WASHINGTON UNIVERSITY
SCHOOL OF ENGINEERING AND APPLIED SCIENCE
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

A THESIS ON ACCELERATION OF NETWORK PROCESSING ALGORITHMS

by

Sailesh Kumar

Prepared under the direction of Prof. Jonathan S. Turner and Prof. Patrick Crowley

---

A thesis presented to the School of Engineering and Applied Science of
Washington University in partial fulfillment of the
requirements for the degree of
DOCTOR OF SCIENCE

May 2008
Saint Louis, Missouri

WASHINGTON UNIVERSITY
SCHOOL OF ENGINEERING AND APPLIED SCIENCE
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

ABSTRACT

A THESIS ON ACCELERATION OF
NETWORK PROCESSING ALGORITHMS

by
Sailesh Kumar

ADVISORS: Prof. Jonathan S. Turner and Prof. Patrick Crowley

May 2008
St. Louis, Missouri

Modern networks process and forward an increasingly large volume of traffic and 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 to efficiently implement the various network features. In this thesis, we focus on two core network features namely: *i*) IP packet forwarding, and *ii*) packet content inspection. We thoroughly investigate the existing methods to realize these two features and evaluate their usability on modern implementation platforms like network processors. Afterwards, we introduce a number of novel algorithms which not only improve the performance theoretically, but also better utilize the capabilities available with the modern hardware. The major contributions of this work include the design and architecture of an ASIC to perform longest prefix match operations on packet headers that uses substantially less memory, an embedded memory based design for regular expressions based packet content inspection, and a general purpose algorithm to cost-efficiently implement regular expressions signatures used in current security systems. We evaluate the proposed algorithms using network processor platforms and cycle accurate ASIC models, which provides us a first order estimate of the usability of our methods.

Contents

List of Tables

List of Figures

Acknowledgements

The work on this thesis has been an inspiring, often exciting, sometimes challenging, but always interesting experience. It has been made possible by many other people who have supported me. First of all, I would like to express my deepest sense of gratitude to my advisor Prof. Jonathan S. Turner. I came to Washington University four years ago with a lot of energy but no clear direction and focus. Prof. Turner taught me how to do research, rigorously define a problem and then pursue for a solution. He not only gave me complete freedom to explore the research topics of my interest, but also provided me an unflinching encouragement and support in various ways. His truly ingenious intuition, tremendous patience and diligent mentorship has made him a source of inspiration, which enriched my growth as a student, a researcher and a scientist. I am indebted to him more than he knows.

I gratefully acknowledge my co-advisor Prof. Patrick Crowley for his early guidance, supervision, and crucial contributions. From the very beginning, he treated me as a peer and played an instrumental role in keeping me engaged and motivated. He quickly grasped my research interests and strengths and guided me in the right direction by introducing me to the wonderful world of computer architecture and network processors, and set a course for long-term collaboration.

The genesis of this thesis can be traced back to my summer internship at Cisco Systems working with Will Eatherton and John Williams. Being one of the leaders at Cisco, Will and John had an early intuition of the performance issues surrounding packet content inspection, and they suggested me to focus my energy in this direction. I continued working in this area after returning from Cisco, which eventually became the backbone of this thesis.

I am grateful to Prof. Michael Mitzenmacher and Prof. George Varghese for agreeing to collaborate and guide me on a number of ideas, which led to key publications that became an integral part of this thesis. Their consummate emphasis on clarity and understanding has provided me a new perspective, and nurtured and encouraged me to produce the highest quality research that I could.

I am grateful to the other members of my committee, Prof. John Lockwood, Prof. Roger Chamberlain, and Prof. Bob Morley for their valuable feedback and advice. My experience as a graduate student was enhanced by the talented group of students and friends around me. Many thanks go to Sarang Dharmapurikar for the long brainstorm sessions, which helped me in shaping up my research direction. Inexpressible thanks also go to Christoph Jechlitschek, and Michela Becchi for developing a number of ideas, and spicing up my stay in St. Louis. I would also like to express my sincere gratitude to my past and present roommates Dushyanth Balasubramanian, Shakir James, and Nigel Thomas for patiently supporting me during my tough times.

I would also like to thank Myrna Harbison, Jean Grothe, Sharon Matlock, Peggy Fuller, and Stella Sung for their invaluable and diligent effort in making the life of graduate students easy. A number of Washington University staff and faculty members have generously provided their wisdom, encouragement and assistance. In particular, I would like to thank John DeHart, Fred Kuhns, and Prof. Sergey Gorinsky.

Finally, I would like to thank my parents, sisters, brother-in-laws and my girlfriend Sangeeta Bhattacharya for their tremendous support, and unconditional love. Without them, it would not have been possible for me to reach at this stage in my career.

Sailesh Kumar

Washington University in St. Louis
May 2008

Chapter 1

Introduction

The Internet has undergone profound transformation during the last decade. What started as a fairly simple data network used mostly by computer researchers, has become a global communications medium with mission-critical importance for the national and international economies and society. The number of Internet users is continuously growing and new Internet services are rapidly emerging, which are driving the Internet traffic volume to new levels. The growth in the Internet traffic is outpacing the rate at which hardware and memory technologies advance. New complications unanticipated by the original designers of the Internet are also arising in the form of erosion of trustworthiness, security threats, widespread use of mobility, etc, which often requires complex workarounds at both the protocol and the infrastructure level. These are introducing a variety of new challenges in the design and implementation of future networking equipment, which are already burdened with an increasingly large number of functions, and demanding performance pressures.

This dissertation addresses some of these concerns by proposing an array of novel algorithms and methods to efficiently realize two key network functions – *i*) IP packet forwarding, and *ii*) packet content inspection – which are both challenging to implement and critical to the functioning of the Internet. Our solutions concentrate on using embedded memory in innovative ways that involve a combination of architectural and algorithmic techniques and advance the state-of-the-art in both performance and efficiency. Considerable attention has been paid to the current levels of embedded memory density and hardware support, and their future trends, thereby enabling the proposed solutions to remain useful in the foreseeable future.

1.1 Internet – The Beginning

The Internet is a complex and vast networking infrastructure interconnecting millions of devices throughout the world, which provides services to numerous distributed applications. The roots of the Internet can be traced back to the development of packet switching [Baran et al. 1964] in the early 1960s. The communications infrastructures at that time were based primarily on circuit switching technologies. Unlike circuit switching, in which bits are the unit of information carriage and they are transmitted at constant intervals, packet switching employs packets of multiple bytes as the unit of information carriage. These packets are transferred between nodes over data links shared with other traffic. Due to the cross traffic, the packets may be buffered and queued in each node, which results in variable delay.

Packet switching appeared to be an efficient approach to handle traffic originating from bursty sources like applications running on an array of computers. The early work of Leonard Kleinrock [Kleinrock 1964] laid the mathematical foundation of packet networks and elegantly demonstrated the effectiveness of packet switching. Later, Paul Baran, and Donald Davies independently developed detailed concepts of multi-node packet switching networks, utilizing the ideas of Kleinrock and packet queuing. The work of Baran and the promise of packet switching helped influence ARPANET, the world's first operational packet switched network, to adopt the technology. In 1969, under the supervision of Kleinrock, the first packet switch, referred to as Interface Message Processors (IMPs), was installed at UCLA. Three additional IMPs were installed shortly thereafter and all four were interconnected to each other. By the end of 1969, these four nodes constituted the direct ancestor of the current Internet.

A number of applications were soon written for the ARPANET, including e-mail in 1971, file transfer in 1973, and voice traffic in 1973. These applications helped spur the

growth of the APRANET; there were 9 nodes by 1970 and the APRPANET extended from the west coast of the U.S. to the east coast. By 1973, there were 40 nodes, and the ARPANET included two satellite links, to Hawaii and Norway across the Pacific and Atlantic oceans, respectively. During the same time frame, an array of proprietary networks were developed, to interconnect the computer systems within a limited geographical region, *e.g.* ALOHANET to link universities in Hawaii, and IBM's SNA to connect the computing resources within a corporation.

