HEXA: Compact Data Structures for Faster Packet Processing

Sailesh Kumar, Jonathan Turner, Patrick Crowley
Washington University
Computer Science and Engineering
{sailesh, jst, pcrowley}@arl.wustl.edu

Michael Mitzenmacher
Harvard University
Electrical Engineering and Computer Science
michaelm@eecs.harvard.edu

***Abstract*—Data structures representing directed graphs with edges labeled by symbols from a finite alphabet are used to implement packet processing algorithms used in a variety of network applications. In this paper we present a novel approach to represent such data structures, which significantly reduces the amount of memory required. This approach called *History-based Encoding, eXecution and Addressing* (HEXA) challenges the conventional assumption that graph data structures must store pointers of $\lceil \log_2 n \rceil$ bits to identify successor nodes. We show how the data structures can be organized so that implicit information can be used to locate successors, significantly reducing the amount of information that must be stored explicitly. We demonstrate that the binary tries used for IP route lookup can be implemented using just two bytes per stored prefix (roughly half the space required by Eatherton's tree bitmap data structure) and that string matching can be implemented using 20-30% of the space required by conventional data representations.**

Compact representations are useful, because they allow the performance-critical part of packet processing algorithms to be implemented using fast, on-chip memory, eliminating the need to retrieve information from much slower off-chip memory. This can yield both substantially higher performance and lower power utilization. While enabling a compact representation, HEXA does not add significant complexity to the graph traversal and update, thus maintaining a high performance.

***Index Terms*— content inspection, IP lookup, string matching**

I. Introduction

Several common packet processing tasks make use of directed graph data structures in which edge labels are used to match symbols from a finite alphabet. Examples include tries used in IP route lookup and string-matching automata used to implement deep packet inspection for virus scanning. In this paper, we develop a novel representation for such data structures that is significantly more compact than conventional approaches. This compactness can lead to higher performance in implementation contexts where we have small on-chip memories with ample memory bandwidth and larger off-chip memories with more limited bandwidth. These characteristics are common to conventional processors, network processors, ASICs and FPGA implementations.

We observe that the edge-labeled, directed graphs used by some packet processing tasks have the property that for all nodes u, all paths of length k leading to u are labeled by the same string of symbols, for all values of k up to some bound. For example, tries satisfy this condition trivially, since for each value of k, there is only one path of length k leading to each node. The data structure used in the Aho-Corasick string matching algorithm [2] also satisfies this property, even though in this case there may be multiple paths leading to each node. Since the algorithms that traverse the data structure know the symbols that have been used to reach a node, we can use this "history" to define the storage location of the node. Since some nodes may have identical histories, we need to augment the history with some discriminating information, to ensure that each node is mapped to a distinct storage location. We find that in some applications the amount of discriminating information needed can be remarkably small. For binary tries for example, two bits of discriminating information is sufficient. This leads to a binary trie representation that requires just two bytes per stored prefix for IP routing tables with more than 100K prefixes. We call the technique used to construct these compact data representations, *History-based Encoding, eXecution and Addressing* (HEXA).

In Section II, we introduce HEXA and apply it to binary tries. We show that the problem of selecting discriminators corresponds to finding a perfect matching in a bipartite graph; we also show how the data structure can be incrementally modified. In Section III, we describe a variant of HEXA in which the discriminator specifies the amount of history information that has to be used to identify the storage location of a node. We then apply this technique to the data structure used by the Aho-Corasick string matching algorithm as well as the bit-split version of the algorithm [6]. In Section IV we report on the results of our evaluation of HEXA for binary tries and string matching. Section V covers the related work and the paper ends with concluding remarks in Section VI.

II. Introduction to HEXA

Directed graphs are commonly used to implement various packet processing algorithms which are used in a variety of network applications, some of which are listed below:

- **Longest prefix match IP lookup**: IP routing involves a longest prefix match, where destination IP address of a packet is matched against a large but finite set of prefixes and the longest matching prefix determines the next hop.

1-4244-1588-8/07/$25.00 ©2007 IEEE

Tries, which essentially are a directed graph without any cycles, are often used to implement such operations.

- **Packet classification**: Packet classification involves a multi-dimensional search on packet's 5-tuple (source/ destination addresses, ports and protocol). Search in each dimension often consists of a longest prefix match, which is commonly implemented using tries. These tries usually have a similar structure as an IP lookup trie.
- **String matching**: Commercial network security devices like network intrusion detection systems (NIDS), and application layer firewalls often use string based pattern matching to identify malicious packets. String matching is usually performed with the aid of a finite automaton (*e.g.* Aho-Corasick, Wu-Manber etc), which is a directed graph with labeled edges. Nodes of these graphs usually have much higher and varying out-degrees.
- **Regular expression matching**: Modern security systems specify the patterns of interest using regular expressions. Regular expressions are also used to enable advanced network services like content based routing, metering, etc. Finite automata are usually used to implement regular expressions, which are again a labeled directed graph. Complex expressions usually lead to relatively complex graphs, as compared to a string based automaton.
- There are several other applications, which use directed graph structures. Some examples are a **web indexing** and **search engines**, an **access control list** (ACL), or even a **file system**. In this paper, we will mostly focus on the first four applications.

Since such a wide variety of network applications employ some form of directed graph traversal, a large body of research literature has focused on improving its performance. For example, [11] propose a multi-bit trie representation, where multiple nodes of a binary trie are merged into a single node. There are also schemes to compactly encode these multi-bit trie nodes [13]. Another class of directed graphs is finite automaton; in [5] authors present techniques to improve the parsing performance of a finite automaton, which is used to perform string matching. It uses a similar technique, where multiple states of the automaton are merged into a single state and represented compactly. In [6], authors propose an alternative technique to reduce the space by reducing the number of transitions from every node of the graph.

Most of these solutions are too specialized; fine tuned and optimized for their respective applications, however a common link between them is that they reduce the memory by either reducing the number of transitions in the graph or by reducing the number of nodes. They also demonstrate that the space reduction achieved by reducing the number of nodes and/or transitions may also enhance the parsing performance of the graph, by utilizing the fast but limited on-chip memory.

With or without the reductions in the number of nodes or transitions, to our best knowledge, directed graphs are always implemented in the following conventional manner. Each node in the n node graph is denoted by a unique $\lceil \log_2 n \rceil$ bit identifier, which also determines the memory location of the node. At this memory location, all transitions of the node (identifiers of the subsequent "next nodes") are stored, along with some auxiliary information. The auxiliary information may be a flag indicating if the node corresponds to a match in a string matching automata or a valid prefix in an IP lookup trie, and an identifier for the string, or the next hop for the matching prefix. The auxiliary information usually requires only a few bits and is kept once for every node; on the other hand, identifiers of the "next node" use $\lceil \log_2 n \rceil$ bits each and are required once for every transition. Thus, in large graphs (say a million nodes) containing multiple transitions per node (say two), the memory required by the identifiers of the "next node" (20-bits per identifier, 2 such identifiers per node) can be much higher than the memory required by the auxiliary information.

