

# Doctoral Dissertation Proposal: Acceleration of Network Processing Algorithms

Sailesh Kumar  
Washington University  
Computer Science and Engineering  
St. Louis, MO 63130-4899  
+1-314-935-4306  
sailesh@arl.wustl.edu

Research advisors: Jonathan Turner and Patrick Crowley

## ABSTRACT

Modern networks process and forward an increasingly large volume of traffic; the rate of growth of the traffic often outpaces the improvements in the processor, memory and software technology. In order for networking equipment to maintain an acceptable performance, there is a need for architectural enhancements and novel algorithms that can efficiently implement the various network features. While there is a variety of network features of significance, in this research proposal, we consider three core features namely: *i*) packet buffering and queuing, *ii*) packet header processing, and *iii*) packet payload inspection.

We propose to thoroughly investigate the existing methods to realize these features and evaluate their usability on modern implementation platforms like network processors. Afterwards, we plan to introduce novel algorithms to implement these features which will not only improve the performance theoretically, but also better utilize the capabilities available with modern hardware. We also intend to evaluate some of the proposed algorithms using network processor platforms, which will provide us a first order estimate of the usability of our methods. The research is likely to contribute to the efficient design and implementation of the aforementioned network features. The proposed research is likely to take one year.

## 1. INTRODUCTION

High performance network devices perform an array of operations upon receiving a packet, all of which have to be finished within a limited time budget in order to maintain a high packet throughput and low processing latency. While the performance constraints of these systems are normally stringent, there are two trends which put additional pressure on the performance: *i*) new network features are regularly introduced, many of which are exercised on a per packet basis, and *ii*) the increase in the packet arrival rates often outpaces the rate at which hardware and memory technology advances. Due to these performance pressures, an efficient implementation of the network features

becomes critical. While it is crucial to efficiently implement the newly introduced features, the exiting features also need to be updated with the advances in technology. A sound realization of any network feature requires a thorough understanding of the classical algorithms and data-structures as well as the hardware and system technologies, thereby making it an interesting research problem. Besides, due to the importance of these methods, they have received an enormous attention in the networking research community.

Two core network features which have remained the focus of researchers are: *i*) *packet buffering and scheduling*, which generally involves a fast packet storage component coupled to a queuing and scheduling module, and *ii*) *packet header processing*, which includes header lookup and packet classification – the former determining the next hop for the packet and the latter prioritizing the packets based upon their source and destination addresses and protocol. A third class of network feature which has recently experienced a wide adoption is *deep packet inspection*, in which the packet payload is matched against a set of pre-defined patterns. Deep packet inspection is often used in the emerging application layer packet forwarding applications and intrusion detection systems.

Due to the importance and broad deployment of these three network features, a collection of novel methods have been introduced to implement them efficiently. These methods often consider the constraints and capabilities of the current hardware platforms and employ a complex mix of ideas drawn from theoretical computer science & systems and hardware technology. Since the systems and hardware technology evolves rapidly, there is a constant need to upgrade these implementations; nevertheless, there is also room to enhance them in an abstract and theoretical sense. In this proposal, we plan to undertake these tasks which can directly contribute to the efficient design of the network features. Our aim is to split the efforts evenly between the

combination of the packet buffering and header processing features and the deep packet inspection feature.

The first two network features we are focusing on were comprehensively studied in the past, and there may be little room for any fundamental improvements. However, the evolution of new implementation platforms like network processors has opened up the possibility of novel ideas and implementation methods. Network processors are software-programmable devices and their feature sets are specifically targeted for networking applications. They contain a collection of memory banks of varying capacity, running at different operating frequencies thereby creating memories of varying size, access latency and bandwidth. A general trend is that large memories have relatively low bandwidth and high access latency.

The presence of such a diverse collection of memories presents new levels of opportunities and challenges in utilizing them. For example, if the data-structures used to implement the memory intensive features like packet buffering and header lookup, are spread out across various memories and the fast but small memories are prudently used, then the performance can be dramatically enhanced. One of the main objectives of the research is to develop innovative methods to distribute the data-structures across the different memories such that both the bandwidth and space can be effectively utilized. The distribution mechanism can be either static or dynamic - the former will pre-allocate memories to different data-structure segments while in the latter, portions of the data-structure will be allowed to migrate from one memory to the other. Traditional caches, which improve the average-case performance, are one instance of such a dynamic distribution mechanism.

Current network processor devices also contain specialized engines, like hash accelerators and Content Accessible Memory (CAM), that offer further possibilities. Popular hash based techniques like hash tables and Bloom filters can now be cost-effectively employed. The presence of hashing also eases the use of randomized methods which can provide strong probabilistic performance guarantees at a reduced cost. CAM, on the other hand, can be used to employ an associative caching scheme, which can help in improving the average-case performance of queuing and buffering features. In the proposed research, we aim to explore such possibilities of utilizing the specialized hardware resources in enhancing the performance.

The third network feature, deep packet inspection, which is our second primary focus, has recently gained a widespread adoption. The key reason is that many emerging network services handle packets based on payload content, in addition to the structured information found in packet headers. Forwarding packets based on content requires new levels of support in networking equipment, wherein every

byte of the packet payload is inspected in addition to examining the packet headers. Traditionally, this deep packet inspection has been limited to comparing packet contents to sets of strings. However, new systems are replacing string sets with regular expressions for increased flexibility and expressiveness. Several content inspection engines have recently migrated to regular expressions, including: Snort [43], Bro [42], 3Com's TippingPoint X505 [58], and various network security appliances from Cisco Systems [59]. Additionally, layer 7 filters based on regular expressions [66] are available for the Linux operating system.

While flexible and expressive, regular expressions traditionally require substantial amounts of memory. Further, the state-of-the art DFA based algorithms to perform regular expression matching are unable to keep up with the ever increasing link speeds. The state space blowup problem in DFA appears to be a serious issue, and limits their practical applicability. The proposed research will aim primarily at developing the innovative algorithmic solutions to implement regular expressions that can enable high parsing rates at low implementation costs.

We propose to begin the research by systematically studying the trade-offs involved in finite automata based regular expressions implementation and the hardware capabilities that are needed to execute these automata. A preliminary analysis suggests that current hardware is not only capable of concurrently executing several automata but can also implement machines which are more complex than finite automata. Consequently it is possible to envision new implementation approaches, which either employs multiple automata or uses a new machine altogether which trades-off the space and performance more effectively. Such approach may also utilize probabilistic methods and exploit the fact that the likelihood of completely matching any pattern is generally remote in many networking applications. One of our key objectives is to develop these machines.

Another significant goal of the research is to develop algorithms to efficiently store a given machine (*e.g.* a finite automaton, a push-down automaton or any new machine) in memory. The primary objective of such storage algorithms is to reduce the memory size and bandwidth, needed to store and execute the machine, respectively. Traditional table compression techniques are known to be inefficient in storing finite automata which generally appear in the networking applications. To an extent, memory compression algorithm developed in the recently proposed CD<sup>2</sup>FA appears promising; however its applicability to more complex machines is not yet known. Therefore, we intend to extend these algorithms so that they can be applied to more general machines.

An orthogonal research direction is to investigate the possibilities to further reduce the memory by eliminating

the overheads involved in explicitly storing the transitions. Conventional dictates that each transition of an automaton requires  $\lceil \log_2 n \rceil$  bits, where  $n$  is the total number of states in the automaton. It appears that, with the aid of hashing, each state can be represented with fewer bits thereby reducing the total memory needed to store the transitions. Our preliminary analysis suggests that conventional methods require 20-bits to represent each transition in a one million state machine, while the stated technique will require only 4-bits.

To summarize, in this proposal we plan to work on three important network features. For each feature, we plan to evaluate the existing implementation methods and the challenges that arise with the introduction of new platforms and hardware capabilities. Additionally, for each specific feature, we plan to undertake the following tasks:

1. Packet buffering and queuing (25% of the total effort)
  - 1.1. Using randomization techniques to improve the buffering performance
  - 1.2. Building an efficient buffering subsystem using a collection of memories of different size, bandwidth and access latency
  - 1.3. Hashing and Bloom filter based buffering and caching subsystems
2. Header lookup (25% of the total effort)
  - 2.1. Architecture of header lookup engines capable of supporting tera-bit data rates
  - 2.2. Algorithms to compress lookup data-structure and implication of caching on lookup performance
3. Deep packet inspection (50% of the total effort)
  - 3.1. Evaluation of the patterns used in current deep packet inspection systems
  - 3.2. Trade-off between NFA and DFA methods; analysis of the worst and average performance
  - 3.3. Evaluation of intermediate approaches like lazy DFA
  - 3.4. Introduction to novel machines, different from finite automaton, which can implement regular expressions much more cost-effectively
  - 3.5. Memory compression schemes (*e.g.* Delayed input DFAs (D<sup>2</sup>FA) and Content addressed delayed input DFAs (CD<sup>2</sup>FA))

The remainder of the proposal is organized as follows. Background on the three network features and related work are presented in Section 2. Section 3 describes the agenda for the packet header processing, while section 4 covers the plan for the buffering and queuing research. Details of our proposed research for the deep packet inspection are highlighted in Section 4. The proposal ends with concluding remarks in Section 5.

## 2. BACKGROUND AND RELATED WORK

We split this section into three subsections. Each subsection will cover some background and relevant related work for the “packet buffering and queuing”, “packet header lookup” and “deep packet inspection” features, respectively.

### 2.1 Packet buffering and queuing

Buffers in modern routers require substantial amounts of memory to store packets awaiting transmission. Router vendors typically dimension packet storage subsystems to have a capacity at least equal to the product of the link bandwidth and the typical network round-trip delay. While a recent paper [27] has questioned the necessity of such large amounts of storage, current practice continues to rely on the bandwidth-delay product rule. The amount of storage used by routers is large enough to necessitate the use of high density memory components. Since high density memories like DRAM have limited random access bandwidth and short packets are common in networks, it becomes challenging for them to keep up with the increasing link bandwidths. In order to tackle these problems, a number of architectures have been proposed.

In reference [29], the authors propose a ping-pong buffer, which can double the random access bandwidth of a memory based packet buffer. Such a buffer has been shown to exhibit good utilization properties; the memory space utilization remains as high as 95% in practice. In references [30][31], Iyer *et al.* have shown that a hybrid approach combining multiple off-chip memory channels with an on-chip SRAM can deliver high performance even in the presence of worst-case access patterns. The on-chip SRAM is used to provide a moderate amount of fast, per-queue storage, while the off-chip memory channels provide bulk storage. Unfortunately, the amount of on-chip SRAM needed grows as the product of the number of memory modules and the number of queues, making it practical only when the number of individual queues is limited.

