes at each level in the tries; of these, only the black nodes contain a prefix. The 4-bit trie has a smaller depth (1 vs 4 memory accesses) but a larger number of nodes compared to the 1-bit trie.

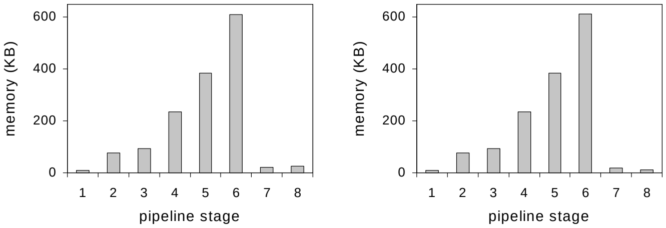
programming technique in controlled prefix expansion works as follows. First, a 1-bit auxiliary trie is constructed from all the prefixes (e.g., see Figure 4(b)). Let $nodes(i)$ be the number of nodes in the 1-bit trie at level i. If we terminate some trie level at bit position i and the next trie level at some bit position $j > i$, then each node in $nodes(i+1)$ gets expanded out to 2^{j-i} nodes in the multi-bit trie (see [4] for details). Let $T[j,r]$ be the optimal memory requirement (in terms of the number of trie nodes) for covering bit positions 0 through j using r trie levels (assuming that the leftmost bit position is 0). Then $T[j,r]$ can be computed using dynamic programming as (see also Figure 5(a)):

$$\begin{aligned} T[j,r] \;&=\; \min_{m\in\{r-1,\ldots,j-1\}} (T[m,r-1] + \\ &\qquad nodes(m+1)\times 2^{j-m}) \quad (1) \\ T[j,1] \;&=\; 2^{j+1} \quad (2) \end{aligned}$$

Here, we choose to terminate the $(r-1)^{th}$ trie level at bit position m, such that it minimizes the total memory requirement. For prefixes with at most W bits, we need to compute $T[W-1,k]$, where k is the number of levels in the trie being constructed. This algorithm takes $O(k \times W^2)$ time; for IPv4, $W = 32$.

B. Implications for memory usage and update performance

The controlled prefix expansion algorithm finds the fixed-stride trie with the minimum *total memory*. It can easily be applied to a pipelined lookup architecture by fixing the number of trie levels to be the number of pipeline stages (or some multiple of it). However, the algorithm does not attempt to distribute the memory equally across the different pipeline stages. As a consequence, some stages may be heavily loaded, while others may be very sparse. Figure 6 shows the amount of memory allocated in each stage of an 8-stage pipeline for the two routing tables rrc03 and rrc04. Each fixed-stride trie was constructed with controlled prefix expansion and uses leaf pushing. The memory allocations are highly variable across the different stages. In particular, the stage that contains 24-bit prefixes has the highest memory requirement. Besides increasing the chance of memory overflows in this stage, this imbalance can negatively impact the update performance: the overloaded stage contains the 24-bit prefixes and therefore typically gets many more writes than the other stages. This makes it difficult to pack the pipeline writes into a small number of bubbles. Note that this follows directly from observation **O–I** in Section IV.



(a) rrc03 (b) rrc04

Fig. 6. Memory allocation for the forwarding trie in different stages of the pipeline using controlled prefix expansion for the two tables: (a) rrc03, and (b) rrc04.

C. A New Algorithm for Pipelined Architectures

We have developed an algorithm (based on controlled prefix expansion) that attempts to evenly allocate memory across the different stages. Our algorithm assumes that each pipeline stage contains exactly one level in the lookup trie, and constructs a trie that satisfies the following constraints:

- Each level in the fixed-stride trie must fit in a single pipeline stage.
- The maximum memory allocated to a stage (over all stages) is *minimized*. This ensures that the memory allocation is balanced across all the pipelined stages.
- The total memory used is minimized subject to the first two constraints (we explain why this is important later in this section).

More formally, as before, let $T[j,r]$ denote the total memory required for covering bit positions 0 through j using r trie levels, when the above conditions are satisfied. Furthermore, let P denote the size of each pipeline stage. Then, the first and the third constraints are satisfied by the following equations:

$$\begin{aligned} T[j,r] \;&=\; \min_{m\in S(j,r)} (T(m,r-1)+ \\ &\qquad nodes(m+1)\times 2^{j-m}) \quad (3) \\ T[j,1] \;&=\; 2^{j+1} \quad (4) \end{aligned}$$

where

$$\begin{aligned} S(j,r) &= \{m | r-1 \le m \le j-1 \text{ and} \\ &\qquad nodes(m+1) \times 2^{j-m} \le P\} \end{aligned}$$

To satisfy the second constraint, we first introduce some additional notation. For a partition p of bit positions 0 through j into r levels in the multi-bit trie, let $Space[j,r,p]$ denote the memory allocated to the rth level in the multi-bit trie. In other words, we have:

$$Space[j,r,p] = nodes(m+1) \times 2^{j-m} \tag{5}$$

where bit positions 0 through m are covered by $r-1$ levels in the trie. (Note that this implies that the rth level in the trie covers bit positions $m+1$ through j). We then define $MaxSpace[j,r,p]$ as the maximum memory allocated to any trie level when partition p is used to split bit positions 0 through j among r levels in the multi-bit trie (see Figure 5(b)). More formally, we have:

$$MaxSpace[j,r,p] = \max_{1 \le m \le r} Space[\sum_{i=1}^{m} l_i, m, p]$$

where l_i denotes the stride-size of the ith level in the trie, and $\sum_{i=1}^{r} l_i = j$. Now let $MinMaxSpace[j,r]$ be the minimum value of $MaxSpace[j,r,p]$ for all possible partitions p of bit positions 0 through j into r levels. Then,

$$MinMaxSpace[j,r] = \min_{p \in \text{Part}} MaxSpace[j,r,p]$$

where Part is the set of all possible partitions. Then, in addition to the equations (3) and (4), the following equation must also be satisfied by the variable m:

$$\begin{aligned} MinMaxSpace[j,r] &= \\ &\min_{m \in S(j,r)} (\max(nodes(m+1) \times 2^{j-m}, \\ &\qquad MinMaxSpace[m, r-1])) \quad (6) \\ MinMaxSpace[j,1] &= 2^{j+1} \end{aligned}$$

Equation (6) selects the bit position m to terminate the $(r-1)^{th}$ level of the trie such that it minimizes the size of the largest trie level. Here the largest level (in terms of memory requirement) is the bigger of the last (r^{th}) level, and the largest level of the first $r-1$ levels.