Another complicating factor in the conventional design approach is that, the transitions or the identifiers of the "next node" are read for each symbol in the input stream, while the auxiliary information is read only upon a match. This necessitates that the "next node" identifiers be stored in a high speed memory (e.g. SRAM or embedded) in order to enable high parsing rate. For instance, a high performance lookup trie may store the set of "next nodes", for every node, in a fast memory along with a flag indicating whether the node corresponds to a prefix. On the other hand, the next hop information can be kept with a shadow trie, stored in a slow memory like DRAM. Similarly, in string matching automaton, in addition to the "next node" identifiers, only a flag per node is needed in the fast memory, which will indicate whether the node is a match. The prime motivation of such separation of fast and slow path is to reduce the high speed memory, which is often expensive and less dense. The advantages are however undermined as the identifiers of the "next node" represent a large fraction of the total memory. While there is a general interest in reducing the total memory, clearly there are increased benefits in reducing the memory required to store these "next node" identifiers.

In this paper, we propose a new method to store directed graph structures that we dub HEXA (History based Encoding, eXecution, and Addressing). While conventional methods use $\lceil \log_2 n \rceil$ bits to identify each of n nodes in a graph, by taking advantage of the graph structure, HEXA employs a novel method that can use a fixed constant number of bits per node for structured graphs such as tries. Thus, when HEXA based identifiers are used to denote the transitions of the graph, the fast memory needed to store these transitions can be dramatically reduced. The total memory is also reduced significantly, because auxiliary information often represents a small fraction of the total memory.

The key to the identification mechanism used by HEXA is that when nodes are not accessed in a random ad-hoc order but in an order defined by its transitions, the nodes can be identified by the way the parsing proceeds in the graph. For instance, in a trie, if we begin parsing at the root node, we can

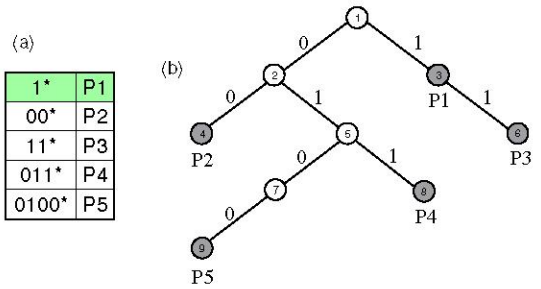


Figure 1: a) routing table, b) corresponding binary trie.

reach any given node only by a unique stream of input symbols. In general, as the parsing proceeds, we need to remember only the previous symbols needed to uniquely identify the nodes. To clarify, we consider a simple trie-based example before formalizing the ideas behind HEXA.

A. *Motivating Example*

Let us consider a simple directed graph given by an IP lookup trie. A set of 5 prefixes and the corresponding binary trie, containing 9 nodes, is shown in Figure 1. We consider first the standard representation. A node stores the identifier of its left and right child and a bit indicating if the node corresponds to a valid prefix. Since there are 9 nodes, identifiers are 4-bits long, and a node requires total 9-bits in the fast path. The fast path trie representation is shown below, where nodes are shown as 3-tuple consisting of the prefix flag and the left right children (NULL indicates no child):

1. 0, 2, 3
2. 0, 4, 5
3. 1, NULL, 6
4. 1, NULL, NULL
5. 0, 7, 8
6. 1, NULL, NULL
7. 0, 9, NULL
8. 1, NULL, NULL
9. 1, NULL, NULL

Here, we assume that the next hops associated with a matching node are stored in a shadow trie which is stored in a relatively slow memory. Note that if the next hop trie has a structure identical to the fast path trie, then the fast path trie need not contain any additional information. Once the fast path trie is traversed and the longest matching node is found, we will read the next hop trie once, at the location corresponding to the longest matching node.

We now consider storing the fast path of the trie using HEXA. In HEXA, a node will be identified by the input stream over which it will be reached. Thus, the HEXA identifier of the nodes will be:

1. -
2. 0
3. 1
5. 00
6. 01
7. 11
7. 010
8. 011
9. 0100

These identifiers are unique. HEXA requires a hash function; temporarily, let us assume we have a minimal perfect hash function f that maps each identifier to a unique number in [1, 9]. (A minimal perfect hash function is also called a one-to-one function.) We use this hash function for a hash table of 9 cells; more generally, if there are n nodes in the trie, n_i is the HEXA identifier of the i^{th} node and f is a one-to-one function mapping n_i's to [1, n], Given such a function, we need to store only 3 bits worth of information for each node of trie in order to traverse it: the first bit is set if node corresponds to a valid prefix, and second and third bits are set if node has a left and right child. Traversal of the trie is then straightforward. We start at the first trie node, whose 3-bit tuple will be read from the array at index $f(-)$. If the match bit is set, we will make a note of the match, and fetch the next bit from the input stream to proceed to the next trie node. If the bit is 0 (1) and the left (right) child bit of the previous node was set, then we will compute $f(n_i)$, where n_i is the current sequence of bits (in this case the first bit of the input stream) and read its 3 bits. We continue in this manner until we reach a node with no child. The most recent node with the match bit set will correspond to the longest matching prefix.

Continuing with the earlier trie of 9 nodes, let the mapping function f, has the following values for the nine HEXA identifiers listed above:

1. $f(-) = 4$
2. $f(0) = 7$
3. $f(1) = 9$
4. $f(00) = 2$
5. $f(01) = 8$
6. $f(11) = 1$
7. $f(010) = 5$
8. $f(011) = 3$
9. $f(0100) = 6$

With this one-to-one mapping, the fast path memory array of 3-bits will be programmed as follow; we also list the corresponding next hops:

	1	2	3	4	5	6	7	8	9
Fast path	1,0,0	1,0,0	1,0,0	0,1,1	0,1,0	1,0,0	0,1,1	0,1,1	1,0,1
Next hop	P3	P2	P4			P5			P1

This array and the above mapping function are sufficient to parse the trie for any given stream of input symbols.

This example suggests that we can dramatically reduce the memory requirements to represent a trie by practically eliminating the overheads associated with node identifiers. However, we require a minimal perfect hash function, which is hard to devise. In fact, when the trie is frequently updated, maintaining the one-to-one mapping may become extremely difficult. We will explain how to enable such one-to-one mappings with very low cost. We also ensure that our approach maintains very fast incremental updates; *i.e.* when nodes are added or deleted, a new one-to-one mapping can be computed quickly and with very few changes in the fast path array.