More recently, multichannel packet storage systems [7][8] have been introduced that use randomization to enable high performance in the presence of arbitrary packet retrieval patterns. Such an architecture requires an on-chip SRAM, whose size is proportional only to the number of memory modules and doesn’t grow as the product of the number of memory modules and the number of queues, making it practical for any system irrespective of the number of queues. It has been shown that, even for systems which use DRAM memories with a large number of banks, the overall on-chip buffering requirements depend mostly on the number of channels and not the product of the number of channels and the number of banks, thereby making such an approach highly scalable.

While packet storage is important, modern routers often employ multiple queues to store the packet headers. These queues are used to realize advanced packet scheduling

policies, QoS, and other types of differentiated services applied to packet aggregates. The problem of scheduling real-time messages in packet switched networks has been studied extensively. Practical algorithms can be broadly classified as either timestamp or round-robin based. Timestamp based algorithms try to emulate GPS [37] by sending packets, approximately, in the same order as sent by a reference GPS server. This involves computation of timestamps for various queues, and sorting them in increasing order. Round-robin schedulers [38] avoid the sorting bottleneck by assigning time slots to the queues and selecting the queues in the current slot. Each selected queue can transmit multiple packets such that their cumulative size is less than or equal to a maximum sized packet.

Many routers use multiple hierarchies of queues in order to implement sophisticated scheduling policies, *e.g.* the first set of queues may represent physical ports, the second classes of traffic and the third may consist of virtual (output) queues. When there is a large number of queues then off-chip memory is required to store the queuing and scheduling data-structure which complicates the design of the buffering and queuing subsystem. In fact, the off-chip memory can be a significant contributor to the cost of the queuing subsystem and can place serious limits on its performance, particularly as link speeds scale beyond 10 Gb/s. A recent series of papers has demonstrated queuing architectures which alleviate these problems and maintain high throughput.

In [6], the authors show how queuing subsystems using a combination of implicit buffer pointers, multi-buffer list nodes and coarse grained scheduling can dramatically improve the worst-case performance, while also reducing the SRAM bandwidth and capacity needed to achieve high performance. In [4], the authors propose a cache based queue engine, which consists of a hardware cache and a closely coupled queuing engine to perform queue operations. Such cache based engines are also available on modern network processors like Intel the IXP [10]. While such NP based queuing assists have limited capability, a number of more sophisticated commercial packet queuing components are available, which focus on advanced QoS and traffic management applications. They often employ custom logic and memory to achieve high performance. However, a programmable queuing component, which uses commodity memory and processing component is more desirable.

## 2.2 Packet header lookup

An Internet router processes and forwards incoming packets based upon the structured information found in the packet header. The next hop for the packet is determined after examining the destination IP address; this operation is often called IP lookup. Several advanced services determine the treatment a packet receives at a router by examining the

combination of the source and destination IP addresses and ports; this operation is called packet classification. The distinction between IP Lookup and Packet classification is that IP Lookup classifies a packet based on a single field in the header while Packet Classification classifies a packet based on multiple header fields. The core of both functions consists of determining the longest prefix matching the header fields within a database of variable length prefixes.

Longest prefix match algorithms have been widely studied. Some well known mechanisms include TCAM [19][20], Bloom filters [16] and hash tables [11]. The hardware based approaches, such as TCAM, have become popular; however they consume a substantial amount of power. In order to cope with the power concerns, several algorithmic solutions have been introduced. Such solutions often employ a trie to perform the longest prefix lookup. A trie is built by traversing the bits in each prefix from left to right, and inserting appropriate nodes in the trie. The resulting trie can then be traversed to perform the lookup operations. A substantial number of papers have been written in this space, which attempts to efficiently implement these tries so that the total memory consumption can be reduced while maintaining high lookup and update rates [12][13][14][15].

With the current memory technology, trie based implementations of header lookup can easily support a data throughput of 10 Gbps. However, at 40 Gbps data rates, a minimum sized 40-byte packet may arrive every 8 ns, and it may become challenging to perform lookup operations with a single memory. A number of researchers have proposed a pipelined trie [17][18]. Such tries enable high throughput because when there are enough memory stages in the pipeline, no stage is accessed more than once for a search and during each cycle, each stage can service a memory request for a different lookup.

Recently, Baboescu et al. [25] have proposed a circular pipelined trie, which is different from the previous ones in that the memory stages are configured in a circular, multi-point access pipeline so that lookups can be initiated at any stage. At a high-level, this multi-access and circular structure enables more flexibility in mapping trie nodes to pipeline stages, which in turn maintains uniform memory occupancy. A refined version of circular pipeline called CAMP has been introduced in [2], which employs a relatively simple method to map the trie nodes to the pipeline stages, thereby improving the rate at which the trie can be updated. CAMP also presents relatively simple but effective methods to maintain high memory utilization and scalability in the number of pipeline stages.

Such circular pipeline based lookup can not only provide a high lookup rate but also improve the memory utilization and reduce the power consumption. It will be extremely valuable to evaluate the feasibility of incorporating such specialized engines in modern network processors.

## 2.3 Deep packet inspection

Deep packet inspection has recently gained widespread popularity as it provides the ability to accurately classify and control traffic in terms of content, applications, and individual subscribers. Cisco and others today see deep packet inspection happening in the network and they argue that “Deep packet inspection will happen in the ASICs, and that ASICs need to be modified” [57]. Some important applications requiring deep packet inspection are listed below:

- Network intrusion detection and prevention systems (NIDS/NIPS) generally scan the packet header and payload in order to identify a given set of signatures of well known security threats.
- Layer 7 switches and firewalls provide content-based filtering, load-balancing, authentication and monitoring. Application-aware web switches, for example, provide scalable and transparent load balancing in data centers.
- Content-based traffic management and routing can be used to differentiate traffic classes based on the type of data in packets.

Deep packet inspection often involves scanning every byte of the packet payload and identifying a set of matching predefined patterns. Traditionally, rules have been represented as exact match strings consisting of known patterns of interest. Naturally, due to their wide adoption and importance, several high speed and efficient string matching algorithms have been proposed recently. Some of the standard string matching algorithms such as Aho-Corasick [45] Commentz-Walter [46], and Wu-Manber [47], use a preprocessed data-structure to perform high-performance matching. A large body of research literature has concentrated on enhancing these algorithms for use in networking. In [49], Tuck et al. presents techniques to enhance the worst-case performance of the Aho-Corasick algorithm. Their algorithm was guided by the analogy between IP lookup and string matching and applies bitmap and path compression to Aho-Corasick. Their scheme has been shown to reduce the memory required for the string sets used in NIDS by up to a factor of 50 while improving performance by more than 30%.

Many researchers have proposed high-speed pattern matching hardware architectures. In [50] Tan et al. propose an efficient algorithm that converts an Aho-Corasick automaton into multiple binary state machines, thereby reducing the space requirements. In [51], the authors present an FPGA-based design which uses character pre-decoding coupled with CAM-based pattern matching. In [52], Yusuf et al. use hardware sharing at the bit level to exploit logic design optimizations, thereby reducing the area by a further 30%. Other work [62][63][64][65] presents several efficient string matching architectures; their

performance and space efficiency are well summarized in [52].

In [39], Sommer and Paxson note that regular expressions might prove to be fundamentally more efficient and flexible as compared to exact-match strings when specifying attack signatures. The flexibility is due to the high degree of expressiveness achieved by using character classes, union, optional elements, and closures, while the efficiency is due to the effective schemes to perform pattern matching. Open source NIDS systems, such as Snort and Bro, use regular expressions to specify rules. Regular expressions are also the language of choice in several commercial security products, such as TippingPoint X505 [58] from 3Com and a family of security appliances from Cisco Systems [59]. Although some specialized engines such as RegEx from Tarari [60] report packet scan rates up to 4 Gbps, the throughput of most such devices remains limited to sub-gigabit rates. There is great interest in and incentive for enabling multi-gigabit performance on regular expressions based rules.

Consequently, several researchers have recently proposed specialized hardware-based architectures which implement finite automata using fast on-chip logic. Sindhu et al. [53] and Clark et al. [54] have implemented nondeterministic finite automata (NFAs) on FPGA devices to perform regular expression matching and were able to achieve very good space efficiency. Implementing regular expressions in custom hardware was first explored by Floyd and Ullman [56], who showed that an NFA can be efficiently implemented using a programmable logic array. Moscola et al. [55] have used DFAs instead of NFAs and demonstrated significant improvement in throughput although their datasets were limited in terms of the number of expressions.

These approaches all exploit a high degree of parallelism by encoding automata in the parallel logic resources available in FPGA devices. Such a design choice is guided partly by the abundance of logic cells on FPGAs and partly by the desire to achieve high throughput as such levels of throughput might be difficult to achieve in systems that store automata in memory. While such a choice seems promising for FPGA devices, it might not be acceptable in systems where the expression sets needs to be updated frequently. More importantly for systems which are already in deployment, it might prove difficult to quickly re-synthesize and update the regular expressions circuitry. Therefore, regular expression engines which use memory rather than logic, are often more desirable as they provide higher degree of flexibility and programmability.

Commercial content inspection engines like Tarari’s RegEx already emphasize the ease of programmability provided by a dense multiprocessor architecture coupled to a memory. Content inspection engines from other vendors [67][68], also use memory-based architectures. In this context, Yu et

al. [48] have proposed an efficient algorithm to partition a large set of regular expressions into multiple groups, such that overall space needed by the automata is reduced dramatically. They also propose architectures to implement the grouped regular expressions on both general-purpose processor and multi-core processor systems, and demonstrate an improvement in throughput of up to 4 times.

Emphasizing the importance of memory based designs, a recently proposed representation of regular expressions called delayed input DFA (D<sup>2</sup>FA) [3] attempts to reduce the number of transitions while keeping the number of states the same. D<sup>2</sup>FAs use *default transitions* to reduce the number of labeled transitions in a DFA. A default transition is followed whenever the current input character does not match any labeled transition leaving the current state. If two states have a large number of “next states” in common, we can replace the common transitions leaving one of the states with a default transition to the other. No state can have more than one default transition, but if the default transitions are chosen appropriately, the amount of memory needed to represent the parsing automaton can be dramatically reduced.