As the number of such networks grew, it became important to interconnect them. Vinton Cerf and Robert Kahn in 1974 developed an interconnection protocol, thereby paving the way to the *network of networks* concept and introduced the term *internet*. Since different networks used a diverse variety of links and communication methods, the interconnection required that everyone agree on at least one common protocol to communicate with each other. Transmission Control Protocol (TCP) developed by Cerf and Kahn became this common language for communicating over the interconnection network (Internet), and it became necessary for every host which is part of the Internet to implement this protocol. TCP provides reliable in-sequence delivery of data with end system retransmissions, and the basic mechanism has hardly changed since its inception.

The introduction of TCP and the Internet Protocol (IP), which provides end-to-end packet delivery, the free and open access to its specifications and the early implementation helped the Internet to flourish. As the scale of the Internet increased, several major changes occurred to address the associated management issues. First, three network classes (class A, B and C) were introduced to accommodate the range of networks of different sizes and number of hosts. Second, the hosts were assigned names that were much easier to remember than the IP addresses. The Hierarchical Domain Name System (DNS) was subsequently developed to resolve these host names into Internet addresses and Classless Inter-Domain Routing (CIDR) was deployed to enable efficient utilization and aggregation of the IP addresses.

The second phase of the Internet growth, which was much more rapid, started with the emergence of the World Wide Web developed by Tim Berners-Lee at CERN. The development of the graphical browser by Marc Andreessen at NCSA gave a significant boost to the popularity of the World Wide Web and its easy-to-use interface made the Internet a viable communication medium for the general public. Several independent commercial networks were built and interconnected and it became possible to route traffic from one continent to another without passing through the government funded Internet backbone. An array of new Internet Service Providers (ISPs) arrived to provide access to the Internet from home, much like telephone connections. As the number of users started to grow at phenomenal rates, a number of services emerged, like web based email, online messaging, peer to peer file sharing. A number of new businesses provided these services, often for free, which further boosted the popularity of the Internet. Today more than one billion people regularly use the Internet, and conservative estimates suggest that there are more than thirty billion pages on the World Wide Web. Today, the economic impact of the Internet is unparalleled and it has become the growth engine of the world's economy. A study conducted in early 2000 [Barua, Whinston, and Yin 2000] estimated that the Internet economy generated more than $500 billion worldwide and created several million jobs, and it has been growing at phenomenal rates since then.

1.2 Internet – Current Infrastructure and Performance Challenges

The current Internet provides connectivity to the end hosts via access networks. An access network may be a wireless or wired local area network or a residential ISP reachable via DSL, cable modem, or a dial-up modem. These access networks are situated at the edge of the Internet infrastructure, which is organized as a tiered hierarchy of ISPs. Access networks are at the bottom of the hierarchy; at the top is a relatively small number of *tier-1 ISPs*. Tier-1 ISPs, often referred as the *Internet backbone*

networks, have direct connectivity to each other and an international presence. They also provide connectivity to a large number of tier-2 ISPs and other customer networks. Tier-2 ISPs usually have regional or national coverage, and they are connected to a small number of tier-1 ISPs. While many large enterprise networks are connected directly to the tier-1 or tier-2 ISPs, most of the end user Internet connectivity is provided by tier-2 or lower tier ISPs.

Since the links of Tier-1 ISPs carry the bulk of the Internet traffic, nodes of the tier-1 ISP network are built with the largest and the most capable routers, often called core routers. The current generation of core routers are capable of receiving, processing and transmitting traffic at hundreds of different interfaces simultaneously, with each interface operating in the range of 2.5 to 40 Gbps. The state-of-the-art Cisco Carrier Routing System (CRS-1) supports multi-chassis configurations that extend to more than 2000 interfaces, each running at 40 Gbps, thereby providing an aggregate bandwidth of 92 Tbps. In addition to these high performance core routers, a number of routing devices are used in the network, such as provider edge routers placed at the edge of an ISP network, inter-provider border routers which interconnect different ISPs, and access routers which are located at customer sites providing connectivity with the ISP.

The performance of a network device is limited by its slowest component; thus in order for the device to meet a given level of performance, it is critical that that all its functions meet the performance goal. For example, a 10 Gbps data throughput can be achieved only if each and every function in the data path runs at a 10 Gbps rate. (Notice that, the functions along the data path are those that process every packet or a large fraction of total packets.) The data paths of the current high performance routing devices include a large variety of functions, which range from routine tasks such as IP address lookup, packet buffering, header checksum verification, policing, and marking, to advanced functions such as packet classification, fair queuing, and traffic shaping. Recently, there is a growing demand that the networking devices also examine the content of data packets in addition to the structured information present in the packet header in order

to provide application-specific services such as application-level load-balancing and security-oriented filtering based on signatures. Forwarding packets based on both header and content is challenging and it is becoming increasingly challenging for the equipment vendors to implement such functions at high rates. In the future, as the best-effort Internet service model evolves and networks become more application aware, the difficulties are likely to exacerbate further. These rising difficulties can be attributed to a number of recent developments, which we can group into three main categories:

- *Growing traffic volume*: One of the primary sources of difficulty is the ever-increasing traffic volume. Internet traffic has grown at exponential rates in the past and shows no signs of decline. The growth in Internet penetration, the popularity of streaming video, and the arrival of high-definition media are likely to trigger unprecedented growth in traffic volumes in the near future, both within the network core and in the other parts such as network edge, and high user concentration sites (*e.g.* enterprise networks and metro area networks). As a wide range of high-bandwidth business and consumer services gain further momentum, ISPs will face a unique set of challenges to continuously upgrade their networking equipment to increase bandwidth and the higher number of interfaces.

- *Growing network functions*: The second primary cause of the challenges is the continuously increasing number of functions which are integrated into the devices. While a number of researchers have advocated to keep the network core simple [Stoica 2001], network equipment providers continue to see value in integrating more and more functions into core networking components. It is now common for routers to examine the layer-4 and higher layer's packet header as well as the packet content. In the future, routers are expected to thoroughly examine every portion of a packet before forwarding it, thereby adding substantial computation overhead.

- *Increasing complexity*: The last concern is the ever-increasing complexity of the functions that are incorporated into various devices; the increased complexity is due to two key factors. The first is the growing number of end hosts and intermediate nodes in the network, which leads to a higher number of address identifiers in use, adding complexity to the functions such as IP address lookup. The growing number of end-to-end flows also directly impacts the per-flow based functions such as stateful packet content inspection and fair queuing. The second factor is the constant upgrades and refinement of these functions, such as an increasing number of rules used to classify the packets to enable more fine-grained control of traffic, or rising number of virus signatures to combat increasing security threats. A higher number of classification rules, IP addresses or virus signatures requires larger amounts of memory and often also deteriorate the performance by requiring more computation and memory bandwidth.

While advancements in fiber optics and signal transmission technologies such a Dense Wavelength Division Multiplexing (DWDM) today enable up to terabits/sec bandwidth over a single fiber, silicon hardware has been unable to keep up. The advances in semiconductor and systems technology are not solely sufficient to combat the above three trends and design the next generation networking equipments which are capable of providing the required levels of performance. There is a pressing need of architectural enhancements and new innovative algorithms that can efficiently implement the existing and newly introduced network functions, in order for the Internet to continue to evolve.

There is another dimension, which arises due to the rapidly changing implementation platforms and evolution of new ones such as network processors. Early networking equipments used general purpose processors to implement most packet forwarding functions. As the link speeds have grown, it became necessary for high performance systems to employ ASICs to perform the key functions. ASIC solutions provide high performance, but they are generally difficult to update and reprogram. Network

processors, on the other hand, are software-programmable devices whose feature sets are specifically targeted for networking applications, and they provide a much greater degree of flexibility and programmability in implementing various network functions. Consequently, network processors are a desirable platform for implementing those services that are updated or upgraded frequently, while ASICs are preferable for implementing those that have been standardized and are unlikely to change, and which require significant amounts of computation.