B. *Devising One-to-one Mapping*

We have seen that we can compactly represent a directed trie if we have a minimal perfect hash function for the nodes of the graph. More generally, we might seek merely a perfect hash function; that is, we map each identifier to a unique element of [1, m] for some $m \geq n$, mapping the n identifier into m array cells. For large n, finding perfect hash functions becomes extremely compute intensive and impractical.

We can simplify the problem dramatically by considering the fact that HEXA identifier of a node can be modified without changing its meaning and keeping it unique. For instance we can allow a node identifier to contain few additional (say c) bits, which we can alter at our convenience. We call these c-bits the node's *discriminator*. Thus, HEXA identifier of a node will be the history of labels on which we will reach the node, plus its c-bit discriminator. We use a (pseudo)-random hash function to map identifiers plus

discriminators to possible memory locations. Having these discriminators and the ability to alter them provides us with multiple choices of memory locations for a node. Each node will have 2^c choices of HEXA identifiers and hence up to 2^c memory locations, from which we have to pick just one. The power of choice in this setting has been studied and used in multiple-choice hashing [23] and cuckoo hashing [1], and we use results from these analyses.

Note that when traversing the graph, when trying to access a node we need to know its discriminator. Hence instead of storing a single bit for each left and right child, representing whether it exists or not, we store the discriminator if the child exists. In practice, we may also optionally reserve the all-0 c-bit word to represent NULL, giving us only 2^c-1 memory locations.

This problem can now be viewed as a bipartite graph matching problem. The bipartite graph $G = (V_1+V_2, E)$ consists of the nodes of the original directed graph as the left set of vertices, and the memory locations as the right set of vertices. The edges connecting the left to the right correspond to the edges determined by the random hash function. Since discriminators are c-bits long, each left vertex will have up to 2^c edges connected to random right vertices. We refer to G as the *memory mapping graph*. We need to find a *perfect matching* (that is, a matching of size n) in the memory mapping graph G, to match each node identifier to a unique memory location.

If we require that $m = n$, then it suffices that c is $\log \log n + O(1)$ to ensure that a perfect matching exists with high probability. More generally, using results from the analysis of cuckoo hashing schemes [1], it follows that we can have constant c if we allow m to be slightly greater than n. For example, using 2-bit discriminators, giving 4 choices, then $m = 1.1n$ guarantees that a perfect matching exists with high probability. In fact, not only do these perfect matchings exist, but they are efficiently updatable, as we describe in Section II.C.

Continuing with our example of the trie shown in Figure 1, we now seek to devise a one-to-one mapping using this method. We consider $m = n$ and assume that c is 2, so a node can have 4 possible HEXA identifiers, which will enable it to have up to 4 choices of memory locations. A complication in computing the hash values may arise because the HEXA identifiers are not of equal length. We can resolve it by first appending to a HEXA identifier, its length and then padding the short identifiers with zeros. Finally we append the discriminators to them. The resulting choices of identifiers and the memory mapping graph is shown in Figure 2, where we assume that the hash function is simply the numerical value of the identifier modulo 9. In the same figure, we also show a perfect matching with the matching edges drawn in bold. With this perfect matching, a node will require only 2-bits to be uniquely represented (as $c = 2$).

We now consider incremental updates, and show how a one-to-one mapping in HEXA can be maintained when a node is removed and another is added to the trie.

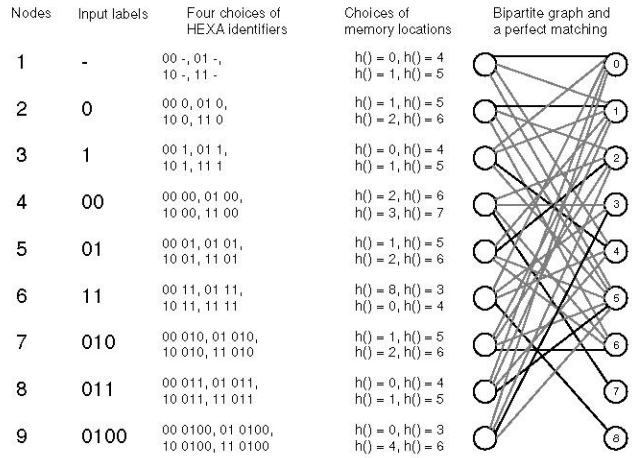


Figure 2: Memory mapping graph, bipartite matching.

C. *Updating a Perfect Matching*

In several applications, such as IP lookup, fast incremental updates are critically important. This implies that HEXA representations will be practical for the applications only if the one-to-one nature of the hash function can be maintained in the face of insertions and deletions. Taking advantage of the choices available from the discriminator bits, such one-to-one mappings can be maintained easily.

Indeed, results from the study of cuckoo hashing immediately yield fast incremental updates. Deletions are of course easy; we simply remove the relevant node from the hash table (and update pointers to that node). Insertions are more difficult; what if we wish to insert a node and its corresponding hash locations are already taken? In this case, we need to find an augmenting path in the memory mapping graph, remapping other nodes to other locations, which is accomplished by changing their discriminator bits. Finding an augmenting path will allow the item to be inserted at free memory location, and increasing the size of the matching in the memory mapping graph. In fact for tables sized so that a perfect matching exists in the memory mapping graph, augmenting paths of size $O(\log n)$ exist, so that only $O(\log n)$ nodes need to be re-mapped, and these augmenting paths can be found via a breadth first search over $o(n)$ nodes [1]. In practice, a random walk approach, where a node to be inserted if necessary takes the place of one of its neighbors randomly, and this replaced node either finds an empty spot in the hash table or takes the place of one of its other neighbors randomly, and so on, finds an augmenting path quite quickly [1].

We also note that even when $m = n$, so that our matching corresponds to a minimal perfect hash function, using $c = O(\log \log n)$ discriminator bits guarantees that if we delete a node and insert a new node (so that we still have $m = n$), an augmenting path of length $O(\log n / \log \log n)$ exists with high probability. We omit the straightforward proof.

We will demonstrate in our experiments that the number of changes needed to maintain a HEXA representation with node insertions and deletions is quite reasonable in practice. Again,

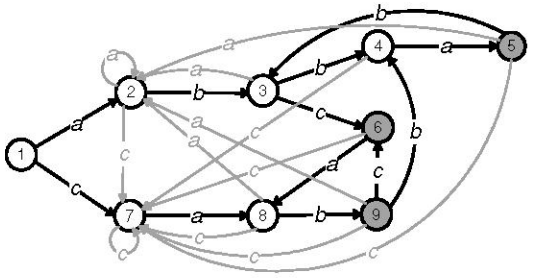


Figure 3: Aho-Corasick automaton for the three strings `abc`, `cab` and `abba`. Gray indicates accepting node

similar results can be found in the setting of cuckoo hashing.