Unfortunately, the use of default transitions also reduces the throughput, since no input is consumed when a default transition is followed, but memory must be accessed to retrieve the next state. In [1], authors develop an alternate representation for D<sup>2</sup>FAs called the Content addressed D<sup>2</sup>FA (CD<sup>2</sup>FA) that allows them to be both fast and compact. A CD<sup>2</sup>FA is built upon a D<sup>2</sup>FA, whose state numbers are replaced with content labels. The content labels compactly contain information which are sufficient for the CD<sup>2</sup>FA to avoid any default traversal, thus avoiding unnecessary memory accesses and hence achieving higher throughput. The authors argue that while a CD<sup>2</sup>FA requires a number of memory accesses equal to those required by a DFA, in systems with a small data cache, CD<sup>2</sup>FA surpasses a DFA in throughput, due to their small memory footprint and higher cache hit rate.

### 3. PACKET HEADER LOOKUP – NEW DIRECTIONS

As part of the proposed research, we intend to develop a collection of ideas that can enhance the performance of header lookup operations. Header lookup operations in IP networks generally involve determining the longest prefix matching the packet header fields within a database of variable length prefixes. We focus on two novel methods to enhance the longest prefix match operation. The first method called HEXA is directly applicable to trie based algorithms and it can reduce the memory required to store a trie by up to an order of magnitude. Such levels of memory reduction may lead to an improvement in the lookup rate, because the compressed trie is more suitable to support a larger stride and the memory being much smaller in size is

also likely to run at much higher clock speeds. Our first order analysis suggests that HEXA based tries also preserves the fast incremental update properties, which are often desirable.

Our second method attempts to improve the performance of header lookup in a more general sense. A series of recent papers [11][16] have advocated the use of hash tables to perform the header lookup. Since the performance of a hash table can deteriorate considerably in the worst-case, they have been coupled with Bloom filter based techniques. We expand these techniques by introducing a novel hash table implementation called Peacock hashing. Our preliminary analysis suggests that Peacock hash tables have several desirable properties, which can lead to a more efficient implementation of header lookup operations. We now elaborate these two research directions.

#### 3.1 HEXA

HEXA employs a new method to address the nodes of a trie which we call History based Encoding, eXecution, and Addressing. HEXA challenges the conventional assumption that we need  $\lceil \log_2 n \rceil$  bits to identify a node in a trie containing a total  $n$  nodes and shows that only  $\lceil \log \log_2 n \rceil$  bits are sufficient. This can dramatically reduce the memory required to represent a trie structure, which mostly consists of the transitions (address of the next nodes). The total memory needed to implement the lookup operation will also be reduced significantly, because auxiliary information other than the trie 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 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.

#### 3.2 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):

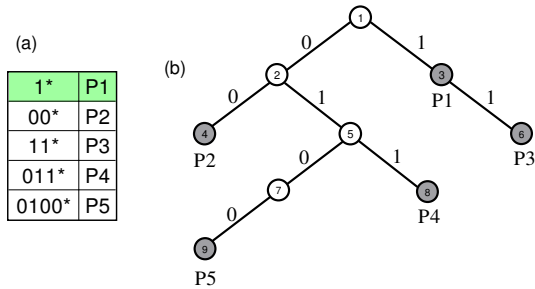


Figure 1: a) routing table, b) corresponding binary trie.

- |               |                  |                  |
|---------------|------------------|------------------|
| 1. 0, 2, 3    | 4. 1, NULL, NULL | 7. 0, 9, NULL    |
| 2. 0, 4, 5    | 5. 0, 6, 9       | 8. 1, NULL, NULL |
| 3. 1, NULL, 7 | 6. 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. - | 4. 00 | 7. 010  |
| 2. 0 | 5. 01 | 8. 011  |
| 3. 1 | 6. 11 | 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  $i^{\text{th}}$  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. $f(-) = 4$ | 4. $f(00) = 2$ | 7. $f(010) = 5$  |
| 2. $f(0) = 7$ | 5. $f(01) = 8$ | 8. $f(011) = 3$  |
| 3. $f(1) = 9$ | 6. $f(11) = 1$ | 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	0,1,1	0,1,1	1,0,1	1,0,0	0,1,1	1,0,0	0,1,0	1,0,0	1,0,0
Next hop			P1	P2		P3		P4	P5

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.

### 3.3 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 [71] and cuckoo hashing [70], 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  $2c-1$  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 [70], 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.

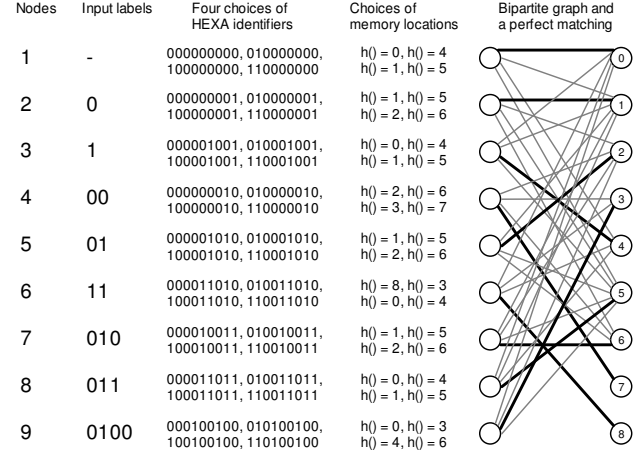


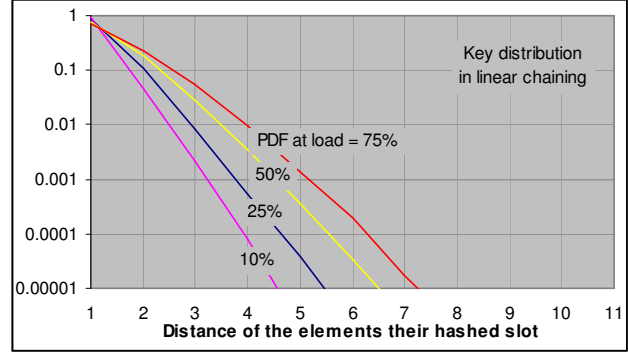
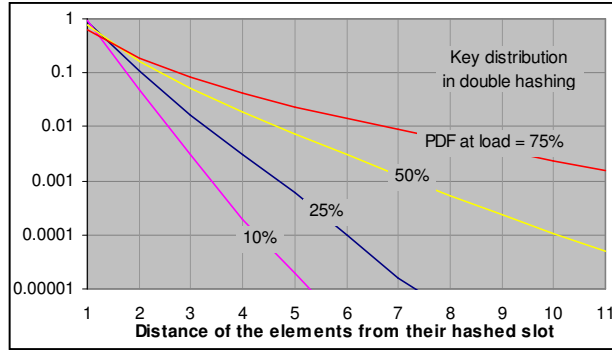
Figure 2: Memory mapping graph, bipartite matching.

### 3.4 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 [70]. 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 [70].

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.



**Figure 3. PDF (base 10 log scale) of distance of elements from their hashed slots in double hashing and linear chaining**

In the proposed research, we intend to implement HEXA and evaluate its effectiveness on real IP lookup databases. We also plan to perform a thorough evaluation of the incremental updates properties of HEXA. Finally, we plan to extend HEXA such that the memory requirements can be further reduced. The intuition behind optimization is that we can employ the discriminator bits of a parent node to identify its children, thus increasing the choices of HEXA identifiers and potentially reducing the discriminator bits.

### 3.5 Peacock Hashing

The second component of our research concerns the implementation of efficient and deterministic hash tables, which are often used to implement the packet header lookup operations. Traditional hash tables provide an  $O(1)$  lookup time, which enables good average performance. While the worst-case lookup time is  $O(n)$  for  $n$  items in the table, in theory, high throughput can be maintained by employing queues to hold the incoming lookup requests awaiting service. In practice however, certain complications arise due to the discrepancy between memory latency and bandwidth. With high access latency, in order to better utilize high memory bandwidth, multiple threads are employed; a thread can be viewed as an instance of the processing algorithm handling one packet. Each thread grabs an incoming packet, and processes it; here we assume that the processing involves a hash lookup. At the end of the processing threads are synchronized; this is essential to maintain the processed packets in the same order in which they have arrived. Such synchronization implies that the overall throughput will be determined by the slowest thread or the thread that finishes the last. Clearly, traditional hash table in such multi-threaded in-order processing environment may yield low throughput.

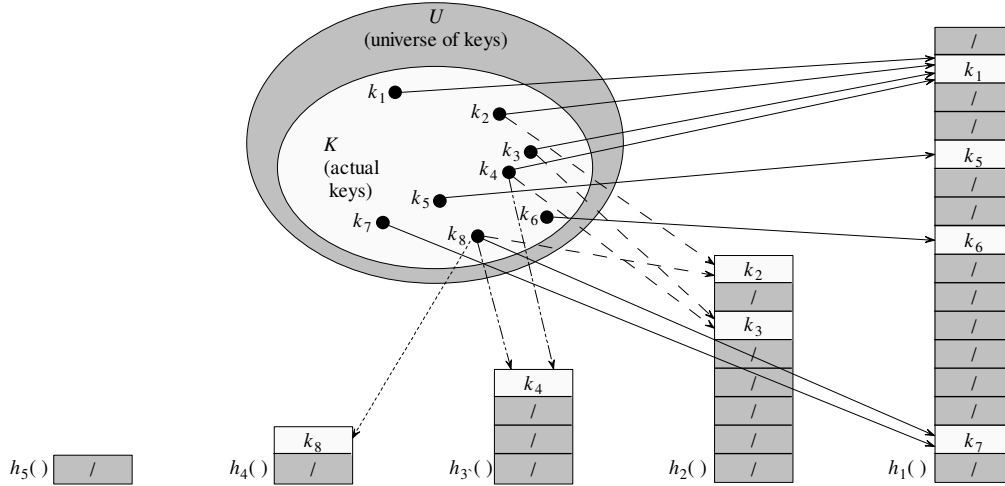
To understand more objectively, we examine the traditional hash table lookups. We plot the probability density function (PDF) of the distance of the inserted elements from their hashed slot in Figure 3. At 25% load, approximately 1% and 2% elements remain at a distance of three or higher, for chaining and double hashing respectively. At 75% load,

approximately 10% of the elements remain at a distance of three or higher; thus more than 10% elements will require three or more memory references. With seven or more total threads, at least one thread is expected to be processing such an element, resulting in a 3x reduction in throughput. With more threads, the performance will degrade further. Notice that the performance of traditional hash tables also remains vulnerable to adversarial or attack traffic patterns.

