Dynamic Pipelining: Making IP-Lookup Truly Scalable

Jahangir Hasan T. N. Vijaykumar

{hasanj, vijay} @ecn.purdue.edu

School of Electrical and Computer Engineering, Purdue University

Abstract

A truly scalable IP-lookup scheme must address five challenges of scalability, namely: routing-table size, lookup throughput, implementation cost, power dissipation, and routing-table update cost. Though several IP-lookup schemes have been proposed in the past, none of them do well in all the five scalability requirements. Previous schemes pipeline tries by mapping trie levels to pipeline stages. We make the fundamental observation that because this mapping is static and oblivious of the prefix distribution, the schemes do not scale well when worst-case prefix distributions are considered. This paper is the first to meet all the five requirements in the worst case. We propose scalable dynamic pipelining (SDP) which includes three key innovations: (1) We map trie nodes to pipeline stages based on the node height. Because the node height is directly determined by the prefix distribution, the node height succinctly provides sufficient information about the distribution. Our mapping enables us to prove a worst-case per-stage memory bound which is significantly tighter than those of previous schemes. (2) We exploit our mapping to propose a novel scheme for incremental route-updates. In our scheme a route-update requires exactly and only one write dispatched into the pipeline. This route-update cost is obviously the optimum and our scheme achieves the optimum in the worst case. (3) We achieve scalability in throughput by simultaneously pipelining at the data-structure level and the hardware level. SDP naturally scales in power and implementation cost. We not only present a theoretical analysis but also evaluate SDP and a number of previous schemes using detailed hardware simulation. Compared to previous schemes, we show that SDP is the only scheme that scales well in all the five requirements.

Categories & Subject Descriptors

C.2.6 [**Internetworking**]: Routers

General Terms

Algorithms, Design, Performance

Keywords

IP-lookup, Scalable, Pipelined, Tries, Longest Prefix Matching

1 Introduction

The pervasive use of the Internet and advances in fiber optics enabling high line-rates are resulting in an explosion in total traffic and in the number of hosts on the Internet. Studies have shown that the traffic is doubling almost every three months [18] and the number of hosts is tripling every two years [5]. These trends translate into two major problems for IP-lookup mechanisms in core routers. First, routers will soon need to look up their routing tables at the rate of about 2 ns per packet (for a 160 Gbps line-rate and minimum packet size of 40 bytes). Second, routers will have to search through a large number of prefixes in their routing tables (e.g., routers today hold a few hundred thousands of prefixes).

While the demand has been increasing, the supply has not been scaling up. The key component in IP-lookup is the routing-table memory which is used to search through the prefixes to locate the one that matches the incoming packet. The IP-lookup scheme has to satisfy five key scaling requirements: (1) Because memory size directly affects system cost, lookup speed and power dissipation, the total memory required should grow slowly with the number of prefixes. IP-lookup should scale well in memory size and be efficient in storing the ever-increasing number of prefixes. (2) IP-lookup must scale in throughput, forwarding packets at increasingly higher speeds to keep up with the ever-increasing line-rates. (3) To keep the complexity of heat removal and the cost of cooling reasonable, power dissipation of IP-lookup must scale well. The power should grow slowly with line-rates and number of prefixes, and avoid becoming prohibitive. (4) Because a routing table is typically unavailable for lookups during the time that it is being updated, applying updates should be quick and efficient. Though updates may be infrequent, a router with slow updates will require partial or full duplication of routing-table memory in order to avoid dropping an increasing number of packets as line-rates grow. Therefore, IP-lookup must allow simple, incremental, and fast updates independent of the routing table size. (5) IP-lookup must scale well in implementation cost and complexity to remain feasible for future table sizes and line-rates. Accordingly, the chip area should grow slowly with line-rates and number of prefixes. Because routers *must* provide worst-case guarantees for all the five aspects, meeting these requirements is especially hard.

We propose an IP-lookup mechanism which meets all the five scalability requirements in the worst case. The problem of scalable IP lookup is not new; there have been several papers on the topic [1][3][4][11][13][14][15][17][19] which may lead one to believe that the problem is well-researched, and satisfactorily solved. However, *all* previous schemes satisfy *only two or three* of the requirements but *not all five*. The unsatisfied requirements will likely render the schemes infeasible in the future. Meeting all the five requirements with worst-case guarantees for the first time is the key contribution of this paper. We not only present a theoretical analysis but also evaluate our scheme and a number of previous schemes using detailed hardware simulation.

Previous IP-lookup schemes can be classified into two categories: TCAMs and trie-based. We list their shortcomings here and explain the detailed reasons for the shortcomings in Sections 3.1,

SIGCOMM'05, August 21–26, 2005, Philadelphia, Pennsylvania, USA.

3.2, and 4. TCAMs do not scale well in power and implementation cost at high line-rates. Tries scale well in power but they do not scale well in throughput if they are not pipelined. Two approaches for pipelining tries are hardware-level pipelining (HLP) [15], and data-structure-level pipelining (DLP) [1]. HLP pipelines the routing-table memory at the hardware level. However, HLP does not scale well in power and implementation cost because it requires extremely deep pipelines at high line-rates. DLP pipelines the trie at the data structure level by placing each trie level in a different memory, so that different packets simultaneously probe different levels. Because DLP does not require HLP's deep hardware pipelining, DLP scales well in power and implementation cost.

However, DLP has three shortcomings: (1) DLP does not scale well in size due to large worst-case memory. (2) DLP's route-update cost can be made *O(1)* by using Tree Bitmap [4], if memory management overhead is ignored. However, Tree Bitmap almost doubles the worst-case memory size due to its inability to use leaf-pushing, and requires over 100 memory accesses, in the worst case, for a route-update if memory management overhead is considered. (3) DLP scales for throughput by partitioning the trie into pipeline stages. However, a trie cannot be partitioned into more stages than its total height. Hence, DLP's scalability in throughput is limited by the maximum height of the trie (i.e., the maximum prefix length), which is constant.

DLP pipelines the trie by mapping a specific prefix bit (i.e., a specific trie level), to a specific pipeline stage (e.g., the 12^{th} bit is mapped to the 2^{nd} stage). We make the fundamental observation that DLP incurs its first shortcoming because this mapping is completely *static* and oblivious of the prefix distribution. For instance, a trie node examining the 12^{th} bit remains mapped to the same stage irrespective of changes to the distribution caused by route updates. Depending on the distribution, as many nodes as all the prefixes may fall into the same level (or equivalently, same stage). Unfortunately, providing worst-case guarantees for *any* prefix distribution implies that most stages have to be large enough to hold as many nodes as *all* the prefixes. Thus the static mapping's obliviousness of the prefix distribution results in large per-stage memory which limits scalability in size and throughput.

To solve DLP's problems, we propose *scalable dynamic pipelining (SDP)* which takes prefix distribution into consideration. We map a trie node to its pipeline stage based on the node height (e.g., nodes of height 3 are mapped to the 8^{th} stage). Because the node height is directly determined by the prefix distribution, the node height succinctly provides sufficient information about the distribution. Node heights change when the prefix distribution changes upon route updates, causing our mapping to be *dynamic*. In contrast to the node height, the node level provides no information about the distribution. This dichotomy exists because the level is measured from the root whose position remains fixed whereas the height is measured from the leaves whose positions reflect the distribution. For instance, a trie node at height 3 is guaranteed to have at least three prefixes in its subtree as long as the subtree uses path compression to address a peculiar feature of tries. We exploit this guarantee to prove a significantly tighter bound on the worst-case per-stage memory than that of DLP. The height-to-stage mapping is our first innovation which addresses DLP's first shortcoming of scalability in size.

The above-mentioned peculiar feature of tries distorts the correlation between node height and distribution: In general, an internal trie node examines a few prefix bits (e.g., 4). Depending on the length of a given prefix, many internal nodes are traversed to match the prefix. If a subtree contains only one prefix, every node in the subtree has only one child, and the traversal goes through a string of such one-child nodes. Such strings artificially increase the height of the subtree's nodes, distorting the correlation. To remove this distortion, we propose a loss-less adaptation of path compression proposed by PATRICIA tries [9]. We collapse each string of one-child nodes into a *jump node*, which examines as many bits as the string length. Thus jump nodes restore the node's true height.

Upon route updates, the nodes whose heights are affected need to be migrated to the correct stage based on their new height. It may seem that our dynamic mapping would incur high route-update cost due to such migrations. Surprisingly, while our height-to-stage mapping causes this problem we exploit the very same mapping to solve the problem via a novel scheme for incremental route-updates. In our scheme, a route-update requires exactly and only one write dispatched into the pipeline (at every stage at most one memory write is done). This route-update cost is obviously the optimum for any pipelined scheme, and our scheme achieves the optimum in the worst case. In addition, our memory management overhead is exactly one operation. Though Tree Bitmap's [4] route-update cost including memory management is *O(1)*, the constant factor is $2^{\text{largest stride}} + 100$; whereas SDP's cost is exactly 1. SDP employs leaf-pushing and therefore, does not incur the size and throughput penalties of Tree Bitmap. The route-update scheme is our second innovation which addresses DLP's second shortcoming of scalability in route-update cost.