III. BOUNDED HEXA (BHEXA)

Our current description of HEXA is useful when graph is acyclic and the total number of input symbols that we parse is bounded. However, in cyclic graphs, the HEXA identifiers may become unbounded if we continue traversing a loop and receiving input symbols. One way to enable bounded HEXA identifier is to restrict it to say previous k symbols, where k may be different for different nodes. However, this requires that all incoming k-long paths into all nodes of the graph have identical sequence of labels. Clearly, nodes of a general cyclic graph will not meet this requirement even for k=1 as there may be multiple incoming transitions into a node labeled with different symbols. Fortunately, a large number of cyclic graphs which are used in networking applications do not exhibit this property, and ensure that all incoming transitions into a node are labeled with identical symbol. In fact, all incoming k-long paths into a node are labeled with identical sequence of symbols, thus potentially creating long unique identifiers; notice that here k is different for different nodes.

The well known and widely used Aho-Corasick based string matching automata is one such cyclic graph. All k-long (k>0) paths leading into any node have identical sequence of labels, with root node being an exception. Several variants of string matching automata (*e.g.* Wu-Manber [4] and Commentz-Walter [3]), including the recently proposed bit-split version of Aho-Corasick [6], which is one of the fastest known embedded implementation, exhibit similar characteristics.

For such graphs, we introduce an extension called bounded HEXA (bHEXA) which examines a variable but finite number of symbols in the history to identify a node, instead of examining the entire history. Since the number of history symbols that we examine may be different for different nodes, bHEXA identifiers require additional bits to indicate this length. While these bits add up to the memory, having variable length identifiers also opens up another dimension of multiple choices of identifiers for the nodes, which helps in finding a one-to-one mapping and reduce the dependence on discriminator bits or even avoid using them. To clarify, we consider a simple string-based example.

A. *Motivating Example*

Let us consider Aho-Corasick automaton for the 3 strings: `abc`, `cab` and `abba`, defined over the alphabet {`a`, `b`, `c`}. The automaton (shown in Figure 3) consists of 9 nodes (all symbols for which a transition is not shown in the figure are assumed to lead to state 1). A standard implementation of this automaton will use 4-bit node identifiers. These identifiers will determine the memory location where the transitions of the node will be stored. There are three transitions per node (over symbols `a`, `b` and `c`, respectively) and assuming that a match flag is required for every node, the fast path memory will store four entries for each of the nine nodes, as shown below:

1. no, 2, 1, 7	4. no, 5, 1, 7	7. no, 8, 1, 7
2. no, 2, 3, 7	5. match, 2, 3, 7	8. no, 2, 9, 7
3. no, 2, 4, 6	6. match, 8, 1, 7	9. match, 2, 4, 6

Since node identifiers are 4-bits, in this case a node requires 13-bits of fast path memory. We now attempt to use bHEXA to represent this automaton. Since bHEXA allows identifiers to contain variable number of input symbols from the history, our first objective is to identify the legitimate bHEXA identifiers for the nodes. Clearly, we would like to keep the identifier unique for each node, irrespective of the path that leads to the node. The identifier of the root node is "–", as it is visited without receiving any input symbol (zero path length). The identifiers of the nodes which are one transition away from the root may contain up to one symbol from the history because all single transition path that will lead to such nodes will be labeled with identical symbol. As an example, all incoming edges into node 2 are labeled with `a`; thus its identifier can either be – or `a`. Similarly, the identifier of node 7 can be – or `c`. In general, a node which is k transitions away from the root may have the bHEXA identifier of any length up to k symbols. For example, both paths $1 \xrightarrow{a} 2 \xrightarrow{b} 3$ and $9 \xrightarrow{b} 4 \xrightarrow{a} 5 \xrightarrow{b} 3$ leads to the node 3, and the last two symbols in these paths are identical; consequently, its bHEXA identifier can either be – or `b` or `ab`. Choices of bHEXA identifiers for the remaining nodes are listed below:

1. –	5. –, `b`, `bb`, `abb`	7. –, `c`
2. –, `a`	6. –, `a`, `ba`, `bba`, `abba`	8. –, `a`, `ca`
3. –, `b`, `ab`	7. –, `c`, `bc`, `abc`	9. –, `b`, `ab`, `cab`

Notice that each of the above bHEXA identifier is legitimate. However, we must ensure that, the ones we choose are unique, so that no two nodes end up with identical identifiers. If we employ c-bit discriminators with bHEXA identifiers then we may allow up to 2^c nodes pick identical identifiers and then use different discriminator values to make them unique. The memory mapping method that we present in the next section enforces these constraints and ensures that bHEXA identifier of each node is unique.

B. *Memory Mapping*

The next step is to select a bHEXA identifier for every node, such that they are mapped to unique memory locations. A large fraction of nodes, being away from the root node, are likely to have several choices of bHEXA identifiers, which will improve the probability of a one-to-one mapping. These choices however come at a cost; if a node has k choices (can have up to k–1 symbols long bHEXA identifier) then up to

$\lceil \log_2 k \rceil$ additional bits may be needed to indicate the length of its identifier. During the graph traversal, these bits will be required to determine the exact number of history symbols that forms the bHEXA identifier of the node. In our example automaton, node 5 has 5 choices; hence 3-bits may be needed to indicate the length of its bHEXA identifier. We can however omit the last choice from its set of legitimate identifiers, thereby keeping the bHEXA identifiers within four symbols and requiring only 2-bits. For completeness, we also keep c-bit discriminators (c may be zero, if we do not need them). Notice that instead of storing the complete bHEXA identifier, only $c+\lceil \log_2 k \rceil$ bits worth of information is required to be stored; this information along with the history of input symbols are sufficient to re-generate the complete bHEXA identifier of any given node.

Continuing with our example, we construct a memory mapping graph (as described in Section II.B), which is shown in Figure 4. In the graph we use m=10, thus an extra memory cell is available for the nine nodes. We also limit the bHEXA identifiers contain up to three history symbols and do not use discriminators. The edges of the graph are determined by the hash function h, which is:

$$h = \left(\sum_{i=1}^{k} s_i \times i\right) \bmod 10 ; \quad \text{for the bHEXA identifier } s_1 \ldots s_k$$

In this formula, the input symbols are assumed to take these numerical values: –=0, a=1, b=2, c=3.