We give equation (6) precedence over equation (3) when choosing m. When multiple values of m yield the same value of $MinMaxSpace[j,r]$, equation (3) is used to choose between these values of m. In other words, our primary goal is to reduce the maximum memory allocation across the pipeline stages, and the secondary goal is to minimize the total memory allocation. We found that it was important to maintain this secondary goal of memory efficiency to produce tries with low update overheads. A memory-efficient trie typically has smaller strides and hence less replication of routes in the trie; a lower degree of route replication results in fewer trie nodes that need to be modified for a given route update.

For a set of prefixes where the longest prefix length is W bits, and the maximum number of lookups is k (i.e., k trie levels), this algorithm takes $O(k^2 \times W)$ operations (same as the controlled prefix expansion algorithm [4]). We refer to this algorithm as the MinMax algorithm.

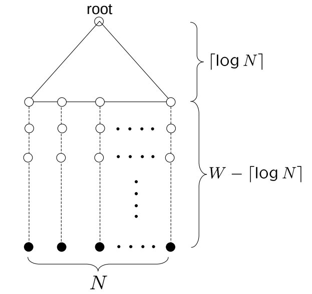


Fig. 7. The 1-bit auxiliary trie corresponding to the worst possible input of N prefixes for the MinMax algorithm; the longest prefix length is W bits. There is maximum fan-out (i.e., each node has two descendants) till we reach level $\lceil \log N \rceil$. Thereafter, every node has a single descendant till level W is reached—all the N prefixes are of length W and lie in level W of the 1-bit trie.

D. Worst Case Memory Bound

The MinMax algorithm is intended for use in hardware design — an important measure of performance in such cases is the worst case memory usage (in addition to lookup and update times). In other words, given a k-stage pipeline, and a prefix table of size N with a maximum prefix length of W, we would like to compute the worst-case memory size for a pipeline stage. Let the worst possible input of N prefixes to the MinMax algorithm be WC, and let max_{WC} be the size of the maximum pipeline stage as computed by the MinMax Algorithm.

We shall now construct the 1-bit auxiliary trie that corresponds to the input WC and prove that WC is indeed the worst possible input. The construction of the 1-bit auxiliary trie is shown in Figure 7. The trie that is constructed has two halves. In the upper half, which consists of levels 1 through $\lceil \log N \rceil$, each node has two children — in other words, there is maximum fan out in the upper half. In the lower half, which spans levels $\lceil \log N \rceil + 1$ through W, each node has exactly 1 child. Note that each prefix represented by this trie has length W, which is the maximum allowed.

In order to prove that WC is the worst possible input, we introduce some more notation. Let D denote any input of N prefixes to the MinMax algorithm. We denote by max_D, the size of the maximum pipeline stage as computed by the MinMax algorithm for the input D. Furthermore, let $nodes_{WC}(m)$ be the number of nodes in level m of the 1-bit auxiliary trie for input WC. The quantity $nodes_D(m)$ is similarly defined. We now prove the following lemma.

Lemma 1: For any level m in the auxiliary trie, $nodes_D(m) \le nodes_{WC}(m)$.

Proof: We first observe that the auxiliary trie for input WC has the maximum possible fan out for each node in the first $\lceil \log N \rceil$ levels. Therefore, at any level $m \le \lceil \log N \rceil$, the number of nodes in the auxiliary trie for input WC must be the maximum possible. Now consider any level $m > \lceil \log N \rceil$. The number of nodes at level m in the auxiliary trie for input WC is N. Suppose there is some input consisting of N prefixes where the auxiliary trie has $L > N$ nodes at level m. If we traverse down this trie, each node at level m must lead to a leaf node which contains a prefix. This implies that the auxiliary trie has $L > N$ prefixes, which is a contradiction. ■

Now let p_D be the partition produced by the MinMax algorithm for the input D, and let p_{WC} be the partition produced for the input WC. Consider the 1-bit trie for input D with the partition p_{WC} superposed on it. Let $max_D(WC)$ be the size of the largest pipeline stage in this case. Note that max_D is the largest pipeline stage corresponding to the partition p_D which is optimal for the input D. Therefore, we conclude that

$$max_D \leq max_D(WC) \tag{7}$$

We now consider the levels in the 1-bit auxiliary trie (for input D) that are spanned by the pipeline stage with size $max_D(WC)$. Let the start level be $max(m)$ and the number of levels be x. Then, we have (using equation (5)),

$$max_D(WC) = nodes_D(max(m)) \times 2^x \tag{8}$$

Using lemma 1 and equation (8), we assert that

$$\begin{aligned} max_D(WC) &= nodes_D(max(m)) \times 2^x \\ &\leq nodes_{WC}(max(m)) \times 2^x \\ &\leq max_{WC} \end{aligned} \tag{9}$$

Using (7) and (9), we conclude that

$$max_D \leq max_{WC} \tag{10}$$

This leads to the following memory bound.

Theorem 1: For any given set of N prefixes of maximum length W, the maximum memory per pipeline stage required to build a k-level trie using the MinMax algorithm is bounded by $2^{\lceil \frac{W+(k-1)\lceil \log N \rceil}{k} \rceil}$ trie nodes.

Proof: Let N' be the smallest power of 2 that is greater than or equal to N. Then $\log N' = \lceil \log N \rceil$. We will compute the memory bound for $N' \geq N$ prefixes; the same upper bound should hold for N prefixes.

Using equation (10), we conclude that the worst possible input with N' prefixes for the MinMax algorithm is shown by the construction in Figure 7. Now, we need to compute the bit positions at which each of the k levels will be terminated by the MinMax algorithm.

We can show that there is at least one level in the resulting trie with over N' nodes in it. Therefore, there is no benefit from terminating the first level at less than $\log N'$ bits. There may however, be some benefit from terminating the first level at a bit value *greater* than $\log N'$, so that the sizes of the lower levels can be reduced. Let the optimal bit value at which the first level terminates be $\log N' + x$ (where $x \geq 0$ is an integer). Then the first level has $2^{\log N' + x}$ nodes.

