Range Tries for Scalable Address Lookup

Ioannis Sourdis, Georgios Stefanakis, Ruben de Smet, and Georgi N. Gaydadjiev

Computer Engineering
TU Delft
The Netherlands
{sourdis, gstefanakis, ruben, georgi}@ce.et.tudellft.nl

ABSTRACT

In this paper we introduce the Range Trie, a new multiway tree data structure for address lookup. Each Range Trie node maps to an address range $[N_a, N_b)$ and performs multiple comparisons to determine the subrange an incoming address belongs to. Range Trie improves on the existing Range Trees allowing shorter comparisons than the address width. The maximum comparison length in a Range Trie node is $\lceil log_2(N_b - N_a) \rceil$ bits. Address parts can be shared among multiple concurrent comparisons or even omitted. Addresses can be properly aligned to further reduce the required address bits per comparison. In so doing, Range Tries can store in a single tree node more address bounds to be compared. Given a memory bandwidth, more comparisons are performed in a single step reducing lookup latency, memory accesses per lookup, and overall memory requirements. Latency and memory size scale better than related works as the address width and the number of stored prefixes increase. Considering memory bandwidth of 256-bits per cycle, five to seven Range Trie levels are sufficient to store half a million IPv4 or IPv6 prefixes, while memory size is comparable and in many cases better than linear search. We describe a Range Trie hardware design and evaluate our approach in terms of performance, area cost and power consumption. Range Trie 90-nm ASIC implementations, storing 0.5 million IPv4 and IPv6 prefixes, perform over 500 million lookups per second (OC-3072) and consume 3.9 and 11.4 Watts respectively.

Categories and Subject Descriptors

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

General Terms

Algorithms, Design

Keywords

Internet Router, Address Lookup, IP Lookup

1. INTRODUCTION

Address lookup is the core function for IP routing and packet classification [7, 14, 18]. Internet backbone routers use the packet's destination address to determine the next hop of a packet. They contain hundreds of thousand entries in their tables and require to perform millions of lookups per second. Packet classification requires multi-Gbps throughput having multiple fields to lookup and tens of thousand table entries. The growing size of routing-tables and rapid growth of internet make more difficult to keep pace with the increasing need for faster processing rates. IPv6 growth rate tripled in the past two years [12] and coupled with the IPv4 exhaustion poses the need for solutions scalable with the address width.

In general, there are many challenging issues which need to be addressed when performing address lookup. *Performance* (fast lookup and high throughput) is the primary objective for such algorithms. *Memory size* is also important and often determines performance (memory access delay). *Memory bandwidth* is limited and its efficient utilization may also affect performance. As the routing tables get larger *performance needs to scale well with the number of entries* (i.e., number of prefixes), while moving from IPv4 to IPv6 indicates that *performance scaling with the address width* is also necessary. Finally, the *latency of incremental updates* as well as *power consumption* are also important.

Although a plethora of algorithms has been proposed, several of the above challenges remain open. On one hand, Trie-based structures have lookup latency which scales linearly to the address width and memory requirements which scale exponentially to the length of strides. On the other hand, Range Trees are limited by the available memory bandwidth, and hence their lookup latency and memory requirements increase significantly for larger tables and wider addresses, such as IPv6.

In this paper we introduce the Range Trie, a new approach for address lookup, and attempt to address the above challenges. A Range Trie is a multi-way tree structure that can be classified between the "search on values" (i.e. Range Trees) and "search on length" (i.e. Tries) approaches, following the Ruiz-Sanchez et al. taxonomy of IP lookup algorithms [14]. On one hand, Range Trees search on the value dimension performing comparisons on complete addresses. On the other hand, Tries search on length matching parts of addresses. *Range Tries perform comparisons on parts of addresses.* Addresses that define ranges are placed sparser or denser in the address space creating longer or shorter ranges, respectively. Comparisons of fewer address bits may be sufficient for sparser areas in the address space. Denser areas need better precision but may have long common address parts that can be shared. We capitalize on this observation to reduce the number of address bits stored and processed in a tree node, improving memory bandwidth utilization, and reducing lookup latency. The performance of the proposed scheme scales well with the address width and the number of address ranges stored in the tree.

The remainder of this paper is organized as follows: Section 2

ANCS'09, October 19-20, 2009, Princeton, New Jersey, USA.

offers some background on address lookup algorithms. In Section 3 we describe the Range Tries data structure and in Section 4 the Range Trie hardware design. Then, in Section 5 we evaluate the Range Trie performance, memory size, power consumption, area cost, and scalability and compare with existing address lookup data-structures. Finally in Section 6 we draw our conclusions.

2. BACKGROUND

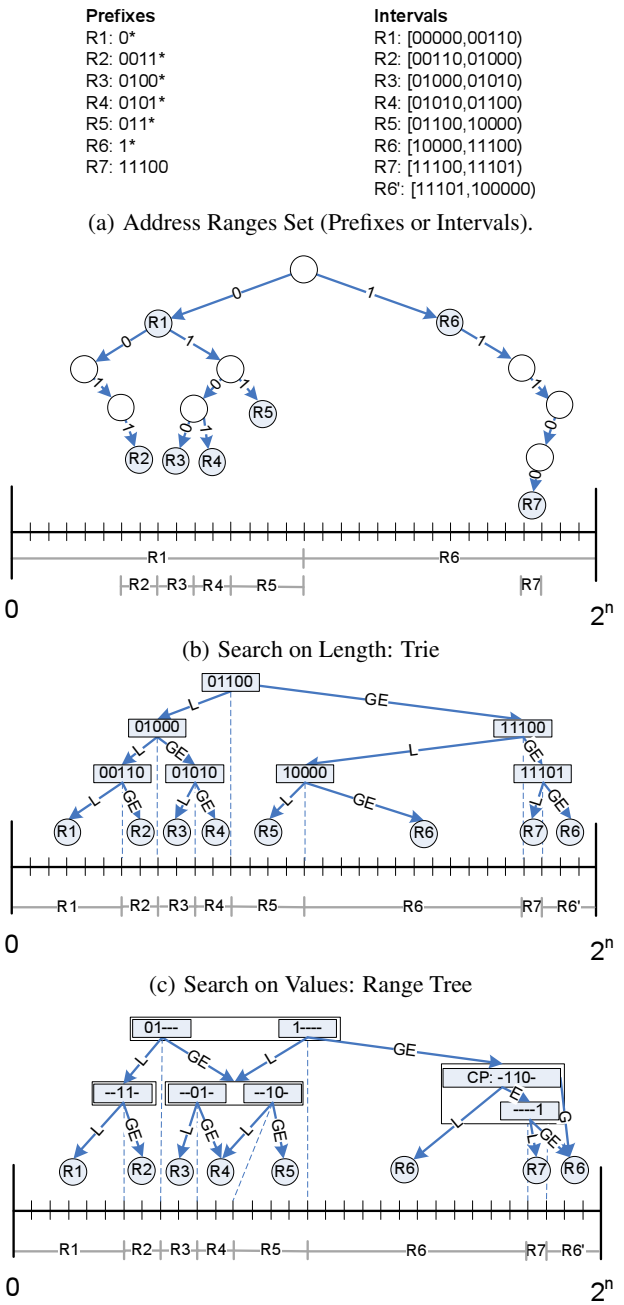
Many address lookup algorithms have been proposed in the past; most of them have been summarized in the survey by Ruiz-Sanchez et al. [14]. We follow the taxonomy of [14] and briefly discuss different approaches below.

A set of address ranges can be expressed, as shown in Figure 1(a), either directly as *intervals*, where the complete bit representation of the addresses can be compared to perform a lookup, or as *prefixes* out of which the longest matching one should be reported. As indicated in [14] address lookup involves searching in two dimensions: length and value. Consequently, existing address lookup data structures are categorized according to the dimension their search is based on, namely in "search on length" and "search on values" approaches.

