Performance Modeling for Fast IP Lookups

Girija Narlikar Francis Zane
Bell Laboratories, Lucent Technologies
700 Mountain Ave, Murray Hill, NJ 07974
{girija,francis}@research.bell-labs.com

ABSTRACT

In this paper, we examine algorithms and data structures for the longest prefix match operation required for routing IP packets. Previous work, aimed at hardware implementations, has focused on quantifying worst case lookup time and memory usage. With the advent of fast programmable platforms, whether network processor or PC-based, metrics which look instead at average case behavior and memory cache performance become more important. To address this, we consider a family of data structures capturing the important techniques used in known fast IP lookup schemes. For these data structures, we construct a model which, given an input trace, estimates cache miss rates and predicts average case lookup performance. This model is validated using traces with varying characteristics. Using the model, we then choose the best data structure from this family for particular hardware platforms and input traces; we find that the optimal data structure differs in different settings. The model can also be used to select the appropriate hardware configurations for future lookup engines. The lookup performance of the selected data structures is competitive with the fastest available software implementations.

1. INTRODUCTION

Routing of IP packets requires fast operation of a tight inner loop: for each packet, extract its destination IP address, look this address up in a table, and take an appropriate action (eg, send the packet out along a given link) based on the result. Obviously, the number of possible destinations in the Internet is too large to permit this simple approach to work in practice; instead, the tables are described in a compact way by aggregating individual IP addresses into larger groups for which the same actions should be taken.

This is done by storing routing table entries in *classless-interdomain routing (CIDR)* format; each entry consists of an *(address, length)* pair and represents all addresses which match *address* in the first *length* bits of their binary representations. For example, 204.178.0.0/16 represents all addresses of the form 204.178.*.*. To make it possible to efficiently subdivide these networks, it is permissible to have more than one entry which covers a given address. If more than one entry contains a given address, then the action corresponding to the most specific entry (i.e., the one with the longest prefix) containing that address should be used. Here, if the routing table consists of 204.0.0.0/8 and 204.178.0.0/16, then 204.178.1.1 matches the latter entry, while 204.179.1.1 matches the former.

This key operation, *longest prefix match*, is often implemented directly in hardware on high-speed routers for performance reasons, since it must be performed on each packet. The emphasis on hardware implementations has guided much of the work in this area. The algorithms used are simple, with well-understood worst-case behavior, because of the complexity of implementing them in hardware. Similarly, total memory usage is a key design criteria on hardware devices with fixed amounts of memory.

However, recent improvements in PC hardware and algorithms make it possible to scale to speeds of a few million packets per second on conventional hardware. This allows for improved performance of software which depends on longest prefix match to analyze packet data (for example, the clustering algorithm used in [11]). In addition, chip manufacturers like Intel [9] and C-Port/Motorola [3] are producing network processors aimed at allowing rapid development of high performance network applications, including both edge and core routers. The Intel IXP1200 configuration, for example, consists of six microengines (small CPUs), up to 8MB of SRAM, and up to 256MB of SDRAM, and supports multiple context threads per microengine. In such settings, fast lookup code, optimized for the specific system environments available to these devices, will enable higher performance solutions.

On such platforms, new design decisions are possible: Programming, even for non-PC network processors, is much easier than hardware design, allowing a wider space of algorithms to be used. Total memory usage is less of a constraint, since more memory can be added cheaply, but the behavior of the memory hierarchy becomes important. Finally, worst-case behavior becomes less of a concern; complex buffering is not required to allow some packets to be processed more quickly than others. Instead, we can evaluate algorithms based on their average-case performance on realistic inputs, and search for ones with good performance

in practice.

In this paper, we present techniques for testing and tuning the average-case performance of algorithms for this problem, given a table of prefixes and a statistical summary of an input trace. We begin by describing a family of data structures for IP lookup generated by composing known techniques, particularly tries and splay trees. In order to evaluate these data structures, we need two things: First, our goal of average-case performance must be defined with respect to some distribution of inputs. To make it a meaningful measure, we must choose input distributions which are representative of real settings. We examine several input traces, both real and synthetic, and identify properties distinguishing these traces. Second, we need a model that allows us to efficiently estimate the performance that a given design will provide. Here, a key component of the model is estimating cache miss rates in various data structures, since cache behavior is a significant factor in overall performance. We validate our model using the traces, showing that the model accurately predicts performance on different platforms across different input traces. By applying this model to specific system configurations, we are able to select the data structure from this family which optimizes performance for that system for the different traces. The data structures we find with this method achieve performance comparable to or better than the best previous implementations. Interestingly, the optimal data structure found is different for different systems and for different packet traces. In addition, this model can be used in guiding system design, allowing designers to predict the performance of different hardware configurations. To illustrate this, we examine the sensitivity of the performance to system parameters like cache size and cache speed, enabling us to identify components which have a significant impact on overall performance.

2. IP LOOKUP ALGORITHMS AND DATA STRUCTURES

In designing our algorithms for longest prefix match, we proceed by composing two known building blocks.

The first goes by a number of names, alternately described as expanded prefix tries [18], generalized level-compressed tries [4], and could easily be described as a tree-of-arrays or tree-of-hashes. For lack of a consistent name, we will refer to it simply as a *trie* throughout, with the understanding that we mean it in this generalized sense.

Regardless of the name, the idea is straightforward. The simplest version, a binary trie, represents a set of prefixes as a binary tree, associating with each prefix a path from the root in the tree. Looking up the prefix associated with a string x in such a structure is easy: At each node n, read the next unread bit of x. If it is 0, move to the left child, if 1, move to the right child. Repeat this process until a leaf is reached, which corresponds to the longest matching prefix. Since each prefix corresponds to exactly one node, we can store any information associated with the prefix in that node, and look it up when we arrive there.

The trie data structure we use generalizes this to larger strides or branching factors at each node in the natural way: Instead of each node having at most two children, and reading one bit to determine which one to move to, the number of children a node has is 2^k (the *branching factor*) for some integer k (the *stride*), and the next k bits of x are used to select which child to visit. This stride, and thus the number of children, can be different for different nodes in the trie. However, to simplify both the algorithm and the optimizations we perform later, we restrict our attention to levelled tries. In such tries, the stride is fixed for all nodes at the same depth in the trie. We will refer to specific parameter choices using the notation T2(16,24) to denote a two-level trie, where the first level is indexed by bits 1-16 and the second level by bits 17-24. In our implementation, a trie node consists of two pointers (8 bytes). There are three possible outcomes of visiting a trie node, depicted in Figure 1: If the node has children (node A), it points to a table of its children. If it is a leaf entirely covered by a single prefix, or a leaf uncovered by any prefix (node B), we store a pointer to the appropriate data directly in the node. Finally, if a trie node contains multiple prefixes (node C), or if the address range represented by a trie node is not completely covered by the single prefix it contains (node D), it requires a pointer to the splay tree data structures described below.[1]

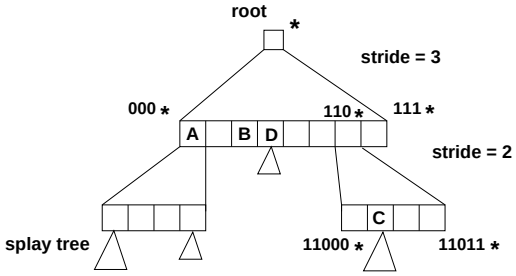


Figure 1: An example trie with two levels

In this non-binary trie, there is no longer a one-to-one correspondence between prefixes and nodes of the tree; for example, if we take $k = 3$ bits in the first step, then the length one prefix "1∗" will actually correspond to four children "100∗" through "111∗" at the first level of the trie, causing the prefix to be *replicated* four times. Grouping several bits together in this way reduces the number of sequential memory accesses required at the expense of increasing memory usage due to replication of prefixes into multiple nodes.