Both ASICs and network processors today are capable of integrating a variety of external and embedded memories of various capacities, running at different clock rates. Modern VLSI technology supports integration of more than a billion transistors [Burger, and Goodman 1997] in a single chip. It is now possible to support several large embedded memories on a single die; for example, IBM's ASIC fabrication technology [IBM 2005] can integrate up to 300 Mbits of embedded memory on one chip. In the future, the available computing power and the quantity of embedded memories are likely to increase further. Network processors will integrate higher numbers of independent processing units, specialized accelerators, and on-chip memory modules. ASICs will become denser, capable of packing much faster and many more transistors. The increased computational capabilities and the higher density and diversity in the memory subsystem will offer new levels of challenges in efficiently utilizing them and open up unparalleled opportunities for enhancing the overall performance.

To summarize, as the Internet continues to evolve, it is becoming increasingly difficult to implement high performance network devices due to the rapidly expanding feature sets and the pressing need to keep up with the increasing traffic volume. While the modern implementation platforms such as network processors and dense ASICs are capable of integrating abundant computing resources and on-chip memory, proper utilization and management of these parallel resources remains a challenging problem. An effective realization of network functions on these increasingly capable platforms

therefore will require a variety of techniques at both the architectural and the algorithmic level, which makes it an interesting research problem.

1.3 Dissertation Focus – Main Contributions

In this dissertation, we concentrate on two important classes of network functions. The first class consists of *IP packet forwarding* functions, such as header lookup and packet classification – the former determining the next hop for the packet and the latter prioritizing the packets. Packet header lookup is an integral component of every routing system; on the other hand, packet classification is frequently used in high performance systems to enable Differentiated Services and Quality of Service (QoS). The second class of functions we focus on is *packet content inspection*, which involves examination of the entire packet payload and matching it against a predefined set of patterns. Packet content inspection has recently experienced a rapid adoption in the emerging application layer packet forwarding applications and intrusion detection systems.

Due to the importance and broad deployment of the aforementioned two classes of network functions, a collection of novel methods has been developed to efficiently implement them. In this dissertation, we comprehensively evaluate the existing state-of-the-art methods and develop novel solutions, which improve upon them both in theory as well as in real implementation contexts. We introduce our solutions and the main contributions in the following sections.

1.3.1 IP Packet Forwarding

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, which is often called *IP address 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 central component of both functions consists of determining the longest prefix matching the header fields within a database of variable length prefixes.

In this dissertation, we introduce two techniques to enable high performance longest prefix match operations. These methods use a trie data-structure which is a power efficient approach to implement longest prefix match. Our first contribution is a novel representation of tries, called *History-based Encoding, eXecution and Addressing (HEXA)* that uses implicit information present in a trie structure to locate the successor trie nodes, thereby significantly reducing the amount of information that must be stored explicitly. The key observation in HEXA is that for any given node of a trie there is only one path that leads to it, and this path is labeled by a unique string of symbols. Since the algorithms that traverse the trie know the symbols that have been used to reach a node, we can use this "history" to define the storage location of the node. Since no nodes have identical histories, each node can be mapped to a distinct storage location by hashing its history value; provided that we have a perfect hash function. In practice, however, devising a perfect hash function is difficult; HEXA therefore employs a few discriminating bits for every node, which are hashed along with their history values. Since the discriminating bits can be altered, it provides multiple choices of storage locations for a node. We find that the amount of discriminating information needed to enable a perfect hashing is just two bits, which leads to a binary trie representation that requires just two bytes per stored prefix for IPv4 routing tables with more than 100K prefixes, a 2-fold memory reduction compared to the state-of-the-art representations.

The resulting memory compactness leads to higher performance in implementation contexts where there are small on-chip memories with ample memory bandwidth and larger off-chip memories with more limited bandwidth. These characteristics are

common in conventional implementation platforms such as general purpose processors, network processors, ASICs and FPGAs.

Our second contribution is a novel embedded memory based pipelined trie, which delivers high lookup and update throughput. The proposed pipelined trie called *Circular Adaptive and Monotonic Pipeline (CAMP)* is different from a regular pipeline 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, when compared to a linear pipeline with static entry and exit stages, this multi-access and circular pipeline structure enables much more flexibility in mapping trie nodes to pipeline stages.

CAMP exploits this flexibility in mapping algorithm by applying controlled prefix expansion on the first few levels of the trie to obtain a modest number of sub-tries. For example, an expansion of all IPv4 prefixes which are shorter than 8 bits to 8 bits would yield at most 256 sub-tries with maximum height of 24. Each of these sub-tries is mapped to our circular pipeline by first choosing a memory stage for the root of the sub-trie, and then assigning subsequent levels of the trie to subsequent stages in the pipeline, including a wrap around. By choosing mappings for all sub-tries judiciously—and fairly simple heuristics are effective—the system can maintain uniform and near-optimal memory utilization with high probability. Moreover, this balance is preserved after an extended period of inserts and deletes. Thus, the system eliminates the deficiencies of previous approaches, which were based on linear pipeline and static height- or level-based node mapping [Hasan, and Vijaykumar 2005] [Basu, and Narlikar 2003] which often led to under-utilized memory. For real routing tables storing 100K prefixes, our approach can achieve 40Gbps throughput with a power consumption of 0.3 Watts. Projections on 250 thousand prefix tables show a power consumption of 0.4 Watts at the same throughput.

To summarize, we propose two novel embedded memory based architectures to realize longest prefix match operations. The solutions called HEXA and CAMP are orthogonal

and together they can enable *i*) high performance by exploiting the abundant memory bandwidth available on-chip, and *ii*) efficiency by economically using the scarce on-chip memory resources.

1.3.2 Packet Content Inspection

Today, many critical network services handle packets based on payload. Traditionally, this packet content inspection has been limited to comparing packet content to sets of strings. State-of-the-art systems, however, are replacing string sets with regular expressions, due to their increased expressiveness. Several content inspection engines have recently migrated to regular expressions, including: Snort, Bro, 3Com's TippingPoint X505, and various network security appliances from Cisco Systems. While flexible and expressive, regular expressions have traditionally required substantial amounts of memory, which severely limits performance in the networking context. In this dissertation, we introduce an array of novel techniques to efficiently implement regular expressions in networking. Our techniques are based upon innovative machines that are capable of recognizing regular expressions languages.

Our first contribution is a representation of Deterministic Finite Automaton (DFA) that substantially reduces the number of transitions associated with each state, thereby reducing the memory and enabling a high performance embedded implementation. The key observation is that groups of states in a DFA often have very similar outgoing transitions and we can use this duplicate information to reduce memory requirements. For example, suppose there are two states s_1 and s_2 that make transitions to the same set of states, $\{S\}$, for some set of input characters, $\{C\}$. We can eliminate these transitions from one state, say s_1, by introducing a default transition from s_1 to s_2 that is followed for all the characters in $\{C\}$. Essentially, s_1 now only maintains unique next states for those transitions not common to s_1 and s_2 and uses the default transition to s_2 for the common transitions. We refer to a DFA augmented with such default transitions as a

Delayed Input DFA (D^2FA). In practice, the proper and effective construction of the default transitions leads to a tradeoff between the size of the DFA representation and the memory bandwidth required to traverse it. In a standard DFA, each input character leads to a single transition between states; in a D^2FA, an input character can lead to multiple default transitions before it is consumed along a normal transition.

The approach achieves a compression ratio of more than 95% on typical sets of regular expressions used in networking applications. Although each input character potentially requires multiple memory accesses, the high compression ratio enables the data-structure to be stored in on-chip memory modules, where the increased bandwidth can be provided efficiently. To explore the feasibility of this approach, we describe a single-chip architecture that employs a modest of amount of on-chip memory, organized in multiple independent modules in order to provide ample bandwidth. However, in order to deterministically traverse the automata at high rates, it is important that the memory modules are uniformly populated and accessed over any short period of time. To this end, we develop load balancing algorithms that map our compressed automata to the memory modules in such a way that deterministic worst-case performance can be guaranteed. Via experiments, we demonstrate that our algorithms can provide high parsing throughput while simultaneously traversing thousands of automata.