Tries are considered a "search on length" approach as they perform a sequential search on the length dimension, matching at step n prefixes of length n (Figure 1(b)). Improvements on the basic Trie structure may include prefix expansion for multibit strides [16] with or without leaf pushing, compressed Tries [11], the Lulea bitmaps to reduce the storage requirements [4], or Tree bitmaps [5]. Trie-based structures inherently support longest prefix match, but their major drawback is that the number of tree-levels scales linearly to the address width [14]. Multibit tries reduce the decision tree height but do not improve scalability in the address width, while significantly increase memory size.

Range Tree is a "search on values" approach, it avoids the length dimension performing comparisons on the expanded prefixes (complete addresses). As depicted in Figure 1(c), Range Trees perform address comparisons creating a balanced decision tree. They store complete addresses to be compared at each node and therefore consume considerable memory size. Multiway Range Trees read and compare at every step multiple addresses [19]. The number of comparisons per node, however, is limited by the available memory bandwidth, which, consequently, reduces scalability with respect to the address width. As described in [9, 14, 19], Range Trees need to store additional information in order to support longest prefix match.

In general, Tries perform *exact match in parts of addresses*, while Range Trees perform *comparisons of full addresses*. The proposed Range Tries attempt to combine the advantages of the above performing *comparisons in parts of addresses*. Figure 1(d) illustrates a Range Trie example and shows that comparing fewer address bits can be sufficient for address lookup; the above are explained in detail in Section 3.1. At the root node, comparing the two most significant bits "01---" and the most significant bit "1----" is the equivalent of comparing the complete addresses "01000" and "10000". In the second iteration and after taking the middle root branch, we do not need to compare the two most significant bits since after the first step we know the incoming address is "01xxx". Similarly, after taking the right branch of the root node we know that the most significant bit is "1xxxx". Then, the two addresses to be compared ("11100" and "11101") have a common prefix ("-110x") which is shared and compared separately. The decision of that node is based on the outcome of the common prefix comparison and (if needed) the comparison of the least significant bit. The created Range Trie of Figure 1(d) is well balanced and shorter than the one



(a) Address Ranges Set (Prefixes or Intervals).

(b) Search on Length: Trie

(c) Search on Values: Range Tree

(d) Range Trie.
CP: Common Prefix, '-': omitted bit, L: less, E: equal, GE: greater or equal, G: greater.

Figure 1: The Trie, Range Tree, and the proposed Range Trie data structures.

of the Range Tree in Figure 1(c) using less memory bandwidth. In the hardware description of the Range Trie (Section 4), we show how a single comparator can be used to perform the above variable length comparisons.

Recently, several related address lookup approaches target ASIC implementations using embedded on-chip memory and pipelining to reduce memory latency and increase throughput [2, 5, 8]. However, it is expected that the routing tables will continue to grow in size; this combined with the transition to IPv6 will increase the

number of pipeline stages and the memory requirements making harder to fit such solutions on-chip. As described in Section 4, Range Tries target hardware implementations addressing the above limitations.

3. THE RANGE TRIE DATA-STRUCTURE

The Range Trie is a multiway tree structure that gradually *compares parts of addresses* to perform an address lookup. A Range Trie reduces the number of address bits to be compared and therefore, given a memory bandwidth, increases the total number of comparisons performed in a single step. In doing so, the decision tree has more branches per node and, consequently, tree height is reduced compared to a Range Tree that uses the same memory bandwidth and compares complete addresses. The Range Trie is based on the following three observations:

- Nodes closer to the root compare addresses that are sparser in the address space and therefore their suffixes can be omitted without creating imbalance on the tree.
- Nodes closer to the leafs compare addresses that are denser in the address space and therefore their prefixes can be either omitted or shared.
- A single node, which may need to compare addresses placed sparser or denser to each other, can process different number of bits for each individual address offering less or better "precision", respectively. In doing so, the decision tree can be kept balanced.

Considering the above observations, it is expected that a Range Trie will start comparing address prefixes at the root node omitting the suffixes. Then, gradually at the next levels will continue with the address infixes and will end up comparing suffixes as approaching the leafs. Intuitively, as we traverse the Range Trie the common address prefixes will gradually get longer and the shared or omitted address suffixes will get shorter. That is because the search space will gradually reduce and better "precision" will be required.

The Range Trie has the following properties:

- A node maps to an address range $[N_a, N_b)$ of the address space. The union of the children node address ranges is the address range of their parent node.
- The maximum number of required address bits per comparison at a Range Trie node is $\lceil \log_2 (N_b - N_a) \rceil$.
- Address suffixes can be omitted from processing, when they are *zero*[1].
- Common address parts are shared among concurrent address comparisons.
- Addresses compared in a node can be aligned properly to maximize address part sharing.

3.1 Range Trie Rules

The Range Trie exploits five rules to increase the number of branches per node given a specific memory bandwidth and hence reduce the depth of the decision tree. In this paragraph, we describe these rules and provide examples of Range Trie nodes that use them. We consider a Range Trie node N, as illustrated in Figure 2, that maps to the address range $[N_a, N_b)$ and divides it in $k+1$ subranges R_1, R_2, ..., $R_k + 1$ defined by the unique addresses $A_i \in [N_a, N_b), \forall i \in \mathbb{N},\ i \leq k$, such that $R_1 = [N_a, A_1)$, ..., $R_i = [A_{i-1}, A_i)$, ..., $R_{k+1} = [A_k, N_b)$. Then an incoming address $A_{IN} \in [N_a, N_b)$ needs to be compared against the addresses

[1]During a Range Trie construction, one may force an address suffix to zero, losing in precision, but reducing the required address bits.

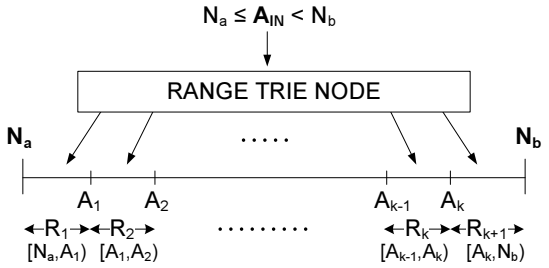


Figure 2: Generic view of a Range Trie node.

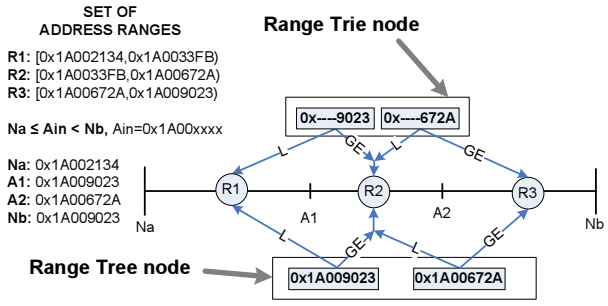


Figure 3: A Range Trie node with *common node prefix* vs. a Range Tree node.

A_i in order to determine the subrange R_i to which it belongs. It is noteworthy that the address width is W. A single comparator reports whether the A_{IN} is "Less", "Equal" or "Greater or Equal" to an address part of A_i. The construction of a Range Trie is based on the following rules[2]:

Rule 1: Omit Node Common Prefix. The common prefix, of length L bits ($L < W$), of all addresses in the node address-range $[N_a, N_b)$ can be omitted from the comparisons between A_{IN} and A_i.

That is because addresses $A_i, A_{IN} \in [N_a, N_b)$ will have the same common prefix of their L most significant bits. This means that the comparison of the L MSbits between A_{IN} and A_i will always result in equality, enabling the comparison of only the $W - L$ LSbits to produce the result.

Figure 3 illustrates an example of the *node common prefix* rule, where all addresses in the address range of the node have a common node prefix 0x1A00----. The bits corresponding to the common node prefix can be omitted from the comparisons of A_{IN} and A_i as opposed to a Range Tree node which requires to store and compare them completely.

Rule 2: Omit address Zero Suffix. Let an address A_i have a suffix of length L bits, where $L < W$, that is zero. Then, this suffix of A_i does not need to be compared against the L last bits of A_{IN}.