In addition, it is possible that a leaf of the trie corresponds to a range of addresses which is not covered by a single prefix. The hope is that by using the trie at higher levels, there are not too many prefixes sent to this leaf. What we would like is a data structure to provide fast access to the most popular ones, while having a memory requirement which depends only on the (small) number of prefixes in this leaf. This second data structure we use is a *splay tree* [17], a type of self-adjusting binary search tree, which dynamically restructures itself to bring frequently searched nodes close to the root of the tree. To apply splay trees to our longest

[1]While we use two pointers for this, it could potentially be reduced to 1 pointer by more aggressive manipulation of pointer bits as flags for these cases. Instead, we use the additional space to store the number of hits to different prefixes, which is an input to our performance model.

prefix match problem, we view the space of addresses as a sequence of 32-bit integers, and divide this space into *intervals* separated by the beginning and endpoints of each prefix when represented in this integer form. This decomposition of prefixes into intervals for binary search was used in previous work [8, 12]. There, they optimize the construction of trees of this form. Here, we will use these trees in many places in our data structure. Rather than explicitly optimizing each tree, we use self-adjusting trees to get good, if not optimal, performance without requiring detailed optimization. In our implementation, each splay tree node requires 24 bytes, since it stores two child pointers, a pointer to the result of the lookup, as well as interval begin/end pointers and a hit counter. Because we use a self-adjusting tree, bounding the worst-case performance of our data structures is problematic. If a worst-case bound is desired, the splay trees can be easily replaced by balanced binary trees, or the more optimized weight-balanced binary trees [8].

In summary, the data structures we consider are leveled tries. Because of this leveled nature, the trie structure for a given prefix table can be described completely by a vector of strides.

2.1 Comparison with Previous Work

The trie technique described above appears in various forms in numerous works. For example, the BSD kernel contains a trie-based implementation [16]. As mentioned above, Srinivasan and Varghese [18] also use the idea of allowing replication of prefixes in order to construct tries with fewer lookups. They consider a family of such tries with different parameters, and find the optimal trie structure, as we do in subsequent sections. The distinction, however, is that they minimize the *memory* required, subject to some worst-case performance guarantees. Here, we maximize the *performance* achieved in the average case; the memory required is important to us only in its impact on caching behavior.

Both Waldvogel et al. [20] and Lampson et al. [12] mention the idea of improving practical (i.e., average-case) performance by adding an array lookup based on the first 16 bits of the address as a preparatory step to other algorithms. Our algorithm is a natural extension of this idea. As we will show later, in improving average-case performance, this is a very powerful technique; indeed, as we will show, the optimal choice will often be to take more than 16 bits in the first step, and the optimal number of bits is sensitive to the properties of the inputs used.

Moving closer to the details of our schemes, several previous papers make use of similar trie data structures, such as the LC-tries of Nilsson and Karlsson [14]. In particular, Degermark et al. [2] construct a data structure which (at a high level) can be described in our notation as a T3(16,24,32), supplemented by several compression techniques. The compression at the lower levels, which deals separately with leaves depending on the number of prefixes they hold, is aimed at addressing the same problems as our use of splay trees, though it uses different techniques. Again, the overall goal of this compression is to minimize storage requirements, allowing a hardware implementation to use less SRAM, rather than to directly improve performance. In a similar vein, the table-based scheme of Crescenzi et al. [5] can be seen as an implicit representation of a two-level trie: A row is indexed by using a function of the first 16 bits (first level), and then a position is indexed by a function of the second 16 bits (second level). By using functions of these bits, rather than the bits directly, and by combining prefixes with the same next hop, they are able to compress the representation of the trie, leading to less storage requirements and better cache performance. In a different direction, Gupta el at [8] examine the problem of improving average case lookup times while simultaneously bounding the worst case lookup time. However, their key data structure is a binary tree, requiring many sequential memory lookups, so their performance in our setting is not competitive.

Finally, Cheung and McCanne [4] take an approach very similar to ours: They consider a family of multilevel tries and model the performance of such search algorithms on systems with a memory hierarchy. Using their model, they optimize performance, either directly with a pseudopolynomial-time dynamic programming algorithm or approximation via Lagrangian relaxation. They describe how to implement this approach on systems with explicit memory management, and extend it to memories with caches by marking items explicitly as cached or uncached. In our work, we look at the cache behavior at a much more detailed level; in particular, accounting for non-trie tree accesses (not present in their data structures) requires more detailed modeling. We also model standard caches, where items are brought in on demand and subsequently replaced.

3. DATA

To evaluate the average-case performance of different data structures, we apply our models to several specific input distributions. For us, the input to the IP lookup problem consists of two parts: a *table* of prefixes and a *trace* consisting of a sequence of IP addresses. We describe the table and traces used in our experiments, and give characterizations of the real traces which highlight their differences with simple synthetic traces, differences which will cause algorithms optimized on synthetic traces to perform sub-optimally on real data.

3.1 Data Sources

In all our experiments, we use a table obtained in May 2000 from the MAE-East peering point [10], which we take to be representative of a large routing table used near the core of an IP network. It contains 52857 prefixes, distributed among lengths 8-32. As shown in Figure 2(a), about half of all prefixes have length 24, with most of the remainder distributed between 16 and 23 bits. Similar experiments could be performed with other tables; we have chosen to focus on this one table both for simplicity and because one of our traces was anonymized in a table-dependent fashion. This table does not contain the 0 length prefix 0/0, and addresses not matching any of the prefixes in the table have been removed from all traces. Without these modifications, mismatches between where the table is used and where the data is collected skew the data, producing significant traffic for this default prefix.

Our experiments also make use of 4 traces, 2 real and 2 synthetic. The first real trace, *ISP*, consists of 1M addresses spanning 172K unique addresses obtained from packets seen

by an ISP core router. The addresses have been anonymized, but in a manner which is one-to-one and respects the prefix boundaries of the MAE-East table (i.e., both an address and its anonymized version belong to the same prefix). This trace will be our primary benchmark in evaluating performance. The second real trace, *SDC*, contains 10M addresses (63K unique) obtained from an edge router at the San Diego Supercomputer Center [15]. Because it is collected at the edge of the network, this trace is less similar to the distributions we expect to see in practice than the ISP trace; however, access to two very different real traces provides greater confidence in validating the model. In particular, the distribution of addresses in the SDC trace is in some ways too easy for IP lookup: It has a large contribution from a small number of popular addresses. Therefore, one could apply further optimizations (caching of destination IP addresses, for example) which would improve performance on this trace, but which would not generalize to other settings. As we will see, however, even these two very different real traces share several common characteristics not shared by simple synthetic traces.

The synthetic traces we compare to are generated using simple, natural heuristics. In *RandIP*, each address is generated by choosing uniformly from among the addresses covered by some prefix. In *RandNet*, first a prefix is chosen uniformly at random from the set of prefixes, and then an address belonging to that prefix is chosen, again uniformly at random. These represent the natural choices for traffic distribution in the absence of any information about real traces. To distinguish them from more sophisticated traffic generation models, we will refer to them as "random." As we will show, however, real traces different significantly from random traces, even at a statistical level. However, these traces, particularly RandIP, are often used to benchmark IP lookup algorithms. Several papers [20, 12] make use of RandIP in attempting to characterize average-case performance; in [4], RandNet is used explicitly (called iid prefix), as well as a variant of RandIP called scaled prefix, where prefixes are chosen with weights inversely proportional to exp(prefix length).