Our second contribution is Content Addressed D^2FA (CD^2FA), which builds upon a D^2FA. CD^2FAs replace state identifiers of a D^2FA with content labels that include part of the information that would normally be stored in the table entry for the state. This makes selected information available earlier in the state traversal process, which makes it possible to avoid unnecessary memory accesses. Specifically, a CD^2FA requires a single memory access before consuming any given input character, thereby matching the performance of an uncompressed DFA, while simultaneously keeping a small number of transitions per state, thereby enabling a compact memory footprint. CD^2FAs employ a perfect hashing technique to map the content labels to memory locations; thus the content labels are directly used to locate the table entry for the next state labels. This

avoids overheads such as explicitly storing the characters and state labels as hash keys, leading to additional memory reduction.

Our third set of solutions to implement regular expressions is specific to network intrusion detection systems. A unique characteristic of these security systems is that the packet contents of normal traffic rarely match more than the first few characters of the patterns. Traditional patterns matchers however employ the entire signatures to construct the DFA, which creates DFA that is so large that off-chip memories are required to store it. This approach is wasteful; rather, the tail portions of the signatures can be isolated, and represented by a compact but slow structure such as an NFA. For normal traffic, the slow path will remain asleep, and activated during those anomalous situations when the packet content begins to match the entire signature. We introduce such a packet processing architecture which we call *bifurcated packet processing*, which splits the signatures into prefixes and suffixes. The splits are computed such that normal data streams will rarely match an entire prefix. Subsequently, the packet processing is divided into two components: fast path and slow path. The fast path parses the entire content of each flow and matches them against the prefixes of all signatures. The slow path parses only those flows which have found a match in the fast path, and matches them against those suffixes, whose corresponding prefixes are matched.

Such a splitting into fast and slow path can enable high speed parsing economically. Signatures used in the fast path are small, thus they can be represented with fast structures such as a single composite DFA. Such a composite DFA would otherwise explode in size and become impractical if the entire signatures were used. The slow path signatures, on the other hand, will parse a small fraction of traffic, thus they can be implemented with slow but compact structures like an NFA. Based upon experiments carried out on real signatures drawn from a collection of networking systems, bifurcated packet processing can reduce the memory requirements by up to 100 times, while simultaneously enabling a two to three fold increase in the packet throughput.

To complement the bifurcated architecture and enable a high performance fast path implementation, we describe a new representation of regular expressions called History based Finite Automaton (*H-FA*), which consists of an automaton augmented with a history buffer. The contents of the history are read and updated during execution. Like in a NFA, the transition function in a H-FA may return multiple next states for certain states and input characters; however, only one next state from this set is chosen, which is decided by the history buffer; thus the transitions in a H-FA are conditional upon the state of the history buffer. Only one state is active at any time in a H-FA which enables it to yield a throughput equal to that of a DFA; besides, an appropriate construction of H-FA results in a dramatic reduction in the number of states over a DFA. We also describe a variant of H-FA called counting H-FA (H-cFA), which addresses the inability of table driven finite automata implementations to efficiently handle length restrictions specified for certain sub-expressions. A number of security signatures consist of length restrictions, thus H-cFA results in dramatic memory reductions in their implementation.

To summarize, this dissertation makes several key contributions in the area of packet content inspection. First, it re-iterates the notion, albeit much more quantitatively, that the central issue in regular expressions implementations is the trade-off between space and time. At one end, DFA based techniques enable a single state of execution, but require prohibitive amounts of memory. At the other end, NFAs are compact but require multiple active states. To enable high performance, a small number of active states is desirable, thus DFAs are preferable. Small memories however clock at higher rates, which brings another dimension to the tradeoff: multiple active states can be acceptable if it reduces the memory significantly. A number of new representations are introduced that fall between the NFA and DFA on the tradeoff-curve, and take advantage of the higher clock rates provided by small memories. D^2FAs enable up to 100-fold memory reduction over DFA and enables higher parse rates by employing parallel embedded memories. CD^2FAs are relatively less compact, 50-fold over DFA; however they enable parsing rates that surpass that of a DFA even without using multiple memories. A bifurcated packet content inspection architecture coupled with H-

FA based machine has also been proposed that enable NIDS to attain high parsing performance at a much reduced implementation cost.

1.4 Evaluation

Our evaluation methodology consists of three components: the performance metrics that we use to characterize a solution, the workloads that are used to evaluate it, and the implementation platforms we consider. Below, we discuss each of these.

1.4.1 Performance Metrics

We use the following four key performance metrics to evaluate the solutions proposed in this dissertation.

- *Efficiency*: We define efficiency as the inverse of the short- and long-term cost of the resources required to implement a function. For example, if the task is to implement IP lookup for 100,000 prefixes at OC192 rate, then the efficiency would depend upon both the short-term cost such as cost of memory and logic, and the long-term cost such as power dissipation and area required on the board. Efficiency is critically important for those networking systems that are deployed in high volumes, in order for the equipment vendors to keep the prices low, compete in the marketplace and generate profit.
- *Raw Performance*: We define raw performance as the peak packet rate that can be sustained with the available technology. When performance varies for varying inputs, we report both the average- and the worst-case results while discussing the likelihood of the worst-case scenarios. Solutions with high raw performance are important to implement state-of-the-art systems that need to forward traffic at the highest possible rates, and where cost is a secondary concern.

- *Scalability*: We define scalability as how well the solution performs when the size of the problem grows. For instance, in an IP lookup engine, scalability is the rate at which the system cost increases as we start increasing the number of prefixes and/or packet rate. Scalability is a decisive factor in the widespread deployment of any solution; partly because high performance networking devices usually have unusually long shelf life spanning decades, and partly to keep up with the continuously increasing traffic volumes and the expanding networks.
- *Vulnerability*: We define vulnerability of a system as the possibility of a dramatic degradation in the performance due to anomalous circumstances that either arise during the normal course of network operations or as the result of a deliberate attack. For example, in the case of a network intrusion detection system, such situations may arise if the contents of several flows frequently match one or many signatures, thereby raising security alarms "too often". In the current Internet, where users can no longer be trusted, it is important that the deployed solutions are capable of handling anomalous conditions.

1.4.2 Workloads

With the above four primary performance metrics, we use a combination of real-world, and synthetically generated workloads. Real-world workloads such as IP prefix tables, data traces, and packet header logs collected from various routing systems, are essential in characterizing the performance of a solution during normal traffic conditions. Efficiency and raw performance of a system are the two performance metrics that are evaluated solely with real-world workloads. A mixture of real-world and synthetically generated workloads, extrapolated from the real-world workloads, are used to evaluate the scalability of a solution.

Synthetically generated workloads, on the other hand, are essential in evaluating how a system will behave during extreme conditions that may not arise frequently, such as

abnormally high routing table update traffic during a link failure and/or congested link. Synthetic workloads are also used to characterize security threats such as Denial of Service (DoS) attacks which exploit a weakness in the architecture. An example of such a workload is a carefully crafted packet stream that falsely raises the security alarm in a NIDS, or which leads to a false positive rate much higher than the normal theoretical rate in a Bloom filter based system. Such anomalous workloads are required to characterize the system performance with respect to the vulnerability performance metric. For every solution we propose in this dissertation, in the respective chapters, we explain a number of attacks and anomalous conditions and describe the synthetic workloads that are used to evaluate the mechanisms we propose to safeguard against these attacks.

1.4.3 Implementation Platforms

The final component of our evaluation consists of the implementation platforms that we consider and the settings that we use in our evaluation. Our primary implementation platform remains configurable ASIC architectures, customized to perform a given networking function. The primary motivation behind focusing solely on such specialized platforms is to be able to maintain high data rates. The settings such as clock frequency, transistor density, memory access latency and memory bandwidth that we choose to evaluate our solutions remain inline with the current technology trends.

Programmability is essential in order to keep an implementation up-to-date in the face of changing workloads and continuously evolving functions; therefore our main focus is to keep our solutions highly programmable in spite of taking an ASIC approach. There are two mechanisms to achieve a high degree of programmability in this setting. First, one can employ a collection of programmable processors whose instruction sets are optimized to efficiently carry out the operations required by the function. Such network processor oriented mechanism will lead to very high degree of programmability but may

limit the performance by limiting the computation parallelism, which will be equal to the total number of processor cores. An alternative approach partitions the problem into two components, a *static computation part* and a set of *dynamic states*. The static portion will consist of all such operations that will not change in the future, and thus can be realized using logic gates. The dynamic portion will include a number of states that may change when the function is updated; thus these states are stored in memory. To clarify, let us consider a simple example. A finite automaton can be realized entirely in circuit by using logic gates to implement all its transitions; however, such solutions are undesirable if the automaton is frequently updated. A more programmable approach is to partition the problem into a transition table stored in embedded memory, and a set of circuitry that reads the transitions and traverse through the states. We focus on such hybrid circuit and memory based solutions.