In order for the comparison of the L-bit suffix to affect the final result of the comparison, the $W - L$ prefix comparison needs to result in equality (which translates into "GE"). In that case however, the comparison between the A_{IN} and A_i L-bit suffix will always be "GE" since the A_i L least significant bits are zero. Then, since "GE" is already the result reported by the prefix comparison we can omit the zero suffix comparison.

Figure 4 illustrates an example of the *zero-suffix* rule, where the A_1 and A_2 have a zero suffix of length 2 and 3 bytes, respectively. Zero suffixes of A_1 and A_2 can be omitted. On the contrary, as illustrated at the bottom part of Figure 4, a Range Tree node requires to store these bits and consider them in the comparisons.

[2]The formal proofs of the rules are omitted due to lack of space.

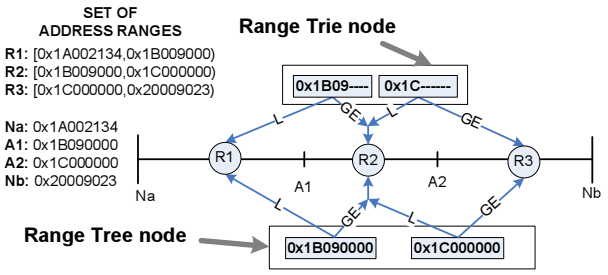


Figure 4: A Range Trie node that compares addresses A_i with *zero-suffixes* vs. a Range Tree node.

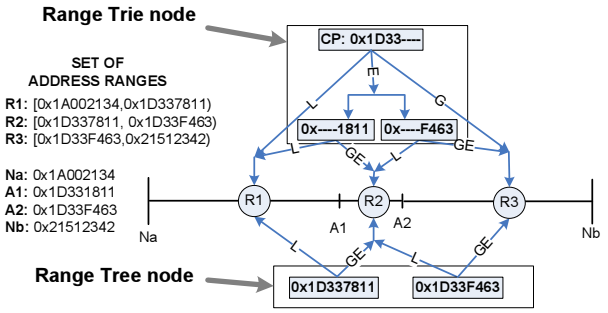


Figure 5: A Range Trie node that compares addresses with a Common Prefix.

It is worth noting, that we may exploit the benefits of this rule by changing the value of an address A_i so that it has a long suffix of zero. This way we economize in the number of bits to be stored and compared in a node in exchange of losing "precision", since the more bits we modify in an address suffix the more we draw away from the original value of A_i[3].

Rule 3: Share addresses' Common Prefix. The Common Prefix A_i^{CP} of the addresses A_i of length L ($L < W$) can be shared among concurrent comparisons and processed separately. Then, if the A_{IN} prefix of length L is *Less* than A_i^{CP}, then $A_{IN} \in R_1$ (the first address range of the node), if it is "Greater" than A_i^{CP} then $A_{IN} \in R_{k+1}$ (the last address range of the node), otherwise we consider the comparison result of the $W - L$ bits suffix to determine the range A_{IN} belongs to.

Figure 5 shows an example of the *Common Prefix* rule, where A_1 and A_2 have a common prefix of 2 bytes 0x1D33----. The common prefix is stored in the node and compared with the respective A_{IN} part only once. In case A_{IN} is less or greater than the prefix, R_1 or R_3 match respectively. In case of prefix equality, the suffixes of the addresses are considered as illustrated in Figure 5. As opposed to the Range Trie, a Range Tree node would need to store and process the addresses common prefix twice for A_1 and A_2.

Rule 4: Share addresses' Common Suffix. The common suffix A_i^{CS} of the addresses A_i of length L ($L < W$) can be shared among concurrent comparisons and processed separately. Let $R_p = [A_{p-1}, A_p)$ (where $1 \leq p \leq k+1$, $A_0 = N_a$, and $A_{k+1} = N_b$) be the address range that the comparisons of the $W - L$ MSbits of A_i and A_{IN} indicate. Then $A_{IN} \in R_{p-1} = [A_{p-2}, A_{p-1})$ IFF all three statements below are true:

(i) the A_{IN} $W - L$ MSbits are equal to the ones of A_{p-1},
(ii) the A_{IN} suffix of length L is less than A_i^{CS},

[3]When forcing the suffix of an address A_i to zero, the original address A_i is compared in a subsequent tree node.

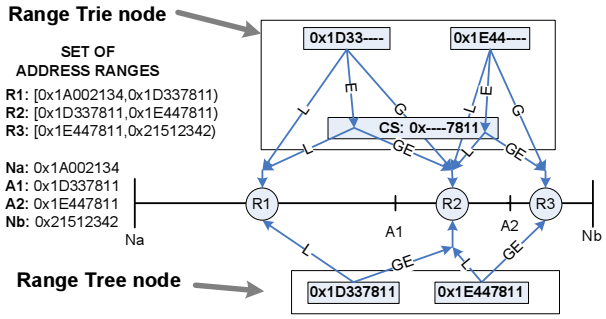


Figure 6: A Range Trie node that compares addresses with a Common Suffix.

(iii) $R_p \neq R_1$.
Otherwise, $A_{IN} \in R_p$.

Figure 6 illustrates an example of the *Common Suffix* rule, where A_1 and A_2 have a common suffix of 2 bytes 0x----7811. The common suffix is stored in the node and compared with the respective A_{IN} part only once. When a prefix comparison of either A_1 or A_2 results in equality, then the common suffix comparison is considered as shown in Figure 6. A Range Tree node would need to store and process the addresses common suffix twice for A_1 and A_2.

Rule 5: Address Alignment. The lookup of address A_{IN} in node N, as described in Figure 2, is equivalent of the lookup of address $A_{IN} - N_a$ in node N' that maps to the address range $[0, N_b - N_a)$ and divides it in $k+1$ subranges R'_1, R'_2, ..., R'_{k+1} defined by the addresses $A'_i = (A_i - N_a) \in [0, N_b - N_a)$, where $i = 1, 2, ..., k$, such that $R'_1 = [0, A_1 - N_a)$, ..., $R'_i = [A_{i-1} - N_a, A_i - N_a)$, ..., $R'_{k+1} = [A_k - N_a, N_b - N_a)$.

Figure 7 illustrates an example of the *Address Alignment* rule. In this example, a node maps to $[N_a, N_b)$ and compares addresses A_1 and A_2. Although the node maps to a narrow address range there are no common address parts to be shared or omitted according to any of the previous rules. The length of the node address range $[N_a, N_b)$ can be represented in one byte ($\lceil \log_2(N_b - N_a) \rceil = 8$). The node can be replaced by a node N' which maps to $[N'_a, N'_b)$ and compares addresses A'_1 and A'_2 as shown in the example. As opposed to node N, node N' has a long common node prefix of 3 bytes which can be omitted according to Rule 1. Then, only the least significant byte of each address A'_i is required. The least significant byte of N_a is first subtracted from the corresponding byte of the incoming address A_{IN}. Subsequently, the result is compared to the last byte of A'_1 and A'_2 in order to select the correct node branch. The Range Trie node will produce the same result as that of a Range Tree node shown at the bottom of Figure 7. However, in the Range Trie case, storing and processing only the one byte per address A_i and one byte to be subtracted is sufficient.

Rule 5 maximizes the benefits of rule 1 and in essence is the means to achieve the most essential Range Trie characteristic: *the maximum number of address bits per comparison, a Range Trie node needs to process, is equal to the number of bits needed to represent the length of the node address range* $\lceil \log_2(N_b - N_a) \rceil$. The above rule and Range Trie characteristic is the main improvement over our previous work, the Range-Trees with Variable Length Comparisons (RT-VLC) [15].