To attack DLP's third shortcoming, we make the key observation that each stage of a data-structure pipeline can be hardware-pipelined further (similar to [15]). We hardware-pipeline each SDP stage into a *different* number of hardware stages as per the desired access rate (e.g., SDP stage 2 has three hardware stages, SDP stage 3 has five hardware stages, and so on). Once we internally pipeline the data-structure stages at the hardware level, the throughput can continue to scale irrespective of the maximum height of the trie (i.e., the maximum prefix length). By combining hardware-level and data-structure-level pipelining, we avoid [15]'s high implementation cost and [1]'s lack of throughput scalability. This combining is our third innovation.

Using hardware simulation, we show that for 1 million prefixes at 160 Gbps line-rate, TCAM requires 6 MB, dissipates 174 W, and takes up 8.9 cm^2 (chip area is a measure of implementation cost); HLP requires 75 MB, dissipates 146 W, and takes up more than 200 cm^2; DLP requires 88 MB, dissipates 10 W, takes up 27 cm^2 and fails to work beyond 40 Gbps; In contrast, SDP requires only 22 MB, dissipates 22 W, and takes up 14.9 cm^2. Thus, SDP achieves the four goals of scalability in size, throughput, power and implementation cost. SDP's route-update cost, which is the remaining goal, is the theoretical minimum of one write.

The rest of the paper is organized as follows: In Section 2 we provide some background on IP-lookup mechanisms. We describe the details of HLP, DLP and SDP in Section 3. In Section 4 we briefly review TCAM-based schemes. We describe our evaluation methodology in Section 5. In Section 6 we present our results, and finally, we conclude in Section 7.

2 Background

Because we design our IP-lookup scheme based on tries, we first present some background details on IP-lookup and trie-based schemes.

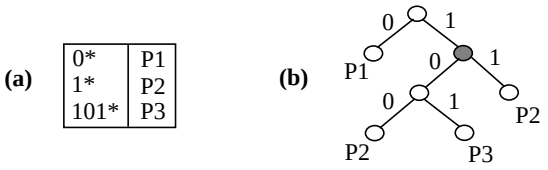

Fig. 1. (a) The prefixes in a routing table (b) a trie constructed from the given prefixes

The IP-lookup mechanism accepts an IP-address, performs a search-and-match through a routing table, and upon a match, returns the appropriate link identifier. The IP-lookup task is complicated by a number of requirements: (1) To avoid denial-of-service attacks and instabilities in the network [7], a router *must* sustain a worst-case IP-lookup throughput that can handle minimum sized packets streaming in at full line-rate. (2) Given the number of prefixes to design for, the IP-lookup mechanism *must* provide enough memory to hold all the prefixes regardless of their distribution. (3) Because of wildcard bits in prefixes, a given destination IP-address may match with multiple prefixes. IP routing protocols require that the lookup must choose the prefix with the longest match.

2.1 Trie-Based IP-lookup Schemes

One of the approaches to matching a destination IP-address against a set of given prefixes is to match it one bit at a time, narrowing the field of search with each successive bit. A *trie* is a tree-like data-structure designed specifically for such bit-by-bit searching. For example, given the set of prefixes shown in Figure 1(a) we can construct the trie shown in Figure 1(b). Each leaf contains the longest matching prefix corresponding to the bits encountered along the path from the root to that leaf. We perform an IP-lookup by starting at the root and traversing down the trie. At any internal node of level k (root being level 0), the k^{th} bit (bit 0 being the most significant) in the destination IP-address determines whether to follow the left child or the right child. The trie traversal eventually ends at a leaf.

Starting from the shaded node in Figure 1(b), any path that corresponds to a mismatch with prefix P3 must be terminated with a leaf containing prefix P2 (i.e., the longest prefix that has already been entirely matched). This method of constructing the trie is called *leaf pushing*. Unfortunately, updating a leaf-pushed trie may be complicated (e.g., if P2 is deleted or modified).

It is possible to construct tries without leaf pushing by placing prefix information inside internal nodes. However, such schemes almost double the trie node size, resulting in considerably larger worst-case memory. The bandwidth demand on the memory is also increased as the lookup process must read both a prefix and pointer at each node. In addition, as we traverse down the trie, we must explicitly check for and remember the longest matching prefix at each internal node.

2.2 Multiple-bit Stride Tries

When the IP-lookup is not pipelined in any manner, the total delay for one IP-lookup determines the maximum lookup rate. The worst-case delay for one lookup is proportional to the trie depth. IP-lookup rate can be improved by reducing the trie depth which in turn can be reduced by *striding* more than one bit at each internal node. If the stride is 2 bits at each internal node, the worst case depth of the trie is reduced by a factor of 2 (i.e., 16), and so on.

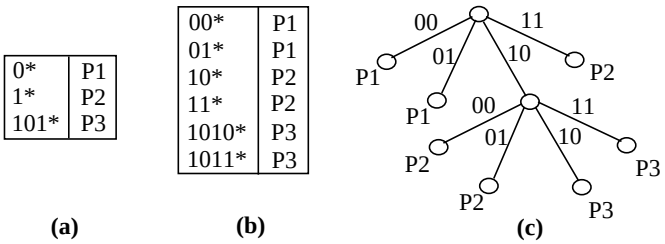

Fig. 2. (a) The routing table after controlled prefix expansion (b) The 2-bit stride trie constructed from the table in (a)

In Figure 2(a), prefix P1 has a length of only one bit. If we wish to stride 2 bits at the root, P1 must be expanded into all the 2-bit combinations implied by the original prefix P1. The process of expanding prefixes in order to align them with stride boundaries is called *controlled prefix expansion* [17]. Figure 2(b) shows the routing table of Figure 2(a) after controlled prefix expansion for a 2-bit stride at each node.

Unfortunately, controlled prefix expansion causes a non-deterministic increase in the routing-table size due to replication of pointers and prefixes, and consequently increases the total memory space. The size of the routing-table in Figure 2(b) is twice that of the table in Figure 2(a) for the same original number of prefixes. If the underlying 1-bit trie is sparse, controlled prefix expansion will inadvertently inflate the data structure's size. Striding multiple bits also aggravates the route-update cost in leaf-pushed tries.

In the example we have presented, each internal node has the same stride. However, using the same stride is neither necessary nor optimal in terms of storage space or lookup delay. *Variable stride* tries and *Level Compressed* (LC) tries [11] determine the stride at each node in accordance with whether the trie is sparse or dense at that particular location. In contrast, compression schemes like *Lulea* [3] and *Tree Bitmap* [4] maintain a fixed stride trie and compress away the redundant replication instead. Tree Bitmap may be additionally extended to support variable strides. However, for a worst-case prefix distribution, variable striding and compression-based schemes do not benefit the total memory size much, as we discuss in Section 6.

2.3 The Need for Pipelined Tries

Tomorrow's routers will have to perform IP-lookups into routing tables of hundreds of thousands of prefixes, at the rate of a few nanoseconds per lookup. With such a large number of prefixes, trie-based schemes require such a large amount of worst-case memory that performing even one memory access may take longer than the packet inter-arrival time. The problem is aggravated by the fact that trie-based schemes perform multiple memory accesses for one lookup. To meet the demand for high lookup rates under such constraints, we obviously need to pipeline IP lookup so that performing multiple lookups in parallel delivers a net lookup rate that meets the demand.

3 Pipelined and Scalable IP-Lookup

The observation that pipelining can be used to solve the scalability problem of IP-lookup is not new. Previous proposals [15] [1] have addressed this problem with some form of pipelining. [15] can be thought of as a hardware-level pipelined (HLP) scheme, whereas [1] can be thought of as a data-structure-level pipelined (DLP) scheme.

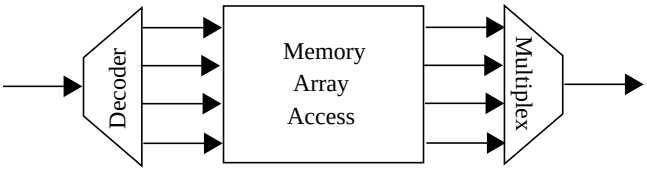

Fig. 3. The hardware steps involved in a memory access

We describe these schemes to explain why they do not scale well and then explain our scheme and how it scales.

3.1 Hardware-Level Pipelining

We can view the IP-lookup process as *k* memory accesses, where *k* is the number of levels in the multi-bit trie. For a given line-rate we know the required IP-lookup rate, say one lookup every *t* seconds. For the given number of prefixes we can determine the total memory required by the trie, and hence *d*, the total delay of one memory access. In order to meet the demanded lookup rate, HLP [15] hardware-level pipelines the entire memory holding the trie into $k*d/t$ stages.