We propose to work on a novel hash table implementation called *Peacock Hashing*, which can enable much higher performance in the multi-threaded processing environment. Recall that a naïve approach to solve the problems of traditional hash table is to allocate a large memory to keep the load at sufficiently low levels and discard elements which are inserted far from their hashed slot. Unfortunately, the discard rates can't be reduced to an arbitrarily low constant unless an arbitrarily oversized hash table is allocated. A Peacock hash table tackles this problem by employing a hierarchy of hash tables with the table sizes following a geometric progression (GP). The objective is to keep the load in each segment uniform and limit the longest probe sequence to a small constant. Since the limit on probe sequence in every segment is equal, the fraction of elements which will overflow from a larger segment to a smaller one will remain equal; hence a GP dimensioning is sufficient.

#### 3.5.1 Architecture and the Basic Operation

The high level schematic of a peacock hash table is shown in Figure 4. In this specific example, the scaling factor,  $r$  (ratio of the size of two consecutive hash tables) is kept at 2 and size of the largest segment is kept at 16. As shown, 8 keys are inserted into the table and the longest probe sequence,  $c$  is limited to 1. The number of hash tables,  $s$  is 5 and a hash function is used for each table. Clearly, the last hash function is the simplest one, since it always returns zero. Grey cells represent empty slots while the light ones are occupied. Since collision threshold is kept at 1, every single collision results in an insert into a backup segment. In the particular example, keys  $k_1$  through  $k_4$  are hashed to the same slot in the main segment. Assuming that  $k_1$  arrived



**Figure 4. High level schematic and operation of a peacock hash table with 5 segments; scaling factor,  $r$  is kept at 2 and size of the largest segment,  $n_0$  is 16. 8 keys are inserted with the collision threshold,  $c$  set at 1.**

earliest, it is inserted and the remaining are considered for insertion in the first backup segment. Since a different hash function is used in this segment, the likelihood of these keys to again collide is low. In this case, however, keys  $k_3$  and  $k_4$  collides;  $k_3$  is inserted while  $k_4$  is considered for insertion in the second backup segment. In the second backup segment,  $k_4$  collides with a new key  $k_8$ .  $k_4$  this time is inserted while  $k_8$  finds its way into the third backup segment.

In general peacock hashing, trials are made to insert an arriving key first into larger segments and then into smaller ones. Keys are hashed into segments using distinct hash functions and then following the probe sequence (or the linked-list in chaining) until an empty slot is found or  $c$  probes are made. Key is discarded if it doesn't find an empty slot in any segment. The pseudo-code for the insertion procedure is shown below assuming an open addressing collision policy. The procedure for chaining will be similar.

---

```

PEACOCK-INSERT( $T, k$ )
 $i \leftarrow 1$ 
repeat
  if SEGMENT-INSERT( $T_i, k$ ) > -1 then
    return  $i$ 
  else  $i \leftarrow i + 1$ 
until  $i = s$ 
error "A discard occurred"

SEGMENT-INSERT( $T_i, k$ )
 $i \leftarrow 0$ 
repeat  $j \leftarrow h(k, i)$ 
  if  $T_i[j] = \text{NIL}$ 
    then  $T_i[j] \leftarrow k$ 
    return  $j$ 
  else  $i \leftarrow i + 1$ 

```

```

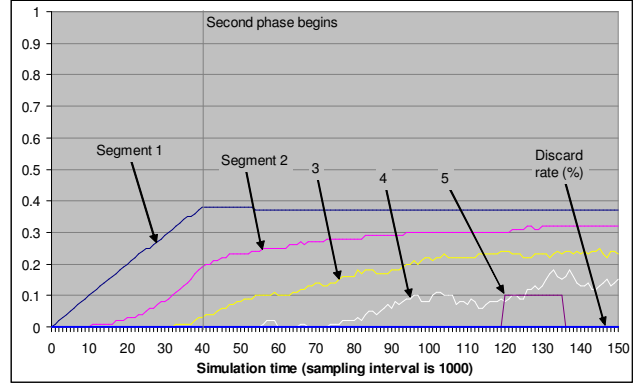
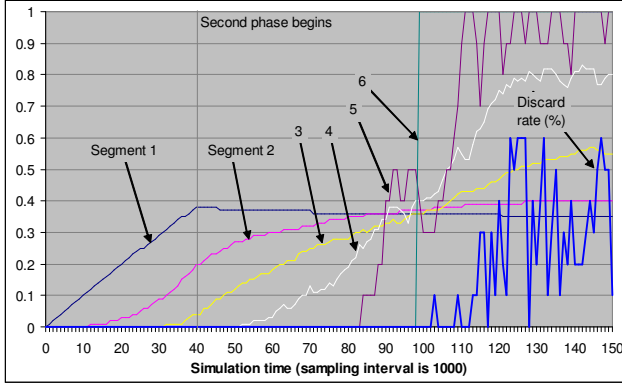
until  $i = c$ 
return -1

```

---

Search requires a similar procedure wherein the largest to the smallest segments are probed. Note that this order of probing makes sense because an element is more likely to be found in a larger segment, assuming that both have equal load. Search within every segment requires at most  $c$  probes because of the way inserts are performed. Thus the worst-case search time is  $O(s \times c)$ . Note that  $s$  equals  $(1 + \log_{1/r} n)$  in an ideal GP dimensioned peacock hash table. Thus, the worst-case search time is of the order of log of the total number of slots. Also, note that this represents the absolute worst-case search time instead of the expected worst-case. If  $c$  were set at a small constant, then the expected worst-case search time will be  $\Theta(\lg \lg n)$  for open addressing as well as chaining. This is assuming that every segment is equally loaded and the probability that an element is present in any given segment follows a geometric distribution. This represents a modest improvement over a naïve open addressing and almost no improvement over a naïve linear chaining. The average case search time will be  $\Theta(\lg \lg n)$ , which in fact represents a degradation. In real practice, however, things will not be as gloomy as it may seem, because the scaling factor,  $r$ , inverse of which is the base of the log scale, will generally be kept pretty low, like 0.1. This represents the condition, when every backup segment is 10% of the size of its predecessor. Our preliminary set of experiments suggests that a small  $r$  indeed results in an improvement in the average and worst-case performance.

The real benefit of peacock hashing, however, is that it ensures a constant time lookup operation. This is achieved by employing filters for every backup segment (except the main segment), which predicts the membership of an element with a failure probability independent of the size of segment. Such filters can be implemented using an



**Figure 5. Plotting fill level of various segments and the percent discard rate a) without balancing, b) with balancing**

enhanced Bloom filter as discussed in [5], at the moment we take the filter as a black box, and represent it by  $f_i$  for the  $i^{\text{th}}$  segment. It serves the membership query requests, with a constant **false positive** rate of  $p_f$ . We also assume that the filter ensures a zero false negative probability. With such filters, the search procedure looks at the membership query response and searches the segment whose filters have indicated the presence of the element. If all responses were false positives, the remaining segments are probed in the largest to smallest order. The pseudo-code for the search procedure is shown below:

---

```

PEACOCK-SEARCH( $T, k$ )
 $i \leftarrow 1$ 
repeat
  if FILTER-MEMBERSHIP-QUERY( $k$ ) > -1 then
    if SEGMENT-SEARCH( $T_i, k$ ) > -1 then
      return  $i$ 
    else  $i \leftarrow i+1$ 
  until  $i = s$ 
 $i \leftarrow 1$ 
repeat
  if FILTER-MEMBERSHIP-QUERY( $k$ ) = -1 then
    if SEGMENT-SEARCH( $T_i, k$ ) > -1 then
      return  $i$ 
    else  $i \leftarrow i+1$ 
  until  $i = s$ 
error "Unsuccessful Search"

SEGMENT-SEARCH( $T_i, k$ )
 $i \leftarrow 0$ 
repeat  $j \leftarrow h(k, i)$ 
  if  $T_i[j] = k$  then
    return  $j$ 
  else  $i \leftarrow i+1$ 
until  $i = c$ 
return -1

```

---

The average as well as the expected worst-case search times can be easily shown to be  $O(1 + f^{\lg n})$ . The exact average

search time in terms of various peacock parameters can also be computed.

### 3.5.2 Deletes and Re-balancing

While insert and lookup is straightforward, delete operation requires re-balancing of the segments because during deletes, the equilibrium loading of segments is disturbed. In an equilibrium loading state, a series of deletes followed by inserts can result in relatively high load at smaller segments which can eventually overflow them, resulting in high discard rates. At present, we only provide an intuitive reasoning for this phenomenon. For simplicity, we will consider the case when collision threshold is set at 1, so that collision resolution policy is not needed. Let us assume that we are in equilibrium loading state and  $m_i = r \times m_{i-1}$ , where  $m_i$  is the number of elements in the  $i^{\text{th}}$  segment. Thereafter a stream of "one delete followed by one insert" occurs.

The first observation is that, the rate at which elements arrive at the  $i^{\text{th}}$  segment ( $i > 1$ ) is equal to the load in the  $i-1^{\text{st}}$  segment. Thus, if  $m_i(t)$  is the number of elements in the  $i^{\text{th}}$  segment at time  $t$ , then during inserts, we have

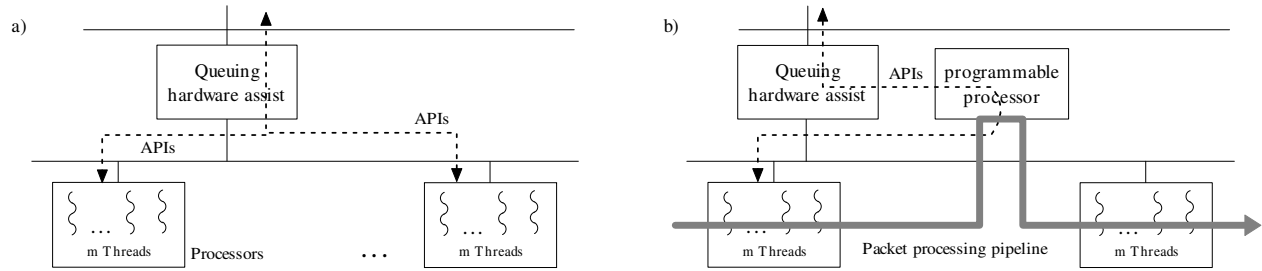
$$m_{i+1}(t) = m_{i+1}(t-1) + \prod_{j=0}^{i-1} m_j(t-1)/n_j(t-1)$$

for all  $i > 1$ . Thus, if elements are deleted and inserted repeatedly since the equilibrium state, the rate of deletion from a segment  $i$  will be  $m_i / \sum m_i$ , assuming that elements being deleted are picked randomly, while the insertion rate will be. The insertion rate will exceed the deletion rate in a segment  $i$  if:

$$m_{i-1}/n_{i-1} > m_i / \sum m_i$$

For smaller segments, this condition is likely to be satisfied easily; thus they will quickly fill up and overflow.