A question arises regarding applying more than one of the above rules in parallel. Rules 1-4 can be applied independently as they do not affect each other. For instance, we can omit common node prefix (Rule 1), omit any zero suffix (Rule 3), and then share address common prefix (Rule 2) and suffix (Rule 4) of the remaining

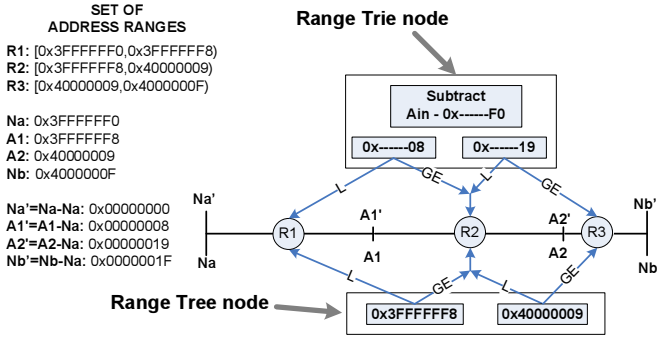


Figure 7: A Range Trie node that aligns incoming address and compared addresses to maximize the common node prefix.

address bits. Rule 5 however is more difficult to apply combined with the rest. Rule 5 aims at maximizing common node prefix, consequently, it can be combined with Rule 1, but needs to be applied before the common prefix rule (Rule 2) since the address prefixes change after the subtraction. Regarding zero and common address suffixes, Rule 5 can be applied independently.

3.2 Constructing a Range Trie

Given a set of k addresses A_i that define $k+1$ basic address ranges R_i at an address space, expressed as prefixes or intervals, a Range Trie is constructed based on the above rules. This is performed by selecting addresses to be compared at each tree node targeting a low tree depth. There are two objectives when constructing a Range Trie. The first one is to select addresses which require fewer bits to be stored and processed in order to maximize the number of node branches. Second, the node should be branching to subtrees of equal or similar depth, so that the entire tree is balanced. It may be the case that the above objectives contradict each other; maximizing the number of branches may not necessarily keep the tree balanced and vice versa. We developed heuristic methods to construct a Range Trie, rather than seek an optimal solution which would possibly have unacceptable complexity. Apart from these two objectives there are other parameters to be consider pertaining to the the Range Trie implementation, such as the memory bandwidth, the available lengths of comparisons, and address alignment restrictions.

We developed two types of heuristic methods to construct a Range Trie based on an arbitrary set of address ranges following either a *top-down* or a *bottom-up* approach.

In a top-down approach a heuristic creates first the root node and then similarly moves to its children and towards the leafs of the tree. Each node selects addresses A_i to be included in the node considering the five rules. At each node the heuristic attempts to balance between "precision"[4], number of node branches, and tree-balance.

Bottom-up heuristics construct first the leaf nodes and subsequently their node bounds are used for the upper tree level; this is repeated until the root of the tree is reached. Starting from A_1, a node includes as many addresses A_i as the resources allow (e.g., comparators, memory size and bandwidth), after applying first the Range Trie rules. Then, the upper bound of the node is selected, so that it has a long zero suffix and hence benefit from Rule 2.

As opposed to a top-down approach, a bottom-up heuristic creates balanced Range Tries that have all their leafs at the same level. As shown in Section 5, in general the bottom-up heuristic achieves a better worst case latency, while the top-down heuristic results in a lower lookup latency for the average case. A more detailed description of the Range Trie heuristics can be found in [3].

[4]Number of bits per comparison as explained in Rule 2.

3.3 Longest Prefix Match & Incremental Updates

In order for a Range Trie to support longest prefix match, each node needs to store more information than just the parts of addresses to be compared. To our advantage, however, is the fact that the Range Trie can be mapped one-to-one to a Range Tree of unlimited memory bandwidth and number of branches per node. As shown in Section 3.1 the external characteristics of a Range Trie and a Range Tree node are the same; in both cases, a node maps to an address interval, determines an address subrange to which A_{IN} belongs to, and accordingly branches to the next level. Consequently, the Range Tree techniques for supporting longest prefix match, as described in [9, 14, 19], apply to the Range Trie as well.

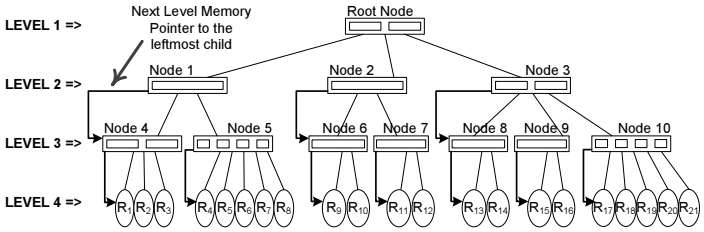
The prefixes are stored and updated in a Range Trie as described in [9, 14, 19] for a Range Tree. The main idea is that a prefix can be stored in internal nodes rather than only leafs of the tree. This improves the update time from $O(n)$ to $O(\log n)$ as updating only a parent node that maps to a prefix is equivalent to updating all its children. Slightly modifying the above techniques, supporting longest prefix match and incremental updates in Range Tries requires the following:

- a pointer to a prefix[5], along with its prefix length, is stored at every node the address range of which is part of the prefix, but the address range of its parent node is not;
- each address compared in a node keeps a value to count the number of prefixes having an endpoint on the address;
- a new prefix is inserted by:
  - inserting its end points or updating the corresponding counters if an endpoint already exists, and
  - by storing a pointer to the prefix in every node
    (i) the range of which is a subset of the prefix,
    (ii) the prefix pointer is not stored at its parent node, and
    (iii) the prefix is longer than any previously stored one in the node;
- for a prefix to be deleted we need to know the prefix to replace it with; then,
  - the prefix endpoints are deleted or the corresponding counters are decremented, and
  - the prefix pointer is replaced in each node it is stored.

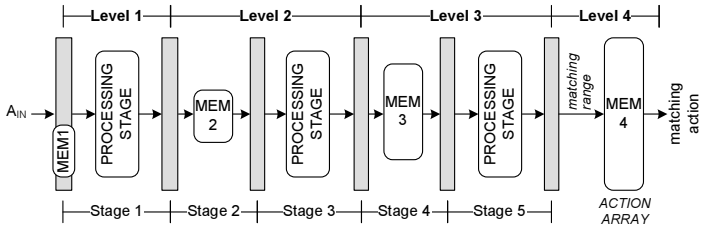
Updating a Range Trie requires to insert or delete addresses that define address ranges (i.e., prefix endpoints). This can be achieved by updating the affected leaf node or subtree performing splits or merges. Similar to a Range Tree, the complexity of inserting or deleting an address is $O(k \log_k n)$ where n is the number of addresses stored in the tree, and k is the number of branches in a node. In Range Tries however k is larger than in a Range Tree with the same memory bandwidth.

Finally, it is worth noting that there are cases where a Range Trie does not need to support longest prefix match. For example, port ranges for packet classification are more efficient to be represented as intervals rather than prefixes. Then, storing overlapping intervals can be supported similarly to storing prefixes.

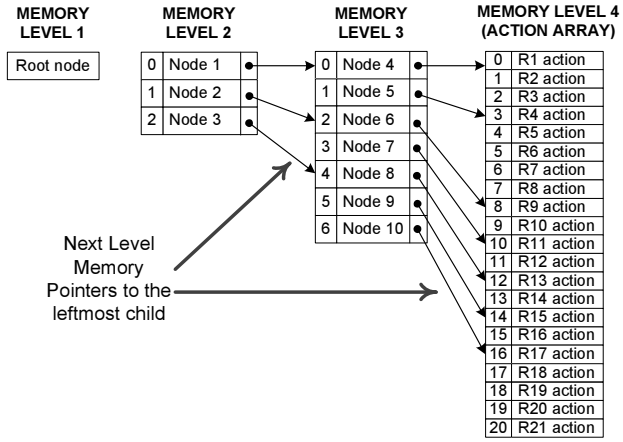
[5]To reduce memory requirements of the tree, the prefix pointer stored in a node can be identical to the pointer to the node. This may come at the cost of increased storage requirements at the end-result memory which stores the actions associated with each prefix.



(a) An example of a Range Trie structure.



(b) Range Trie Hardware Block Diagram.



(c) Range Trie Memory levels.

Figure 8: Range Trie top-level hardware design and memory levels.