Now, the worst case 1-bit trie (from Figure 7) has exactly N' nodes in every level at bit position greater than $\log N'$. Therefore, MinMax will simply attempt to split them at equal intervals to create the remaining $k - 1$ levels. Thus, the remaining $(W - \log N' - x)$ bit positions are split evenly into $k - 1$ levels. The largest of these levels will cover $\lceil \frac{W - \log N' - x}{k-1} \rceil$ bits. This level will contain $N' \cdot 2^{\lceil \frac{W - \log N' - x}{k-1} \rceil}$ nodes.

Table	rrc01	rrc03a	rrc03b	rrc04	me-b	me-a	mw
Overhead (%)	17.2	17.6	16.4	16.5	16.5	17.8	13.8
Max reduc. (%)	42.2	41.7	42.0	41.8	44.3	44.5	44.2

TABLE III

THE ADDITIONAL (TOTAL) MEMORY OVERHEAD OF THE MINMAX ALGORITHM, AND THE REDUCTION IN THE MEMORY REQUIREMENT OF THE MOST POPULATED PIPELINE STAGE. BOTH VALUES ARE RELATIVE TO THE CORRESPONDING VALUES FOR CONTROLLED PREFIX EXPANSION.

MinMax will select the value of x such that the size of the first partition equals the largest of the lower partitions. Therefore,

$$\begin{aligned} 2^{\log N' + x} &= N' \cdot 2^{\lceil \frac{W - \log N' - x}{k-1} \rceil} \\ &= 2^{\log N' + \lceil \frac{W - \log N' - x}{k-1} \rceil}, \quad \text{or,} \\ \log N' + x &= \log N' + \lceil \frac{W - \log N' - x}{k-1} \rceil \end{aligned}$$

Solving for x gives us

$$x = \lceil \frac{W - \log N'}{k} \rceil \tag{11}$$

Therefore, the number of nodes in the largest partition is bounded by

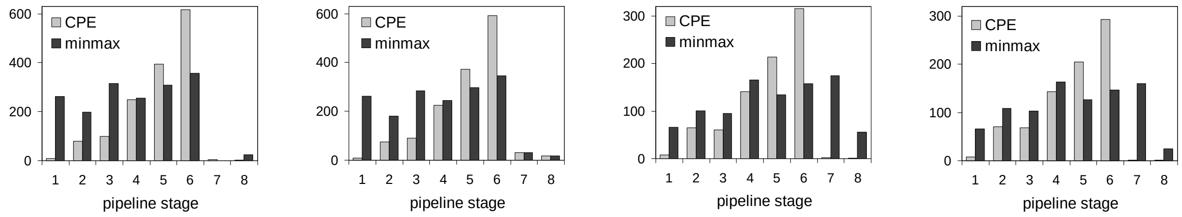
$$2^{\lceil \frac{W+(k-1)\log N'}{k} \rceil} = 2^{\lceil \frac{W+(k-1)\lceil \log N \rceil}{k} \rceil} \tag{12}$$

■

For N = 2^{20} (1M) prefixes, $k = 8$, and $W = 32$ this bound amounts to 2^{22} (4M) trie nodes per stage. Assuming a pointer size of 22 bits (to address each of the 4M entries in a stage), we get a maximum memory requirement of 11MB per stage. For N = 128K (a typical size for a current core routing table), a similar calculation yields 1.19MB per stage.

Performance. The performance of the MinMax algorithm is shown in Figure 8; it reduces the maximum memory allocation across the pipeline stages (by over 40%) at the cost of a slightly higher (13–18%) total memory overhead compared to controlled prefix expansion (see Table III). We also point out that in each of the graphs in Figure 8, some of the levels show disproportionately low memory usage even after the MinMax algorithm is applied (for example, levels 7 and 8 in Figure 8(a)). In each of these cases, the levels in question are the ones that are assigned to bit positions 24 to 31 (recall that bit positions are numbered from 0). Since there are very few prefixes of length more than 24, the number of prefixes terminating in these levels is low, hence the low memory usage. The MinMax algorithm assigns one level solely to the 24-bit prefixes (which typically form largest level in the trie). Therefore, it cannot further balance out the trie levels by combining the 24-bit prefixes with the sparsely populated prefixes of length greater than 24.

Finally, we note that instead of minimizing the maximum, minimizing some other metrics could also balance out the memory allocation across the multiple stages. We have experimented with three other strategies, namely, (a) minimizing the standard deviation of all the $Space[j, r, p]$'s, (b) minimizing



(a) rrc01 (b) rrc03-a (c) me-a (d) mw

Fig. 8. Memory allocation (in KB) in the different pipeline stages using the MinMax algorithm for trie construction, compared to using controlled prefix expansion.

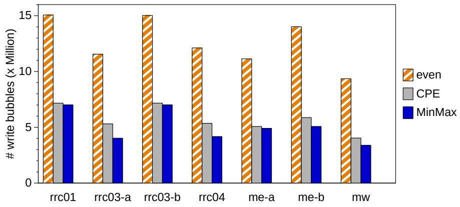


Fig. 9. Number of write bubbles sent to the pipeline when different trie construction algorithms are used: “even” denotes using equal (4-bit) strides in each of the 8 levels, “CPE” denotes using controlled prefix expansion, and “MinMax” denotes using MinMax.

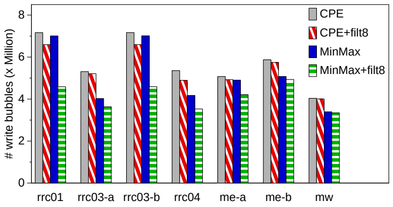


Fig. 10. Reduction in the number of write bubbles when short prefixes (≤ 8 bits) are filtered out and stored in a separate 256-entry table. Bubbles include the cost of updating this separate table. CPE is controlled prefix expansion, and MinMax is our new tree building algorithm. “filt8” denotes the runs when the short prefixes were stored separately.

the sum of squares of all the $Space[j, r, p]$’s, and (c) minimizing the difference between the maximum $Space[j, r, p]$ and the minimum $Space[j, r, p]$. However, the quality of the results for the MinMax algorithm proves to be just as good or better than the other (computationally more intensive) algorithms. For the rest of the paper, we shall use the MinMax algorithm for calculating stride lengths.