This scenario is illustrated in Figure 5a, where the load of various segments is plotted. The experiment was conducted on a peacock hash table with 6 segments, each using double hashing as the collision resolution policy and with a scaling factor,  $r$  of 0.1 (*i.e.* the largest segment has 100,000 slots). The collision threshold is set at 2. In the first phase, 40,000 elements are inserted one after another with no intermediate



**Figure 6. Two alternative organizations of the queuing hardware assist**

deletes. In the second phase, an element is deleted and another inserted one after another for an extended period of time. The sampling interval for the above plot is once per 1000 events. It is clear that smaller segments fill up quickly during the second phase in absence of any re-balancing, even when the total number of elements remains the same.

The results of an experiment, where rebalancing is done by exhaustively searching the backup segments and moving the elements upwards, are illustrated in Figure 5b. It is clear that rebalancing can improve the performance dramatically. However, at present, it is not clear which is the best strategy to perform rebalancing efficiently. Our general motive is to perform a thorough evaluation of peacock hashing and propose algorithms to perform rebalancing in an efficient way. An orthogonal research agenda is to consider the case when access latency and bandwidth of the segments are different.

#### 4. PACKET BUFFERING AND QUEUING – NEW DIRECTIONS

The second prime focus of the proposed research is packet buffering and queuing techniques. In the packet buffering area we plan to expand a recently proposed multi-channel packet buffering architecture to such memories that consist of multiple banks. In the packet queuing area, we plan to develop novel network processor centric architectures to implement the popular queuing algorithms. The architecture may consist of a specialized set of instructions on a network processor or a hardware assist or a combination of the both. We discuss these in the following sections.

##### 4.1 NP based packet queuing

Current network processors employ a chip multiprocessor (CMP) architecture, which consists of a cluster of processors to collectively and co-operatively perform any given job. In order to expedite the queuing operations which are often limited by the memory access latency, these CMPs contain a limited set of hardware assists, like a software-controlled on-chip memory to cache the queue headers and lengths. A recent paper [4] has thoroughly investigated such designs and proposed an alternative called queuing cache which consists of a hardware cache coupled with a queuing engine to realize the queue operations. The

efficiency of queuing cache is due to *i)* its effectiveness in avoiding the software instructions to manage the specialized cache and *ii)* its effectiveness in cutting the access latencies between the processors and the on-chip queue headers.

While queuing cache is shown to improve the queuing performance, it lacks the programmability of a software based approach; additionally its functions are limited to the management of a simple linked-list. Several advanced fair queuing algorithms use much more complex data-structures. For instance, recent developments have advocated the use of a data-structure which consists of a linked-list of nodes such that a node can store at most  $n$  elements. Such batched nodes can help in hiding the memory access latency. Our experience with the implementation of the queue manager in GENI project has shown that software implementation of such data-structures are complex and require hundreds of instructions which reduces the throughput. Besides, the traditional queuing cache design or NP based queuing assist becomes unusable for these structures.

We propose the design of a new hardware assist which can aid in efficient and simple implementation of such complex queuing structures. The proposed design is intended to use batching to maintain all linked-list structures, including the ones which are used to store the packet and queue headers. While batching helps in combating the memory latency problems, it also creates new challenges such as allocation of a new batch or management of the elements which are freed up from any given batch. The proposed design incorporates all functions associated with the maintenance of batched linked-list in hardware and provide application protocol interface (API) to the software. The hardware is also intended to integrate the functions associated with the management of the free nodes, thus relieving the software from the management of the freed up memory.

Clearly, the APIs which will be provided by the hardware assist is crucial and must maintain a right balance between the simplicity of the hardware and the flexibility provided to the programmers. Flexibility is important so that complex fair queuing policies can be implemented. An evaluation of the known fair queuing policies suggests that even though a large number of policies have been invented, the widely adopted ones are the weighted deficit round robin (WDRR)

and some of its variants like stratified round robin (SRR). These popular fair queuing policies employ a single or multiple circular linked-lists which holds the queues which are awaiting their schedule. Thus, the basic linked-list operations must be provided. Besides, the APIs must also provide support for operations on a batch of individual elements. We plan to develop these APIs in this proposed research; below we present an initial list.

API	Arguments	Function
initialize_list	max_length, batch_size	Initializes a list of the given length and batch size
add(rem)_elt(batch)_at_tail(head)	elt(batch), list	Adds (removes) an element (batch) at the tail (head) of the list
return_length	list	Returns length of list
join_list	list1, list2	Joins list1 and list2 and returns a new list
prefetch_head	List, n	Pre-fetches n nodes near head

Once the APIs are finalized, there are several architectural decisions in developing the hardware. First, one may couple the hardware assist with a programmable processor, thereby enabling a highly programmable queue management subsystem (see Figure 6b). Such queue management subsystem can be placed at any position in the processing pipeline provided that the hardware assist has physical connectivity to all processors. Such connectivity can be provided either by the shared interconnect between the processors and the hardware resources or by dedicated point to point links. An alternative architecture may keep the queuing hardware accessible by all processors; in which case any processor can use the APIs to implement the queuing and scheduling functions (see Figure 6a). From a performance point of view, the first choice is likely to yield higher performance; on the other hand the second choice may provide better flexibility. We plan to explore these trade-offs.

Another dimension is an efficient design of the hardware assist which will provide the queuing APIs. Our first order analysis in [4] suggests that the hardware implementation of a basic linked-list processor requires only a few thousand gates. In our case, batching and free memory management capabilities are likely to complicate the design, but only to an extent that the entire design will still require less than a hundred thousand gates. The most complex components of the design are likely to remain the cache array and the interconnection network which represented more than 80% of the total logic in our previous design. We also aim at *i*) exploring efficient alternatives to design the interconnection network and *i*) performing a trace-driven analysis to dimension the cache array.

## 4.2 Extending multichannel packet buffers

In our previous work [7], we have studied the performance of multichannel packet storage systems that use randomization to enable high performance in the presence of arbitrary packet retrieval patterns. The architecture

employs an on-chip SRAM, whose size is proportional only to the number of memory modules and doesn't grow as the product of the number of memory modules and the number of queues, making it practical for any system irrespective of the number of queues. In this research proposal, we plan to extend the idea to systems where the memory channels are implemented using multi-bank DRAM. Our preliminary analysis suggests each bank of a DRAM can be treated as a separate slow memory channel, thus the size of the on-chip SRAM will be multiplied by the number of banks. Since the number of banks in a DRAM is generally small, the on-chip SRAM is likely to remain small. In practice, individual banks of a DRAM are not as slow, for example banks in a four bank DRAM device are only two to three times slower than the memory interface. Therefore, the on-chip SRAM can be kept comparable to the size of the on-chip SRAM in a SRAM based multichannel packet buffer.

There are several other opportunities for extending the work presented in [7]. One direction that can be explored involves replacing the timestamp-based resequencer with one that uses sequence numbers. A sequence number based resequencer can forward chunks earlier than the timestamp-based one which holds chunks back until their age exceeds a fixed threshold. This does sacrifice the constant delay that can be obtained using timestamp-based resequencers, but can substantially reduce the minimum and average delay. The results obtained with our simplified DRAM modeling reflect such behavior and we plan to extend this model by incorporating the timing and interface properties of the commercial DRAM devices.

Another issue that is worth exploring has to do with the memory bandwidth inefficiency that can result from the use of fixed length chunks. Modern routers handle variable length packets that are typically divided into fixed length chunks for storage in off-chip memory. While fixed length chunks are much more convenient for the storage system designer, they can lead to significant inefficiencies, if packet lengths don't match the chunk size. In particular, packet lengths that are just slightly too long to fit in a single chunk can lead to effective bandwidth reductions of close to 50%. One way to reduce this loss of effective bandwidth is to allow chunk sizes to vary across a limited range of sizes. This would allow one to divide packets into chunks that are all at least equal in length to the minimum chunk size, eliminating the loss of memory bandwidth that occurs when a chunk with only a small amount of data is transferred to/from memory. Of course, the use of variable size chunks does complicate the mechanisms that distribute the traffic randomly across memory channels, requiring significant extension of the architecture. We plan to evaluate such architectural extensions and their implication on the system performance.

## 5. DEEP PACKET INSPECTION – NEW DIRECTIONS

Our third research focus is deep packet inspection, which is employed by network services that handle packets based on payload content, in addition to the structured information in packet headers. Forwarding packets based on content requires new levels of support in networking equipment, wherein the packet payload is compared against a set of patterns describe by regular expressions. Regular expressions are generally implemented using either a NFA or a DFA with DFA being the preferred method. DFAs are fast, however they require prohibitive amounts of memory.

In this proposal, we attempt to thoroughly investigate the memory problems associated with the DFA based pattern matchers. We argue that there are three principal reasons behind the memory problems. First, traditional approach takes no interest in exploiting the fact that normal data streams rarely match more than first few symbols of any signature. In such situations, if one constructs a DFA for the entire signature, then most portions of the DFA will be unvisited. Thus the approach of keeping the entire automaton active appears wasteful; we call this deficiency *insomnia*. Second, a DFA usually maintains a single state of execution, due to which it is unable to efficiently follow the progress of multiple partial matches. They employ a separate state for each such combination of partial match, thus the number of states can explode combinatorially. It appears that if one equips an automaton with a small auxiliary memory which it will use to register the events of partial matches, then a combinatorial explosion can be avoided; we refer to this drawback of a DFA as *amnesia*. Third, DFA is inefficient in counting; for example a DFA will require 4 billion states to implement a 32-bit counter. We call this deficiency *acalulia*.

In this proposal, we propose to work on novel solutions to tackle these three drawbacks. We introduce our initial set of solutions in the following sections.