Figure 3 shows, at a high level, the hardware steps involved in accessing a memory. Memory is typically organized as a two-dimensional array. The *decode* step uses *x* higher-order bits of the *x+y*-bit memory address to identify which of the 2^x rows is being accessed. The *memory array access* step performs the actual access of the chosen memory row. And the *multiplex* step selects the desired word from the 2^y words in the row and feeds it to the output. To optimize access times, circuit designers subdivide the memory array into many subarrays. Using the subarrays reduces memory (sub)array access time but increases decode and multiplex times for an overall reduction in access time. The decode and multiplex steps essentially look like decision trees and they can be pipelined into smaller stages by splicing up these trees. The memory-array-access step consists of reading from (writing to) the memory cells to (from) bitline wires. Because designing the bitline wires to carry multiple values is hard, for all practical purposes this step is atomic and cannot be pipelined. Therefore, even if decode and multiplex steps are pipelined into many fast stages, the throughput would be limited by the delay of the memory-array-access step. The time taken to perform this atomic step is proportional to the size of the memory array.

To reduce the delay of the atomic step, HLP [15] aggressively divides the memory array into a larger number of smaller subarrays. Such division does not come for free, however. It makes the decode and multiplex complicated, and does not scale well in terms of power dissipation and implementation cost. As we show in the experimental evaluation, such aggressive pipelining leads to prohibitive chip area (implying high implementation cost) and power dissipation. Therefore, HLP is not a scalable solution.

3.2 Data-Structure-Level Pipelining

We have seen that if the entire *k*-level trie resides in one large memory, then the bandwidth demanded by that memory is *k* times the lookup rate needed. To solve this problem, DLP [1] places each level of the trie in a different memory, so that each memory is accessed only once per packet lookup. Therefore, the bandwidth that each memory must individually supply does not incur the factor-of-*k* multiplier.

Because DLP partitions the trie data-structure such that each level is placed in a separate memory stage, DLP can overlap the lookups for multiple packets by accessing different levels (in different memories) for different packets at the same time. Thus, DLP dissociates the lookup rate from the total delay of one lookup. Because DLP does not rely on expensive memory technologies or deep hardware pipelining, it scales well in power and implementation cost. There are, however, three remaining challenges that must be addressed in order to make DLP truly scalable, namely: scalability in memory size, in route-update cost, and in lookup throughput.

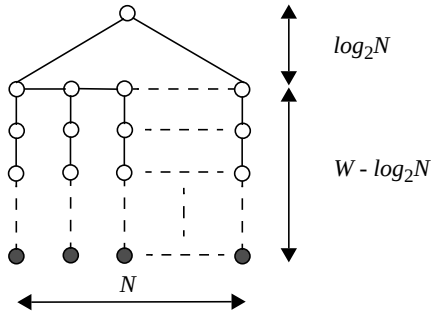

Fig. 4. The 1-bit trie corresponding to the worst-case prefix distribution. *N* is the number of prefixes, *W* is the length of an IP-address in bits

3.2.1 DLP's Scalability Problems in Memory Size

The total memory requirement of DLP is the sum of the memory size of each stage. In order to provide worst-case guarantees, the space provided at each memory stage should be sufficient for *any* prefix distribution. Because DLP assigns each node to a stage based on which trie level the node belongs to (i.e., which bit each node examines), the worst-case per-stage memory size is determined by the worst-case node count per level of the trie. It is important to note that the well-known average-case properties of randomly-built trees are not relevant here because we are concerned with worst-case guarantees.

In the ensuing analysis of worst-case per-stage memory size for DLP, we consider a 1-bit trie for simplicity. Striding multiple bits causes inflation in memory size due to controlled prefix expansion, and will not lower the worst-case bound. As such, our bound applies to multibit tries as well.

Imagine a prefix distribution in which all *N* prefixes have length *W* (*W* being the length of an IP-address), and the first log_2N bits of each prefix are unique (i.e., the prefixes cover all *N* values that the first log_2N bits can take). The 1-bit trie corresponding to this prefix distribution is shown in Figure 4. To establish that this trie, indeed, represents the worst-case memory requirement at each level, we make two observations: (1) In a 1-bit trie, each node can have two children, therefore no prefix distribution can have more than 2^k nodes at the k^{th} level. (2) There are only *N* prefixes therefore no level can have more than *N* nodes. Thus we see that the 1-bit trie in Figure 4 does in fact represent the worst-case memory requirement per level. Accounting only for the rectangular bottom-half of the trie in Figure 4, we see that the total memory required by this trie is greater than $N * (W-log_2N)$ nodes. It is important to note that the rectangular part is not due to leaf-pushing, rather it is a result of the specific prefix distribution. For a million prefixes DLP's memory requirement exceeds 80 MB, in contrast the storage needed for just the prefixes is only 6 MB, illustrating DLP's scalability problem in memory size. Note that, though a variable-stride trie may reduce

total space in the average case, for the prefix distribution shown in Figure 4 its worst-case memory size would be no better.

3.2.2 DLP's Scalability Problems in Route-update Cost

Because DLP uses a multibit trie with leaf pushing, a single route-update may affect an entire subtree which has arbitrarily many nodes. [1] proposes a number of optimizations for applying fast incremental route-updates in a pipelined fashion. However, all the optimizations are heuristics which improve only the average-case route-update cost. The worst-case route-update cost of DLP remains unbounded even with the optimizations.

Techniques like Tree Bitmap [4] can be used to achieve an *O(1)* bound on the route-update cost. By avoiding leaf-pushing Tree Bitmap ensures that an update needs to modify only one trie node achieving the *O(1)* bound. However, because Tree Bitmap cannot use leaf-pushing, it almost doubles the size of each trie node (see Section 2.1). [4] explains an implementation to avoid the doubling of the node size, where only the pointers are stored in the nodes and the prefixes are stored in a parallel copy of the trie. Obviously, the second copy must also be maintained in fast memory (as it must be accessed at IP-lookup rates), almost doubling the total memory size. Further, the trie nodes in Tree Bitmap have variable sizes due to variation in strides and compressions. Route-updates result in repeated allocations and deallocations of such variable sizes, causing fragmentation and under-utilization of memory. This fragmentation necessitates a complex memory management scheme for compaction [4][14], which must be invoked whenever memory for a new node is allocated. The memory accesses for the compaction appear as an overhead in the route-update cost. We found that the worst-case memory management overhead of Tree Bitmap [4] exceeds 100 memory accesses for a single route-update. ([4] reports 1852 memory accesses based on an analysis which is more conservative than ours.) Though a pipelined update scheme such as [1] could be leveraged to reduce the effective compaction cost, such reduction would be sensitive to the distribution of the memory accesses across the pipeline stages. In the worst case there may be no reduction at all. Hence, we see that previous schemes do not scale well in worst-case route-update cost.

3.2.3 DLP's Non-Scalability in Throughput

Because the pipeline's throughput is limited by the slowest stage DLP proposes a dynamic programming algorithm to minimize the size of the largest stage. This algorithm takes as inputs a prefix distribution and the number of levels in a fixed-stride trie, and returns the strides for each level such that the size of the largest level is minimized. The size of the largest stage can be lowered by increasing the number of levels in the trie (i.e., reducing the stride at each level). In the limit, even if each level strides only one bit, there can be only as many levels as bits in an IP-address (32). With 1 million prefixes, the 1-bit trie shown in Figure 4, has a largest memory stage of 5 MB which, realistically speaking, may not be accessed faster than 6 ns or so. When the demanded lookup-rate exceeds this limit, DLP does not work. For truly scalable throughput the depth of the pipeline should not be limited by the number of bits in an IP-address (32).

3.3 Scalable Dynamic Pipelining

To address the problems of IP-lookup scalability, we propose *scalable dynamic pipelining (SDP)*. We begin by taking a closer look at why traditional tries have such a large worst-case memory

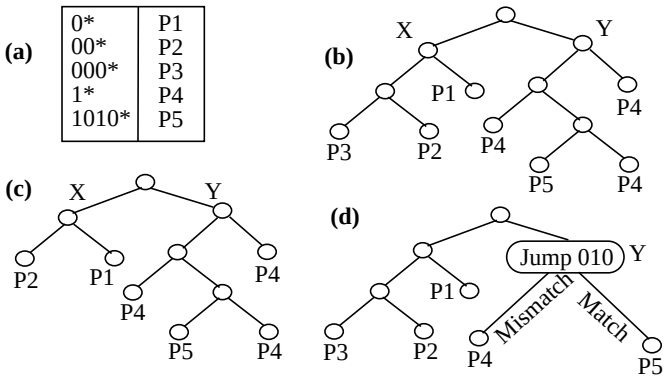

Fig. 5. (a) A table of prefixes (b) The corresponding 1-bit trie (c) The 1-bit trie with P3 deleted (d) The trie with jump nodes

requirement. We observe that previous schemes like [1] pipeline the trie by mapping a specific level of the trie to a specific stage. This mapping is strictly *static* and oblivious to the prefix distribution. Consider, for example, the set of prefixes shown in Figure 5 (a), and the corresponding 1-bit trie in Figure 5 (b). The node labelled *X* is in the second level of the trie and hence placed in the second stage of the pipeline. Imagine that we remove prefix P3 from the table, the resulting trie is shown in Figure 5 (c). Even though the structure and the memory requirements of the subtree rooted at *X* have changed significantly, *X* remains mapped to the second stage of the pipeline, oblivious of this change.