As shown in Figure 9, both controlled prefix expansion (CPE) and MinMax are fairly effective in reducing the number of write bubbles generated by the packing component, when compared to the baseline case of using a trie with equal strides (4 bits) at each level. We also point out that the CPE algorithm was not explicitly designed to solve the pipelined architecture problem. However, for the rest of the paper, we omit the baseline case (of tries with equal strides) and only present comparison figures for our optimizations and the CPE algorithm since it is the most competitive algorithm we could find.

VI. Reducing Write Bubbles

Minimizing the disruption to the fast-path lookup pipeline by route updates is an important goal of this work. We now describe a series of optimizations aimed at reducing the number of write bubbles that are sent to the pipeline when routes are added to or withdrawn from the forwarding trie. These optimizations are implemented in either the trie component or the packing component from Figure 3. In some cases, the packing component applies an optimization based on writes that are specially tagged by the trie component. As we describe each optimization, we show its benefits when applied incrementally along with the previous optimizations.

A. *Separating out updates to short routes*

The number of short routes (≤ 8 bits) in all the tables is very small (**O–II**, Section IV). However, even a small number of updates to these routes can cause a big disruption to the pipeline. For example, if the trie root has a stride of 16, the addition of a 7-bit route can cause up to $2^{16-7} = 512$ writes to the first stage of the pipeline. These writes cannot be packed into a smaller number of bubbles since they all target the same pipeline stage.[6] Since the trie construction algorithms do not take into account such update effects when determining the strides in the trie, we instead suggest storing all short routes, of up to 8 bits length, in a separate table with $2^8 = 256$ entries. Assuming 32 bits per entry, this only requires an additional 1KB of fast memory. The IP lookup process now searches the pipeline first. If no route is found, an additional lookup is performed in this table using the first 8 bits of the destination address. This lookup can also be pipelined similar to the trie lookups. Figure 10 shows the benefit of using this simple optimization. The figure includes the cost of writing to the new table. For example, an update to a 7-bit route now causes 2 writes to the new table (which cannot be packed into the same bubble). The benefit of this simple optimization ranges between 1.6%–35% when using MinMax to construct the trie. The routing tables for rrc01 and rrc03b benefit more than the rest because they have a larger number of updates to very short routes, including 1-bit routes.

An alternative to adding a separate table for short prefixes would be to simply add another stage (with a stride of 8 bits) at the *beginning* of the pipeline. However, with such a pipeline

[6] In fact, we have even observed the addition and deletion of a 1-bit route in some of our traces.

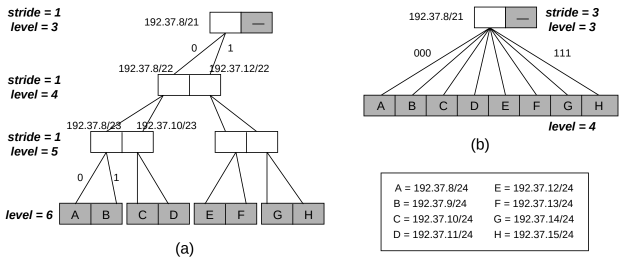
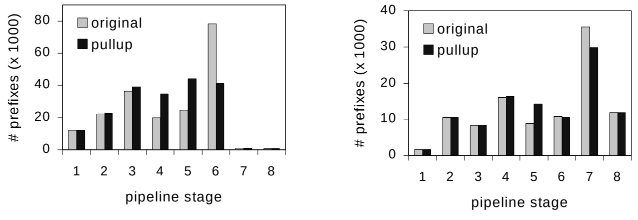


Fig. 11. A subtree (a) before, and (b) after node pullup has been performed. The 24-bit prefixes A, B, . . ., H can be pulled up 2 levels. The number of trie nodes can at best decrease (in this case from 16 to 10) after node pullups.



(a) rrc01

(b) me-a

Fig. 12. Number of prefixes in each level of the pipelined trie before and after node pullup. For both the datasets, the tallest bar in the original trie represents the stage with the 24-bit prefixes.

stage, many copies of the short prefixes would be pushed down into lower parts of the trie (due to leaf pushing) — adding a separate table at the *end* of the pipeline avoids this overhead.

B. Node pullups

In fixed-stride tries, all 24-bit prefixes lie in a single level of the trie. Since most updates are to 24-bit prefixes (**O–I**, Section IV), a large fraction of the pipeline writes are directed to that level, which resides in a single stage of the pipeline. This makes it harder to pack the writes efficiently into bubbles. Node pullups are an optimization aimed at spreading out the 24-bit prefixes in the trie. Given that there are many groups of neighboring 24-bit prefixes in the trie (**O–III**, Section IV), we can move entire such groups above the level that contains the 24-bit prefixes. This pullup can be performed by increasing the stride of the node that is the lowest common ancestor of all the neighboring prefixes in the group. Let l be the level that contains the 24-bit prefixes. Consider a node in a level k above level l; say k terminates at bit position n (where $n < 24$). For some node in level k, if all of the 2^{24-n} possible 24-bit prefixes that can be descendants of this node are present in the trie, we pull all of them up into level k. The stride of the parent of the pulled-up node is increased by $24 - n$. We start examining nodes to pull up in a top-down manner, so that the 24-bit prefixes are pulled as far up as possible.

The node pullup optimization ensures that the memory requirement of the transformed trie can possibly reduce, but not increase (see Figure 11). Thus, the MinMax algorithm constructs a strictly fixed-stride trie; node pullups subsequently modify the strides of some nodes in a controlled manner. This optimization is similar to the Level Compression scheme described in [11]. However, the motivation here is different (we use it to reduce update overheads) and we have also developed a modification to enhance the performance of this optimization (see the next paragraph).

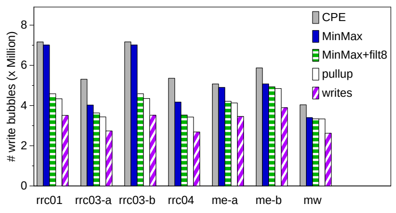


Fig. 13. Reduction in the number of write bubbles after node pullups have been performed (labeled "pullup"). "writes" shows the further reduction in write bubbles when excess writes are eliminated. For both these optimizations, the trie was constructed using the MinMax algorithm, and shorter prefixes were filtered out.