### 5.1 Curing DFA from Insomnia

Traditional approach of pattern matching constructs an automaton for the entire regular expression (reg-ex) signature, which is used to parse the input data. However, normal flows rarely match more than first few symbols of any signature. Thus, the traditional approach appears wasteful; the automaton unnecessarily bloats up in size as it attempts to represent the entire signature even though the tail portions are rarely visited. Rather, the tail portion can be isolated from the automaton, and put to sleep during normal traffic conditions and woken up only when they are needed. Since the traditional approach is unable to perform such selective sleeping and keeps the automaton awake for the entire signature, we call this deficiency *insomnia*.

In other words, insomnia can be viewed as the inability of the traditional pattern matchers to isolate frequently visited portions of a signature from the infrequent ones. Insomnia is dangerous due to two reasons *i*) the infrequently visited tail portions of the reg-exes are generally complex (contains closures, unions, and length restrictions) and long (more than 80% of the signature), and *ii*) the size of fast representations of reg-exes (*e.g.* DFA) usually increases exponentially with the length and complexity of an expression. Thus, without a cure from insomnia, a DFA of hundreds of reg-exes may become infeasible or will require enormous amounts of memory.

An obvious cure to insomnia will essentially require an isolation of the frequently visited portions of the signatures from the infrequent ones. Clearly, frequently visited portions must be implemented with a fast representation like a DFA and stored in a fast memory in order to maintain high parsing rates. Moreover, since fast memories are less dense and limited in size, and fast representations like DFA usually suffer from state blowup, it is vital to keep such fast representations compact and simple. Fortunately, practical signatures can be cleanly split into simple prefixes and suffixes, such that the prefixes comprise of the entire frequently visited portions of the signature. Therefore, with such a clean separation in place, only the automaton representing the prefixes need to remain active at all times; thereby, curing the traditional approach from insomnia by keeping the suffix automaton in a sleep state most of the times.

There is an important tradeoff involved in such a prefix and suffix based pattern matching architecture. The general objective is to keep the prefixes small, so that the automaton which is awake all the time remains compact and fast. At the same time, if the prefixes are too small then normal data streams will match them very often, thereby waking up the suffixes more frequently than desired. Notice that, during anomalous conditions the automaton representing the suffixes will be triggered more often; however, we discuss such scenarios later. Under normal conditions, the architecture must therefore balance the tradeoff between the simplicity of the fast automaton and the dormancy of the slow automaton.

We refer to the automaton which represents the prefixes as the *fast path* and the other automata as the *slow path*. Fast path remains awake all the time and parses the entire input data stream, and activates the slow path once it finds a matching prefix. There are two expectations. First, the slow path should be triggered rarely. Second, the slow path should process a small fraction of the input data; hence it can use a slow memory technology and a compact representation like a NFA, even if it is relatively slow. In order to meet these expectations, we must ensure that the normal data streams either do not match the prefixes of the

signatures or match them rarely. Additionally, even after a prefix match, the slow path processing should not continue for a long time. The likelihood that these two expectations will be met during normal traffic conditions will depend directly upon the signatures and the positions where they are split into prefixes and suffixes. Thus, it is critically important to decide these split positions and we describe our procedure to compute these in the next section.

### 5.1.1 Splitting the regular expressions

The dual objectives of the splitting procedure are that the prefixes remain as small as possible, and at the same time, the likelihood that normal data matches these prefixes is low. The probability of matching a prefix depends upon its length and the distribution of various symbols in the input data. In this context, it may not be acceptable to assume a uniform random distribution of the input symbols (*i.e.* every symbol appears with a probability of  $1/256$ ) because some words appear much more often than the others (*e.g.* “HELO” in an ICMP packet). Therefore, one needs to consider a *trace driven probability distribution* of various input symbols. With these traces, one can compute the matching probability of prefixes of different lengths under normal and attack or anomalous traffic. This probability will establish the rate at which slow path will be triggered.

In addition to the “matching probabilities”, it is important to consider the probabilities of making transitions between any two states of the automaton. This probability will determine how long the slow path will continue processing once it is triggered. These transition probabilities are likely to be dependent upon the previous stream of input symbols, because there is a strong correlation between the occurrences of various symbols, *i.e.* when and where they occur with respect to each other. The transition probabilities as well as the matching probabilities can be assigned by constructing an NFA of the regular expressions signatures and parsing the same against normal and anomalous traffic.

More systematically, given the NFA of each regular expression, we determine the probability with which each state of the NFA becomes active and the probability that the NFA takes its different transitions. Once these probabilities are computed, we determine a cut in the NFA graph, so that *i)* there are as few nodes as possible on the left hand side of the cut, and *ii)* the probability that states on the right hand side of the cut is active is sufficiently small. This will ensure that the fast path remains compact and the slow path is triggered only occasionally. While determining the cut, we also need to ensure that the probability of those transitions which leaves some NFA node on the right hand side and enters some other node on the same side of the cut remains small. This will ensure that, once the slow path is triggered, it will stop after processing a few input symbols. Clearly, the cut computed from the normal traffic traces and

from the attack traffic are likely to be different, thus the corresponding prefixes will also be different. We adopt the policy of taking the longer prefix. Below, we formalize the procedure to determine cuts in the NFA graphs.

Let  $p_s : Q \rightarrow [0, 1]$  denote the probability with which the NFA states are active. Let the cut divides the NFA states into a fast and a slow region. Initially, we keep all states in the slow region; thus the slow path probability  $p$  is  $\sum p_s$ .

Afterwards, we begin moving states from the slow region to the fast region. The movements are performed in a breadth first order beginning at the start state of the NFA, and those states are moved first, whose probabilities of being active are higher. After a state  $s$  is moved to the fast region,  $p_s[s]$  is reduced from the slow path probability  $p$ . We continue these movements, until the slow path probability,  $p$  becomes smaller than  $\epsilon$ , the slow processing capacity threshold. This method gives us the first order estimate of the cut between the fast and the slow path. Such a cut will ensure that the slow path processes only  $\epsilon$  fraction of the total bytes in the input stream. The procedure is pseudo-code form described below.

For a large majority of the signatures which are used in the current systems, this method will cleanly split the regular expressions into prefix and suffix portions. However, for certain types of regular expressions, the above method will not result into a clean split. For instance an expression  $ab(cd|ef)gh$ . may be cut at the states which corresponds to the locations of the prefix  $abc$  and  $abe$ . We propose to split such types of expressions by extending the prefixes until a clean split of the expression is possible. Thus, in the above example, we will extend the cut to the states which corresponds to the prefix  $abcd$  and  $abef$ ; thus the prefix portion will become  $ab(cd|ef)$  and the suffix will be  $gh$ .

---

```

procedure find-cut(nfa  $M(Q, q_0, \delta_n, A, \Sigma)$ , map  $p_s$  :
state $\rightarrow[0,1]$ );
(1) heap  $h$ ;
(2) map  $mark$ : state $\rightarrow$ bit;
(3) set  $state$   $fast$ ;
(4) float  $p = \sum p_s$  ;
(5)  $h.insert(q_0, p_s(q_0))$ ;
(6) do  $h \neq []$  and  $p > \epsilon \Rightarrow$ 
(7)   state  $s := h.findmax()$ ;  $h.remove(t)$ ;
(8)    $mark[s] = 1$ ;  $fast = fast \cup s$ ;    $p = p - p_s(s)$ ;
(9)   for char  $c \in \Sigma \Rightarrow$ 
(10)    for state  $t \in \delta_i(s, c) \Rightarrow$ 
(11)     if not  $mark[t] \Rightarrow h.insert(t, p_s(t))$ ; fi
(12)   rof
(13) rof
(14) od
end;

```

---

The above splitting procedure would provide a method to split the reg-ex signatures into a fast path and a slow path. The method first attempts to keep the combined probability of the states in the fast path very high compared to that of the slow path. At the same time, during the fast path construction, it selects only those states that have high activation probabilities compared to others. Thus both of our objectives are fulfilled: the slow path is triggered rarely and it remains active only for a short duration.

### 5.2 H-FA: Curing DFAs from Amnesia

DFA state explosion occurs primarily due to amnesia, or the incompetence of the DFA to follow multiple partial matches with a single state of execution. Before introducing the cure to amnesia, we re-examine the connection between amnesia and state explosion. As suggested previously, DFA state explosions usually occur due to those signatures which comprise of simple patterns followed by closures over characters classes (e.g.  $*$  or  $[a-z]^*$ ). The simple pattern in these signatures can be matched with a stream of suitable characters and the subsequent characters can be consumed without moving away from the closure. These characters can begin to match either the same or some other reg-ex, and such situations of multiple partial matches have to be followed. In fact, every permutation of multiple partial matches has to be followed. A DFA represents each such permutation with a separate state due to its inability to remember anything other than its current state (amnesia). With multiple closures, the number of permutations of the partial matches can be exponential, thus the number of DFA states can also explode exponentially.

An intuitive solution to avoid such exponential explosions is to construct a machine, which can remember more information than just a single state of execution. NFAs fall in this genre; they are able to remember multiple execution states, thus they avoid state explosion. NFAs, however, are slow; they may require  $O(n^2)$  state traversals to consume a character. In order to preserve the fast execution, we would like to ensure that the machine maintains a single state of execution. One way to enable single execution state and yet avoid state explosion is to equip the machine with a small and fast *cache*, which will act as a history buffer and register key events which may occur during the parse, such as encountering a closure. Recall that the state explosion occurs because the parsing get stuck at a single or multiple closures; thus if the history buffer will register these events then the automaton may avoid using several states. We call this class of machines History based Finite Automaton (H-FA).

The execution of the H-FA is augmented with the history buffer. Its automaton is similar to a traditional DFA and consists of a set of states and transitions. However, multiple transitions on a single character may leave from a state (like in a NFA). Nevertheless, only one of these transitions is

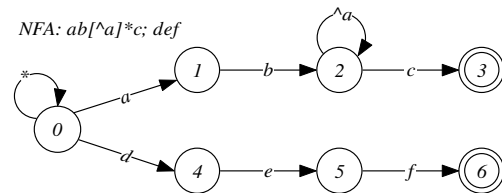
taken during the execution, which is determined after examining the contents of the history buffer; thus certain transitions have an associated condition. The contents of the history buffer are updated during the machine execution. The size of the H-FA automaton (number of states and transitions) depends upon those partial matches, which are registered in the history buffer; if we judiciously choose these partial matches then the H-FA can be kept extremely compact. The next obvious questions are: *i)* how to determine these partial matches? *ii)* Having determined these partial matches, how to construct the automaton? *iii)* How to execute the automaton and update the history buffer? We proceed with the discussion of H-FA which attempts to answer these questions.