We make the key observation that while the *level* of *X* does not change, the *height* of *X* does change in response to the new prefix distribution (height of leaves being zero). This dichotomy exists because the height is measured from the leaves whose positions reflect the distribution, whereas the level is measured from the root whose position remains fixed. Specifically, the height of *X* is 2 in Figure 5 (b) and becomes 1 in Figure 5 (c). We see that the height of *X* is correlated to the number of prefixes in the subtree rooted at *X*. Because the node height is directly determined by the prefix distribution, it succinctly provides information regarding the distribution which is sufficient for achieving a tight worst-case bound on memory. However, there is one peculiar feature of tries which can distort the correlation between node height and prefix distribution. We first turn our attention to this distorting feature before presenting an analysis of worst-case per-stage memory size for SDP.

3.3.1 Jump Nodes

The way a trie is constructed, an internal node that strides *k* bits must have an array of 2^k pointers, one for each possible child. Often there may be only one child and the remaining pointers are null (leaf pushing may eventually insert a longest matching leaf in place of such nulls). In Figure 5 (a) and (b) the prefix distribution is such that it results in a long string of one-child nodes (ignoring the leaf-pushed copies of P4). The height of node *Y* is 3, though the number of unique prefixes in the subtree rooted at *Y* is just 2. The presence of the string of one-child nodes artificially increases the height of *Y*. Because the correlation becomes distorted in such a case, the height of *Y* does not faithfully inform us about the underlying prefix distribution.

To address this problem we collapse strings of one-child nodes into a single *jump node*. We call it a jump node because it allows the lookup to jump over the string of one-child nodes. Jump nodes are similar to *skip nodes* in [4] and can be thought of as an adaptation of

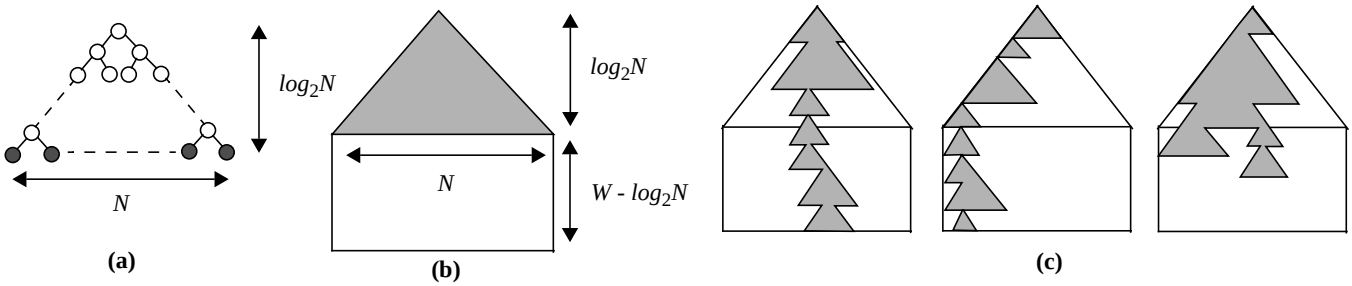

Fig. 6. (a) A binary search tree with *N* leaves (b) memory size of a trie with jump-nodes for the worst-case prefix distribution of Figure 4, compared to size of 1-bit trie (c) The space taken at various levels by a trie with jump-nodes, for various prefix distributions

path compression in PATRICIA tries [9]. A jump node collapses strings using *loss-less compression* by storing and matching all the jump bits. In contrast, PATRICIA tries use a form of *lossy compression* that examines only the first bit out of the string being compressed.

In addition to pointers, a jump node also stores the *jump bits*, the string of bits corresponding to the path collapsed. A jump node need store only two pointers, one for the path that matches the jump bits, and the other for a mismatch. Because any node may be a jump node, the default size for every SDP node must budget for the jump bits in addition to two pointers. Because a string of one-child nodes may have any arbitrary length, each SDP node must budget for the maximum number of jump bits possible (i.e., 32). Although the space for jump bits causes an increase in the overall size of every node, we show in Section 6.2 that the drastic reduction in the number of worst-case per-stage nodes (due to dynamic pipelining), dominates this increase to result in a much lower worst-case total memory bound compared to a statically pipelined trie. The increase in trie-node size also increases the bandwidth demanded from the memory. However, because 1-bit trie nodes are small compared to multibit trie nodes, the eventual bandwidth demand stays relatively small.

Figure 5(d) shows the trie of Figure 5(b) after the string of one-child nodes has been collapsed into a single jump node. In Figure 5(d) the height of *Y* has been reduced to 1 which is correlated to the number of prefixes in the subtree rooted at *Y*. Thus, jump nodes remove the artificial increase in height due to strings of one-child nodes, and restore the correlation between node height and prefix distribution. On a tangent, note that the jump node also removes all but one copy of the leaf-pushed node P4. Because jump-nodes remove one-child nodes, in a 1-bit trie they effectively remove all the nulls that leaf-pushing would try to fill up. Consequently, a route-update modifies *only one* trie node and does not propagate down to entire subtrees.

The use of jump nodes results in two key properties which we will use in Section 3.3.2 to prove a worst-case per-stage memory bound for dynamic pipelining: (1) Because we stride only one bit at trie nodes, there is no controlled-prefix-expansion, and hence no replication of the same prefix. Jump nodes remove nulls, eliminating unnecessary copies of leaf-pushed nodes. Thus, the number of leaves in the data-structure is equal to the number of prefixes. (2) In SDP every internal node, whether that be a 1-bit trie node or a jump node, is guaranteed to have two children.

3.3.2 Per-Stage Memory Bound

As observed in [11], before pipelining, the total memory required by a trie with jump nodes does not exceed the number of nodes in a binary search tree (i.e., 2*N* for *N* prefixes), as shown in Figure 6(a). However, it is important to realize that bounding the total memory does not bound the per-stage memory. Figure 6(b) shows the total memory required by an *unpipelined* trie with jump nodes (shaded triangular region) compared to the total memory required for a 1-bit trie (the containing boundary), for the worst-case prefix distribution shown in Figure 4. However, as Figure 6(c) illustrates, when the trie is pipelined by partitioning it across stages, the per-stage memory usage varies greatly depending on the prefix distribution. In order to provide worst-case guarantees, the space provided at each memory stage should be sufficient for *any* prefix distribution. If we were to assign a node to a particular memory stage based on which level of the trie it belongs to (as DLP does), then even with jump nodes the worst-case per-stage memory requirement remains equal to the impractical size derived in Section 3.2.1.

As noted in Section 3.3, the height of nodes is correlated to the prefix distribution. If we assign nodes to pipeline stages based on the height of the nodes, then we expect to obtain a tighter per-stage memory bound. Formally, the height *h* of an internal node is defined to be the length of the longest path from that node to any of the leaves below it. ([14] briefly discusses how tries may be pipelined and suggests mapping levels to stages, but the paper erroneously uses the term *height* when it actually means *depth*.) For example, the height of the shaded node in Figure 7(a) is 4. Let *W* be the number of bits in an IP address. *W* is both the maximum height of the trie, and the total number of memory stages in the dynamic pipeline. Utilizing the two key properties stated in Section 3.3.1, we now prove a bound on the worst-case per-stage memory size for dynamically pipelined tries with jump nodes.

Lemma 1: The number of leaves in a subtree rooted at a particular node is no less than the height of that node.

Proof: If the height of a node is *h*, then there is at least one path *P*, from that node to some leaf, that is *h* nodes long including the leaf. Figure 7(b) shows such a path for some arbitrary node. There are *h-1* internal nodes along *P*. For each of the *h-1* internal nodes, there is an alternate path that could be taken instead of *P* when traversing the trie. In Figure 7(b) we indicate, as shaded nodes, the first node along every such alternate path. Each shaded node must either be a leaf

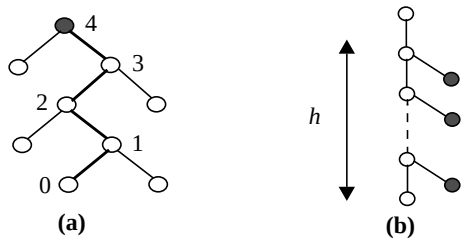

Fig. 7. (a) The height of a node (b) The relation between height and the number of leaves beneath a node.

itself, or must be an internal node that leads to at least one leaf. Because all leaves are unique, the alternate paths must contain at least *h-1* leaves in total. Thus the number of leaves in the entire subtree is at least *h*.

Lemma 2: Given any distribution of *N* prefixes, there can be no more than *N*/*h* internal nodes with height *h*.

Proof: In an SDP trie there are as many leaves as there are prefixes. Therefore the total number of leaves in the trie can be no more than *N*. From lemma 1 we know that each node of height *h* accounts for at least *h* unique leaves. Assume that the number of nodes with height *h* exceeds *N*/*h*, then there must be more than *N* leaves in total, contradicting the initial property of *N* leaves. Therefore, the number of internal nodes with height *h*, can be no more than *N*/*h*.

Theorem 1: If we assign all nodes with height *h*, in an SDP trie, to the $(W\text{-}h)^{th}$ pipeline stage, then we need to provide space for only $min(\ N/(W\text{-}k)\ ,\ 2^k\ )$ nodes at the k^{th} pipeline stage.