In the same figure, a maximum matching in the memory mapping graph is highlighted, which assigns a unique memory location to each node of the automaton. According to this matching, the bHEXA identifiers of the nodes are chosen as:

Nodes	1	2	3	4	5	6	7	8	9
bHEXA	–	a	ab	bb	bba	bc	c	ca	b
length	0	1	2	2	3	2	1	2	1

Notice again that we only store the length of bHEXA identifiers in the memory (and discriminators, if they are used). During the graph traversal, the length and the history of input symbols are sufficient to reconstruct the complete bHEXA identifier. Since the length can be encoded with 2-bits in this case and there are no discriminators, the fast path will require total 7 bits per node: a match flag and 2-bits each to indicate the length of the bHEXA identifiers of the three "next nodes" for the symbols a, b and c, respectively. The resulting programming of the fast path memory is shown below:

Mem. location	node	match flag	a	b	c
0	1	0	01	00	01
1	2	0	01	10	01
2	9	1	01	10	01
3	7	0	10	00	01
4	8	0	01	01	01
5	3	0	01	10	10
6	4	0	11	00	01
7					
8	6	1	10	00	01
9	5	1	01	10	01

Compared to a standard implementation (13-bits per node), bHEXA uses about half memory (7-bits per node). There may however be circumstances when a perfect matching does not exist in the memory mapping graph. There are two possible solutions to resolve this problem. The first solution is *upward expansion*, in which additional memory cells are allocated; each new cell improves the likelihood of a larger matching. The second solution is *sideways expansion*, in which an extra bit is added, either to the discriminator of the bHEXA identifier or to its length, whichever leads to larger matching. Notice that each such extra bit doubles the number of edges in the memory mapping graph, which is likely to produce significantly larger matching. Unfortunately, sideways expansion also increases the memory rapidly. For example, if the current bHEXA identifiers require 3-bits, then a single bit of sideways expansion will increase the total memory by 33%.

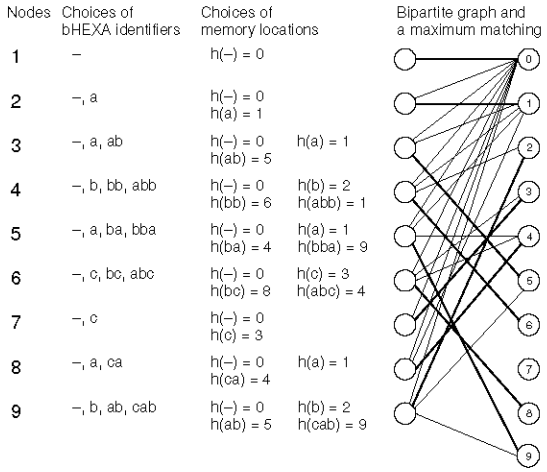


Figure 4: Memory mapping graph, bipartite matching.

A memory efficient way of finding one-to-one mapping should iterate between two phases. In the first phase, upward expansion will be applied until the added memory exceeds the memory needed by a single bit of sideways expansion. If one-to-one mapping is not yet found then the second phase will begin, which will reset the previous upward expansion and perform a bit of sideways expansion. If a one-to-one mapping is still not found, the first phase is repeated (without resetting the sideways expansion). This method is expected to find a one-to-one mapping while also minimizing the memory. In real bHEXA implementations, however, some new challenges may also arise, which we discuss in the coming section.

C. *Practical Considerations*

The challenges that may appear during the implementation of bHEXA are likely to depend primarily on the characteristics of the directed graph. The first challenge may arise when the directed graph contains long paths, all of whose edges have identical labels. Consider the Aho-Corasick automaton for l characters long string such as aaaaa… There will be l+1 nodes in the automaton and the legitimate bHEXA identifier for the i^{th} node will be any such string (aaa…) of length less than i. In this case, if we attempt to find a one-to-one mapping without using any discriminator then the bHEXA identifier of any i^{th} node will be i–1 characters long. Since there are l+1

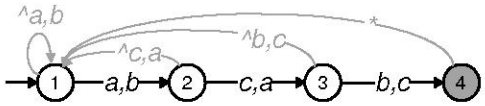
nodes, the longest bHEXA identifier will contain l symbols and $\lceil \log_2 l \rceil$ bits will be required to store its length. If we employ c discriminator bits then the longest bHEXA identifiers can be reduced by a factor of 2^c, nevertheless the total number of bits that will be stored per bHEXA identifier will remain the same. Clearly, large l will undermine the memory savings achieved by using bHEXA. While such strings are not common, we would still like to decouple the performance of bHEXA from the characteristics of the strings sets.

One way to tackle the problem is to allow the length bits to indicate superlinear increments in bHEXA identifier length. For instance, if there are three length bits available then they may be enabled to represent the bHEXA lengths of 0, 1, 2, 3, 5, 7, 12, and 16, thereby covering a much larger range of bHEXA lengths. Of course, the exact values that the length bits will represent will depend upon the strings database. Second way to tackle the problem is to employ a small on-chip CAM to store those nodes of the automaton that could not be mapped to a unique memory location due to the limited number of length and discriminator bits. In our previous example, if l is 9, and the bHEXA lengths are represented with 3-bits, then at least 2 nodes of the automaton can not be mapped to any unique memory location. These nodes can be stored in the CAM and can be quickly looked at during the parsing. We refer to the fraction of total nodes that can not be mapped to unique memory location as the *spill fraction*. In our experiments, we find that for real world string sets, the spill fractions remains low, hence a small CAM will suffice.

D. Challenges with General Finite Automaton

Modern network security appliances use regular expressions matching and employ finite automata to represent them [xxx]. Since complex regular expressions generally lead to large and complex automaton, it is important to reduce their memory footprint to enable an on-chip implementation and high parsing speed. Therefore, we investigate if it is possible to use some variant of bHEXA be to represent a general finite automata and save memory. Unfortunately, our early analysis suggests that for the finite automaton representation of the regular expressions used in current systems, it is difficult to save memory by using bHEXA. The primary reason is the extensive use of character classes in these regular expressions. We consider the following simple example to illustrate this.

Consider the simple regular expressions `[ab][ca][bc]`; such expressions are commonly used. The resulting automaton is shown below.



In this automaton, none of the nodes have all of its incoming paths labeled with unique sequence of symbols. Thus, it is difficult to use bHEXA identifiers to identify them. One may add new symbols in the alphabet, which will represent those character classes that are present in the regular expressions, thereby enabling paths with unique sequences of symbol. This however is likely to significantly expand the alphabet size, which will significantly increase the number of outgoing transitions from every node[1]. For instance, we find that, the regular expressions sets used in modern security appliance from Cisco Systems [xxx] have several thousand different character classes. Other sets [xxx] of regular expressions exhibit similar characteristics. This is likely to offset any memory reduction achieved with the bHEXA identifiers.