1.5 Organization

In the next chapter, we discuss the background and related work on IP address lookup and deep packet inspection. A number of well known and deployed algorithms are discussed and their main advantages and disadvantages are pointed. We also set the stage for a fair comparison between these and our proposed methods.

In Chapter 3, two algorithms and the associated architectures that target an ASIC implementation are proposed. It has been argued with quantitative data points that these solutions can enable current Internet routers to forward IP packet at high speeds.

In Chapter 4 and 5, we propose methods to implement regular expressions based deep packet inspection at high speeds. Our implementation platforms are ASIC and network processors, respectively.

Chapter 6 summarizes out contributions.

Chapter 2

Background and Related Work

An essential function of a network router is to examine the IP headers of packets arriving at various input ports and forward them to appropriate output ports. These forwarding decisions require a lookup in a routing table that consists of a large number of variable length IP address prefixes and their associated destination ports. This function, often referred as *IP address lookup*, determines the longest prefix matching the destination address field within the routing database, and then forwards the packet to the destination port associated with the matching prefix. In addition to basic packet forwarding, modern systems are increasingly identifying different classes of traffic and providing them different levels of service. The allocation of packets into different classes of end-to-end flows requires examination of multiple fields of the packet header, as flow classes are identified by five tuples of the IP header: a pair of source and destination addresses, a protocol, and a pair of source and destination ports. In practical sets, a large fraction of the rules include prefixes. While port numbers are most often specified as ranges, they can also be represented as a set of prefixes. Thus the process of classification of packets based upon five tuples translates into several longest prefix matches and a number of high performance packet classification algorithms such as [Gupta, and McKeown 1999] and [Baboescu, Singh, and Varghese 2003] employ some form of longest prefix match.

Due to the widespread use of longest prefix match, both in packet forwarding and in packet classification, it has been studied extensively. Some well known hardware approaches include the use of TCAM, Bloom filters and hash tables. Another class of

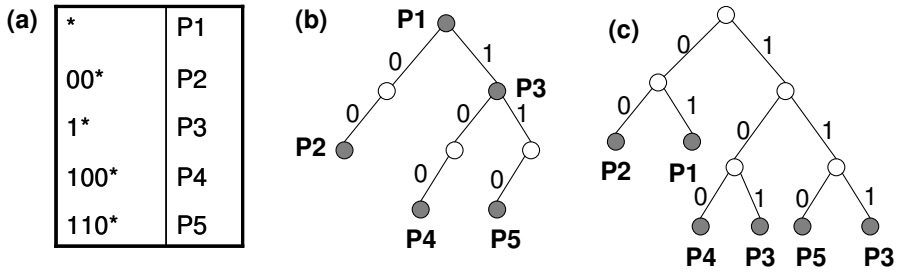


Figure 2.1 (a) Routing table; (b) corresponding unibit trie; (c) corresponding leaf-pushed unibit trie.

solutions employs a trie to implement the longest prefix match operations. We begin our discussion with these trie based methods.

2.1 Trie based Longest Prefix Match

A trie is an ordered tree data structure associating a string s_x to each node n_x; s_x is not explicitly stored at the tree, but can be derived by concatenating the symbols labeling the edges on the path from the root of the trie to the node n_x. A basic property of tries is that all descendants of a node n_x share a common prefix, represented by the string associated with n_x. In the context of IP address lookup, a binary trie representing a routing table can be built by traversing each prefix from the leftmost to the rightmost bit, and inserting nodes into the trie as needed, a left child for each 0 and a right child for each 1. For an example, see Figure 2.1 (a) and (b). Nodes corresponding to valid prefixes must be marked with a prefix pointer that gives the location of the next hop info. Lookup is performed by traversing the trie according to the bits in the IP address. When a leaf or a node with no matching outgoing edge is reached, the last marked node traversed corresponds to the longest matching prefix.

As illustrated in Figure 2.1 (b), each node contains two child pointers and one prefix pointer. To reduce memory usage, leaf pushing (Figure 2.1 (c)) has been proposed [Waldvogel et al. 1997], wherein prefixes at non-leaf nodes (*e.g.*: P1, P3) are pushed down to the leaves. Leaf pushing makes every node have two children or none; thus,

(a)

*	P1
00*	P2
10*	P3
11*	P3
1000*	P4
1001*	P4
1100*	P5
1101*	P5

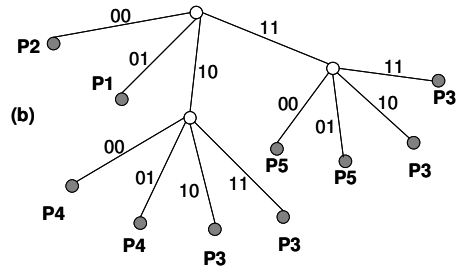


Figure 2.2 (a) Routing table expanded with stride 2; (b) corresponding multibit trie.

each node requires a single pointer to locate the prefix or the array of children. However, leaf-pushed nodes may need to be replicated at several leaves (*e.g.*: P3); therefore on average, leaf pushing results in less than a 2-fold memory reduction. Moreover, it also complicates updates.

If several bits are scanned for each node traversal, then the resulting data structure is a multibit trie. The number of bits scanned at once is called the stride. A node with stride *d* will have a maximum of 2^d child nodes. In a multi-bit trie, some prefixes may be expanded to align to the stride boundaries, which may increase the size of the routing table, as illustrated in Figure 2.2. However, during a node traversal, multiple bits are scanned at once, which reduces the number of steps. Since the time to complete a lookup is determined by the trie depth, the choice of stride depends upon the lookup time-memory tradeoff: smaller strides allow a more compact data structure but require more memory accesses, whereas larger strides reduces the lookup time at the cost of more memory.

Controlled prefix expansion has been introduced in order to address the above issue [Srinivasan, and Varghese 1999]. Given the maximum number of memory accesses allowed for a lookup (*i.e.*: trie depth), this technique uses dynamic programming to determine the stride leading to the minimum total memory. However, this involves two important limitations: first, it is suitable for building a trie from scratch but does not support incremental updates; second, while reducing the total memory, this technique

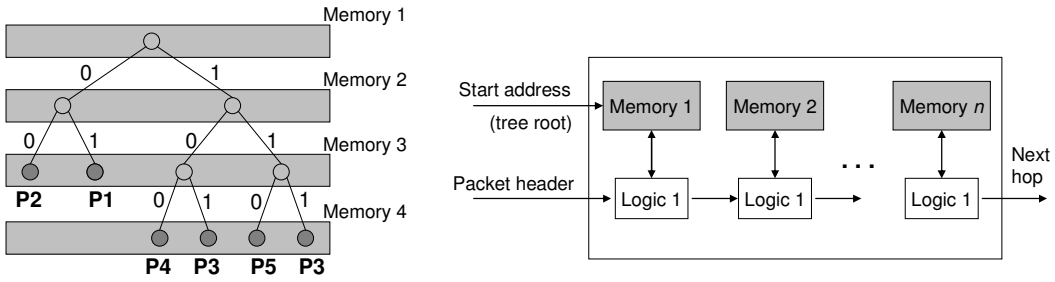


Figure 2.3 (a) Pipelining a lookup trie, each level forms a stage; (b) Corresponding pipelined lookup architecture.

does not control the per level memory occupancy in a pipelined trie. The reason for this will be explained shortly.