4. RANGE TRIE HARDWARE DESIGN

It is more efficient to implement a Range Trie in hardware rather than in software where multiple bit-manipulation instructions are required to shift addresses, select parts of addresses to be compared and select the matching address range. Nevertheless, software implementations can also benefit from the Range Trie data-structure given that reducing the number of memory accesses would substantially improve performance. Bellow, we describe our Range Trie hardware design.

The Range Trie hardware design, as illustrated in Figure 8(b), is pipelined as interleaving processing stages and memory access stages, the number of which is equal to the number of implemented Range Trie levels. Figures 8(a) and 8(b) show an example of a Range Trie and the way each level maps to the hardware implementation. The first level consists of only the root node and therefore a few registers are sufficient to store the node configuration. The incoming address and the root node configuration feed the first processing stage which performs the necessary computations to determine the first node branch to be taken. Subsequently, the second level of memory is accessed to provide the next node configuration to the second processing stage. When reaching a leaf, we read the action related to the matching range from the last memory level.

Figure 8(c) depicts the memory organization and contents for the above example. Each memory line stores the configuration of a node in the corresponding level. Memory addressing is performed similarly to that in the Tree Bitmap Tries [5]; each node stores a single pointer to its leftmost child and a subsequent addition between this pointer and the processing stage outcome determines the address of the child node to be read at the next level. In so doing, only one pointer per node needs to be stored with the restriction of storing contiguously all children nodes of a given Range Trie node.

Figure 9 offers the block diagram of a Range Trie processing stage which performs the necessary computations in a Range Trie node to determine its next branch for a given incoming address. The configuration of a given node to be processed, previously read from the memory of the respective tree-level, provides the address parts to be compared along with the necessary control bits which determine: (i) the incoming address parts to be compared, (ii) address alignment information (Rule 5), (iii) comparison lengths, (iv) common prefix/suffix comparisons, and (v) a pointer to the next memory level.

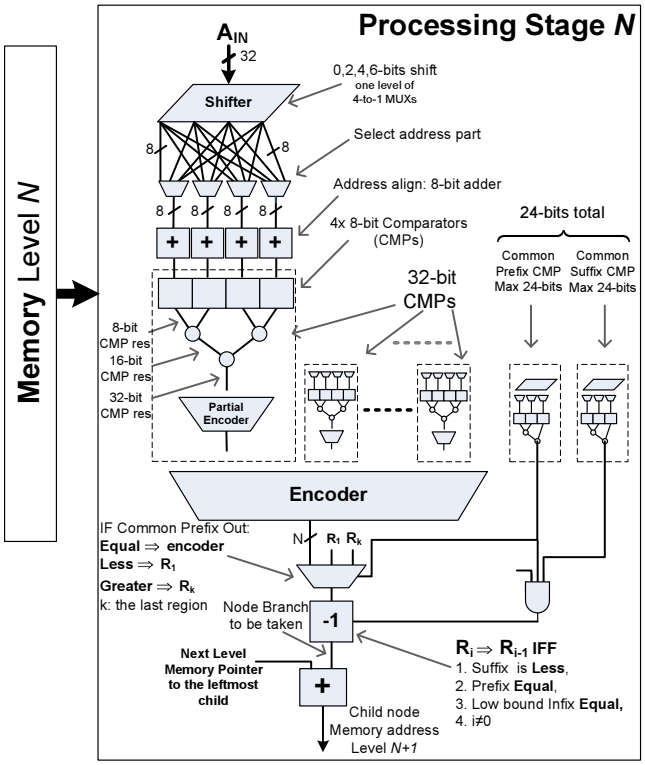
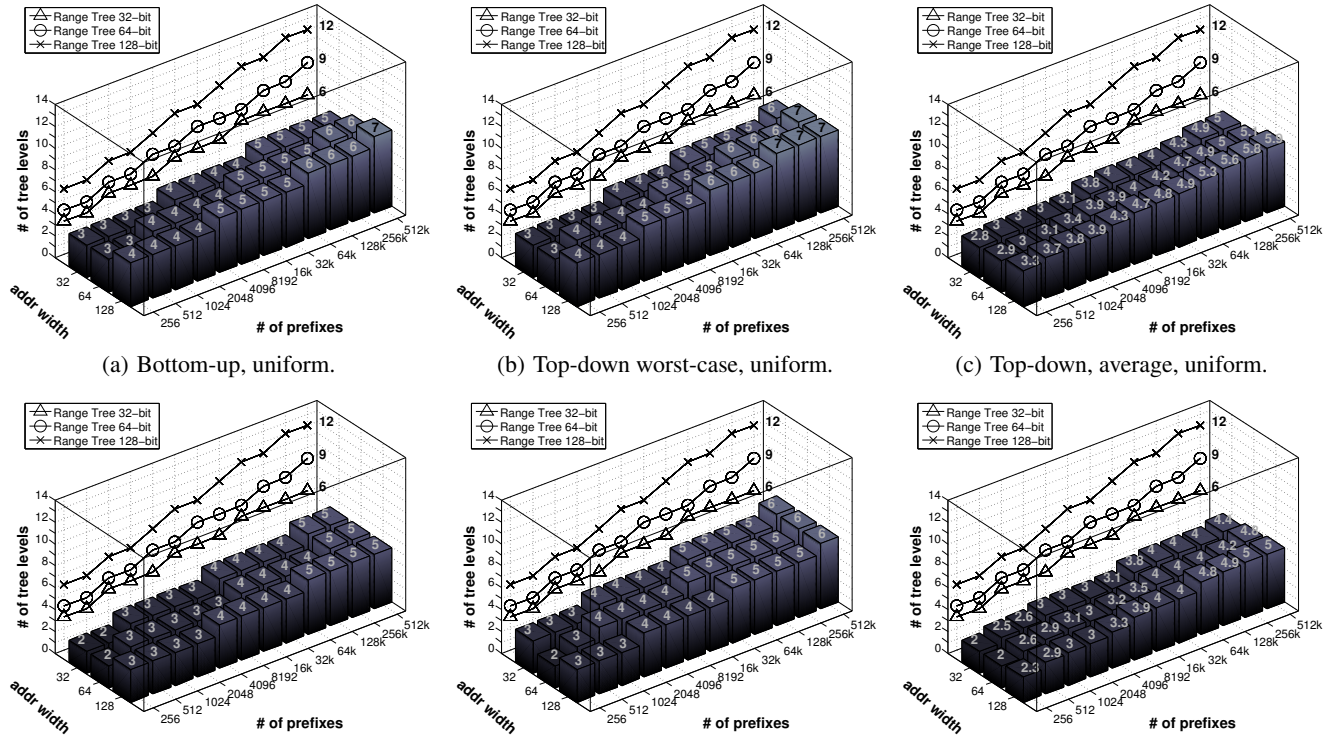


Figure 9: Range Trie processing stage.

The incoming address is first shifted properly (byte align) and a part of it is selected according to the node configuration. Subsequently, a constant value is subtracted from the incoming address parts according to the Range Trie Rule 5[6]. The result feeds multiple comparators of length equal to the address width W[7], which in this example is 32-bits. The second input of the comparators are the Range Trie address parts of the node. Each 32-bit comparator comprises of four 8-bit comparators, the results of which can be

[6]In case Rule 5 is disabled, the subtraction value is equal to *zero*.
[7]It is also possible comparators length to be less than W.



(a) Bottom-up, uniform. (b) Top-down worst-case, uniform. (c) Top-down, average, uniform.

(d) Bottom-up, Gaussian. (e) Top-down worst-case, Gaussian. (f) Top-down, average, Gaussian.