### 5.3 Motivating example

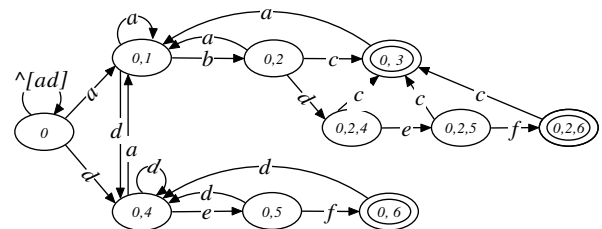
We introduce the construction and executing of H-FA with a simple example. Consider two reg-ex patterns listed below:

$$r_1 = .*ab[^a]^*c; \quad r_2 = .*def;$$

These patterns create a NFA with 7 states, which is shown below:

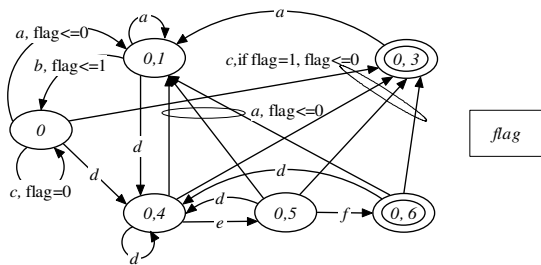


Let us examine the corresponding DFA, which is shown below (some transitions are omitted to keep the figure readable):

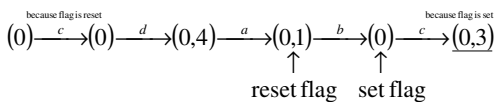


The DFA has 10 states; each state corresponds to a subset of NFA states, as shown above. There is a small blowup in the number of states due to the presence of the Kleene closure  $[^a]^*$  in the expression  $r_1$ . Once the parsing reaches the Kleene closure (NFA state 2), subsequent input characters can begin to match the expression  $r_2$ , hence the DFA requires three additional states (0,2,4), (0,2,5) and (0,2,6) to follow this multiple match. There is a subtle difference between these states and the states (0,4), (0,5) and (0,6), which corresponds to the matching of the reg-ex  $r_2$  alone: DFA states (0,2,4), (0,2,5) and (0,2,6) comprise of the same subset of the NFA states as the DFA states (0,4), (0,5) and (0,6) plus they also contain the NFA state 2.

In general, those NFA states which represent a Kleene closure appear in several DFA states. The situation becomes more serious when there are multiple reg-exes containing closures. If a NFA consists of  $n$  states, of which  $k$  states represents closures, then during the parsing of the NFA, several permutations of these closure states can become active;  $2^k$  permutations are possible in the worst case. Thus the corresponding DFA, each of whose states will be a set of the active NFA states, may require total  $n2^k$  states. These DFA state set will comprise of one of the  $n$  NFA states plus one of the  $2^k$  possible permutations of the  $k$  closure states. Such an exponential explosion clearly occurs due to amnesia, as the DFA is unable to remember that it has reached these closure NFA states during the parsing. Intuitively, the simplest way to avoid the explosion is to enable the DFA to remember all closures which has been reached during the parsing. In the above example, if the machine can maintain an additional flag which will indicate if the NFA state 2 has been reached or not, then the total number of DFA states can be reduced. One such machine is shown below:



This machine makes transitions like a DFA; besides it maintains a flag, which is either set or reset (indicated by  $\leq 1$ , and  $\leq 0$  in the figure) when certain transitions are taken. For instance transition on character  $a$  from state (0) to state (0,1) resets the flag, while transition on character  $b$  from state (0,1) to state (0) sets the flag. Some transitions also have an associated condition (flag is set or reset); these transitions are taken only when the condition is met. For instance the transition on character  $c$  from state (0) leads to state (0,3) if the flag is set, else it leads to state (0). This machine will accept the same language which is accepted by our original NFA, however unlike the NFA, this machine will make only one state traversal for an input character. Consider the parse of the string "cdabc" starting at state (0), and with the flag reset.



In the beginning the flag is reset; consequently the machine makes a move from state (0) to state (0) on the input character  $c$ . On the other hand, when the last input character  $c$  arrives, the machine makes a move from state (0) to state (0,3) because the flag is set this time. Since the

state (0,3) is an accepting state, the string is accepted by the machine.

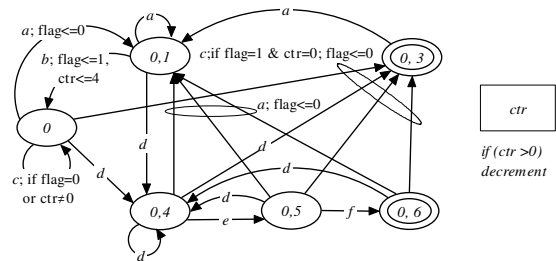
Such a machine can be easily extended so that it will maintain multiple flags, each indicating a Kleene closure. The transitions will be made depending upon the state of the flags and certain flags will also be updated. As illustrated by the above example, augmenting an automaton with these flags can avoid state explosion. However, a more systematic method is needed to construct H-FA, which we plan to develop in the proposed research.

### 5.4 H-cFA: Curing DFAs from Acalulia

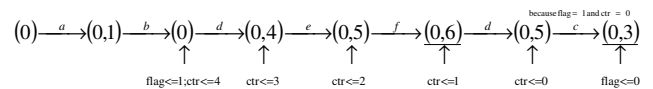
We now introduce "History based counting finite Automata" or H-cFA, which efficiently cures traditional FA from acalulia, due to which a FA is unable to efficiently count the occurrences of certain sub-expressions. We again introduce H-cFA with an example; we consider the same set of two reg-exes with the closure in the first reg-ex replaced with a length restriction of 4, as shown below:

$$r_1 = . *ab[^a]^4c; \quad r_2 = .*def;$$

A DFA for these two reg-exes will require 20 states. The blowup in the number of states in the presence of the length restriction occurs due to acalulia or the inability of the DFA to keep track of the length restriction. Let us now construct an H-cFA for these reg-exes. The first step in this construction replaces the length restriction with a closure, and constructs the H-FA, with the closure represented by a flag in the history buffer. Subsequently with every flag in the history buffer, a counter is appended. The counter is set to the length restriction value by those conditional transitions which set the flag, while it is reset by those transitions which reset the flag. Furthermore, those transitions whose condition is a set flag are attached with an additional condition that the counter value is 0. During the executing of the machine, all positive counters are decremented for every input character. The resulting H-cFA is shown below:



Consider the parse of the string "abdefdc" by this machine starting at the state (0), and with the flag and counter reset.



As the parsing reaches the state (0,1), and makes transition

to the state (0), the flag is set, and the counter is set to 4. Subsequent transitions decrements the counter. Once the last character  $c$  of the input string arrives, the machine makes a transition from state (0,5) to state (0,3), because the flag is set and counter is 0; thus the string is accepted. This example illustrates the straightforward method to construct H-cFAs from H-FAs. Several kinds of length restrictions including “greater than  $i$ ”, “less than  $i$ ” and “between  $i$  and  $j$ ” can be implemented. Each of these conditions will require an appropriate condition with the transition. For example, “less than  $i$ ” length restriction will require that the conditional transition becomes true when the history counter is greater than 0.

From the hardware implementation perspective, a greater than or less than condition requires approximately equal number of gates needed by an equality condition, hence different kinds of length restrictions are likely to have identical implementation cost. In fact, a reprogrammable logic can be devised equally efficiently, which can check each of these conditions. Thus, the architecture will remain flexible in face of the frequent signature updates. This simple cure to acalulia is extremely effective is reducing the number of states, specifically in the presence of long length restrictions. Snort signatures comprises of several long length restrictions, hence H-cFA is extremely valuable in implementing these signatures. Even though the initial results are encouraging, in order to evaluate the true effectiveness of our cures, the proposed research will perform a detailed experimental evaluation of our methods using real life signatures,

### 5.5 Additional Research Directions

In our final research direction, we intend to apply HEXA on a finite automaton, in which case several challenges are likely to appear. The first challenge will arise due to the fact that unlike in a trie, there can be cycles of transitions. With these cycles, the HEXA identifier will become non-deterministic in length. While this problem can be tackled by limiting the length of the HEXA identifiers, the problem may persist for those cycles that are too small. Another problem will appear when multiple transitions will lead to the same next node. In such situations, the destination node may get multiple distinct HEXA identifiers which will clearly challenge our initial assumption that each node has a unique HEXA identifier. This problem may be solved by treating all such nodes as special case; however this will undermine the effectiveness of HEXA. In fact, it is not clear if HEXA will reduce the memory needed to represent an arbitrary automaton at all. Our final objective in this research is to answer these questions.

## 6. ACKNOWLEDGMENTS

I am grateful to Will Eatherton and John Williams for providing the reg-ex signatures used in Cisco security appliances. I feel thankful to Prof. Michael Mitzenmacher

who is actively involved in the development of HEXA architecture. I am also thankful to Prof. George Varghese who is helping us in developing H-FA machines. I would like to thank Prof. Patrick Crowley for his continued support and motivation. Finally, I am thankful to Dr. Jonathan Turner, who has remained a source of inspiration to me. This work is intended to be supported by the NSF Grant CNS-0325298 and a URP grant from Cisco Systems.