All of our experiments are done on IPv4 traffic. While similar modelling techniques may apply to IPv6, we do not have sufficient real data to judge average-case performance.

3.2 Data Characterization

These four traces will be used to validate our model and select and tune data structures for specific scenarios. As we will show in the Section 6, using different traces will lead to different choices in designing data structures for IP lookup. Before doing so, we examine the traces more closely to understand the properties of the traces and the differentiation between real and random traces. We do this by characterizing the traces along two main directions: distribution of addresses by prefix and distribution of addresses by prefix length.

The distribution of addresses in a trace among the prefixes of the table is important because it reveals the extent to which the traffic is concentrated among a small number of prefixes. We illustrate this in Figure 2(b) by plotting the fraction of addresses captured by the most popular network.

As expected, the real traces, particularly SDC, are heavily skewed towards a few popular prefixes. This concentration of real traffic patterns has been noted previously in a number of contexts, including inter-AS traffic [6] and web site access [11, 1].

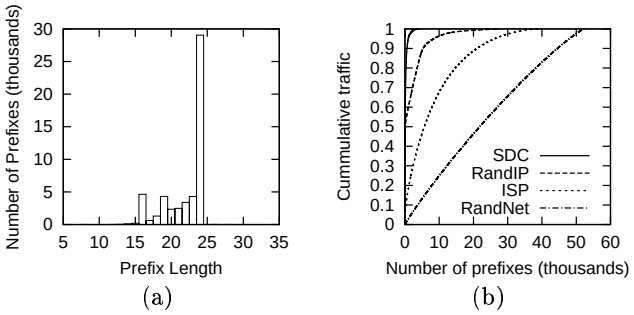


Figure 2: (a) Traffic Distribution of Prefix Lengths, and (b) Traffic Distribution by Prefix

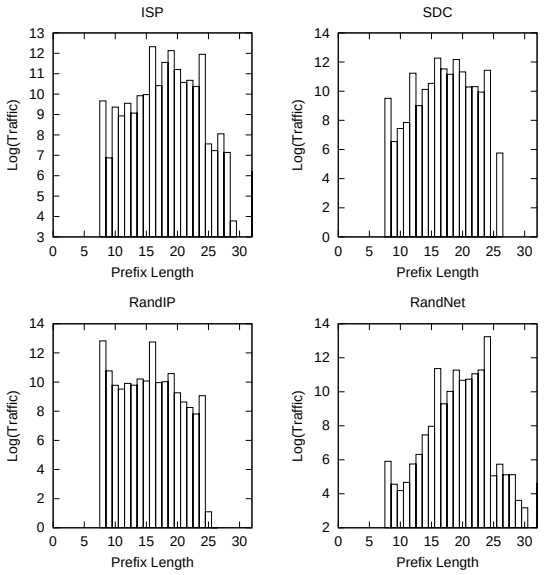


Figure 3: Log(Traffic distribution) by prefix length

However, the performance of our lookup data structures will be affected not only by the number of popular prefixes, but how those popular prefixes are distributed across different prefix lengths. Figure 3 shows the distribution of traffic to different length prefixes. While clearly not identical, the two real traces have a number of points of similarity: a high concentration of traffic at 16-20 and 24 bit prefixes and relatively little traffic outside that range. By contrast, the random traces concentrate traffic in either very short (RandIP) or very long (RandNet) prefixes. Therefore, the commonly-used RandIP distribution is a very easy distribution for most IP lookup algorithms: Hashing on the first 8 and/or 16 bits gives direct answers for a large fraction of the addresses in the trace.

We now examine the *average* number of hits for a prefix of

a given prefix length. For the random traces, the results are as expected: The number of hits per prefix is roughly constant independent of prefix length for RandNet, and for RandIP decreases by a factor of 2 each time the prefix length increases by 1. For the real traces, the answer has two parts. Outside of prefix length of 13-24 bits, the mean number of hits varies unpredictably; however, prefixes outside that range represent only a small portion of the traffic in our real traces. For prefixes with length in the important range 13-24, the number of hits per prefix is decreasing with prefix length (unlike RandNet), but at a rate much slower than RandIP. A least-squares fit shown in Figure 5 of the log of the number of hits to prefix length yields that increasing the prefix length by one decreases the mean number of hits by a factor of about 0.69. These observations could be used in the future to develop more realistic synthetic trace models, which would be an important help in analyzing average-case performance.

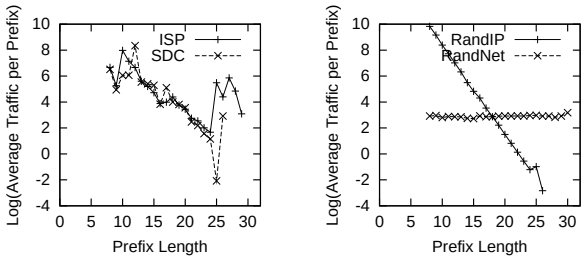


Figure 4: Average traffic by prefix length

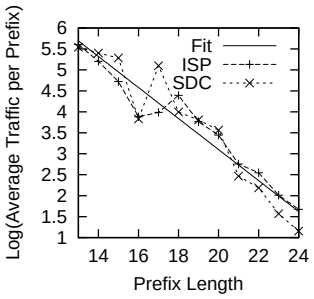


Figure 5: Linear fit of average traffic vs. prefix length

Finally, these characterizations rest on analyzing some short segment of data, from which we wish to conclude some global properties of the trace. To make this possible, the properties of the trace that interest us (eg, distribution of hits among prefix lengths) should not vary widely over time. In order to demonstrate this, we examine the histograms of traffic by prefix and traffic by prefix length taken from different segments of the same trace. We then quantify the difference between two such histograms by computing the total variation distance (essentially, a normalized L_1 distance between the histograms):

$$\frac{\sum_{\text{bins } i} \text{abs}(h_i - h'_i)}{\sum_{\text{bins } i}(h_i + h'_i)}$$

where h_i, h'_i denote the weight given to histogram bin i in the two traces. This distance d is at least 0 and at most 1; intuitively, it says that one histogram can be turned into the other by moving a d fraction of the samples between bins.

	ISP	SDC
By Prefix	0.278	0.389
By Prefix Length	0.006	0.083

Table 1: Total Variation Distance

This total variation distance was computed for the ISP trace (comparing the first 0.5M packets with the last 0.5M packets out of the 1M packets in the trace) and the SDC trace (comparing the first 1M packets against the last 1M packets out of the 10M in the trace; the two segments are over one hour apart). The results are shown in Table 1. While the SDC trace exhibits larger variation in an absolute sense (both because of the properties of the trace and the larger separation between the intervals compared), the conclusions for the two traces are similar. While the access frequencies to individual prefixes change somewhat, the frequencies to different prefix *lengths* remain relatively constant. For trie-based algorithms hashing on bits of the address, this is encouraging, since it indicates that the importance of specific address bits is not varying widely with time. It also indicates that , at this coarse time scale, the traces have only limited temporal locality. Further work, examining these properties over longer traces (which are difficult to obtain in unanonymized form), would be helpful in making these statements more precise.

4. PERFORMANCE MODEL

We now describe how we build the performance model to predict average case lookup time for a lookup data structure, given the prefix table and a summary of the input packet trace consisting of the access frequencies for each of the entries in the table. This summary data is much easier to collect than complete traces, so our model assumes only this knowledge; however, the full traces described in the previous section are used in validation. Our model focuses solely on the problem of lookup time, assuming that other issues like updates are handled separately (say, through a double-buffered implementation). Incorporating update handling into such a model is a problem for future study.