Figure 10: Range Trie vs. Range Tree number of tree levels: uniform and Gaussian distribution of 256-512k prefixes considering 32, 64 and 128-bits address width and 256-bits/cycle memory bandwidth.

combined to report 16-bits or 32-bits comparisons. Based on the node configuration, each 32-bit comparator can be configured to perform comparisons of 8, 16, or 32 bits, or combinations of them. For example a 32-bit comparator can be configured to perform 3 comparisons of 8, 8 and 16-bits. The available memory bandwidth determines the number of 32-bit comparators. Subsequently, the comparators results are encoded in two steps, first per 32-bit comparator and then for the entire node. In parallel, we perform the common prefix and common suffix comparisons, the results of which are considered after encoding, according to the Range Trie Rules 3 and 4. All of the above determine the node branch to be taken according to the node configuration and the incoming address. In order to compute the memory address for the next level, the node-branch id is added to a next-level memory pointer to the leftmost child of the node.

In general, Range Tries are suitable for single chip hardware implementations due to their low memory requirements and number of tree levels (and pipeline stages), as shown in the next Section 5. Although significant processing is required for a Range Trie node, the above design manages to support variable length comparisons in an efficient way and provides a balanced delay comparable to the memory access delay which leads, as explained in the Section 5.3 in high operating frequencies. A more detailed description of the Range Trie hardware design is offered in [17].

5. EVALUATION

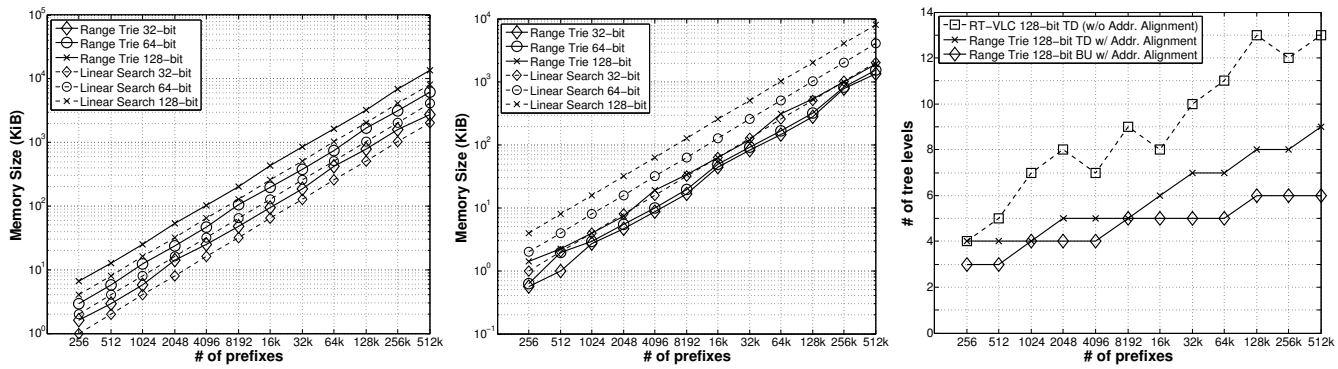
We evaluate the proposed Range Trie data-structure and hardware design and measure performance, memory requirements, power consumption, as well as the scalability of the above with the address width and the routing table size. We first use synthetic and real IPv4 and IPv6 routing tables for constructing a Range Trie and count tree-levels, which correspond to memory accesses (and latency) per lookup, and the required search memory size. Then, we compare the above results with previous address lookup data-structures. Subsequently, we evaluate several Range Trie hardware designs in 90-nm CMOS technology, varying the address width and supported routing table sizes, and measure frequency, throughput, area, and power consumption. In our evaluation, we consider a memory bandwidth per tree-level of 256-bits per cycle.

5.1 Synthetic Routing Tables

We first use synthetic routing tables with a uniform and a Gaussian distribution of prefixes which define address ranges. The above routing tables are not meant to be indicative for IPv4 and IPv6 tables, however, they provide useful input to evaluate the scalability of the proposed approach. Hence, we can generate large synthetic tables of 128-bit addresses, while current real IPv6 routing tables contain only 1.6k prefixes, and also tables of 64-bit addresses in order to have a third point to the address width axis.

We construct Range Tries using both the top-down and bottom up heuristics described in Section 3.2. Range Tries are then compared to the theoretical best-case Multiway Range Tree, which in terms of lookup latency is known to scale better than Trie-based solutions [14]. Multiway Range Trees, as described in [19], are constructed using B-Trees which may not maximize the number of branches per node and therefore may require more levels. In this comparison, we consider that Range Trees have the theoretical minimum number of tree-levels.

We generated synthetic sets of 256-512k prefixes of 32, 64 and 128-bits address width. Range Trie performs better for address ranges generated with Gaussian distribution (Fig. 10(d), 10(e), and



(a) Bottom-up, uniform

(b) Bottom-up, Gaussian

(c) Effect of Range Trie Address Alignment vs. RT-VLC [15].

Figure 11: In (a) and (b) the memory requirements of the Range Trie vs. linear search is shown. In (c) the effect of address alignment in the height of a Range Trie is evaluated.

10(f)) rather than the uniform (Fig. 10(a), 10(b), and 10(c)); that is because in the Gaussian distribution prefix bounds are concentrated in a smaller range and therefore have large common address parts.

Range Tries constructed by the top-down heuristic may have leaf nodes in more than one tree-levels and therefore their worst case lookup latency differs from the average one. On the contrary, the bottom-up heuristic creates Range Tries which have all their leafs in the same tree level, and consequently, worst and average lookup latencies are equal. As shown in Figure 10, the top-down heuristic generates Range Tries with better average lookup latency (Fig. 10(f), 10(c)) compared to the bottom-up (Fig. 10(d), 10(a)), however in the worst case (Fig. 10(e), 10(b)) they are deeper.

In general, Range Trie height and latency scales better than the Range Tree as the number of prefixes increase, even for addresses of 32-bits width. Scalability improves even more as we increase the address width. Moving from 32-bit addresses to 64 and 128 bits adds one or no extra tree level for the Range Trie, as opposed to the Range Tree which requires 3 more levels when doubling the address width for sets of 512k bounds. In all cases, even for 128-bit addresses, a Range Trie needs at most 7 tree levels. On the contrary, a Range Tree for 128-bit addresses and 512k address ranges needs at least 12 tree levels, while any Trie-based structure would need 16 or 32 tree levels considering strides of 8 and 4 bits, respectively.

The Range Trie memory requirements are shown in Figures 11(a) and 11(b) for the bottom-up heuristic (which is the most efficient) and for both the uniform and Gaussian distributions. In both cases, memory size scales similarly to a linear search algorithm. For the uniform distribution the Range Trie memory size is about 2× the one of the linear search, while in the Gaussian distribution -where sharing is higher- Range Tries need about 50% the memory of the linear search. For 512K ranges the Range Trie memory size is 1.4-2.7, 1.5-6.1, and 1.9-13.5 Mbytes for 32, 64 and 128 bits address width, respectively.

Another interesting result is retrieved when evaluating the effect of the Range Trie address alignment rule. In Figure 11(c) we have arranged the mean of the Gaussian distribution of prefixes such that the Range Trie creates nodes similar to the example of Figure 7. These nodes may map in narrow address ranges, the addresses of which however do not have common parts. The only way of sharing address parts in these cases is to perform address alignment as discussed in Rule 5. Figure 7 shows that for the above sets an RT-VLC[8] [15] creates a tree with excessive height; for 128-bit addresses and 512k the RT-VLC has 13 tree-levels. When enabling address alignment, the Range Trie tree-levels are reduced to 9 and 6 for the top-down and the bottom-up heuristics, respectively.