State Tries. Figure 12 shows the distribution of prefixes in different levels of an 8-level trie before and after node pullups have been performed for three of the datasets. Here we used the MinMax algorithm to construct the trie. Node pullups are successful in spreading out the prefixes across the pipeline stages. They also helped reduce the size of the largest pipeline stage by an average of 6.5% compared to MinMax. However, simply using the node pullup optimization is not sufficient. The pullup information (in the form of a changed stride length) is stored in the node where the pullup has occurred.[7] Therefore, if the node itself is deleted, and then re-inserted (due to a route withdrawal, followed by an insertion), this information cannot be reconstructed. Instead, the only information available is the trie level information that has been calculated by the MinMax algorithm. To remedy this shortcoming, we use a *state trie* in software when pullups are applied. The state trie stores the pullup information at each node. When there is a deletion followed by an insertion, the stride size of the inserted node is obtained from the corresponding node in the state trie.

Figure 13 shows the benefit of node pullups (with a state trie) for reducing the total number of write bubbles. The benefits were lower than we expected: updates to neighboring routes often appear together (**O–IV**, Section IV), and all the neighboring routes are typically in the same level, whether or not they have been pulled up. This makes it difficult to pack the resulting writes to that level into write bubbles.

C. Eliminating excess writes

Since neighboring routes are often added in the same timestep (**O–IV**, Section IV), the same trie node can be overwritten multiple times before all the nodes are added. We eliminate these extra writes (by eliminating all except the last write to the same trie node). At the same time, we take care not to create any dangling pointers between consecutive write bubbles. For example (see Figure 14), when neighboring 24-bit routes A, B, C, and D are added in the same timestamp,

[7] 5 bits are sufficient to represent strides of up to 32 bits; these 5 bits can be easily fit into the single 32-bit word that represents a trie node. Assuming 1 bit is used to flag leaf nodes, this leaves 26 bits for addressing, which implies about 64 million locations can be addressed—much more than the size of a pipeline stage.

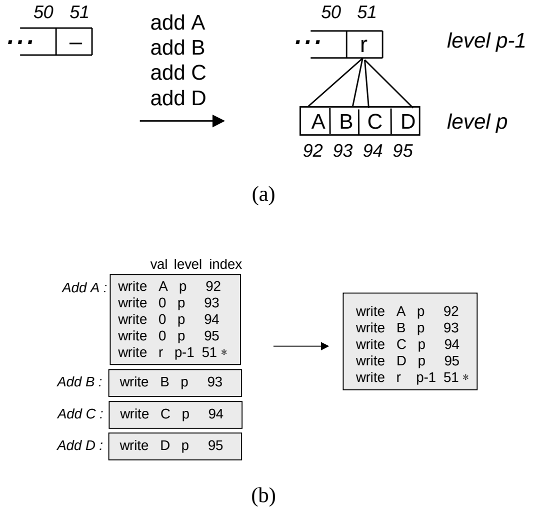


Fig. 14. (a) The portion of the trie before and after four neighboring prefixes A, B, C, and D are added. The numbers (indexes) denote the memory locations in the pipeline stage, and the value 0 denotes a null pointer. (b) Three excess writes (and therefore 3 excess write bubbles) are eliminated by the packing component; the write that adds the root of the new subtree is tagged ($*$) by the trie component, and all trie nodes under it must be written at least once before the tagged write is sent to the pipeline.

the first route (say A) may cause all four new nodes to be created. A pointer to A will be written in one node and a null pointer will be written to its three neighboring nodes (a total of 4 write bubbles). When B, C, and D are added, pointers to them are written in these neighboring nodes. Thus, the first of the two writes to each neighboring node (B, C, and D) can be eliminated.

Note that care must be taken to ensure that the trie is not left in an inconsistent state. Consider the case when the pointer to the node in level p is written before the new node is written in. In this case, if a search operation is inserted in between the two write operations, the search operation may return erroneous results if it tries to access the *dangling pointer* at level $p-1$. To eliminate this kind of update dependencies, the trie update algorithm tags writes that link a whole new subtree to the existing trie. The packing component then makes sure that these tagged writes are executed *after* the subtree that they point to has been written in.

Excess writes can also be eliminated when neighboring routes are withdrawn. Often an entire subtree is deleted when neighboring routes are withdrawn. The trie component from Figure 3 tags any pipeline write that deletes the root of an entire subtree. The packing component then eliminates all the writes in that subtree that occur with the same timestamp before the tagged write. Figure 15 shows how writes are eliminated when the four neighboring routes added in Figure 14 are withdrawn.

The effectiveness of both optimizations increases when a large number of neighboring routes are added or withdrawn in a single timestamp. Figure 13 shows the benefit of eliminating the excess writes. Combined with node pullups, it results in an incremental reduction of 18–25% in the number of write bubbles over using MinMax and filtering out short prefixes.

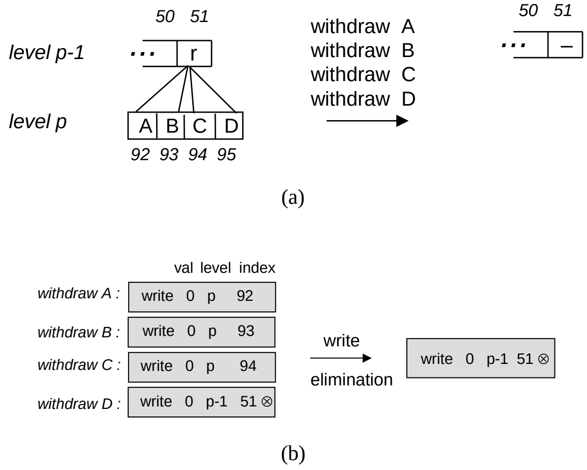


Fig. 15. (a) The portion of the trie before and after four neighboring prefixes A, B, C, and D are withdrawn. (b) Three excess writes are eliminated by the packing component; the write that deleted the entire subtree is tagged by the trie component to aid this optimization.