The lookup engine in our model has a 3-level memory hierarchy: an L1 cache, an L2 cache and main memory (see Figure 6). The number of levels, however, can be easily adjusted for different processors. We make some assumptions and approximations to simplify the analysis of average lookup performance; these assumptions are mentioned throughout this section, and are *highlighted*.

Let t_{L1} be the penalty of missing in the $L1$ cache, which equals the L2 read latency. Let t_{L2} be the *additional* penalty for missing in the $L2$ cache. Since every $L2$ miss also causes an $L1$ miss, $t_{L1} + t_{L2}$ equals the main memory latency of the machine. Because references to the L1 cache depend statically on the compiled code and not on the locality introduced by the input data, we include the cost of accessing the $L1$ cache as part of the of the useful work done in the code.

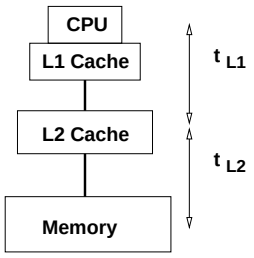


Figure 6: The cache hierarchy in our model.

An IP lookup in our chosen space of data structures consists of visiting one or more trie nodes followed by visiting zero or more nodes in a binary (splay) tree at the leaf of the trie. Therefore the average time t_{avg} to search the data structure can be expressed as

$$t_{avg} = M_1 \cdot t_{L1} + M_2 \cdot t_{L2} + H \cdot t_{trie} + S \cdot t_{tree} + t_{const} \quad (1)$$

where M_1 is the average number of L1 cache misses and M_2 is the average number of $L2$ cache misses per IP lookup. Here t_{trie} is the average amount of time per IP lookup spent visiting a trie node that is in the $L1$ cache, t_{tree} is the average amount of time spent visiting a tree node in the $L1$ cache, and t_{const} is the additional overhead for processing each packet. H is the average number of trie nodes visited while S is the average number of splay tree nodes visited. This assumes that *the useful work done while the CPU is waiting on a miss is negligible*; this is reasonable for the lookup procedure, because it essentially involves pointer jumping. However, this assumption is not valid for multi-threaded lookup engines that can context switch between threads to hide memory latency. Further, Equation 1 assumes that *no other process is contending with the lookup process for access to the memory bus or caches*.

We first explain how the average number of cache misses can be computed in general. We then describe how the average number of accesses to tree and trie nodes can be inferred; this allows us to compute the expected number of cache misses incurred during the lookup procedure.

4.1 Computing cache misses

We compute the cache misses for the L1 and L2 caches separately. Consider a set of N memory blocks being accessed through a cache with C blocks. Let B be the total number of accesses made to the N blocks over a certain period of time. For $i = 1, \ldots, N$ let b_i be the number of accesses to memory block i. To simplify the analysis we assume that *accesses to prefixes are independent*; that is, we do not model small-scale temporal locality in the input traffic. Since real traffic typically has such temporal locality, this approximation causes our model to over-predict cache misses incurred during lookups for real traces. However, in actual lookup experiments we found at most a 10% difference in the measured average lookup time between our ISP packet trace and its random permutation, and at most 15% for the SDC trace, indicating that the resulting inaccuracies are limited.

We approximate the expected number of cache misses while accessing block i as follows. For simplicity, we first assume the cache is direct mapped; we subsequently show how to handle set-associative caches. If the number of memory blocks N accessed is large compared to C, and no small set of blocks is accessed too frequently, it is reasonable to assume that *each cache block is accessed with roughly equal probability*.[2] Therefore, each cache block gets B/C accesses. Now consider a memory block i; it gets mapped to a unique cache block. Every time memory block i is accessed, the access will be a hit if the last access to the cache block was for the same memory block. Since b_i of the B/C accesses to the cache block are for memory block i, the probability q_i (in steady state) that an access to memory block i will result in a miss is given by

$$q_i = \frac{B/C - b_i}{B/C} = 1 - C \cdot b_i/B$$

Since there are a total of b_i accesses to memory block i, the total number of cache misses incurred due to memory block i is $b_i\,(1 - C \cdot b_i/B)$.

Associative caches. If the cache is a-way set associative, the memory block can map to a set of a cache blocks. Let S_i be the set of a cache blocks that memory block i maps to. Then S_i gets $a \cdot B/C$ accesses. If LRU replacement is used between cache blocks in one set, then access to memory block i will not be a miss if at least one of the last a accesses to set S_i was an access to memory block i. The number of misses m_i incurred due to memory block i is then summarized as

$$m_i = b_i \left(1 - \frac{C \cdot b_i}{a \cdot B}\right)^a \quad (2)$$

This expression may overestimate the misses to memory block i because *a reference to it can be a hit even when none of the last a accesses to set S_i was to memory block i*. This may happen when there are repeated references to some other memory block that maps to S_i, while i was still within the last a *unique* memory blocks to be accessed.

4.2 Access counts for each memory block

Each binary tree node and trie table entry is treated as a separate memory block. This assumes that *we get no spatial locality from accesses to consecutive trie nodes*.[3] We compute the number of times each memory block is accessed during the lookup procedure. We then use Equation 2 to predict the cache misses for both the L1 and the L2 caches separately.

The input to the model is the prefix table, and the relative lookup frequencies of the prefixes for a representative packet trace. The ***lookup frequency*** of a prefix is how often the prefix is returned as the answer to the IP lookup problem, when a fixed number of packets from the representative packet trace are looked up. We also need the machine parameters t_{L1} and t_{L2}, and the parameters t_{trie}, t_{tree} and

[2] This assumption is not valid for traffic patterns where a very small number of prefixes significantly dominate the set of prefixes looked up.

[3] Trie nodes, which are 8 bytes each, are implemented as arrays and are likely to have some spatial locality. In principal this can be modeled, but would complicate our modeling code. Splay tree nodes are larger and are not likely to yield spatial locality.

t_{const} which depend on the implementation of the lookup code on the given machine.

For simplicity, we first explain how the access counts are computed for a one-level trie followed by splay trees at its leaves, and then describe how to extend it to a trie with multiple levels. To build the performance model for a prefix table, we perform a depth-first traversal of a 32-level binary tree, which we will call T_B. The goal is to compute, in a single traversal of T_B, the average lookup time for 1-level tries with all possible strides.

We start with all the prefixes at the root; each prefix is assigned a weight that represents its lookup frequency. As we traverse T_B, we split the prefixes at each level of T_B. At level l, all the prefixes with 0 at bit position l are passed down the left branch, while all prefixes with 1 in bit position l are passed down the right branch. Each prefix with length less than l is passed down both branches; its weight is divided by two down each branch. Since we do not know the real lookup frequency for the prefix across each branch, we assume that *it is looked up equally often along each branch.*

When we reach a node of depth l during this depth-first traversal, we know that when the initial stride is l, this node will be a trie leaf and all prefixes contained in it will be stored in a splay tree rooted at this node. By summing the weights of the prefixes at this node, the number of accesses to this trie leaf is obtained.