5.2 Real IPv4 & IPv6 Routing Tables

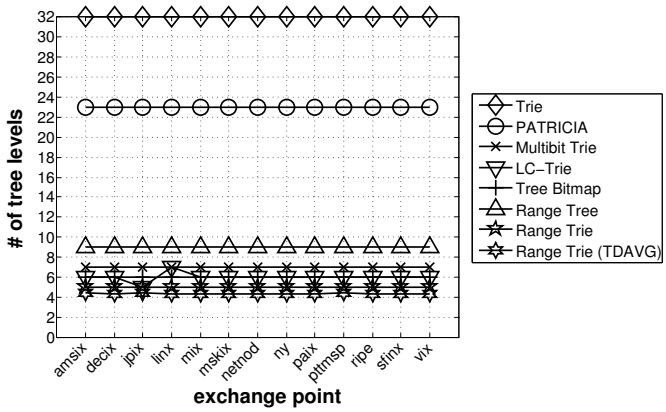
Next, we evaluate the Range Tries lookup latency and memory requirements on real IPv4 and IPv6 routing tables considering again 256-bit memory bandwidth. We further compare with other existing address lookup algorithms. We were able to find implementations of other algorithms for 32-bit IPv4 addresses, however, we were able to only estimate performance of related address lookup data-structures for the case of IPv6.

We used real IPv4 routing tables collected in 8/8/2008 from thirteen locations found in [13], each one having 262,310 to 275,706 prefixes (314,756-330,445 address bounds). As depicted in Figure 12(a) the Range Trie (bottom-up heuristic) requires 5 tree levels in the worst case and 4.1-4.5 on average (top-down heuristic) for the above IPv4 tables. The second best is the Tree bitmap which needs 6 levels considering strides of 13-4-4-4-4-3 bits, as described in [5]. Similar lookup latency is achieved by the LC-tries [11] which for the JPIX table needs 5 levels, for the LINX needs 7, and for the rest needs 6 tree-levels. Multibit tries require 7 levels considering strides of 8-4-4-4-4-4-4. The Multiway Range Tree as described in [19] using B-trees requires 9 levels. Finally, the Patricia algorithm [10] requires 23 tree levels, and a simple Trie needs 32 levels.

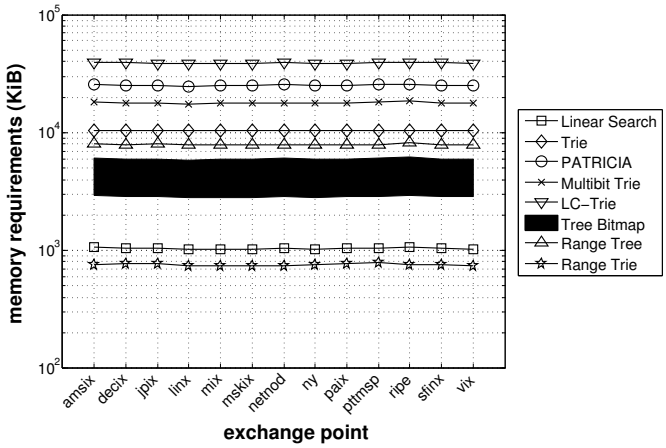
Figure 12(b) depicts the memory requirements of the proposed approach and existing algorithms for the IPv4 routing tables. A Range Trie needs about 75% the memory size of the linear search algorithm, in total about 0.75 Mbytes. Tries using Tree bitmaps, considering 11-23 bytes per prefix as reported in [5], need about 4-5× more memory. Simple Tries and Range Trees require about 10× more memory, while the rest of the algorithms are more than 20× worse.

Figure 12(c) illustrates the number of tree levels for the Range Trie, the best case Range Tree, and Trie-based structures for real IPv6 routing tables retrieved from Hurricane Electric, and BGP (AS6447, AS2.0) [1]. These three sets have only 1631, 1597, and 1625 prefixes, which map to 2997, 2558, and 2819 unique address bounds, respectively. Although, the small size of the tables reduces

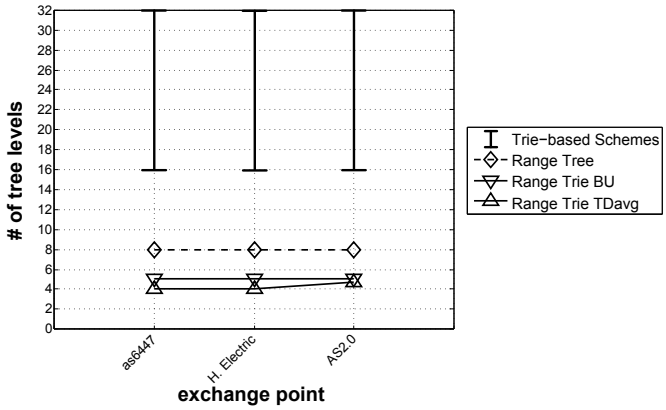
[8] An RT-VLC is a Range Tree that compares parts of addresses, but as opposed to the Range Trie, RT-VLC does not perform address alignment [15]. In addition, each RT-VLC node supports comparisons of only a single length (e.g. a RT-VLC node may compare a single address part of either 8 or 16 or 32 bits).



(a) Number of tree-levels in Range Trie vs. related data-structures when storing RIPE IPv4 routing tables of 270K prefixes [13].



(b) Memory size of Range Trie vs. related data-structures for the RIPE IPv4 routing tables of about 270K prefixes [13].



(c) Number of tree-levels in Range Trie vs. related data-structures when storing Real IPv6 routing tables of 1.6K prefixes [1].

Figure 12: Range Tries latency and memory requirements vs. related data-structures using real IPv4 and IPv6 routing tables.

the benefits of the Range Trie, it still performs better requiring only 5 tree levels in the worst case and 4 on average as opposed to 8 for the theoretical best-case Range Tree. We can estimate that Trie-based schemes would need at least 16 to 32 tree levels considering strides of 8 and 4 bits respectively. It is worth noting, that the most efficient Trie-based structure, the Tree-bitmaps, increases its memory bandwidth requirements 5 times when increasing the strides length from 4 to 8 bits [6]. For the above IPv6 tables, Range Tries need 18, 21 and 64 Kbytes of memory, while a linear search needs about 40Kbytes.

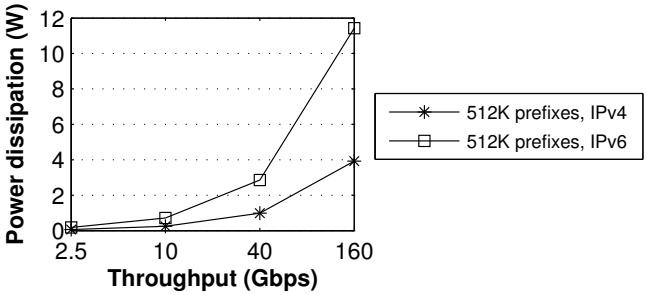


Figure 14: Range Trie Power Consumption for different throughputs.

We discuss next separately a more recent related work, CAMP, which uses multibit tries in a circular pipeline [8]. CAMP needs 5-10 levels for IPv4 and an memory size of roughly 1.9-2.6 Mbytes for the above routing tables. That is at best equal latency with the Range Trie and over 2× more memory; however, for IPv6 tables CAMP is expected to scale linearly both in latency and memory.

5.3 Hardware Implementation Results

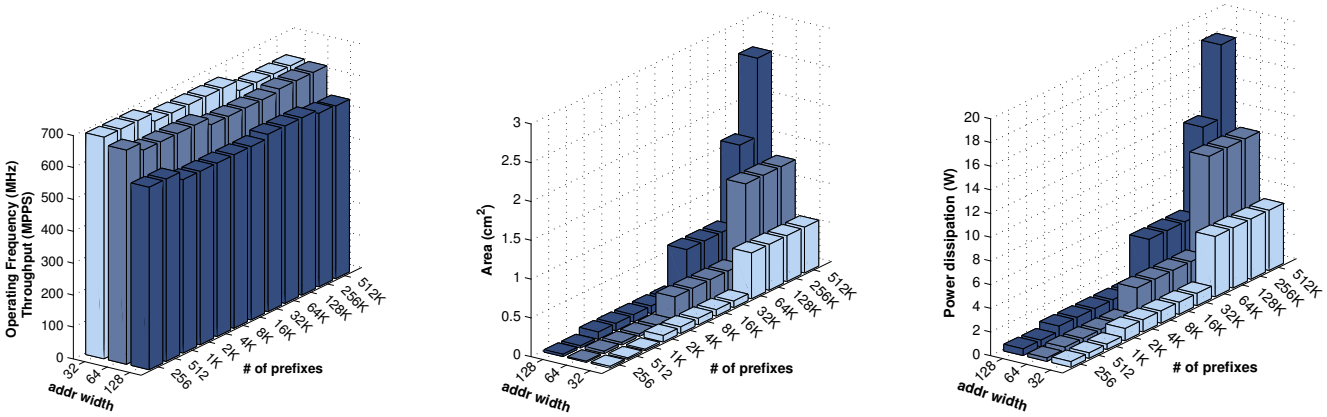
We have implemented several Range Trie design points in hardware, as described in Section 4, considering 90nm UMC CMOS technology and report post-synthesis results using Synopsys ASIC design tools. In order to evaluate the scalability of the designs we implemented Range Tries that fit 256-512k prefixes for address widths of 32-, 64-, and 128-bits. As shown in Figure 13, the above designs are evaluated in terms of area cost, power consumption, and operating frequency, which corresponds also to maximum throughput in processed packets per second (MPPS). Finally, we measured the power consumption of the largest designs when varying the throughput requirements from OC-48 to OC-3072.