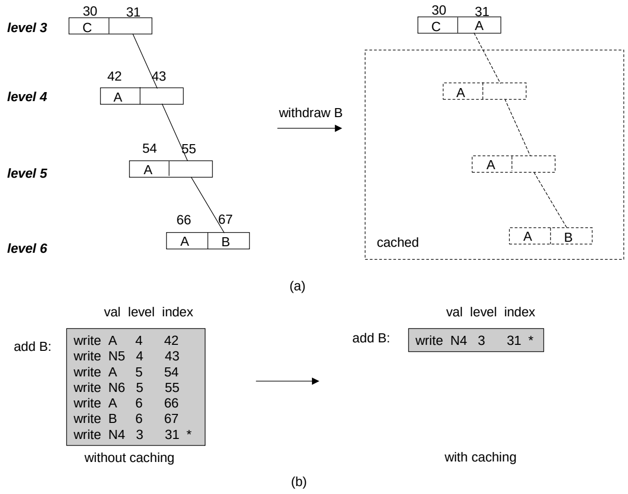


Fig. 16. (a) The caching optimization during the deletion of B, and (b) The corresponding writes that are avoided when B is re-inserted. A denotes the route that has been pushed down the subtree containing B. $N4, N5$, and $N6$ denote pointers to the nodes at levels 4, 5, and 6, respectively, and the numbers denote memory locations.

D. Caching deleted subtrees

A route is often withdrawn, and added back a little later with possibly a different next hop (**O–V**, Section IV). Since the withdrawal and the add often do not appear with the same timestamp, we cannot simply update the next hop table in this case. Instead, when a route withdrawal causes a subtree to be deleted, the trie component caches the subtree in software and remembers the location of the cached trie in the pipeline memory. The deleted subtree contains pointers to the withdrawn route, as well as (possibly) pointers to a shorter routing prefix that was pushed down into the subtree due to leaf pushing.[8]

[8] At most one prefix can be pushed down into any subtree from above the subtree.

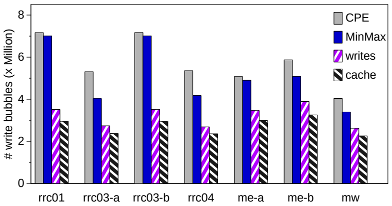


Fig. 17. Effect of caching subtrees that are withdrawn and added back soon after. The label "cache" denotes using the caching optimization (along with all the previous optimizations).

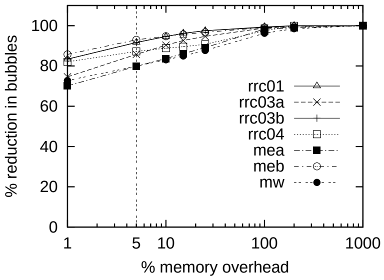


Fig. 18. Memory Requirements for Caching Deleted Nodes.

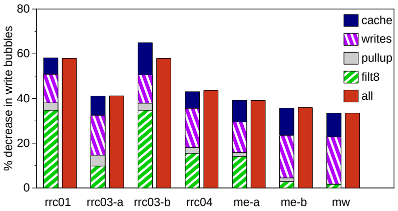


Fig. 19. Percentage reduction in the number of write bubbles; the base case here is a trie constructed using MinMax, with no other optimizations. The effect of each optimization is shown here in isolation (with no other optimizations turned on); "all" shows the effect of turning on all the optimizations. Sometimes, multiple optimizations target (eliminate) the same update; the "all" bar is slightly lower in those cases.

Therefore, the only information that must be stored with the cached subtree is the prefix that was pushed down, and the last route in the subtree that was withdrawn. If the same route is added before any other neighboring route gets added, and the prefix that is pushed down remains unchanged, we can simply add back a pointer to the deleted subtree in the pipeline memory. All this checking is performed in software by the trie component, and the pipeline sees only one write instead of a number of writes. The caching optimization is equivalent to allowing a fast "undo" of one route withdrawal. Figure 16 shows how this optimization works.

When multiple routes withdrawn in the same timestep result in the deletion of a subtree, the caching optimization can conflict with the optimization of eliminating excess writes (Section VI-C). For example, consider Figure 15 after routes A, B and C are withdrawn. If route D is withdrawn now and we cache the deleted subtree, the cached subtree would still contain routes A, B, and C — pipeline writes that deleted these routes were eliminated by the packing component as an optimization. If route D is now added back, and if the cached subtree is reinserted into the trie, routes A, B, and C would also (incorrectly) be added back to the trie. To avoid this error, the trie component caches a subtree only if a single route is deleted from it in one timestamp. By storing timestamps in the shadow trie nodes when they are modified, the trie component checks for this condition before caching a subtree. Figure 17 shows that caching subtrees reduces the number of write bubbles by an additional 13-16%.

Memory Requirements. Not all withdrawn routes are added back soon after. Therefore, caching subtrees can consume precious pipeline memory. However, since a route withdraw is often closely followed by an add, we get most of the benefit of caching subtrees by incurring small memory overheads. Therefore, we limited the amount of caching memory to a fixed size. We observed that nodes withdrawn several timestamps ago are less likely to be added back—hence, we maintained a FIFO list of cached nodes. When the caching memory in use went over the fixed size limit, the oldest cached nodes were deleted. We experimented with the amount of memory required for caching in order to get good performance from the caching optimization. The results are shown in Figure 18. The X-axis shows the caching memory overhead per pipeline stage as a percentage of the trie memory usage (actual allocation, not worst case) in that stage. In the Y-axis, we show how effective the optimization was when compared to the case where we could use unlimited caching memory. Thus, the 100% reduction number (on the Y-axis) refers to the case where we can cache as many deleted nodes as we want. We find that as much as 80% of the optimization benefits can be obtained even if we restrict the cached memory to only 5% of the memory allocated to the trie. The caching results in Figure 17 were obtained using a 5% memory overhead threshold.

E. Summary and discussion

The benefits of applying each optimization individually, and together with the other optimizations is shown in Figure 19 — all the tries in these experiments were built using MinMax. As noted before, the benefits of the node pullup optimization are small. However, it does help to further balance the memory requirements and the routing prefixes across the various pipeline stages. Each of the remaining optimizations appears promising in reducing the number of write bubbles. Compared to using controlled prefix expansion to construct the trie, the optimizations (along with MinMax for trie construction) result in (on average) a factor of 2 fewer bubbles — compared to a trie with even strides (4 bits per stage), the optimizations reduce the number of write bubbles on average by a factor of 5.7 (see Figure 20).