Accounting for the splay tree accesses requires more effort. Although the prefixes are converted (by possibly splitting them) into intervals before insertion into the splay trees at the leaves of the trie, we construct the model assuming that *prefixes themselves form the nodes of the splay tree.*[4] A splay tree is a self-balancing tree data structure that attempts to minimize lookup depth (that is, the average number of nodes touched in a lookup). For a given distribution of requests T, this average depth (for any binary tree) is at least the *entropy* $E(T) = -\sum_{i \in T} p_i \log_2 p_i$ of the distribution of requests; here p_i is the probability of looking up element i. In practice, splay trees come close to achieving this optimum. Theoretical bounds show that they do no worse than $3E(T) + 2$ for sufficiently long sequences of lookups[7]. Therefore, *we approximate the lookup depth for each splay tree T at a trie leaf as $1 + E(T)$ to include for the access to the root of the tree.* Thus if h_i is the lookup frequency of a prefix in a splay tree T, then

$$E(T) = -\sum_{i \in T} p_i \log_2(p_i), \quad \text{where} \quad p_i = \frac{h_i}{\sum_{j \in T} h_j}$$

Let n be a leaf of the trie in the lookup data structure, and let T_n be the splay tree stored under n. Then, every lookup to a prefix under n results in one access to n. Thus n is accessed $H(T_n) = \sum_{i \in T_n} h_i$ times, where h_i is the lookup frequency of node (prefix) i. Every access to trie node n is followed by an average of $1 + E(T_n)$ accesses to the nodes in T_n. Thus, the nodes in T_n get a total of $(1 + E(T_n)) \cdot H(T_n)$ accesses. We cannot precisely determine the distribution of these accesses to the nodes in T_n without explicitly constructing the splay tree (for example, the root of the tree gets accessed every time any prefix in the tree is looked up). We therefore assume as an approximation that *each prefix i in T_n with a lookup frequency of h_i is accessed $E(T_n) \cdot h_i$ times.* The inaccuracy introduced by this approximation becomes significant when the binary splay tree is large and therefore has a large entropy. However, as we show in Section 5, good lookup performance is obtained only with small splay trees, and therefore we did not find it necessary to derive a more accurate model for large splay trees.

[4]Only a small fraction of prefixes get split into two or more intervals because of overlap between prefixes; the remaining prefixes map directly into intervals.

Processor	L1 cache	L1 miss	L2 cache	L2 miss
400 MHz Pentium II	16KB on-chip, 4-way	38 ns	512KB off-chip, 4-way	100 ns
700 MHz Pentium III	16KB on-chip, 4-way	10 ns	256KB on-chip, 8-way	100 ns

Table 2: Relevant details of the hardware platforms we tested our model on. Cache lines are 32 bytes. The Pentium II has 128MB RAM while the Pentium III has 512MB RAM.

Once we know the number of accesses to each node in the trie and the splay trees, we can compute the $L1$ and $L2$ cache misses using Equation 2. For a one-level trie, exactly one trie node is accessed in each lookup. The average number of splay tree nodes accessed is computed as the weighted sum $\sum((1 + E(T)) \cdot H(T)/N)$ over all splay trees T at the leaves of the trie; here N is the total number of IP lookups performed on the data structure.

We maintain global counters for each level $l = 1, \ldots, 32$ of T_B. When visiting a node at level l of the binary tree T_B, we incrementally add to the total number of $L1$ and $L2$ cache misses, and the number of tree or trie nodes visited, when a stride of l bits is used in the final lookup data structure. By the end of the traversal of T_B, we accumulate these statistics for each choice of the stride. We can now compute the average lookup time using Equation 1, and choose the optimal value for the trie's stride.

Extension for multi-level tries. For a two-level trie, when we visit a node in the 32-level binary tree T_B, we must now add in the contributions to the cache misses and average number of trie or tree nodes accessed under two cases: when the current node represents a first-level trie node, or a second-level trie node. Similarly, for a three-level trie, we add contributions to the misses and node access counts in three cases: when the current node is the first-level, middle-level or third-level trie node. These counts can be collected over exactly two traversals of T_B.[5]

5. MODEL VERIFICATION

We now present results to show that our performance model is valid for a range of traces, and for the two different machines listed in Table 2.

[5]The first traversal simply computes the total number of accesses to memory blocks while the second computes the miss rates using the access counts.

5.1 Determining the Model Parameters

We first explain how the various parameters were determined for each machine, and then validate the use of Equation 1 using these parameters.

The L1 and L2 cache miss penalties (that is, the parameters t_{L1} and t_{L2} used in Equation 1) were inferred by running the "lmbench" benchmark [13] on each machine. Note that the memory hierarchies and relative speeds of memory accesses to clock cycles are fairly different on the two machines. We also set a limit of 32MB on the total size of the lookup data structures that the model evaluates.

We used the performance counters available on the Pentiums to measure the actual number of L1 and L2 cache misses during the search procedure, for one million packets from each packet trace. We also counted the average number of tree and trie nodes visited by modifying our code. Thus we have the values for t_{avg}, M_1, M_2, t_{L1}, t_{L2}, H, and T in Equation 1. To infer the values for the cost of visiting a trie node (t_{trie}), the cost for visiting a splay tree node (t_{tree}), and the constant overhead (t_{const}), we used a least-squares fit against the real lookup time with t_{trie}, t_{tree} and t_{const} as the unknowns. We confirmed the results using Vtune [19], a profiler from Intel. For the Pentium II, we find that $t_{trie} = 12.5ns$, $t_{const} = 7.5ns$ and $t_{tree} = 37.5ns$. For the Pentium III, $t_{trie} = 2.5ns$, $t_{const} = 4.5ns$, and $t_{tree} = 16ns$. Figure 7 shows the lookup time computed according to Equation 1 with the above parameter values (and using experimentally measured values for M_1, M_2, H and T), and compares it with the experimentally measured lookup time for a 1-level trie.

5.2 Verification of performance prediction

We now examine the accuracy of the predicted performance when M_1, M_2, H and T are predicted by our performance model. We use the values of t_{L1}, t_{L2}, t_{trie}, t_{tree} and t_{const} as given in Section 5.1. Figure 8 compares the measured performance with the performance predicted by the model for the Mae-East table, using a one-level trie. The measured lookup time is averaged over one million consecutive packets in each trace. Figure 9 shows the prediction accuracy of the model for 2-level tries. In this figure, for each x-value we show the performance of the best 2-level trie with stride x at the upper level. In both figures, the error bars indicate the standard deviation for the measured average lookup time. To keep measurement overheads low, we compute the mean lookup times for sets of 1000 consecutive packets (from a total of 1 million packets for each trace), and compute the standard deviation of the resulting 1000 means. The standard deviations were within 10% of the mean for all the traces.

For 1-level tries with strides smaller than 8, the splay trees are fairly big, making our approximate counting of memory accesses to tree nodes inaccurate. Further, a small stride translates to a small number of contiguously spaced trie leaf nodes, each with high access counts; this leads to spatial locality (which our model does not account for). However, for these settings, the performance of the lookup structure is far from optimal. For the interesting ranges of bits, the performance is predicted fairly accurately. This problem does not arise for the 2-level trie, provided that the stride in the 2nd level is at least 8 bits. The relative errors are listed in Table 3; we do not include the errors for the 1-level trie here when the stride is less than 8 bits. Except for the SDC trace, the prediction errors are typically within 10%. Our model consistently overestimates the cache misses incurred by the lookup procedure for the SDC trace. This is probably because the SDC trace has higher temporal locality and very few unique addresses, resulting in a few prefixes dominating others in the lookup table. (This breaks the assumptions in our model that the consecutive accesses to prefixes are independent, and that all cache blocks are equally likely to be accessed.) However, for such a trace, placing a software cache in front of the lookup data structure can significantly boost performance and extract out this locality.

Trace	Pentium II		Pentium III	
	1-level trie	2-level trie	1-level trie	2-level trie
R-Net	3.57, 6.32	7.67, 14.3	4.16, 14.4	8.66, 11.7
ISP	2.64, 4.69	4.85, 6.9	10.4, 19.9	2.14, 6.10
R-IP	11.2, 16.3	11.4, 16.9	9.1, 17.8	7.45, 18.0
SDC	15.3, 23.9	18.3, 29.2	13.9, 27.9	20.9, 28.8

Table 3: Percentage errors (average, maximum) in predicted running time relative to real running time on the different traces. Here R-Net is RandNet and R-IP is RandIP. For the 1-level trie, only cases with strides of 8 or more are considered.