Figure 13(a) depicts the Range Tries operating frequency and throughput. All designs could operate above 500MHz and up to almost 700 MHz (in the case of 32-bit addresses) showing that the processing stage has similar latency with the on-chip memory blocks. The operating frequency remains almost constant for increasing routing table sizes and scales quite well to the address width. The corresponding throughput is also between 500-700 MPPS which suffices for supporting OC-3072 wire speeds (160 Gbps) considering 40 bytes packet size.

The area cost of the designs is shown in Figure 13(b) and scales linearly to the address width and also to the routing table size. Designs that store 0.5 million entries occupy 0.5, 1.2 and 2.5 cm^2 for address widths of 32-, 64-, and 128-bits, respectively. The primary reason for the linear scalability of area is the over-provisioning of the memory requirements of each design.

Figure 13(c) depicts the power consumption of the Range Trie designs which scales similarly to the area cost. This is to be expected since the area cost is mainly due to the on-chip memory and memory is the main source of power consumption. Range Trie designs of 0.5 million entries consume 5, 10.6 and 18 Watts for address widths of 32-, 64-, and 128-bits, respectively. For some designs unused memory blocks could be turned-off to save power.

The above power dissipation results are for designs that operate at their maximum frequency. Figure 14 shows the power consumption of the largest Range Trie designs (storing 512k prefixes) for required throughput of OC-48, OC-192, OC-768, and OC-3072. In



(a) Range Trie throughput & frequency. (b) Range Trie area. (c) Range Trie power consumption.

Figure 13: Range Trie implementation results for 90-nm UMC technology. Different design points have been implemented for address widths 32-, 64-, and 128-bits, and routing tables of 256-512k entries. The memory bandwidth per stage is 256-bits per cycle.

this evaluation we consider packet sizes of 40- and 60-bytes for IPv4 and IPv6 respectively. In general, the IPv6 design consumes about 3 times more power than that of IPv4. For up to 10 Gbps throughput the power consumption is 0.1-0.7 Watts for both IPv4 and IPv6 cases. For 40 Gbps, IPv4 design dissipates 1 Watt and IPv6 2.8 Watts. Finally, for 160 Gbps throughput IPv4 design consumes 3.9 Watts and IPv6 11.4 Watts.

Compared to CAMP, storing 600k IPv4 prefixes and implemented in the same technology (90-nm CMOS) [8], Range Tries can support the same throughput 160 Gbps. Considering the difference in the number of stored prefixes between the two designs, Range Tries require half the area of CAMP (0.5 vs. 1.25 cm^2) mostly due to the reduced memory requirements. Finally, Range Tries power consumption is 3.9 Watts while CAMP consumes 2.2-4 Watts mainly due to the balanced CAMP memory requirements per pipeline stage. A circular pipeline could be applied to Range Tries as well. Alternatively, the Range Tries power consumption can be further reduced by splitting the large memories of the last stages to multiple banks at the expense of increased area cost. Then, only one bank per stage would be accessed dissipating less power.

6. CONCLUSIONS

We have introduced the Range Trie, a new powerful data structure for address lookup which performs comparisons of parts of addresses. We described five rules employed to construct a Range Trie and to reduce the required address bits per comparison. Furthermore, we offered a Range Trie pipelined hardware design which employs comparators each of which can perform multiple variable length comparisons. We showed the benefits of the proposed approach in terms of performance, memory size and scalability. Considering 256-bits/cycle memory bandwidth, half a million IPv4 and IPv6 prefixes can be stored in 5 and 7 Range Trie levels respectively; that is at least 15% and 40% lower lookup latency than related works. The memory requirements are 50% to 2× that of a linear search and more than 2× better compared to related techniques. The proposed hardware design supports 160 Gbps throughput and consumes 3.9 and 11.4 Watts for IPv4 and IPv6 routing tables, respectively. Range Tries have substantially better scalability in the address width than any known Trie-based approach which scales only linearly, but also better than the Range Trees which are limited by the memory bandwidth; this offers a great advantage for Range Trie in IPv6 routing and packet classification.

7. REFERENCES

[1] http://bgp.potaroo.net/.

[2] F. Baboescu, D. M. Tullsen, G. Rosu, and S. Singh. A tree based router search engine architecture with single port memories. In *ISCA '05*, pages 123–133, Washington, DC, USA, 2005.

[3] R. de Smet. Range trie heuristics for variable-size address region lookup. Master's thesis, TU Delft, Computer Engineering, May 2009.

[4] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink. Small forwarding tables for fast routing lookups. *SIGCOMM Comput. Commun. Rev.*, 27(4):3–14, 1997.

[5] W. Eatherton, G. Varghese, and Z. Dittia. Tree bitmap: hardware/software IP lookups with incremental updates. *SIGCOMM Comput. Commun. Rev.*, 34(2):97–122, 2004.

[6] W. N. Eatherton and Z. Dittia. Data structure using a tree bitmap and method for rapid classification of data in a database. US Patent 6728732, July 2007.

[7] P. Gupta and N. McKeown. Algorithms for packet classification. *IEEE Network*, 15(2):24–32, Mar/Apr 2001.

[8] S. Kumar, M. Becchi, P. Crowley, and J. Turner. CAMP: fast and efficient IP lookup architecture. In *ANCS '06*, pages 51–60, 2006.

[9] H. Lu and S. Sahni. A B-Tree dynamic router-table design. *IEEE Trans. Comput.*, 54(7):813–824, 2005.

[10] D. R. Morrison. PATRICIA—practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, 1968.

[11] S. Nilsson and G. Karlsson. IP-address lookup using LC-tries. *IEEE JSAC*, 17(6):1083–1092, 1999.

[12] NRO: IPv6 Growth Increases 300 Percent in Two Years. www.nro.net/documents/press_release_031108.html, Dec 2008.

[13] RIPE Network Coordination Centre. http://www.ripe.net/.

[14] M. Ruiz-Sanchez, E. Biersack, and W. Dabbous. Survey and taxonomy of ip address lookup algorithms. *IEEE Network*, 15(2):8–23, Mar/Apr 2001.

[15] I. Sourdis, R. de Smet, and G. Gaydadjiev. Range trees with variable length comparisons. In *IEEE Workshop on High Performance Switching and Routing*, Paris, France, France, 2009.

[16] V. Srinivasan and G. Varghese. Fast address lookups using controlled prefix expansion. *ACM Trans. Comput. Syst.*, 17(1):1–40, 1999.

[17] G. Stefanakis. Design and implementation of a range trie for address lookup. Master's thesis, TU Delft, Computer Engineering, July 2009.

[18] D. E. Taylor. Survey and taxonomy of packet classification techniques. *ACM Comput. Surv.*, 37(3):238–275, 2005.

[19] P. Warkhede, S. Suri, and G. Varghese. Multiway range trees: scalable ip lookup with fast updates. *Comput. Netw.*, 44(3):289–303, 2004.