An orthogonal complication concerns with the performance. With the expanded alphabet, one may require additional memory lookups to map any given input symbol into the alphabet symbol representing the appropriate character class. Such additional lookups for every input symbol will adversely affect the parsing performance, and additional memory bandwidth will be required to maintain a given level of parsing rate. Memory bandwidth being much pricier than the memory size [xxx], such trade-offs may not be desirable (assuming that we were able to save some memory with bHEXA).

To conclude, it appears plausible to employ bHEXA for the finite automata used to represent regular expressions used in modern networking equipments, we conclude that it not clear, if this will lead to significant memory saving. The added complexity in parsing and symbol resolution to the character classes will offset the memory saving, if there is any at all. Nevertheless, we leave further investigation of the issue for the future research.

IV. Experimental Evaluation

We have performed a thorough experimental evaluation of the HEXA and bHEXA representations. First, we consider HEXA based representation of real world IP lookup tries. The results demonstrate that, HEXA can dramatically reduce the memory required by a binary trie; at the same time it can also reduce the memory in more sophisticated trie implementations like multi-bit trie and tree bit-map. Second, we employ HEXA to implement the finite automata, which are used to perform string matching operations. We consider two flavors of high performance string matcher, the classic Aho-Corasick automaton, and the recently proposed bit-split version. We show that, in both cases, HEXA reduces memory by up to five times without sacrificing the parsing performance.

A. Results on Tries

BGP tables have grown steadily over the past two decades from less than 5000 entries in the early 1990s to nearly 75,000 entries in 2000 to 135,000 entries today, and the growth is expected to continue in the near future. Binary tries are a standard method to implement these BGP tables and enable fast lookup. High performance implementations of these lookup tries consider multiple input bits at a time, thereby creating multi-bit nodes. The multi-bit nodes can be represented compactly by using tree bit-map tactics. In our

[1] Notice that in a DFA, at any given node, there is an outgoing transition for every symbol in the alphabet.

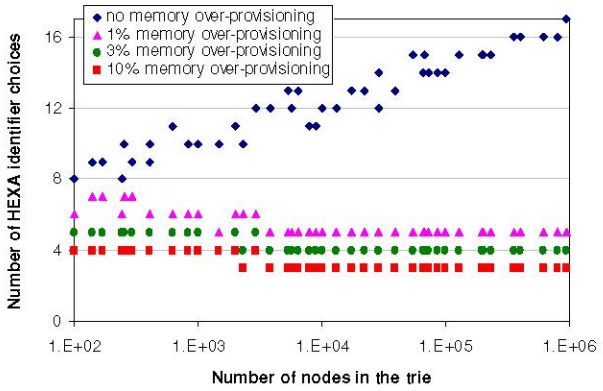


Figure 4: For different memory over-provisioning values and trie sizes, the number of choices of HEXA identifier that is needed to successfully perform the memory mapping

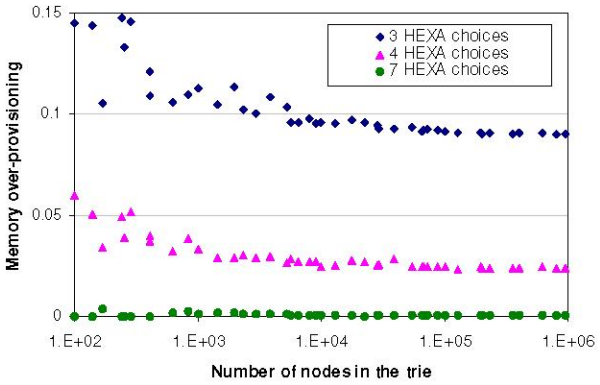


Figure 5: For different number of choices of HEXA identifiers and trie sizes, the memory over-provisioning that is needed to successfully perform the memory mapping

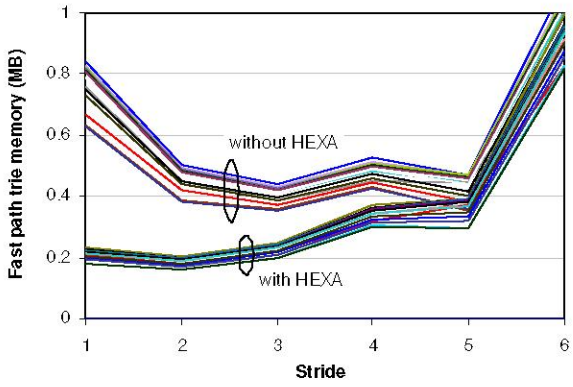


Figure 6: Memory needed to represent the fast path portion of the trie with and without HEXA. 32 tries are used, each containing between 100-120k prefixes.

experiments, we have employed HEXA to implement both binary trie as well as multi-bit trie. Unless otherwise specified, the reported results are based on the prefixes in more than fifty BGP tables obtained from [19].

1) Binary Tries

In Figure 4, for varying trie sizes, we plot the number of choices of HEXA identifiers that are needed to ensure that a perfect matching exists in the memory mapping graph with more than 90% probability. As expected, more choices of HEXA identifiers or increased memory over-provisioning $((m-n)/m)$ improves the chances of a perfect matching. In compliance with the theoretical analysis, for $m=n$, the required number of HEXA identifier choices remains $O(\log\ n)$. However, when m is slightly greater than n (results for 1, 3 and 10% are reported here), the required number of choices becomes constant, independent of the trie size. Recall that the number of HEXA identifier choices determines the number of discriminator bits that are needed for a node, thus a small memory over-provisioning is desirable in order to keep the discriminators constant in size.

From a practical point, we would like to keep the number of choices of HEXA identifiers a power of two minus one, so that one discriminator value will be used to indicate a null child node and all remaining permutations of discriminator values will be used in finding better matching. Thus, we are interested in such number of HEXA choices as 1, 3, 7, etc. Therefore, we fix the number of HEXA choices at these values, and plot the memory over-provisioning needed to successfully perform a one-to-one memory mapping (Figure 5). It is clear that that for 3 HEXA identifier choices, the required memory over-provisioning is 10%. Thus, 2.2 bits are enough to represent each node identifier.

2) Multi-bit tries

We now extend our evaluation of HEXA to multi-bit tries where tree bit-maps are used to represent the multi-bit nodes. Notice that when HEXA is used for such tries, the bit-masks used for the tree bitmap nodes are not affected; only the pointers to the child nodes are replaced with the child's discriminator. The first design issue in such tries is to determine a stride which will minimize the total memory. We accomplish this experimentally by applying different strides to our datasets and measuring the total fast path memory. The results are reported in Figure 6. Clearly, strides of 3, 4 and 5 are the most appropriate choices, when HEXA is not used. When HEXA is employed, large strides no longer remain effective in reducing the memory. This happens because a uni-bit HEXA trie requires just 2-bits of discriminator to represent a node, thus there is little room for further memory reductions by representing a subset of nodes with a bitmap. In fact, with increasing stride, the bitmaps grow exponentially and quickly surpass any memory savings achieved with the tree bitmap based multi-bit nodes.