We have also built the model and implementation of a 3-level trie, which gives a further performance improvement for some of the traces on the Pentium III. These results are discussed in Section 6.

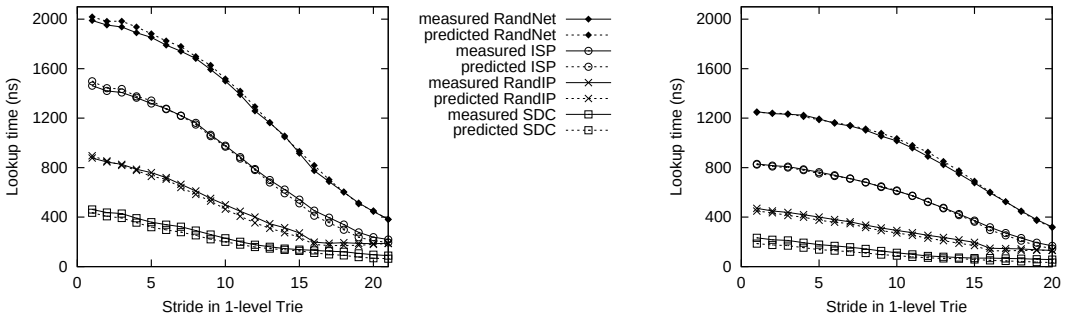
6. IMPLICATIONS AND APPLICATIONS

We now use the model to study the effect of the packet traces and machine architectures on the performance of IP lookup.

6.1 Finding the appropriate data structure

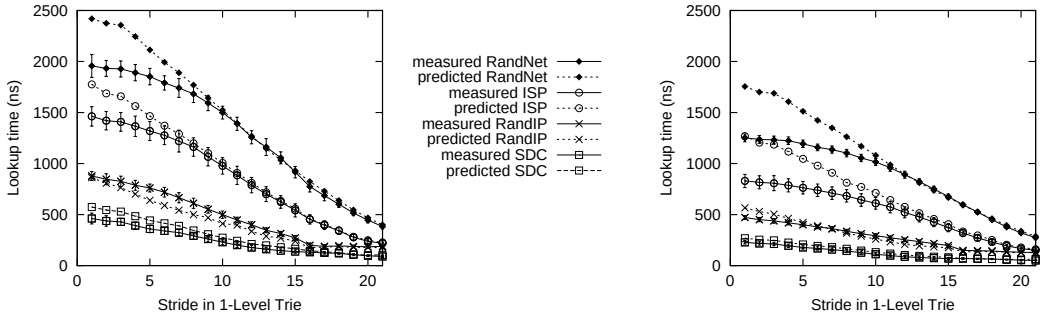
Researchers in the past have often used synthetic traces with IP addresses generated at random with uniform probability over the entire IP address space (as in our RandIP trace). Experiments with such data would suggest that a relatively small stride (say, 16 bits) would be sufficient. For example, for the RandIP trace, it is best to choose a stride of 16 bits at the top level of the trie (see Figures 8 and 9). In fact, this is often a popular choice while designing a multilevel trie. However, for more representative traffic (like the ISP trace), an initial stride of 20 bits gives better performance. Table 4 lists the optimal data structures determined for each trace; our performance model successfully chose the right data structures. The structures that work best for the synthetic traces are not optimal for the real traces. Table 5 shows the loss in performance when the optimal setting for one trace is used on another trace. The results indicate that the lookup time may increase by a fair amount if the data structure is not optimized for the type of packet trace being looked up.

We show in Figure 10 the performance of software IP lookup code by other researchers on our machines; we only report results for which we had the source code. The two methods



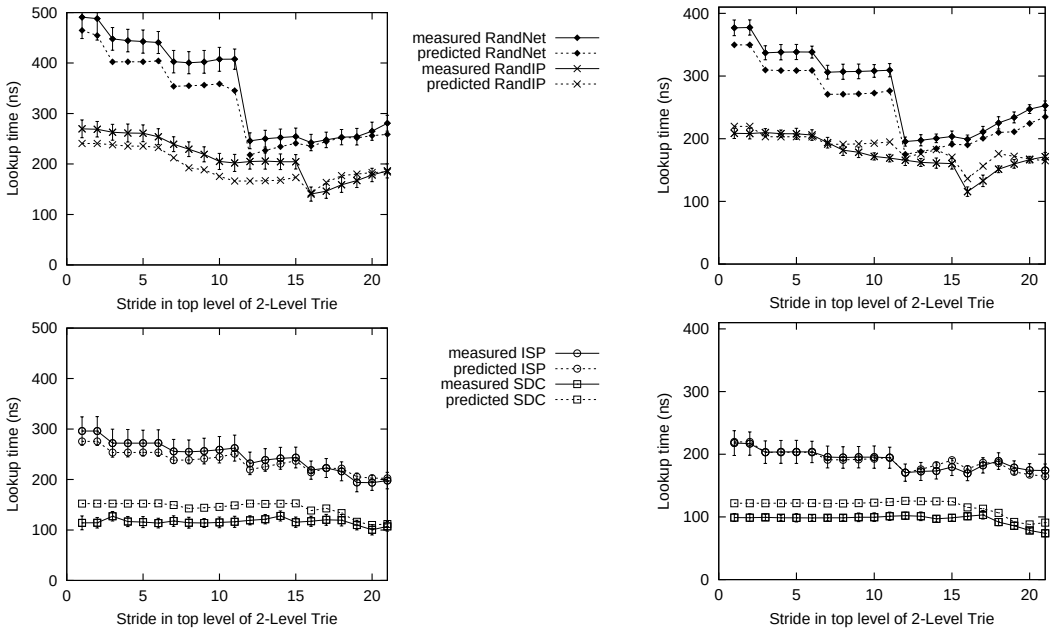
(a) Pentium II (b) Pentium III

Figure 7: The experimentally measured running time for the 4 traces Vs the predicted running time when the exact values of M_1, M_2, H and T are known. All numbers are for a one-level trie with splay trees at its leaves. The x-axis represents the stride of the 1-level trie.



(a) Pentium II (b) Pentium III

Figure 8: The experimentally measured running time for the 4 traces Vs the predicted running time when the values of M_1, M_2, H and T are predicted. All numbers are for one-level tries with splay trees at their leaves. The x-axis represents the stride of the 1-level trie. Error bars indicate standard deviations for measured times.



(a) Pentium II (b) Pentium III

Figure 9: The measured running time (with standard deviation error bars) for the 4 traces Vs the predicted running time when the values of M_1, M_2, H and T are predicted. All numbers are for two-level tries with splay trees at their leaves. The x-axis represents the stride in the top level of the trie.

Trace	Pentium II			Pentium III		
	Meas	Mod	Structure	Meas	Mod	Structure
R-Net	242	235	T2 (16,24)	168	164	T3 (16,24,28)
ISP	192	202	T2 (20,24)	131	149	T3 (21,24,27)
R-IP	140	142	T2 (16,24)	89	108	T3 (16,24,28)
SDC	89	104	T1 (21)	50	62	T1 (21)

Table 4: Optimal data structures for the traces, with lookup times in nanoseconds measured ("Meas") as well as predicted by our performance model ("Mod").