Proof: We need to prove two bounds — $N/(W\text{-}k)$ and 2^k— in order to obtain the expression given in Theorem 1. To obtain the first bound, we observe that if all nodes with height *h* are assigned to the $(W\text{-}h)^{th}$ stage, then from lemma 2 it suffices to provide *N*/*h* space at the $(W\text{-}h)^{th}$ pipeline stage. In other words, for the *k*th stage, it is sufficient to provide space for $N/(W\text{-}k)$ nodes. To obtain the second bound, we recall that internal nodes in an SDP trie have two children, therefore the total number of nodes at the k^{th} *level* of the trie cannot exceed 2^k. But, we need to establish that the number of nodes at the k^{th} *stage* cannot exceed 2^k. Each node along a path from the root must lie in a different stage, therefore an internal node at level *k* of the trie cannot fall in a stage earlier than the k^{th}. Equivalently we can also say that an internal node at level *k* of the trie cannot have a height of more than *W-k*, which means it will not get placed in a stage earlier than *k*. Therefore, the space requirement of the k^{th} stage is no greater than space requirement of the k^{th} level of the trie, thus proving the second bound. For any value of *k*, we need to provide only as much space at the k^{th} stage as the minimum of the two bounds, which proves Theorem 1.

Assigning nodes to stages based on their height is our first innovation. For 1 million prefixes, for instance, the worst-case total memory required by SDP is just 22 MB, a four-fold reduction over the latest static pipelining scheme [1]. We now to briefly describe the overall system architecture for SDP.

3.3.3 System Architecture

SDP is implemented using *W* stages (where *W* is the number of bits in an IP-address), each consisting of an SRAM memory which is sized in accordance with the results of Section 3.3.2. An IP-lookup is provided with the location of the root of the trie, and it is dispatched into the first stage of the pipeline. The lookup performs "NOPs" until it reaches the stage containing the root node. In addition, the lookup also performs "NOPs" in the intervening stages when the heights of a node and its child differ by more than one. When the lookup emerges off the end of the pipeline, the IP-lookup has completed. The pipeline concurrently sustains as many lookups in flight as the number of stages. We will now explain the mechanism of applying route-updates.

To update the SDP trie upon route changes, we need to maintain information about the height of the nodes. However, keeping the node heights and other auxiliary information within the trie itself would increase its size and slow down the lookup rate. Further, we would require frequent interruption of the IP-lookup stream in order to examine or modify this auxiliary information. To address this issue, we borrow the idea of a *shadow trie* from [1]: a copy of the trie containing all the required auxiliary information. The shadow trie is accessed only during the construction or update of the trie. Because route-updates are orders of magnitude less frequent than lookups, not only is it unnecessary to pipeline the shadow trie, but we can implement it using slow and cheap memory (DRAM). Today the cost of 128 MB of high performance DRAM is so trivial that the addition of a shadow trie has no effect on total system cost. Meanwhile, all IP-lookups are performed on the fast, pipelined trie itself.

When the router receives route-updates, we first apply them only to the shadow trie, modifying the data-structure in accordance with the route-updates. Because the modifications access only the shadow trie and the IP-lookups access only the SDP trie, they can both proceed concurrently without interrupting each other. Following the modification of the shadow trie, we compute the eventual changes that are required in the SDP trie. The required changes are formulated into node-writes and are then dispatched to the SDP trie. To apply the changes, we borrow a pipelined *write-bubble* scheme from [1]. In this scheme, a write operation interrupts the stream of IP-lookups by using up the turn of a single IP-lookup. The write operation marches down the pipeline stage-by-stage just like an IP-lookup, except that it performs writes instead of reads. Further, while the write operation is in a particular stage, IP-lookups can access the other stages. This observation allows us to dispatch writes into the pipeline, interleaved with lookups. A write operation is simply equivalent to a "bubble" in the lookup stream. However, the writes must obviously be performed in a manner which ensures that no read operation may encounter the data-structure in an inconsistent or erroneous state. We address this concern after we analyze the cost of pipelined incremental route-updates.

3.3.4 Optimum Cost Incremental Route-updates

The cost of route-updates can be represented by the IP-lookup throughput that is lost to write-bubbles. When a route-update is applied to a trie, it generally causes the insertion or removal of nodes, and can obviously change the height of a number of nodes. We first apply the route-updates to the shadow trie and recompute the heights of affected nodes. Then, in order to maintain the height-to-stage mapping of SDP, we migrate every node whose height has changed to the stage that corresponds to its new height. In addition to the node migrations, a route-update also results in the creation (deletion) of a node for the prefix being added (removed). Recall that because SDP uses a 1-bit trie with jump-nodes, any route-update needs to modify (including insert or remove) *only one* trie node (Section 3.3.1). We refer to this node as the *prefix-node*. Together, the migrations and the prefix-node modification account for the total cost for a route-update. It may seem such migrations may hurt the worst-case route-update cost. However, we make two key observations which enable us to bound the cost of *any* route-update by the optimum of strictly one write-bubble only.

Our first key observation is that, by virtue of the very definition of height, the insertion or deletion of a node can affect the height of *only* its ancestors, and *cannot* affect the height of any descendants.

A node insertion (deletion) may increase (decrease) the height of all its ancestors. In the worst case the number of affected ancestors can be *W-1* (i.e. the maximum height of the trie minus one). Our second key observation is that the prefix-node itself and all the affected ancestors, each belong in a uniquely different stage, both before and after the migration. Hence, a route-update requires one write to every stage of the pipeline in the worst case. Just as an IP-lookup

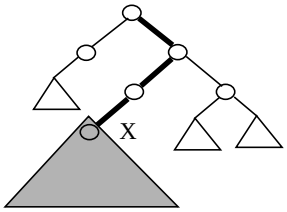

Fig. 8. Difference between the update cost of SDP and that of leaf-pushed trie schemes

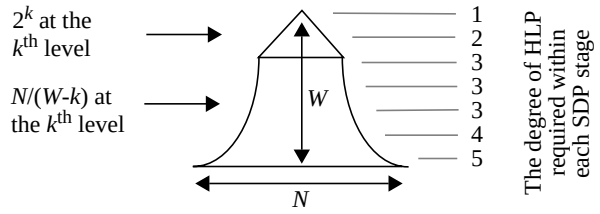

Fig. 9. An example of the per-stage memory requirement for SDP and the corresponding degree of HLP

can perform a read operation in every stage of the pipeline, a write-bubble can perform a write in every stage of the pipeline. Therefore, we can send a single write-bubble into the pipeline and migrate the ancestors to their new stages and write to the prefix node. Obviously, this single write bubble represents the optimum route-update cost in any pipelined IP-lookup scheme.

The write-bubble itself does not contain all the data that are to be written to various stages. The write-bubble simply reserves each pipeline stage for one cycle, ensuring that no lookup is accessing that stage. Typically a lookup processor or custom logic performs the necessary computation at each memory stage during a lookup. The same processor or logic is responsible for supplying the new data through the data bus of each memory stage, when the write-bubble reaches that particular stage.

Unlike multibit trie nodes, the size of 1-bit trie nodes is small and constant, therefore our assumption that a single memory write operation is wide enough to write an entire trie node is justified. In SDP a single write-bubble is sufficient for handling the worst-case route-update. Note that the conclusion of this cost analysis is significant: node migrations are literally *free*; SDP reduces a seemingly unbounded factor in route-update cost to the equivalent of a non-existent factor. Thus, we exploit the dynamic height-to-stage mapping to obtain both scalability in total memory size, and optimum route-update cost.

Figure 8 illustrates an intuitive way to understand this marked difference between the update cost of SDP and that of leaf-pushed trie schemes. In SDP the region that is affected by an update to node *X* is only the highlighted path from node *X* to the root. In contrast, in a leaf-pushed trie scheme an update to node *X* must be propagated down into the entire shaded subtree.

Recall that we mentioned at the end of Section 3.3.3 that when write-bubbles interleave with IP-lookups we must never allow the IP-lookups to read the SDP trie in an inconsistent state. Specifically, each pointer must be valid when dereferenced. We can trivially fulfil this requirement by observing that a write-bubble modifies at most one node in each stage. Because only one node is re-written in any given stage, the stage previous to it contains only one pointer that can be potentially invalid. When a write-bubble modifies a pointer in a node in stage *s*, only the lookups that are upstream to the write-bubble observe the modified pointer. After modifying the pointer the write-bubble arrives into the next stage (*s+1*) and writes out the new node being pointed to. By the time an upstream lookup arrives into stage *s+1* and dereferences the pointer in question, the write-bubble has already written out the new node. Thus, we guarantee data-structure consistency. The lookups that are downstream to the write-bubble read and dereference the pointer before it is ever touched by the write-bubble.

3.3.5 Memory Management Overhead

The process of applying route-updates allocates and deallocates memory for inserting and deleting nodes in the trie. Hence, the ability to apply incremental route-updates necessitates a memory management scheme for the routing-table memory. Because memory management appears as an overhead in the route-update cost, an IP-lookup scheme must include the worst-case memory management overhead for obtaining its *true* worst-case route-update cost. Previous IP-lookup schemes use multibit tries, often with some compression mechanism [3][4][11], resulting in nodes that vary in size. Repeated allocations and deallocations of such non-uniform sizes leads to fragmentation and underutilization of memory. Tree Bitmap [4] and Segmented Hole Compaction [14] use complex memory management schemes to compact away the memory fragmentation. Thus, even though Tree Bitmap guarantees *O(1)* route-update cost, in the worst case its memory management scheme can add an overhead of more than 100 memory accesses to any route-update due to compaction operations.