2.1.1 Pipelined IP Lookup Tries

An effective way to tackle the time-memory tradeoff is to recognize that tries are well suited to data structure pipelining. A common way to pipeline a trie is to assign each trie level to a different stage, as illustrated in Figure 2.3, so that a lookup request can be issued every cycle, thereby increasing the throughput. Besides increased lookup performance, such pipelined implementations are also suitable for handling updates. In fact, as proposed in [Basu, and Narlikar 2003], software preprocessing of prefix insertions and deletions can be exploited in order to determine the necessary per-level modifications to be performed in the trie. In a second phase, those write operations can be inserted in the pipeline in the form of "write bubbles". Because of the sequential character of the pipeline operation, straightforward techniques can prevent write operations from interfering with the concurrent lookups.

In a pipelined implementation, it is desirable for nodes to be distributed uniformly across pipeline stages. [Basu, and Narlikar 2003] applies an extended version of controlled prefix expansion to achieve this objective. Rather than minimizing the total memory, the modified algorithm aims at minimizing the size of the largest trie level, while still keeping the total memory low. Through the use of variable-stride tries (having

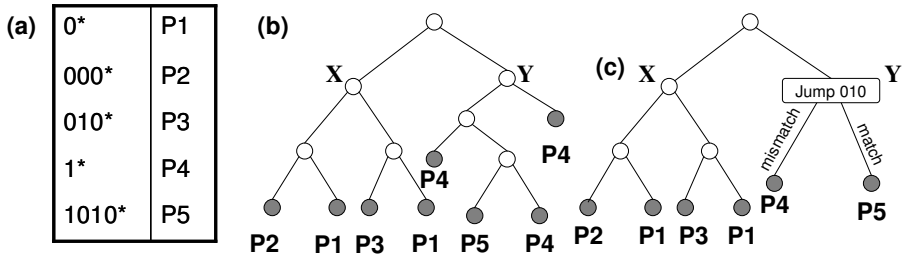


Figure 2.4 (a) Routing table; (b) corresponding unibit leaf pushed trie; (c) unibit trie with jump nodes.

a fixed per level stride but allowing different strides at different levels), it achieves a discretely balanced prefix distribution across pipeline stages.

An alternative approach is presented in [Hasan, and Vijaykumar 2005], where height-based (rather than level-based) pipelining is proposed. The work does not aim at balancing memory utilization; rather, it focuses on guaranteeing worst case performance bounds. In particular, it focuses on leaf-pushed unibit tries using a technique called jump nodes, which limits the number of copies of a leaf-pushed node. The usage of jump nodes is exemplified in Figure 2.4, where all descendants of node Y represent either the prefix P4 (leaf-pushed) or P5. Clearly, all internal nodes in the subtree rooted at Y can be condensed into a jump node carrying the information about the remaining portion of P5. In [Hasan, and Vijaykumar 2005], the authors argue that jump nodes ensure that the number of leaves in a leaf pushed unibit trie is equal to the number of prefixes, which enables $O(1)$ updates. Unfortunately, since not all the copies of leaf-pushed nodes can be removed by using jump nodes (see P1 in Figure 2.4), such claims are incorrect. Moreover, height-based pipelining leads to unbalanced stages; as a workaround, hardware-based pipelining has been proposed, which, adds to the complexity and power consumption.

The most recent and the most efficient pipelined trie has been proposed in [Baboescu et al. 2005], which uses a circular pipeline with dynamic pipeline entry points. It has been shown that such a circular pipeline can enable uniform utilization of the memory available at each stage irrespective of the trie shape, thereby enabling much better

memory utilization. Uniform pipeline stages also avoid any single stage to requiring excessive amounts of memory and hence becoming a performance bottleneck.

2.1.2 Efficient Encoding of Multibit-Trie Nodes

The last relevant aspect studied in the trie literature is the use of compression to reduce memory requirements. Memory compression is achieved by representing a number of nearby nodes of the trie with a bit-map. In particular, the Lulea scheme [Degermark et al. 1997] uses leaf pushing and controlled prefix expansion along with an appropriate stride length, say k, to create a multi-bit expanded trie. Each multi-bit node of this trie requires 2^k words, each representing either a matching prefix or the next node pointer. Since a large number of these 2^k words may be identical due to the use of controlled prefix expansion, Lulea employs a single bit-mask to eliminate such repetitions of words, thereby significantly reducing the total memory. Due to the use of leaf pushing, Lulea however does not exhibit good incremental update properties. Tree Bitmap algorithm [Eatherton, Dittia, and Varghese 2004] on the other hand focuses on non-leaf-pushed multi-bit tries, and uses two separate bit-masks, one to represent the destination ports associated with the prefixes within the multi-bit node, and another to represent the pointers to the sub-tries. Therefore, Tree Bitmap allows $O(1)$ updates unlike Lulea which may require that the entire memory structure be modified, while requiring comparable amounts of memory.

Shape shifting tries (SST) [Song, Turner, and Lockwood 2005] has been proposed as an alternative solution that adapts its node encodings according to the shape of the trie, thereby leading to further memory compression. The core idea is that when a trie is sparse (such as an IPv6 trie), then using the traditional multi-bit representation, where a multi-bit node represents a sub-tree of fixed shape (binary tree with k levels), may lead to wasted space. A shape shifting trie allows its multi-bit nodes to correspond to arbitrarily shaped sub-trees, thereby enabling the sub-trees within the underlying binary

trie to conform to the structure of the trie, and significantly reducing the number of SST nodes and the total memory. Notice, however that SST has an overhead that every node must store a few bits, in addition to the bit-masks, specifying its shape; therefore, it is important in SST to keep the total number of currently used sub-tree shapes to a small value. Two classes of algorithms have been proposed by the authors for the appropriate construction of SST trie. The first algorithm focuses on achieving substantial levels of memory compression by limiting the number of SST nodes and the number of distinct shapes that are used, while the second focuses on significantly reducing the total number of SST node that are traversed to perform a lookup, thereby enabling higher lookup throughput.

2.2 Non-trie based Longest Prefix Match

A number of alternative architectures have been proposed, which avoid using a trie data-structure to perform longest prefix matching. The *Multiway and Multicolumn Search* technique [Lampson, Srinivasan, and Varghese 1999] require $O(W+\log N)$ time and $O(2N)$ memory, where W is the number of bits in the address, and N is the total number of prefixes. The scheme involves a basic binary search for the longest matching prefix problem, an efficient implementation of which requires two techniques: encoding a prefix as the start and end of a range, and pre-computing the best-matching prefix associated with a range. The paper also proposes a number of optimizations for cache based implementations, such as a multi-way search technique, which exploits the fact that most processors prefetch an entire cache line when doing a memory access. These techniques along with the use of an initial precomputed array to lookup the first 16-bits of the address results in a total of 9 memory accesses to perform longest prefix match in the worst case. The primary issue with this algorithm, however, is its linear scaling relative to address length, thus such schemes are not attractive to implement IPv6 lookups and packet classification.

Another computationally efficient algorithm called *Binary Search on Prefix Lengths* has been introduced in [Waldvogel et al. 1997]. The main contribution of this work is to use significant precomputation of the lookup database to bound the number of memory accesses required during the lookup. The algorithm first sorts the prefixes into up to a maximum of W sets based on their length; each set is intended to be examined separately to find the best matching prefix. In order to enable fast examination, a hash table is used for each set; the authors made an assumption that the examination of a set will require a single hash probe. We will later talk why such assumption can lead to low worst case performance, and high performance hash tables are required in order to enable guaranteed throughput [Song, Dharmapurikar, Turner, and Lockwood 2005] [Kumar, and Crowley 2006]. The simpler scheme uses binary search to choose the sequence of sets to probe, beginning with the median length set. Thus if an IPv4 lookup table contains prefixes of all lengths, the search will begin with the probe of the set containing the length 16 prefixes. Markers are placed along the binary search path for prefixes that are of longer lengths, in order to direct the search to the appropriate set. If no matching prefix or marker is present then the search will continue at the shorter set along the binary search path. There is a potential problem of backtracking, *i.e.* for a given IP address, if there is no longer matching prefix in the table then the search may unnecessarily follow a marker. In order to prevent this, the best-matching prefix for the marker is computed and stored with the marker. If a search terminates without finding a match, the best-matching prefix stored with the most recent marker is used to make the routing decision. The authors also propose methods of optimizing the data structure to the statistical characteristics of the database. Empirical measurements using an IPv4 database resulted in memory requirements of about 42 bytes per entry.