We also tested the effectiveness of our optimizations with fewer pipeline stages. Figure 21 shows the results of applying all the optimizations when the pipeline has 4 or 6 stages (and therefore, 4 or 6 trie levels, respectively) instead of 8. The graphs show that we get similar improvements for both the 4 and the 6-stage pipeline—hence, our optimizations are not specific to a given number of pipeline stages.

Prefix Table Dynamics. The MinMax algorithm attempts

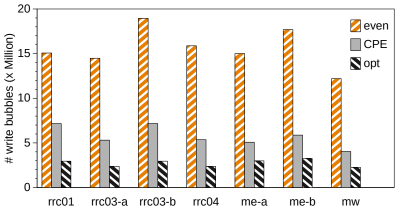
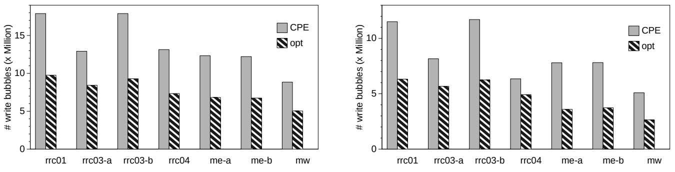


Fig. 20. Number of write bubbles generated for the different tables. Here "even" uses a trie with even (4-bit) strides, "CPE" uses controlled prefix expansion (with no additional optimizations), and "opt" constructs the trie using MinMax and uses all the optimizations listed in this section.



(a) 4 pipeline stages

(b) 6 pipeline stages

Fig. 21. Number of write bubbles when the lookup pipeline has fewer (4 or 6) stages. Here "CPE" denotes the number of bubbles generated when the trie is constructed using controlled prefix expansion (with no additional optimizations), and "opt" is when the trie is constructed using MinMax and all the optimizations listed in this section are applied.

to balance the memory allocations across the pipeline stages — one of the motivations behind this was to avoid frequent rebuilding of the trie after incremental updates. However, performing a large number of incremental updates may cause the trie to gradually become unbalanced. Indeed, it is possible that because of this unbalanced growth, the memory requirements for some of the pipeline stages may exceed the capacity of the pipeline stage when updates occur, and MinMax would need to be re-applied to balance out the memory allocations. Hence it can be argued that optimizing the memory allocation based on an initial prefix table snapshot may not be the right approach.

To explore this idea further, we did some measurements on how much the structure of the optimal trie (as chosen by MinMax) changes when large update sequences are applied to an initial trie. We used the same update traces that we have been using so far and the results are shown in Table IV. The table shows the percentage difference in the size of the largest stage after all the updates are applied, and the size of the largest stage if the MinMax algorithm were applied at the end of the updates to re-balance the trie once more. For all the tables, the basic structure of the trie (in terms of strides at each level) chosen by MinMax remained the same before and after the updates. The small difference in the sizes of the largest stage shown in Table IV is due to some additional nodes being pulled up when the trie is rebuilt at the end of the updates. Note that the optimal trie structure at the end of the updates remained unchanged even for the me-a and me-b tables, which grew by 34% and 61%, respectively. In conclusion, our approach based on optimal memory allocation using an initial prefix table snapshot works reasonably well. Of course, as the size of the table grows significantly, periodic re-balancing (and eventually, more physical memory) will be required.

Table	rrc01	rrc03a	rrc03b	rrc04	me-b	me-a	mw
Diff. in max	3.4%	1.5%	.09%	0.5%	7.9%	1.8%	.38%

TABLE IV

"DIFF. IN MAX" IS THE DIFFERENCE IN THE SIZE OF THE LARGEST STAGE AFTER ALL THE INCREMENTAL UPDATES ARE APPLIED, AND THE SIZE OF THE LARGEST STAGE WHEN MINMAX IS RE-APPLIED AFTER THE UPDATES TO RECONSTRUCT THE TRIE FROM THE NEW SET OF PREFIXES.

VII. RELATED WORK

Packet forwarding in high speed routers has been a well studied area. In particular, several techniques have been suggested to optimize IP lookup using binary trees [8], [10], [18–20], or binary search [4], [9], [12].

Multi-bit tries (such as the ones considered in this paper) have been also been used extensively for fast IP lookups. Gupta et. al. in [2] proposed a 2-level multi-bit trie where the first level had a stride length of 24 and the second level had a stride length of 8. This was based on the observation that most routing table entries have a prefix length of 24 bits or less. Consequently, most routing lookups could be done in 1 memory access — however, updating shorter prefixes can require several writes to memory. In another scheme, Nilsson et. al. [11] use the Level Compression technique (similar to the node pullup technique described in Section VI) to convert binary search tries into multi-bit tries. In their scheme, (nearly) full binary subtrees with k levels are converted (recursively) into a single multi-bit trie node with stride length k to reduce the number of lookups.

Several schemes, such as the Lulea Algorithm [7], apply path compression to optimize multi-bit tries. Crescenzi et. al. [6] use run length encoding to efficiently compress the routing table. The more general problem of constructing multi-bit tries (with either fixed or variable strides) that are optimal in terms of total memory usage for a given number of lookups was solved by Srinivasan and Varghese in [4]; this is the algorithm described in Section V. Cheung and McCanne [5] have developed an algorithm for trie layout that improves the average case performance using dynamic programming and Lagrange multipliers. Narlikar and Zane [21] built a performance model to improve the average-case performance of trie-based lookups in the presence of caching.

Almost all of the work described above deals with non-pipelined architectures, and the focus is to minimize the routing trie size or the lookup time. One exception is work by Sikka and Varghese [3], which presents efficient memory allocators to support fast updates, and points out the memory allocation problem for pipelined architectures. Our work is the first (to our knowledge) to focus on minimizing the overheads of incremental route updates in multi-stage pipelined architectures.

Technologies based on Ternary CAMs (TCAMs) [22], [23] provide an attractive alternative to ASIC-based designs that implement tries. TCAMs are content addressable (fully associative) memories that allow each bit to have a 0, 1, or "don't care" value. A TCAM can return one lookup result per cycle. Multiple matches for a given destination address are typically resolved by selecting the entry with the lowest (highest) memory location, requiring the table to always be sorted with respect to prefix lengths.