In contrast, we use only 1-bit trie nodes without compression, ensuring that all nodes have the same size across all levels and all stages. Hence, all memory allocations and deallocations deal with one size. Consequently, our routing-table memory incurs no fragmentation whatsoever, and we obviate the need for complex memory management schemes like Tree Bitmap [4] and Segmented Hole Compaction [14]. Thus, after accounting for memory management overhead, the total worst-case route-update cost of SDP amounts to exactly and only one write-bubble.

3.3.6 Scalability in Lookup Rate

As we have pointed in Section 3.2.3, the lookup rates for data-structure pipelining cease to scale once the strides have been reduced to 1, and the size of the largest stage has been minimized. For 1 million prefixes, the size of the largest memory stage using SDP is 3.8 MB, which means that SDP pipelined from only a data-structure perspective can manage only upto 40 Gbps line-rate.

The underlying assumption in data-structure pipelining is that, before a packet's lookup can access a particular stage, it must wait for the lookup that is currently in that stage to complete its access. However, we can internally pipeline, at the hardware level, the memory of each SDP stage so that a packet's lookup can access the memory of an SDP stage before the downstream lookups have completed their access of that memory. The degree to which an SDP stage must be hardware-pipelined is equal to the ratio of the required lookup-rate to its access delay, which depends on its size. Thus, different SDP stages may be hardware-pipelined to different number of hardware stages. We see from the example in Figure 9 that the early memory stages are small and may require shallow or no HLP, while the later stages are larger and may require deeper HLP. This combining of hardware-level and data-structure-level pipelining for throughput scalability is our third innovation. The combining makes

throughput independent of the size of the SDP stages, obviating [1]'s minimization of the largest stage.

4 Brief Review of TCAM-based Schemes

A Content Addressable Memory (CAM) is a type of memory that is designed specifically for search tasks. A CAM simultaneously compares all memory locations against the input key to find matching entries. A Ternary Content Addressable Memory (TCAM) is simply a CAM which supports wild card bits in the entries. IP-lookup is performed by supplying the destination IP-address to the TCAM, which finds the matching prefixes in one operation. TCAMs must have an arbitration scheme to choose the longest match when multiple prefixes match. Most arbitration mechanisms generally require sorting the prefixes by their lengths before placing them in the TCAM, complicating the process of route-updates. [13] proposes an efficient way to update TCAMs via incremental and partial-order sorting.

Because a single access activates *all* memory locations, as opposed to just one, a TCAM dissipates a lot more power compared to RAM. [19] presents a scheme to improve TCAM power by reducing the number of memory locations searched. However, [19] needs to restructure the layout of prefixes in the TCAM subbanks when the distribution undergoes non-trivial changes, complicating the route-update cost.

Even today, TCAM access delays are longer than packet inter-arrival times. Therefore, TCAMs are pipelined at the hardware level, which further worsens their power dissipation and implementation cost.

5 Methodology

Because the IP-lookup memory in trie-based schemes must provide high bandwidth, SRAM is the choice of memory technology for tries, both today and in the expected future. TCAMs on the other hand, can be built using CAM-styled memory. To evaluate the implementation cost, power and timing for these two types of memories we utilize CACTI 3.2 [2]. CACTI is a tool that models accurately the area, power, and timing of SRAM and CAM structures. Because the stock version of CACTI cannot handle memories as large as 75 MB, we modify CACTI according to our needs. Using the modified versions of CACTI we determine the area, power and timing details for HLP, DLP, TCAM, and SDP. We validated our evaluation methodology by modelling SRAMs and TCAMs with the parameters of commercially available products, and we verified that our results closely match the power and timings quoted by vendors of such products [8][10].

We evaluate previous schemes and SDP over a wide spectrum of routing-table sizes, and of line-rates. We evaluate each scheme under worst-case guarantees both for lookup-rates, and for prefix distributions. Due to lack of space we present experimental results only for 100nm CMOS technology. We performed the same experimental evaluations for a range of CMOS technologies and found that the results are qualitatively the same.

6 Experimental Results

We now present a detailed experimental evaluation that compares SDP against previously proposed IP-lookup schemes — HLP, DLP, and TCAM. Recall that a truly scalable IP-lookup scheme must meet all the five requirements of scalability in routing-table size, lookup throughput, implementation cost, power dissipation, and routing-table update cost. We present evaluations for four of the five scalability requirements. The remaining requirement is scalability in throughput which is implicit in the x-axes of the graphs we present for the other requirements.

Because HLP places the entire trie in one large memory, it may use any trie scheme that is not pipelined at the data-structure level. For a fair evaluation we must pick the best choice out of the various schemes available. Recall from Section 2.2, that multi-bit strides increase the total memory size of a trie due to redundant replication of pointers and prefixes in trie nodes. Variable-striding and compression-based schemes can help reduce total memory size by eliminating such redundant replication. However, for the worst-case distribution shown in Figure 4, the top half (triangular region) of the trie has no redundant replication whatsoever, and the bottom half (rectangular region) uses up space because of the large number of nodes and not because of inflated node sizes. Hence, schemes that target average case memory size such as variable stride tries, LC tries [11] (which is essentially a variable stride trie [17]), and Lulea scheme [3], will do no better than a fixed stride trie for the worst-case distribution. Though Tree Bitmap [4] may reduce the large number of nodes in the bottom half (rectangular region) of Figure 4, it would require large strides (e.g., 6 or 8) for any significant improvement. Such large strides will adversely affect the worst-case route-update cost as explained in Section 6.5. Further, Tree Bitmap almost doubles the total memory requirement (Section 3.2.2), therefore any saving in the number of trie nodes would be offset by a multiplicative factor of about 2. Hence, in the evaluation of HLP we choose the fixed-stride multi-bit trie of [17] with strides chosen to minimize worst-case total memory size.

By varying k, the number of levels in a multi-bit trie, we can obtain a wide design-space for DLP and HLP. We explore this design space first in order to choose optimal values of k for these two schemes. We then evaluate, in detail, the optimal design-points of DLP and HLP, TCAM, and SDP. We first compare the total worst-case memory requirement of each scheme, and we then compare the power dissipation and implementation cost of each scheme. Finally, we compare the route-update cost of SDP against that of Tree Bitmap [4], the best previous scheme for route-updates.

6.1 Optimal Design Point for Previous Proposals

As we mentioned in Section 3.2, increasing the number of levels in DLP decreases the worst-case size of the largest stage. We expect the largest stage to be minimized when each level strides only one bit (i.e., $k = W$, the number of bits in an IP-address(32)). Figure 10(a) shows the worst-case size of the largest memory-stage plotted against k, for various routing-table sizes. Observe that though the per-stage memory is *minimized* when $k = 32$, it does not decrease appreciably beyond $k = 16$. We therefore choose $k = 16$, so that the per-stage memory is effectively minimized, and the total memory is halved compared to that of $k = 32$ (because there are only half as many stages). Coincidentally $k = 16$ represents the optimal design point for both worst-case per-stage memory size and worst-case total memory size.

Recall that HLP employs one large memory to hold the entire multi-bit trie, and hardware-pipelines the memory to a depth proportional to k. We are interested in reducing the total memory size, while keeping k small in order to reduce hardware complexity.

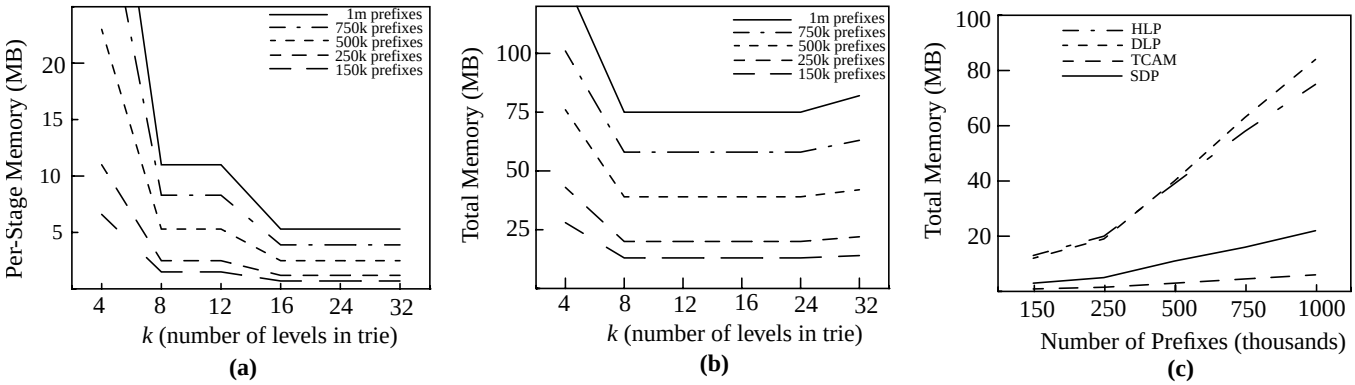

Fig. 10. (a) Worst-case per-stage memory versus trie-levels for DLP (b) Worst-case total memory versus trie-levels for HLP (c) A comparison of total worst-case memory versus routing table size for various IP-lookup schemes