Input Trace	Trace for which lookup structure is optimized							
	Pentium II				Pentium III			
	R-Net	ISP	R-IP	SDC	R-Net	ISP	R-IP	SDC
R-Net	—	15.7	0	58.3	—	15.1	0	39.7
ISP	29.9	—	29.9	11.3	06.8	—	06.8	34.3
R-IP	0	33.1	—	35.0	0	13.7	—	44.9
SDC	31.5	20.4	31.5	—	28.1	09.0	28.1	—

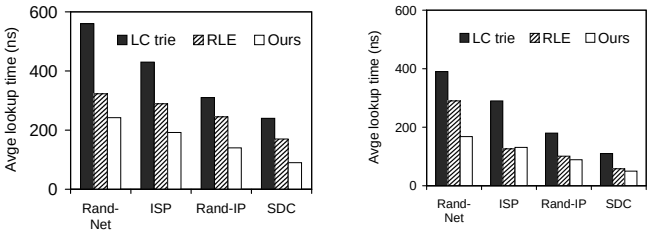
Table 5: Percentage increase in lookup time for each trace, when the data structure that is optimal for another trace is used to perform the lookup. The optimal data structures for each trace are listed in Table 4.

include the LC-trie [14] and the RLE next-hop compression scheme [5]. For the LC-trie we tried expanding 16 bits or 20 bits at the top level, and report the better of the two for each trace. The RLE compression scheme works very well on the Pentium II because they make use of repeated next hops in the route table to compress the lookup structure and get good L2 cache performance. We do not use any information on the next hop in our code, and can therefore store a pointer to any arbitrary information about each prefix in our data structure. We conjecture that ideas from their method could further improve our performance, since next hops do appear to be highly compressible. The results in Figure 10 indicate that the lookup performance of the optimal data structure chosen by our model is competitive with or higher than that of previous approaches. The method by Gupta et al. [8] is equivalent to using splay trees without the trie, since we find that splay trees perform approximately as well as weight-balanced binary trees. However, as shown in Section 5, the performance improves significantly by examining multiple bits in a trie before accessing the binary trees.

Number of trie levels. A 3-level trie works well on the Pentium-III with its smaller L2 cache, while the L2 cache on the Pentium-II is large enough to accommodate the frequently accessed nodes of a 2-level trie, thereby saving one additional access. The SDC trace always picks a 1-level trie because very few prefixes are hit while performing a lookup on the trace. This ensures that the trie nodes accessed are often cached, and whenever a splay tree is accessed, the prefix being searched for is often at the root of the splay tree.

6.2 Selecting the right architecture

Several alternative processor and memory architectures are available for performing IP lookups in software. They range from desktop PCs with small L1 and moderate L2 caches



(a) Pentium II (b) Pentium III

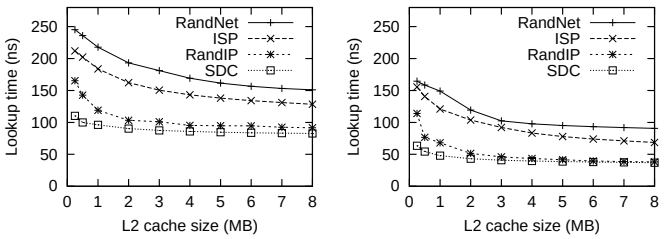
Figure 10: A comparison with lookup times for other software-based IP lookup schemes (in nanoseconds).

to specialized network processors with large off-chip caches. In this section we use our performance model to study the effect of the some of the architectural features that influence the performance of IP lookups.

6.2.1 Size of the L2 cache

The selection of the optimal data structure can depend on the amount (and speed) of the L2 cache. We therefore extended the size of the L2 cache to up to 8 MB, and studied the performance through our model. Current network processors support large SRAMs (eg, the IXP [9] can have up to 8 MB of SRAM), so this seems like a reasonable range. For each value of the L2 cache size, we show in Figure 11 the performance for the optimal data structure found by the model. The L1 cache was fixed at 16 KB, and the memory limit for the data structure was set to 32 MB. For all the traces, we find that increasing the L2 cache size from 256 or 512 KB to 2 or 3 MB provides a very large benefit; beyond that size, the performance is typically improved further by only about 10-15%. This implies that although the selected data structures are often bigger than 8 MB, only a fraction of that memory is frequently accessed. Because most networks are concentrated in small portions of the 2^{32} address space, many leaves of a trie that uses a large ($>$ 16 bit) stride do not contain any prefix, or contain prefixes that are hit very infrequently. The knee of the curve may shift to the right for larger tables. However, the actual prefixes only contribute a small fraction to the total space requirement for a large trie; therefore we do not expect this shift to be large. Things may look very different for IPv6, but the model could still be used in a similar way.

For the Pentium II, the optimal data structures selected look similar to those in Table 4. The only exception is the RandIP trace, which switches to a 3-level trie when the L2 size is 2 MB or more. This switch probably allows most of the common prefixes for RandIP (the shorter prefixes) to fit in the L2 cache. For the Pentium III, which has a very fast (on-chip) L2 cache, the traces that have a large number of accesses to very few networks (SDC and RandIP) prefer a 1-level trie for L2 caches beyond 512 KB. Even the ISP trace switches to a 1-level trie beyond an L2 size of 2 MB. This is because, unlike on the Pentium II, the additional L1 misses caused by using large strides in the 1-level trie can be more easily offset by the reduced number of L2 misses in the larger L2 cache. Only the RandNet trace prefers a 3-level trie, because it contains roughly equal accesses to all



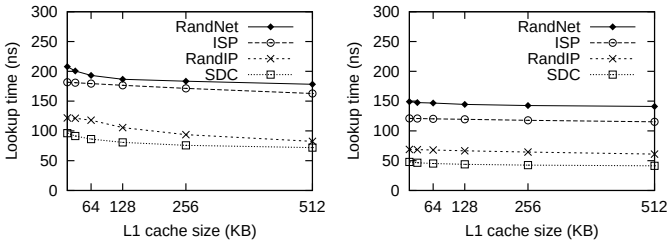
(a) Pentium II (b) Pentium III

Figure 11: The effect of L2 cache size on performance as predicted by our performance model.

prefixes, including very long prefixes; this keeps both the L1 and L2 cache miss rates high for a 1-level trie.

6.2.2 Size of the L1 cache

Figure 12 shows the effect on the lookup performance as the size of the on-chip L1 cache is increased, up to 512 KB. The L2 cache was fixed at 1 MB, and the memory limit was set to 32 MB. The L1 cache appears to have a far less significant effect on performance compared to the L2 cache, mainly because it remains too small to cache any significant portion of the data structure. The data structures selected by the model did not differ significantly from those listed in Table 4.

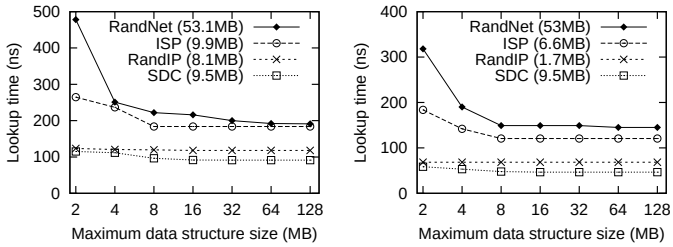


(a) Pentium II (b) Pentium III

Figure 12: The effect of L1 cache size on performance as predicted by our performance model.

6.2.3 Total size of the data structure

So far we have restricted our search to the set of data structures that are under a total of 32 MB in size; we felt this was a reasonable limit for a software implementation. Figure 13 shows the lookup performance as this limit is varied, from 2 MB up to 128 MB. It also shows the size of the optimal data structure when no memory constraint is applied. The L1 and L2 cache sizes were fixed at 32 KB and 1 MB, respectively. For traces that involve hits to a large number of prefixes (RandNet and ISP), the performance is significantly reduced when the data structure size is constrained to below 4 MB. Except for the RandNet trace, however, none of the traces require data structures of size more than 10 MB. Even for the RandNet trace, 85-95% of the performance can be reached using a data structure under 10 MB. Cache misses increase rapidly as prefixes are replicated in



(a) Pentium II (b) Pentium III

Figure 13: The effect performance of limiting the maximum space requirement of the lookup data structure, as predicted by our performance model. Each curve's label includes the size of the largest data structure chosen for the trace under no memory constraint.