Recent work on TCAMs has looked at strategies for efficiently compressing and updating TCAM routing tables [24], [25]. The biggest advantages of TCAMs are their lookup speeds and ease of management. However, they are typically more expensive and consume significantly more power than ordinary random access memories that are used in ASIC-based designs.

VIII. Conclusions and Future Work

In this paper, we have presented a scheme for minimizing route update overheads in forwarding engines that use pipelined ASIC architectures. Our optimizations are driven by certain well-known characteristics of the IPv4 address allocation process (especially related to aggregation), and the BGP routing protocol. On applying these optimizations to a set of real route update traces obtained from the Internet core, we find that they can produce more than a 5-fold reduction in the number of write bubbles that go through the packet lookup pipeline when compared to the baseline case of assigning equal strides to each level. Further, when compared to an existing optimization algorithm that uses controlled prefix expansion, our algorithm shows a 2-fold reduction in write bubble overheads. Future work would include the development and evaluation of schemes for pipelines that require multiple passes through the pipeline for lookups and updates. This would be especially useful for IPv6 lookups.

References

[1] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, 2nd ed. San Francisco, CA: Morgan Kaufmann, 1997.

[2] P. Gupta, S. Lin, and N. McKeown, "Routing Lookups in Hardware at Memory Access Speeds," in *Proceedings of Infocom '98*, San Francisco, CA, April 1998, pp. 1240–1247.

[3] S. Sikka and G. Varghese, "Memory-Efficient State Lookups with Fast Updates," in *Proceedings of SIGCOMM '00*, Stockholm, Sweden, August 2000, pp. 335–347.

[4] V. Srinivasan and G. Varghese, "Fast Address Lookups Using Controlled Prefix Expansion," *ACM Transactions on Computer Systems*, vol. 17, no. 1, pp. 1–40, February 1999.

[5] G. Cheung and S. McCanne, "Optimal Routing Table Design of IP Address Lookups Under Memory Constraints," in *Proceedings of Infocom '99*, New York, NY, March 1999, pp. 1437–1444.

[6] P. Crescenzi, L. Dardini, and R. Grossi, "IP Address Lookup Made Fast and Simple," *Lecture Notes in Computer Science*, vol. 1643, pp. 65–76, 1999.

[7] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink, "Small Forwarding Tables for Fast Routing Lookups," in *Proceedings of SIGCOMM '97*, Cannes, France, September 1997, pp. 3–14.

[8] R. P. Draves, C. King, S. Venkatachary, and B. N. Zill, "Constructing Optimal IP Routing Tables," in *Proceedings of Infocom '99*, New York, NY, March 1999.

[9] B. Lampson, V. Srinivasan, and G. Varghese, "IP Lookups Using Multiway and Multicolumn Search," in *Proceedings of Infocom '98*, San Francisco, CA, April 1998, pp. 1248–1256.

[10] D. R. Morrison, "PATRICIA —Practical Algorithm to Retrieve Information Coded in Alphanumeric," *Journal of the ACM*, vol. 15, no. 4, pp. 514–534, October 1968.

[11] S. Nilsson and G. Karlsson, "Fast Address Look-Up for Internet Routers," *IEEE J. Select. Areas Commun.*, pp. 1083–1092, June 1999.

[12] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, "Scalable High Speed IP Routing Lookups," in *Proceedings of SIGCOMM '97*, Cannes, France, September 1997, pp. 25–38.

[13] C. Systems, "The Evolution of High-end Router Architectures, White Paper," 2001.

[14] C. Semeria, "Implementing a Flexible Hardware-based Router for the New IP Infrastructure," Juniper Networks," White Paper, September 2001.

[15] G. Huston, "Analyzing the Internet's BGP Routing Table," *The Internet Protocol Journal*, vol. 4, 2001.

[16] C. Labovitz, A. Ahuja, and F. Jahanian, "Experimental Study of Internet Stability and Wide-Area Backbone Failures," in *Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, Madison, WI, June 1999.

[17] C. Labovitz, G. R. Malan, and F. Jahanian, "Origins of Internet Routing Instability," in *Proceedings of Infocom '99*, New York, NY, March 1999.

[18] P. Gupta, B. Prabhakar, and S. Boyd, "Near-Optimal Routing Lookups with Bounded Worst Case Performance," in *Proceedings of Infocom '00*, Tel Aviv, Israel, March 2000, pp. 1184–1192.

[19] L. L. Larmore and T. M. Przytycka, "A Fast Algorithm for Optimum Height-Limited Alphabetic Binary Trees," *SIAM Journal on Computing*, vol. 23, no. 6, pp. 1283–1312, December 1994.

[20] K. Sklower, "A Tree-Based Packet Routing Table for Berkeley UNIX," in *Proceedings of the Winter 1991 USENIX Conference*, Berkeley, CA, January 1991, pp. 93–104.

[21] G. Narlikar and F. Zane, "Performance Modeling for Fast IP Lookups," in *Proceedings of ACM SIGMETRICS '01*, Cambridge, MA, June 2001.

[22] "IDT. `http://www.idt.com`."

[23] "Netlogic microsystems. `http://www.netlogicmicro.com`."

[24] H. Liu, "Routing Table Compaction in Ternary CAM," *IEEE Micro*, vol. 22, no. 1, pp. 58–64, January–February 2002.

[25] D. Shah and P. Gupta, "Fast Updating Algorithms for TCAMs," *IEEE Micro*, vol. 21, no. 1, pp. 36–47, January–February 2001.

Anindya Basu received his B. Tech in Computer Science from the Indian Institute of Technology (IIT) Kanpur, and his MS and PhD in Computer Science from Cornell University. He was a Member of Technical Staff for seven years at Bell Labs, the research arm of Lucent Technologies. During this time, he worked on various network related issues including routing protocols, forwarding engine design and network topology design. He currently works in the Fixed Income Division of Morgan Stanley.

Girija Narlikar (M '99) received her B.Tech in Computer Science from the Indian Institute of Technology (IIT), Bombay, and her MS and PhD in Computer Science from Carnegie Mellon University. She is a Member of Technical Staff at Bell Labs, the research arm of Lucent Technologies. Her current research interests include various aspects of data and wireless networking, with an emphasis on routing and scheduling.