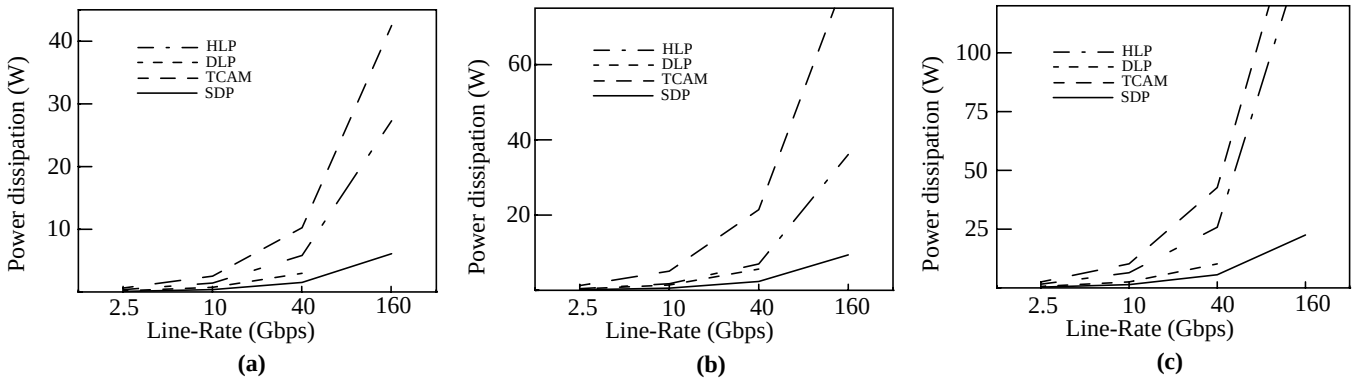

Fig. 11. Comparison of power dissipation versus line-rate for various schemes with tables sizes of (a) 250,000 (b) 500,000 (c) 1 million

Increasing k reduces the total memory size by reducing the extent of controlled prefix expansion. Because the opportunity for this reduction is small when the trie is dense (as is the case in the worst-case prefix distribution of Figure 4), we expect only diminishing returns as k is increased. In Figure 10(b) we show the worst-case total memory size plotted against k, for various routing-table size. Observe that beyond $k = 8$, the total memory size does not decrease appreciably. In order to minimize total memory size while keeping hardware complexity within reason, we choose $k = 8$ as the optimal design point for HLP.

6.2 Worst-case Total Memory Size

In Section 3 we presented expressions for obtaining the worst-case memory sizes for DLP, HLP and SDP. Recall that, due to a much tighter bound, we expect the worst-case total memory size of SDP to be much smaller than that of DLP and HLP. Because the per-bit implementation area of TCAMs is higher than that of SRAMs (used in HLP, DLP and SDP), comparing raw memory sizes is not a useful comparison. However, we show the memory requirement for TCAM because it represents a lower bound on memory size (it is the size for storing only all the prefixes.)

Because the total memory requirement is independent of the line-rate, we need to vary only the routing-table size while evaluating the memory requirement of various schemes. Figure 10(c) shows the worst-case total memory size plotted against the routing-table size, for the various schemes. For a routing table of 1 million prefixes the node sizes for DLP and HLP are 80 bits and 100 bits respectively, whereas for SDP the size is 72 bits because each node must also budget for the jump bits. For a routing-table size of one million prefixes DLP and HLP require memories as large as 84 MB and 75 MB respectively, whereas SDP and TCAM require 22 MB and 6 MB respectively. Across all routing-table sizes, DLP and HLP require roughly the same amount of memory, whereas SDP requires four times smaller memory on average. The memory requirement of TCAM is, on average, another factor-of-four smaller than SDP. We see that DLP and HLP do not scale well in worst-case total memory size as the number of prefixes increase.

6.3 Power Dissipation

Because the power dissipated in accessing a memory varies with both the size of the memory and the rate of access, we must vary both the routing-table size and the line-rate when evaluating the power dissipation of various schemes. Recall that we are evaluating worst-case power dissipations. TCAMs activate all memory locations in a single access in the worst-case, therefore their power dissipation is expected to be much higher than that of the trie-based schemes. We expect HLP's dissipation to be large because it hardware-pipelines the memory aggressively. Recall that DLP does not, therefore we expect its power dissipation to be small. However, for the same reasons, DLP cannot achieve high line-rates such as 160 Gbps. We expect the power dissipation of SDP to be slightly smaller than that of DLP because of a smaller memory size.

Figure 11 shows the power dissipation plotted against the line-rate, for the various schemes. Because we must also vary the memory size while evaluating power dissipation, we present three separate graphs for three different routing-tables sizes: (a) 250,000

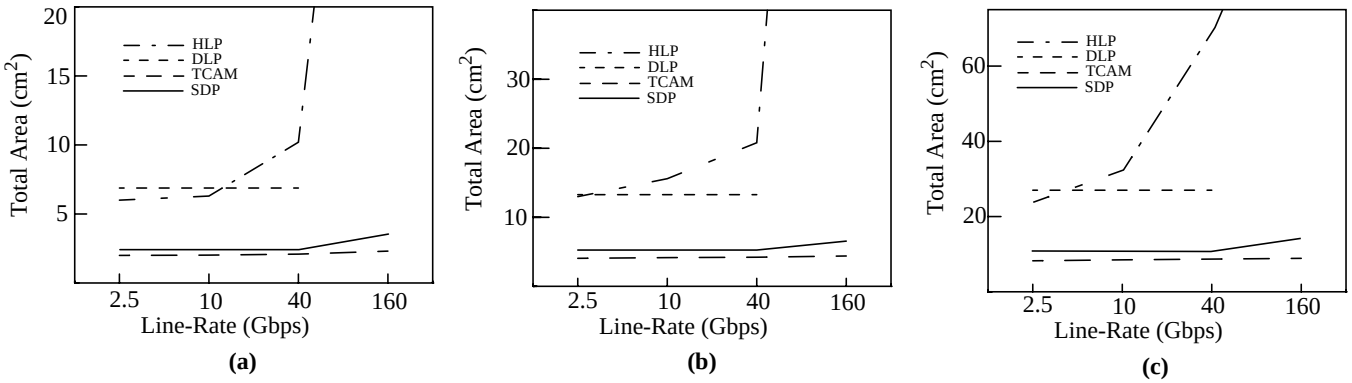

Fig. 12. Comparison of chip area versus line-rate for various schemes with table sizes of (a) 250,000 (b) 500,000 (c) 1 million prefixes

prefixes (b) 500,000 prefixes, and (c) 1 million prefixes. In the evaluation of hardware-pipelined memories, we ignore the area and power overhead of pipeline latches, giving an unfair advantage to HLP. SDP uses hardware-pipelining to a much smaller extent than HLP, therefore its advantage is minimal.

Observe that the results for all routing-table sizes are qualitatively similar, therefore we comment only on the results corresponding to 1 million prefixes (Figure 11(c)). Due to TCAM's brute-force searching it dissipates as much 42 W at 40 Gbps line-rate, and 174 W at 160 Gbps line-rate. For $k = 8$, HLP needs to aggressively pipeline the memory, even more so for high line-rates. When dealing with 160 Gbps line-rate, HLP must access the memory every 0.25 ns. We see that pipelining 75 MB of SRAM to such depth dissipates prohibitive amounts of power. HLP dissipates as much as 25 W for 40 Gbps line-rate, and 146 W for 160 Gbps line-rate. Note that, because DLP does not scale to 160 Gbps line-rate, its 160 Gbps data-point is absent in all graphs. DLP dissipates 10 W at 40 Gbps line-rate. For 160 Gbps line-rate, SDP hardware-pipelines the individual memory-stages, albeit to a less extent than HLP, and hence incurs some penalty in power dissipation. SDP dissipates 5.5 W for 40 Gbps line-rate, and 22 W for 160 Gbps line-rate. The difference in power dissipation between SDP and DLP is primarily due to memory size, whereas between DLP and HLP it is primarily due to aggressive hardware-pipelining.We see that HLP and TCAM do not scale well in power dissipation as the routing-table sizes and line-rates increase.

6.4 Implementation Cost

The cost of implementing chips in silicon is proportional to approximately the fourth power of their area [6]. Hence, we evaluate the chip area of various schemes to ascertain their scalability in implementation cost. Although the total memory requirement of TCAM is fairly small, we do not expect its chip area to be as small because, for circuit-level reasons, CAM-styled memories cannot be designed to have the same high density as RAM. In the absence of hardware pipelining, the area taken up by a memory is proportional to its size in bytes. In the presence of hardware-pipelining the area grows exponentially with the depth of pipelining. Because DLP does not hardware-pipeline the memory, we expect its chip area to stay constant across line-rates. For small values of line-rates, we do not expect any of the schemes to incur any significant hardware-pipelining. Therefore for low line-rates, we expect DLP, HLP and SDP to take up chip areas in accordance with their memory sizes (i.e., we expect DLP and HLP to take up similar areas and SDP to take up four times lesser area than theirs). For high line-rates, we expect the area of HLP to grow in comparison to DLP due to hardware-level pipelining. We also expect the area of SDP to increase at high line-rates, however not by the same trend as HLP because SDP's hardware-pipelining is not nearly as aggressive as HLP.