Note that smaller strides may not be acceptable in off-chip memory based implementations. However, in an embedded implementation such as pipelined trie [26], small stride may enable higher throughput, as reported in [27]. This happens because with small stride, one can employ much deeper pipelines and each pipeline stage can be kept compact and fast.

3) Incremental Updates

We now present the results of incremental updates on tries

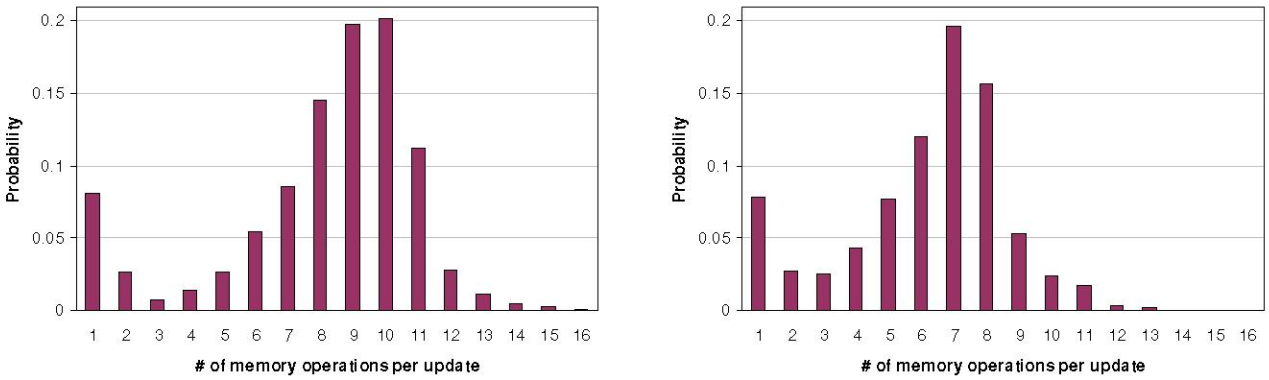


Figure 7: PDF of the number of memory operations required to perform a single trie update. Left trie size = 100,000 nodes, Right trie size = 10,000 nodes.

represented with HEXA. In our experiments, we remove a node and add another to a HEXA trie, and then attempt to find a mapping for the newly added node. The general objective of triggering minimum changes in the existing mapping is achieved by finding the shortest augmenting path in the memory mapping graph, between the newly added node and some free memory location (as described in Section II.C). We find that the shortest augmenting path indeed remains small, thus a small number of existing nodes are remapped. In Figure 7, we plot the probability distribution of the number of nodes that are remapped during an update. It is clear that no update is likely to take more than 19 memory operations and a large majority of updates require less than ten memory operations. Thus, update operations in a HEXA trie can be carried out quickly, irrespective of the trie shape and update patterns.

B. Results on Strings

In this section, we report the results obtained from the experiments in which we use bHEXA to implement string based pattern matchers. We have obtained the string sets from a collection of sources: peptide protein signatures [25], Bro signatures [20], and string components of the Cisco security signatures [21]. We have also used randomly generated signatures whose lengths were kept comparable to the real world security signatures. These strings were implemented with Aho-Corasick automaton; in most experiments we did not use failure pointers as they reduce the throughput. Without failure pointers, an automaton has 256 outgoing transitions per node, and may require large amounts of memory. In order to cope up with such high fan-out issue, we have considered the recently proposed bit-split version of Aho-Corasick, wherein multiple state machines are used, each handling a subset of the 8-bits in each input symbol. For example, one can use eight binary state machines, with each machine looking at a single bit of the 8-bit input symbols, thereby reducing the total number of per node transitions to 16.

First, we report the results on randomly generated sets of strings consisting of a total 64887 ASCII characters. In Figure 8(a), we plot the spill fraction (number of automaton nodes that could not be mapped to a memory location) as we vary the memory over-provisioning. It is clear from the plot that it is difficult to achieve zero spill without using discriminators. With a single bit of discriminator and less than 10% memory over-provisioning, spill fraction becomes zero, even when the bHEXA lengths are limited to 4. Thus, total 3-bits are needed in this case, to identify any given node: one for its discriminator and two to indicate the length of its bHEXA identifier. This represents more than five fold reduction in the memory when compared to a standard implementation, which will require 16-bits to represent a node.

Next we report similar results for real world string sets. In Figure 8(b), we plot the spill fraction for the set of protein strings, and the strings extracted from the Bro signatures, and Cisco security signatures. We only report results of those bHEXA configurations (number of discriminator bits and maximum bHEXA length) that keep the spill fraction at an acceptably low value. For the Bro strings, about 10% memory over-provisioning is needed in order to keep the spill fraction below 0.2%. The spill level corresponds to 11 nodes which remain unmapped in the automaton consisting of total 5853 nodes. The bHEXA configuration in this case does not use any discriminator and limits the length to 8, thus total of 3-bits are needed to identify any given node. For the protein patterns, again a 10% memory over-provisioning is needed in a configuration that uses 1-bit discriminator and up to 8 characters long bHEXA identifiers. Thus, in this case, 4-bits are needed to represent a node.

In the Cisco string set containing total 622 strings, there was one string that consisted of \x04 ASCII symbol repeated 50 times, which creates up to 50 states with identical bHEXA identifiers. This is precisely the issue that we have described in Section III.C. With restricted bHEHA length and limited discriminator bits, it is impossible to uniquely identify each of the resulting 51 nodes. Consequently, in a configuration where we employ 4-bits per bHEXA identifier, 35 nodes remain unmapped even if we arbitrarily increase the memory over-provisioning (refer to third set of vertical columns in Figure 8(b)). As we remove this string from the database, we were able to reduce the spill fraction to 0.1% with no memory over-provisioning and for an identical bHEXA configuration (last set of vertical columns in Figure 8(b)).