## 7. REFERENCES

- [1] Sailesh Kumar, Jonathan Turner and John Williams, "Advanced Algorithms for Fast and Scalable Deep Packet Inspection", IEEE/ACM Symposium on Architectures for Networking and Communications Systems, San Jose, December, 2006.
- [2] Sailesh Kumar, Michela Becchi, Patrick Crowley and Jonathan Turner, "CAMP: Fast and Efficient IP Lookup Architecture", IEEE/ACM Symposium on Architectures for Networking and Communications Systems, San Jose, December, 2006.
- [3] Sailesh Kumar, Sarang Dharmapurikar, Fang Yu, Patrick Crowley and Jonathan Turner, "Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection", ACM SIGCOMM'06, Pisa, Italy, Sep 12-15, 2006.
- [4] Sailesh Kumar, John Maschmeyer, and Patrick Crowley, "Queuing Cache: Exploiting Locality to Ameliorate Packet Queue Contention and Serialization", ACM International Conf. on Computing Frontiers, Ischia, Italy, May 2-5, 2006.
- [5] Sailesh Kumar, and Patrick Crowley, "Segmented Hash: An Efficient Hash Table Implementation for High Performance Networking Subsystems", IEEE/ACM Symp. on Architectures for Networking and Comm. Systems, Princeton, Oct., 2005.
- [6] Sailesh Kumar, Patrick Crowley, and Jonathan Turner, "Buffer Aggregation: Addressing Queuing Subsystem Bottlenecks at High Speeds", IEEE Symposium on High Performance Interconnects, Stanford, August 17-19, 2005.
- [7] Sailesh Kumar, Patrick Crowley, and Jonathan Turner, "Design of Randomized Multichannel Packet Storage for High Performance Routers", IEEE Symposium on High Performance Interconnects, Stanford, August 17-19, 2005.
- [8] Sarang Dharmapurikar, Sailesh Kumar, John Lockwood and Patrick Crowley, "Optimizing Memory Bandwidth of a Multi-Channel Packet Buffer", IEEE Globecom, St. Louis, Nov. 2005.
- [9] Sailesh Kumar, and Patrick Crowley, "A Scalable, Cache-Based Queue Management Subsystem for Network Processors", ASPLOS-XI Building Block Engine Architectures for Computers and Networks, 2004, October 10, 2004.
- [10] Intel IXP series of network processors.
- [11] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, "Scalable High Speed IP Routing Lookups," in *Proc. ACM SIGCOMM'97*, pp. 25-37.
- [12] 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.
- [13] 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.
- [14] M. Degermark, A. Brodnik, S. Carlsson and S. Pink, "Small Forwarding Tables for Fast Routing Lookups", in *Proc. of ACM SIGCOMM 1997*.
- [15] W. Eatherton, Z. Dittia, and G. Varghese, "Tree bitmap: Hardware/software ip lookups with incremental updates", in *ACM SIGCOMM Computer Comm. Review*, 34(2), 2004.

- [16] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor, "Longest prefix matching using Bloom filters," *ACM SIGCOMM 2003*.
- [17] A. Basu and G. Narlikar, "Fast Incremental Updates for Pipelined Forwarding Engines", in *Proceedings of INFOCOM 2003*, 2003
- [18] J. Hasan and T.N. Vijaykumar, "Dynamic Pipelining: Making IP-Lookup Truly Scalable", in *Proc. ACM SIGCOMM 2005*, pp 205-216.
- [19] A. J. McAuley and P. Francis, "Fast Routing Table Lookup Using CAMs", in *Proc. INFOCOM 1993*.
- [20] Francis Zane, Girija Narlikar, and Anindya Basu, "CoolCAMs: Power-Efficient TCAMs for Forwarding Engines", in *Proc. INFOCOM 2003*.
- [21] Haoyu Song, Jonathan Turner and John Lockwood, "Shape Shifting Tries for Faster IP Route Lookup", in *ICNP 2005*, 11/2005.
- [22] S. Suri, G. Varghese, and P. Warkhede, "Multiway range trees: Scalable IP lookup with fast updates", *GLOBECOM 2001*.
- [23] S. Nilsson and G. Karlsson, "Fast Address Lookup for Internet Routers," in *Proc. of IEEE Conf. on BroadBand Communications Tech.*, 1998.
- [24] Timothy Sherwood, George Varghese and Brad Calder, "A Pipelined Memory Architecture for High Throughput Network Processors," In *Proceedings of the 30th Annual ISCA*, pages 288-299, 2003.
- [25] Florin Baboescu, Dean M. Tullsen, Grigore Rosu, Sumeet Singh, "A Tree Based Router Search Engine Architecture with Single Port Memories," in *ISCA 2005*.
- [26] R. Graham, F. Graham, and G. Varghese, "Parallelism versus Memory Allocation in Pipelined Router Forwarding Engines", in the *Proceedings of SPAA'04, Barcelona, Spain, (2004)*, pp. 103—111.
- [27] Appenzeller, G., I. Keslassy and N. McKeown. "Sizing Router Buffers," *ACM SIGCOMM 2004*, 8/04.
- [28] Henrion, M. "Resequencing system for a switching node." U.S. Patent #5,127,000, 6/92.
- [29] Y. Joo, N McKeown, "Doubling Memory Bandwidth for Network Buffers", *IEEE Infocom 1998*, San Francisco.
- [30] Iyer, S., R. R. Compella, and N. McKeown, "Designing Buffers for Router Line Cards," *Stanford University HPNG Technical Report - TR02-HPNG-031001*, 2002.
- [31] Iyer, S., R. R. Kompella, and N. McKeown, "Analysis of a Memory Architecture for Fast Packet Buffers," *IEEE HPSR*, 5/02.
- [32] A. Nikologiannis, M. Katevenis, "Efficient Per-Flow Queuing in DRAM at OC-192 Line Rate using Out-of-Order Execution Techniques," *Proc. IEEE Int. Conf. on Communications (ICC'2001)*, Helsinki, Finland, June 2001.
- [33] DDR-SDRAM, RLDRAM, Micron Technology, Inc.
- [34] FCRAM, Toshiba America Electronic Components, Inc.
- [35] RDRAM, Rambus, Inc.
- [36] C. Villamizar and C. Song, "High Performance TCP in ANSNET," *Computer Communication Review*, Vol. 24, No. 5, pp. 45-60, Oct. 1994.
- [37] A. K. Parekh, "A generalized processor sharing approach to flow control in integrated services networks," Ph.D. thesis, Dept. of Elect. Eng. and Comput. Sci., M.I.T., Feb. 1992.
- [38] M. Shreedhar, G. Varghese, "Efficient Fair Queuing using Deficit Round Robin", *ACM SIGCOMM*, Cambridge, 1995.
- [39] R. Sommer, V. Paxson, "Enhancing Byte-Level Network Intrusion Detection Signatures with Context," *ACM conf. on Computer and Communication Security*, 2003, pp. 262--271.
- [40] J. E. Hopcroft and J. D. Ullman, "Introduction to Automata Theory, Languages, and Computation," Addison Wesley, 1979.
- [41] J. Hopcroft, "An nlogn algorithm for minimizing states in a finite automaton," in *Theory of Machines and Computation*, J. Kohavi, Ed. New York: Academic, 1971, pp. 189--196.
- [42] Bro: A System for Detecting Network Intruders in Real-Time. <http://www.icir.org/vern/bro-info.html>
- [43] M. Roesch, "Snort: Lightweight intrusion detection for networks," In *Proc. 13th Systems Administration Conference (LISA)*, USENIX Association, November 1999, pp 229--238.
- [44] S. Antonatos, et. al, "Generating realistic workloads for network intrusion detection systems," In *ACM Workshop on Software and Performance*, 2004.
- [45] A. V. Aho and M. J. Corasick, "Efficient string matching: An aid to bibliographic search," *Comm. of the ACM*, 18(6):333--340, 1975.
- [46] B. Commentz-Walter, "A string matching algorithm fast on the average," *Proc. of ICALP*, pages 118--132, July 1979.
- [47] S. Wu, U. Manber, "A fast algorithm for multi-pattern searching," *Tech. R. TR-94-17*, Dept. of Comp. Science, Univ of Arizona, 1994.
- [48] Fang Yu, et. al., "Fast and Memory-Efficient Regular Expression Matching for Deep Packet Inspection", *UCB tech. report, EECS-2005-8*.
- [49] N. Tuck, T. Sherwood, B. Calder, and G. Varghese, "Deterministic memory-efficient string matching algorithms for intrusion detection," *IEEE Infocom 2004*, pp. 333--340.
- [50] L. Tan, and T. Sherwood, "A High Throughput String Matching Architecture for Intrusion Detection and Prevention," *ISCA 2005*.
- [51] 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.
- [52] S. Yusuf and W. Luk, "Bitwise Optimised CAM for Network Intrusion Detection Systems," *IEEE FPL 2005*.
- [53] R. Sidhu and V. K. Prasanna, "Fast regular expression matching using FPGAs," In *IEEE Symposium on Field- Programmable Custom Computing Machines*, Rohnert Park, CA, USA, April 2001.
- [54] C. R. Clark and D. E. Schimmel, "Efficient reconfigurable logic circuit for matching complex network intrusion detection patterns," In *Proceedings of 13th International Conference on Field Program*.
- [55] J. Moscola, et. al, "Implementation of a content-scanning module for an internet firewall," *IEEE Workshop on FPGAs for Custom Comp. Machines*, Napa, USA, April 2003.
- [56] R. W. Floyd, and J. D. Ullman, "The Compilation of Regular Expressions into Integrated Circuits", *Journal of ACM*, vol. 29, no. 3, pp 603-622, July 1982.
- [57] Scott Tyler Shafer, Mark Jones, "Network edge courts apps," [http://infoworld.com/article/02/05/27/020527newebdev\\_1.html](http://infoworld.com/article/02/05/27/020527newebdev_1.html)
- [58] TippingPoint X505, [www.tippingpoint.com/products\\_ips.html](http://www.tippingpoint.com/products_ips.html)
- [59] Cisco IOS IPS Deployment Guide, [www.cisco.com](http://www.cisco.com)
- [60] Tarari RegEx, [www.tarari.com/PDF/RegEx\\_FACT\\_SHEET.pdf](http://www.tarari.com/PDF/RegEx_FACT_SHEET.pdf)
- [61] N.J. Larsson, "Structures of string matching and data compression," PhD thesis, Dept. of Computer Science, Lund University, 1999 .
- [62] 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*.

- [63] Z. K. Baker, V. K. Prasanna, "Automatic Synthesis of Efficient Intrusion Detection Systems on FPGAs," in *Field Prog. Logic and Applications*, Aug. 2004, pp. 311–321.
- [64] Y. H. Cho, W. H. Mangione-Smith, "Deep Packet Filter with Dedicated Logic and Read Only Memories," *Field Prog. Logic and Applications*, Aug. 2004, pp. 125–134.
- [65] M. Gokhale, et al., "Granid: Towards Gigabit Rate Network Intrusion Detection Technology," *Field Programmable Logic and Applications*, Sept. 2002, pp. 404–413.
- [66] J. Levandoski, E. Sommer, and M. Strait, "Application Layer Packet Classifier for Linux". <http://l7-filter.sourceforge.net/>.
- [67] SafeXcel Content Inspection Engine, hardware regex acceleration IP.
- [68] Network Services Processor, OCTEON CN31XX, CN30XX Family.
- [69] Will Eatherton, John Williams, "An encoded version of reg-ex database from cisco systems provided for research purposes".
- [70] R. Pagh, F. F. Rodler, Cuckoo Hashing, *Proc. 9th Annual European Symposium on Algorithms*, August 28-31, 2001, pp.121-133.
- [71] 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.