Figure 12 shows the chip area plotted against the line-rate, for the various schemes. Recall that we ignore the area and power overhead of pipeline latches in HLP, giving it an unfair advantage. Because we must also vary the memory size while evaluating chip area, we present three separate graphs for three different routing-tables sizes: (a) 250,000 prefixes (b) 500,000 prefixes, and (c) 1 million prefixes.

We see that the results for all routing-table sizes are qualitatively similar, therefore we comment only on the results corresponding to 500,000 prefixes (Figure 12(b)). For 40 Gbps line-rate HLP takes up 20.8 cm^2, and for 160 Gbps it takes more than 150 cm^2 of chip area. This drastic increase occurs because HLP must access the memory every 0.25 ns, and in order to achieve such an access rate, the memory array must be split to an extremely fine extent. The chip area of DLP stays constant at 13.3 cm^2 for all line-rates except 160 Gbps. Because DLP does not scale to 160 Gbps line-rate, its 160 Gbps data-point is absent in all graphs. SDP takes up an area of 6.3 cm^2 at 40 Gbps, whereas at 160 Gbps it takes up an area of 7.5 cm^2. We see that the SDP's area is larger for high line-rates due to hardware-pipelining. TCAM takes up about 4.1 cm^2 for all line-rates. We see that the chip area of TCAM, unlike its memory size as evaluated in Section 6.2, is not smaller than that of SDP. Note that because we do not model the priority-encoder in TCAM, its evaluated area is a conservative result and we expect the actual area to be larger. We see that HLP does not scale well in implementation cost.

6.5 Cost of Route-Updates

TCAMs can be updated efficiently by using techniques like [13]. [1] proposes, for DLP, a number of optimizations for fast, incremental route-updates, however all the optimizations are heuristics and improve only the average-case update-cost. The worst-case route-update cost of DLP remains unbounded even with the optimizations. HLP also has an unbounded worst-case route-update cost.

However, the route-update scheme of Tree Bitmap [4] can be applied to HLP and DLP to reduce the route-update cost. This scheme, which is the best to date, requires the update of only one trie node in the worst-case. This one node may contain 2^k pointers in the worst-case, where k is the largest stride in the trie. For small values of k (e.g., 2 or 3), a single memory write may be wide enough to suf-

fice, but for larger values of k (e.g., 6 or 8) Tree Bitmap may require multiple memory writes. In addition to writing the trie node upon a route-update, Tree Bitmap can incur substantial worst-case memory management overhead as mentioned in Section 3.3.5. Accounting for this overhead, the eventual worst-case route-update cost can exceed 100 memory operations. Hence Tree Bitmap does not scale well in worst-case route-update cost. In addition, it incurs the penalty of almost doubling the size of each trie node due to the absence of leaf pushing (Section 3.2.2). In our evaluations above (Section 6.2 through Section 6.4), we do not penalize the total memory size of HLP and DLP. The results presented in those sections must be viewed with the understanding that HLP and DLP *do not* scale in route-update cost. If HLP and DLP incorporate Tree Bitmap to achieve a better worst-case route-update cost, their worst-case total memory sizes would be almost twice as large than the sizes shown in Section 6.2. Yet, even with Tree Bitmap their worst-case route-update cost could exceed 100 memory operations.

In contrast, the worst-case route-update cost in SDP is provably optimum, as it amounts to exactly and only one write-bubble. Owing to the uniform size of its trie nodes, SDP does not need complex memory management schemes with compaction and defragmentation. Hence, we see that SDP scales well in worst-case route-update cost.

6.6 Summary of Results

We see that dynamic pipelining is the only IP-lookup scheme that is truly scalable in routing-table size, lookup throughput, implementation cost, power dissipation, and routing-table update cost. In contrast, all other IP-lookup schemes do not scale well in a number of these requirements. HLP does not scale well in total memory size, power dissipation, route-update cost, and implementation cost. DLP does not scale well in total memory size, lookup throughput, and route-update cost. TCAMs do not scale well in implementation cost and power dissipation. For a routing-table of 1 million prefixes, and a line-rate of 160 Gbps, HLP requires 75 MB, dissipates 146 W, and takes up more than 200 cm^2. TCAM requires 6 MB, dissipates 174 W, and takes up 8.9 cm^2. DLP requires 88 MB, dissipates 10 W, takes up 27 cm^2 and fails to work beyond 40 Gbps. In contrast, SDP requires only 22 MB of memory, dissipates 22 W, and takes up 14.9 cm^2.

7 Conclusions

A truly scalable IP-lookup scheme must address five challenges of scalability, namely: routing-table size, lookup throughput, implementation cost, power dissipation, and routing-table update cost. Though several IP-lookup schemes have been proposed in the past, all of the schemes satisfy only two or three of the requirements but not all five. Previous schemes pipeline tries by mapping trie levels to pipeline stages. We made the fundamental observation that because this mapping is static and oblivious of the prefix distribution, the schemes do not scale well when worst-case prefix distributions are considered. This paper is the first to meet all the five requirements in the worst case. We proposed scalable dynamic pipelining (SDP) which includes three key innovations: (1) We map trie nodes to pipeline stages based on the node height, which succinctly provides sufficient information about the distribution. Our mapping enables us to prove a worst-case per-stage memory bound which is significantly tighter than those of previous schemes. (2) We exploit our mapping to propose a novel scheme for incremental route-updates. In our scheme a route-update requires exactly and only one write dispatched into the pipeline. This route-update cost is obviously the optimum and our scheme achieves the optimum in the worst case. (3) We achieve scalability in throughput by simultaneously pipelining at the data-structure level and hardware level. SDP naturally scales in power and implementation cost. Using detailed hardware simulation, we showed that SDP is the only scheme that achieves all the five scalability requirements. Our results confirm that schemes like SDP will be necessary for future routers to keep up with the scaling trends of the Internet.

References

[1] Anindya Basu and Girija Narlikar. Fast Incremental Updates for Pipelined Forwarding Engines. In *Proceedings of INFOCOM '03*, 2003.

[2] CACTI. http://*research.compaq.com/wrl/people/jouppi/CACTI.html*

[3] M. Degermark, A. Brodnik, S. Carlsson and S. Pink. Small Forwarding Tables for Fast Routing Lookups. In *Procedings of SIGCOMM '97* 1997.

[4] W. Eatherton, Z. Dittia, G. Varghese. Tree Bitmap: Hardware/Software IP Lookups with Incremental Updates. *ACM SIGCOMM Compter Communication Review*, 34(2) 97-122, 2004.

[5] Mathew Gray. Internet Growth Summary. *http://www.mit.edu/people/mkgray/net/internet-growth-summary.html*, 1996.

[6] J. L. Hennessy and D. A. Patterson. Compter Architecture: a Quantitative Approach. *Morgan Kaufman Publishers*. 2002

[7] V. Kumar, T. Lakshman and D. Stiliadis. Beyond Best Effort: Router Architectures for Differentiated Services of Tomorrow's Internet. *IEEE Communications Magazine* 36(5) 152-164, 1998

[8] Integrated Device Technology, Inc. *http://www.idt.com*.

[9] D. R. Morrison. PATRICIA - Practical Algorithm to Retrieve Information Coded in Alphanumeric. *Journal of the ACM,* 15(4)514-534, Oct. 1968.

[10] NetLogic Microsystems, Inc. *http://www.netlogicmicro.com*.

[11] S. Nilsson and G. Karlsson. Fast Address Look-up for Internet Routers. In *Proceedings of The IEEE Conference on BroadBand Communications Technology*. 1998.

[12] Routing Information Service. *http://www.ris.ripe.net*.

[13] D. Shah and P. Gupta. Fast Updating Algorithms for TCAMs. In *Proceedings of IEEE MICRO*, 21(1)36-47, Feb 2001.

[14] Sandeep Sikka and George Varghese. Memory-Efficient State Lookups with Fast Updates. In *Proceedings of SIGCOMM '00*, 2000

[15] Timothy Sherwood, George Varghese and Brad Calder. A Pipelined Memory Architecture for High Throughput Network Processors. In *Proceedings of the 30th Annual ISCA*, pages 288-299, 2003.

[16] K. Sklower. A Tree-Based Routing Table for Berkeley Unix. In *Proceedings of the 1991 Winter Usenix Conference*. 1991.

[17] V. Srinivasan and George Varghese. Fast Address Lookups Using Controlled Prefix Expansion. *ACM Transactions on Computer Systems*, 17(1):1–40, February 1999.

[18] Alan Tammel. How to Survive as an ISP. In *Proceedings of Networld Interop 97*, 1997.

[19] F. Zane, G. Narlikar and A. Basu. CoolCAMs: Power-Efficient TCAMs for Forwarding Engines. In *Proceedings of INFOCOM '03*, 2003.