These results suggest that bHEXA based representations

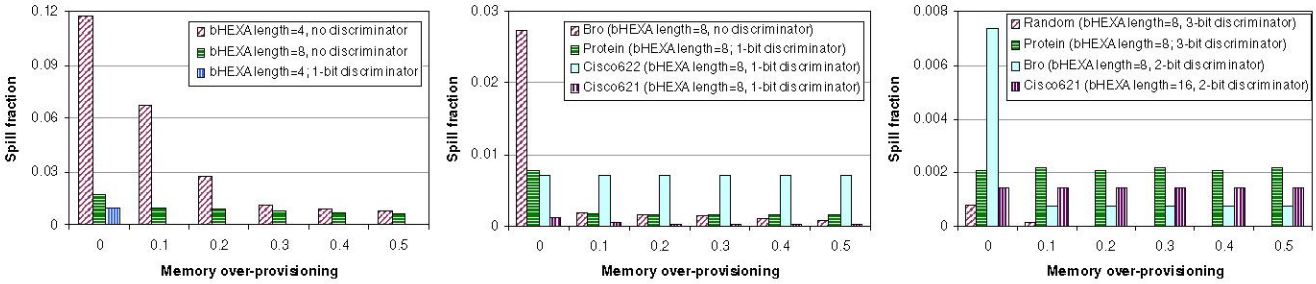


Figure 8: Plotting spill fraction: a) Aho-Coroasick automaton for random strings sets, b) Aho-Coroasick automaton for real world string sets, and c) random and real world strings with bit-split version of Aho-Corasick.

reduces the memory by between 3 to 5 times, when compared to standard representations. In our final set of experiments, we attempted to represent bit-split Aho-Corasick automaton with bHEXA. We have employed four state-machines, each handling two bits of the 8-bit input character. To our surprise, we found that bit-split versions were more difficult to map to the memory, and requires longer discriminators and bHEXA identifiers, which increases the number of bits per node. In spite of employing the techniques we have discussed in section III.C (*e.g.* using superlinear increase in the bHEXA length), we generally require 5 bits to represent each node of a bit-split automaton. This represents approximately 2-3 fold reduction in memory as compared to a standard implementation. The results are plotted in Figure 8(c).

To summarize, bHEXA based representations achieve between 2-5 fold reductions in the memory. Such reductions will not only aid in reducing the on-chip memory but also yield higher throughput at lower power dissipation levels.

V. Related Work

Please refer to our technical report. Due to the space limitations, we are unable to include related work here.

VI. Concluding Remarks

In this paper, we develop HEXA, a novel representation for structured graphs such as tries. HEXA uses a unique method to locate the nodes of the graph in memory, which enables it to avoid using any "next node" pointer. Since these pointers often consume most of the memory required by the graph, HEXA based representations are significantly more compact than the standard representations. We validate HEXA over two well known applications, IP lookup and string matching and find that HEXA indeed reduces the memory by up to five times. Such reduction levels facilitate the use of embedded memory, which can dramatically improve the packet throughput and reduce the power dissipation.

References

[1] R. Pagh, F. F. Rodler, Cuckoo Hashing, Proc. 9th Annual European Symposium on Algorithms, August 28-31, 2001, pp.121-133.

[2] A. V. Aho and M. J. Corasick, "Efficient string matching: An aid to bibliographic search," Comm. of the ACM, 18(6):333–340, 1975.

[3] B. Commentz-Walter, "A string matching algorithm fast on the average," Proc. of ICALP, pages 118–132, July 1979.

[4] S. Wu, U. Manber," A fast algorithm for multi-pattern searching," Tech. R. TR-94-17, Dept. of Comp. Science, Univ of Arizona, 1994.

[5] N. Tuck, T. Sherwood, B. Calder, and G. Varghese, "Deterministic memory-efficient string matching algorithms for intrusion detection," IEEE Infocom 2004, pp. 333--340.

[6] L. Tan, and T. Sherwood, "A High Throughput String Matching Architecture for Intrusion Detection and Prevention," ISCA 2005.

[7] I. Sourdis and D. Pnevmatikatos, "Pre-decoded CAMs for Efficient and High-Speed NIDS Pattern Matching," Proc. IEEE Symp. on Field-Prog. Custom Computing Machines, Apr. 2004, pp. 258–267.

[8] S. Yusuf and W. Luk, "Bitwise Optimised CAM for Network Intrusion Detection Systems," IEEE FPL 2005.

[9] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood, "Deep Packet Inspection using Parallel Bloom Filters," IEEE Hot Interconnects 12, August 2003. IEEE Computer Society Press.

[10] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, "Scalable High Speed IP Routing Lookups," in *Proc. ACM SIGCOMM'97*, pp. 25-37.

[11] V. Srinivasan, and G. Varghese., "Fast Address Lookups using Controlled Prefix Expansion", in *ACM Transactions on Computer Systems*, vol. 17, no. 1, 1999, pp. 1-40.

[12] D. E. Taylor, J. S. Turner, J. W. Lockwood, T. S. Sproull, and D. B. Parlour, "Scalable IP Lookup for Internet Routers," in *IEEE Journal on Selected Areas in Communications*, 2003.

[13] W. Eatherton, Z. Dittia, and G. Varghese, "Tree bitmap: Hardware/software ip lookups with incremental updates", in *ACM SIGCOMM Computer Communications Review*, 34(2), 2004.

[14] A. Basu and G. Narlikar, "Fast Incremental Updates for Pipelined Forwarding Engines", in *Proceedings of INFOCOM 2003*, 2003

[15] J. Hasan and T.N. Vijaykumar, "Dynamic Pipelining: Making IP-Lookup Truly Scalable", in *Proc. ACM SIGCOMM 2005*, pp 205-216.

[16] C. Labovitz, A. Ahuja, and F. Jahanian, "Experimental Study of Internet Stability and Wide-Area Backbone Failures", Proc. 29th Annual International Symp. on Fault-Tolerant Computing, Madison, WI, June 1999.

[17] Routing Information Service. `http://www.ris.ripe.net`

[18] CACTI.www.research.compact.com/wrl/people/jouppi/CACTI.html

[19] BGP Table Data. `http://bgp.potaroo.net`, April 2006

[20] Bro: A System for Detecting Network Intruders in Real-Time. http://www.icir.org/vern/bro-info.html

[21] Will Eatherton, John Williams, "An encoded version of reg-ex database from cisco systems provided for research purposes".

[22] M. Roesch, "Snort: Lightweight intrusion detection for networks," In Proc. 13th Systems Administration Conference (LISA), USENIX Association, November 1999, pp 229–238.

[23] Adam Kirsch, M. Mitzenmacher, "Simple Summaries for Hashing with Multiple Choices," In Proceedings of the Forty-Third Annual Allerton Conference on Communication, Control, and Computing, 2005.

[24] M. Degermark, A. Brodnik, S. Carlsson and S. Pink, "Small Forwarding Tables for Fast Routing Lookups", in *Proc. of ACM SIGCOMM 1997*.

[25] Comprehensive Peptide Signature Database, Institute of Genomics and Integrative Biology, http://203.90.127.70/copsv2/

[26] A. Basu and G. Narlikar, "Fast Incremental Updates for Pipelined Forwarding Engines", in *Proceedings of INFOCOM 2003*, 2003

[27] Florin Baboescu, Dean M. Tullsen, Grigore Rosu, Sumeet Singh, "A Tree Based Router Search Engine Architecture with Single Port Memories," in ISCA 2005.