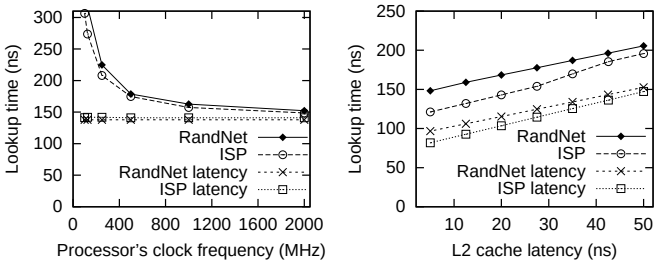
larger data structures (with larger branching factors), and offset any advantage obtained by reducing the total number of memory accesses.

The data structures selected under low memory limits (4 MB or less) vary widely across traces, and are often different from the ones listed in Table 4. For example, for the SDC trace on the Pentium II, a 3-level trie is preferred under 4 MB, while a 1-level trie with a large branching factor works best under larger memory limits. In contrast, the ISP trace consistently performs best with a 2-level trie on the Pentium II and a 3-level trie on the Pentium III (although the tries' branching factors vary).

6.2.4 Processor and cache speeds

Finally, we examine the effect of both processor and L2 cache speeds on the lookup performance. We fixed the sizes of the L1 and L2 caches at 32 KB and 1 MB, respectively. Figure 14(a) shows how the lookup time is reduced as the processor speed increases. As expected, with increasing processor speed, the lookup eventually becomes memory limited. The cache and memory latencies assumed are the same as the Pentium II. At moderate clock speeds, such as those of our Pentium II or of commercial networking processors, CPU overhead is still significant. This indicates that there may be room for further optimizing the lookup procedure; however, since our goal was to design and validate a performance model, we did not focus on optimizing to the last cycle. Also note that the processor time shown here includes time to read data from the L1 cache, which requires a few cycles. Therefore the results shown here assume that L1 speeds will increase in proportion to clock speeds.

Our two Pentium platforms have significantly different L2 latencies (10ns and 50ns). We decided to study the effect on lookup performance when the L2 latency is varied from 5ns to 50ns (see Figure 14(b)). The performance improves by 35-45% over this range. The latency for L2 and main memory remains between 65-75% of total lookup time over this range, again indicating that it may be possible to further optimize the CPU work out of the lookup procedure. The processor speed was fixed at 400 MHz.



(a) Pentium II (b) Pentium II

Figure 14: The effect on lookup performance of (a) processor speed, and (b) L2 latency on the lookup procedure. We also show the portion of the running time the processor spends waiting for data from the L2 cache or main memory (labelled "latency").

6.2.5 *Searching the space of hardware configurations*

Instead of varying one hardware parameter at a time, it would be useful to simultaneously search the space of parameters, including the sizes and latencies of the different caches and the processor speed. For direct-mapped caches ($a = 1$ in Equation 2), it is possible to express the average lookup time in terms of access counts to memory blocks in each candidate lookup data structure. The variables in such an expression are the above hardware parameters. Then, if the cost of the hardware can be expressed as a funtion of these hardware parameters, a non-linear optimizer can find the fastest hardware configuration for each lookup data structure under a given cost limit. Alternatively, the optimizer can find the cheapest hardware configuration for a specified lower limit on the average lookup performance.

7. CONCLUSION

We presented an analytical model that accurately predicts the performance of software-based IP lookups that use hierarchical data structures. Because realistic traces have very different characteristics from synthetic traces such as one with random IP addresses, it is important to find the optimal data structure for the type of traffic being processed. Our model can be used to select the appropriate data structure based on knowing the distribution of packet hits to different networks in the routing table. The model can also be used to select the appropriate processor and memory architecture to perform the lookups. For the traces and table we studied, we found that the size and latency of the L2 cache, and to some degree the processor speed, are critical in determining the resulting lookup performance. Increasing the L1 cache or space requirement of the lookup data structure does not significantly improve performance. Although we do not use highly optimized lookup procedures or highly compressed data structures, the data structures selected by our performance model yield good lookup speeds. Adding further optimizations or different trie schemes is likely to involve simple changes to our current performance model.

ACKNOWLEDGEMENTS

Brenda Baker, Eric Grosse, Dave Presotto, and the anonymous referees provided several useful comments on earlier versions of this paper. We would also like to thank Hans-Werner Braun for the SDSC packet trace.

8. REFERENCES

[1] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *Proc. IEEE Infocom*, pages 126–134, March 1999.

[2] A. Brodnik, S. Carlsson, M. Degermark, and S. Pink. Small forwarding tables for fast routing lookups. In *Proceedings of the ACM SIGCOMM Conference*, pages 3–14, September 14–18 1997.

[3] C-Port Network Processors. `http://www.cportcorp.com/`.

[4] G. Cheung and S. McCanne. Optimal routing table design of IP address lookups under memory constraints. In *Proc. IEEE Infocom*, pages 1437–1444, 1999.

[5] P. Crescenzi, L. Dardini, and R. Grossi. IP address lookup made fast and simple. *Lecture Notes in Computer Science*, 1643:65–76, 1999.

[6] W. Fang and L. Peterson. Inter-AS traffic patterns and their implications. In *Proceedings of Global Internet 99*, December 1999.

[7] D. Grinberg, S. Rajagopalan, R. Venkatesan, and V. Wei. Splay trees for data compression. In *Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 522–530, 1995.

[8] P. Gupta, B. Prabhakar, and S. Boyd. Near-optimal routing lookups with bounded worst case performance. In *Proc. IEEE Infocom*, pages 1184–1192, March 2000.

[9] Intel IXP Network Processor. `http://developer.intel.com/design/network/products/npfamily/`.

[10] Internet Performance Measurement and Performance Analysis Project. `http://www.merit.edu/ipma`.

[11] B. Kristnamurthy and J. Wang. On network-aware clustering of web clients. In *Proceedings of the ACM SIGCOMM Conference*, September 2000.

[12] B. Lampson, V. Srinivasan, and G. Varghese. IP lookups using multiway and multicolumn search. In *Proc. IEEE Infocom*, pages 1248–1256, 1998.

[13] L. McVoy and C. Staelin. LMbench: Portable tools for performance analysis. In *USENIX 1996 Annual Technical Conference*, pages 279–294, Berkeley, CA, USA, January 1996.

[14] S. Nilsson and G. Karlsson. Fast address look-up for internet routers. *IEEE Journal on Selected Areas in Communications*, pages 1083–1092, June 1999.

[15] NLANR Measurement and Operations Analysis Team. `http://moat.nlanr.net/Traces/Traces/`.

[16] K. Sklower. A tree-based packet routing table for Berkeley UNIX. In *Proceedings of the Winter 1991 USENIX Conference*, pages 93–104, Berkeley, CA, USA, January 1991.

[17] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, July 1985.

[18] V. Srinivasan and G. Varghese. Fast address lookups using controlled prefix expansion. *ACM Transactions on Computer Systems*, 17(1):1–40, February 1999.

[19] R. van der Wal. Programmer's toolchest: Source-code profilers for Win32. *Dr. Dobb's Journal of Software Tools*, 23(3):78, 80, 82–88, March 1998.

[20] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable high speed IP routing lookups. In *Proceedings of the ACM SIGCOMM Conference*, pages 25–38, September 14–18 1997.