A variant of this scheme [Dharmapurikar, Krishnamurthy, and Taylor 2003], efficiently narrows the scope of the search by using compact but probabilistic Bloom filters. There are W Bloom filters, one for each prefix length. While the hash tables built from the prefixes are stored in off-chip memory, their associated Bloom filters are stored in an on-chip memory, which has ample bandwidth and low access latency, and therefore can

be probed in parallel. Before accessing the hash tables, the Bloom filters are examined, which probabilistically directs the search to the appropriate hash table within constant time. In order to optimize average performance, the authors introduce asymmetric Bloom filters which allocate memory resources according to prefix distribution and provide viable means for their implementation. Via the use of a direct lookup array and use of Controlled Prefix Expansion (CPE), it has been shown that the worst case performance can be limited to two hash probes and one array access per lookup.

The last but the most frequently used hardware based architecture for longest prefix matching is Ternary Content Addressable Memory (TCAM). A CAM is an associative memory array containing data words. When a user supplies a word to the CAM, it searches its entire memory array to see if that data word is stored anywhere in it. If found, the CAM returns the first location where the word is present along with optional information called a tag. A TCAM is an extension, which allows its data bits to be "don't care" in addition to being 1, or 0. This adds tremendous flexibility in search; for example, a ternary CAM might have a stored word of "1XX0" which will match any of the four search words "1000", "1010", "1100", or "1110". Clearly, TCAMs can be used to search longest matching prefixes, if the prefixes are sorted by their length, beginning with the longer prefixes, and the tag stores the next hop information [McAuley, and Francis 1993]. Due to hardware implementation and optimizations at the transistor level, TCAMs enable longest prefix matching at very high rates. However they consume a significant amount of power, because a search requires a probe into every data word of the memory. It also becomes problematic for the TCAM to ensure fast updates to the prefix database, due to the requirement that the prefixes must remain sorted. A number of papers have been published both in the area of power-efficient implementation of TCAM [Zane, Narlikar, and Basu 2003] and the effective use of TCAM to enable fast incremental updates. Power efficient TCAM architectures usually partition the memory array into smaller segments, thus selectively addressing smaller portions of the TCAM at a time [Spitznagel, Taylor, and Turner 2003]. A number of

schemes have also been introduced to enable fast updates to the prefix database such as [Shah, and Gupta 2001] [Song, and Turner, 2006].

2.3 Packet Content Inspection

Modern systems are expected to examine the packet content in addition to the header in order to make forwarding decisions. Packet content inspection is gaining popularity as it provides capability 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" [Shafer, Jones 2005]. 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, patterns have been specified as exact match strings. Naturally, due to their wide adoption and importance, several high speed and efficient string matching algorithms have been proposed recently. These often employ variants of standard string matching algorithms such as Aho-Corasick [Aho, and Corasick 1975], Commentz-Walter [Commentz, and Walter 1979], and Wu-Manber [Wu, and Manber 1994], and use a preprocessed data-structure

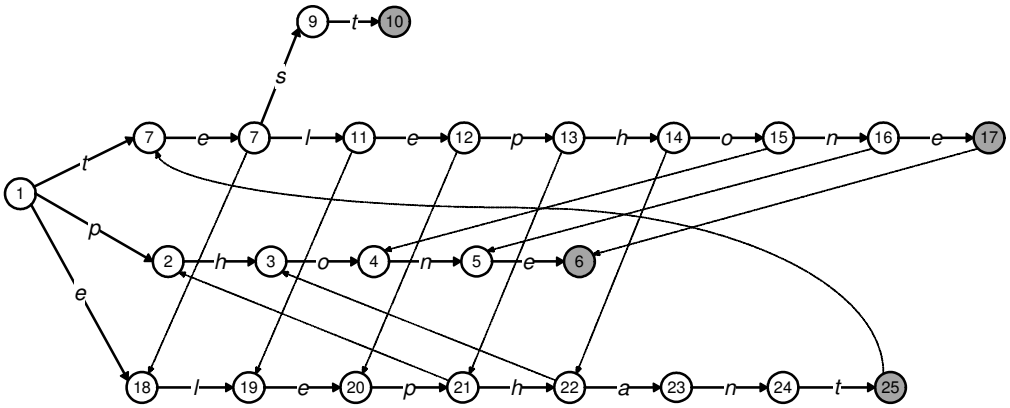


Figure 2.5 Aho-Corasick automaton for the four strings test, telephone, phone and elephant. Gray indicates accepting node. Dotted lines are failure transitions.

to perform high-performance matching. Among these, Aho-Corasick has been adopted most widely and is relevant to our work.

2.3.1 Aho-Corasick Algorithm based String Matching

One of the earliest, efficient algorithms for exact multi-pattern string matching is due to Aho-Corasick [Aho, and Corasick 1975]. The algorithm enables string matching in time linear in the size of the input. Aho-Corasick builds a finite automaton from the strings, whose structure is similar to a trie, and encodes all the strings to be searched in multiple stages. The construction begins with an empty root node which represents no partial match; subsequently nodes are added for each character of the pattern to be matched, starting at the root node and going to the end of the pattern. Strings that share a common prefix also share a corresponding set of ancestor nodes in the trie. Beyond this, there are two variants of Aho-Corasick: deterministic and non-deterministic. In the non-deterministic version, the state machine trie is traversed beginning at the root node and failure pointers are added from each node to the longest prefix of that node that also leads to a valid node in the trie. Figure 2.5, illustrates a simple example. There are four strings: `phone`, `telephone`, `test`, and `elephant`. The automaton consists of 25 nodes in total. The bold transitions are normal ones, while the dotted ones are failure transitions. The operation of this implementation is straightforward. For any given input

character in any given state, the character is consumed if a normal transition for the character is present at the state; else the failure transition is taken. Due to the construction, whenever a failure transition is taken the current input character is not consumed, and used recursively until it is consumed during a normal transition. It is easy to show via amortized analysis that only two state traversals per character of the input string are required to process any given input string.

The deterministic version of Aho-Corasick automaton avoids the use of failure pointers in order to enable one traversal per input character. Instead of using failure pointers, next state from every state for every character in the alphabet is precomputed, and these transitions are added to the automaton. For an ASCII alphabet, such a construction results in 256 transitions at every state, which requires substantial amounts of memory.

A large body of research literature has concentrated on enhancing the Aho-Corasick algorithm for use in networking. In [Tuck et al. 2004], the authors present techniques to enhance the worst-case performance of 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 Aho-Corasick based pattern matching hardware architectures. In [Tan, and Sherwood 2005] the authors propose an efficient algorithm that converts the deterministic version of Aho-Corasick automaton into multiple binary state machines. These state machines have significantly fewer transitions per state, which dramatically reduces the total space requirements. In [Sourdis, and Pnevmatikatos 2004], the authors present an FPGA-based design which uses character pre-decoding coupled with CAM-based pattern matching. In [Yusuf, and Luk 2005], authors use hardware sharing at the bit level to exploit logic design optimizations, thereby reducing the area by a further 30%. Other work [Dharmapurikar et al. 2003][Bakar, and Prasanna 2004][Cho, and Smith 2004][Gokhale et al. 2002] presents several efficient string matching

architectures; their performance and space efficiency are well summarized in [Sourdis, and Pnevmatikatos 2004].

2.3.2 Regular Expressions in Packet Content Inspection

In [Sommer, and Paxson 2003], the authors note that regular expressions might prove to be fundamentally more efficient and flexible as compared to exact-match strings when specifying signatures for packet content inspection. 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, today use regular expressions to specify rules. Regular expressions are also the language of choice in several commercial security products, such as TippingPoint X505 [TippingPoint 2005] from 3Com and a family of network security appliances from Cisco Systems. Additionally, layer 7 filters based on regular expressions are available for the Linux operating system.

The most popular representation of regular expressions is the finite state automata [Hopcroft, and Ullman 1979]. There are two primary kinds: Deterministic Finite Automaton (DFA) and Non-deterministic Finite Automaton (NFA). A DFA consists of an alphabet denoted by Σ, which is a finite set of input symbols, a finite set of states s, an initial state and a transition function δ, which specifies the transition from every state for every symbol in the alphabet. In networking applications, the alphabet generally consists of 256 ASCII characters. A key property of a DFA is that in any given state, the transition function returns a single next state for any given input symbol; thus at any time, only one state is active in a DFA. The distinction between an NFA and a DFA lies in their transition function: instead of returning a single next state, the transition function of a NFA returns a set of states, which may be an empty set. Thus, multiple states can be simultaneously active in an NFA.

A regular expression containing n characters can be represented by an NFA consisting of $O(n)$ states. During the execution of an NFA, $O(n)$ states can be active in the worst case, and the processing complexity for a single input character can be $O(n^2)$. When a DFA is constructed from the same regular expression, it may generate $O(\Sigma^n)$ states in the worst-case. However, only one state will be active during execution, thereby leading to $O(1)$ per character processing complexity. Clearly, there is a space-time tradeoff between NFAs and DFAs. NFAs are compact but slow; DFAs are fast but may require prohibitive amounts of memory. Current implementations of regular expressions patterns used in networking require gigabytes of memory, and their performance remains limited to sub-gigabit parsing rates; which makes this an important and challenging research area.

In order to enable high parse rates, several researchers have proposed specialized hardware-based architectures which implement finite automata using fast on-chip logic. Implementing regular expressions in custom hardware was explored in [Floyd, and Ullman 1982], in which the authors showed that an NFA can be efficiently implemented using a programmable logic array. Sindhu et al. [Sidhu, and Prasanna 2001] and Clark et al. [Clark, and Schimmel 2003] have implemented NFAs on FPGA devices to perform regular expressions matching and were able to achieve very good space efficiency. In [Moscola et al. 2003], the authors have used such forms of NFAs that reduce the total number of simultaneously active states and demonstrated significant improvement in throughput.

These hardware based implementation approaches have two common characteristics: 1) due to the limited amount of on-chip storage, they use an NFA to keep the total number of states small, and 2) they exploit a high degree of parallelism by encoding the automata in the parallel logic resources. These design choices are guided partly by the high degree of computation parallelism available on an FPGA/ASIC and partly by the desire to achieve high throughput. While such choices seem promising for FPGA

devices, they 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 a higher degree of flexibility and programmability.

Commercial content inspection engines like Tarari's RegEx [LSI Co. 2005] already emphasize the ease of programmability provided by a dense multiprocessor architecture coupled to a memory. Content inspection engines from other vendors [SafeXcel 2003][Cavium Octeon 2005], also use memory-based architectures and report packet scan rates up to 4 Gbps. In this solution space, the transitions of the automaton are stored in memory in a tabular form, which is addressed with the state number and the input symbol. Every state traversal requires at least one memory access. Consequently, it becomes critical to keep as few active states as possible in order to limit the number of memory accesses and maintain a high parse rate. DFAs are therefore preferred over NFAs; however a large number of complex regular expressions often creates DFAs with an exponentially large number of states. Rather than constructing a composite DFA from the entire regular expressions set, [Yu et al. 2006] have proposed to partition the set into a small number of subgroups, such that the overall space needed by the automata is reduced dramatically. The proposed partitioning method keeps the number of DFAs small while containing the exponential blowup in the number of states. 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. In order to further boost the parsing performance, a recent paper [Brodie, Taylor, and Cytron 2006] attempts to construct DFAs that can consume multiple characters during a single state traversal. However, for the patterns used in networking, such DFAs tend to have exponentially increasing numbers of states in the number of input characters that are consumed at once. Even though the authors report an increased parse rate, their datasets were limited and it remains challenging to use this approach in networking.

In this dissertation, we extend these high performance memory-based regular expressions implementations. We propose a number of novel representations and algorithms that can enable regular expressions pattern matching at multi-gigabit rates, while also keeping the memory requirements low and preserving the flexibility provided by programmability.

Chapter 3

IP Packet Forwarding

In this chapter, we propose a number of algorithms and architectures that target an ASIC implementation, and can enable current Internet routers to forward IP packet at high speeds. IP packet forwarding routines include IP address lookup and packet classification, both of which generally require longest prefix match operations. Conventional methods of longest prefix match require fast computation and high memory bandwidth to achieve high performance. ASICs can enable such levels of performance by providing the required computation, and packing multiple embedded memory modules, which can be accessed in parallel, thereby providing enormous amounts of bandwidth. With such levels of memory bandwidth and computation power, it may become challenging to utilize them efficiently. Proper utilization of the bandwidth provided by multiple memory modules requires that the data-structure be distributed across these modules in a way that the accesses remain balanced and no single module becomes a performance bottleneck.

Another dimension to the difficulty arises due to the limited amount of memory bits available on-chip to handle the large databases; and that these bits are much more expensive than the commodity off-chip memory bits. It therefore becomes critical to prudently use the embedded memory, and store only those components of the data-structures on-chip that are accessed very frequently. If multiple on-chip memory modules are used to provide high bandwidth, it also becomes important to keep them nearly uniformly occupied at all times, in order to keep high levels of memory space utilization. Thus there are three challenges in devising a high-performance alternative for ASIC implementation: first, on-chip data-structures must be compact; second, they

must be uniformly divided and stored across the set of on-chip memories; and third, the accesses to the memory modules must remain balanced.

We describe two novel and orthogonal architectural solutions which can efficiently implement longest prefix match in an ASIC and achieve high performance by utilizing multiple embedded memories. Our solutions – based on a trie data structure – significantly reduces the on-chip memory compared to the state-of-the art techniques, and maintains a high degree of memory utilization, both in space and bandwidth. The first solution is called History based Encoding, eXecution, and Addressing (HEXA), which is a compact representation of directed acyclic graphs such a tries. HEXA based encoding of a trie results in between two to four times reduction in on-chip memory when compared to Eatherton-Dittia tries, and requires slightly increased computation to maintain the same level of lookup performance, thereby making it ideal for ASIC based systems. Our second solution is a novel pipelined trie implementation called Circular Adaptive and Monotonic Pipeline (CAMP), which efficiently utilizes the bandwidth provided by multiple memory modules to enable fast lookups. Unlike previous approaches such as linear pipelines, CAMP provides near perfect memory utilization when mapping the trie nodes to the pipeline stages, thereby using the scarce and expensive embedded memory bits much more prudently. We begin with HEXA, and keep our description broad enough so that we can later apply an extension to encode more complex graph structures such as the Aho-Corasick finite state automaton.

3.1 HEXA –Encoding Structured Graphs

Several common packet processing tasks make use of directed graph data structures in which edge labels are used to match symbols from a finite alphabet. Examples include tries used in IP route lookup and string-matching automata used to implement deep packet inspection for virus scanning. We describe a novel representation for such data structures that is significantly more compact than conventional approaches. We observe

that the edge-labeled, directed graphs used by some packet processing tasks have the property that for all nodes u, all paths of length k leading to u are labeled by the same string of symbols, for all values of k up to some bound. For example, tries satisfy this condition trivially, since for each value of k, there is only one path of length k leading to each node. The data structure used in the Aho-Corasick string matching algorithm [Aho, and Corasick 1975] also satisfies this property, even though in this case there may be multiple paths leading to each node. Since the algorithms that traverse the data structure know the symbols that have been used to reach a node, we can use this "history" to define the storage location of the node. Since some nodes may have identical histories, we need to augment the history with some discriminating information, to ensure that each node is mapped to a distinct storage location. We find that in some applications the amount of discriminating information needed can be remarkably small. For binary tries for example, two bits of discriminating information is sufficient. This leads to a binary trie representation that requires just two bytes per stored prefix for IP routing tables with more than 100K prefixes.

3.1.1 Introduction to HEXA

A large fraction of current research literature improves the performance of traversing directed graph structures such as tries by either reducing the number of child pointers stored and/or by reducing the number of nodes. With or without the reductions in the number of nodes or pointers, to our best knowledge, directed graphs are always implemented in the following conventional manner. Each node in the n node graph is denoted by a unique $\lceil \log_2 n \rceil$ bit identifier, which also determines the memory location of the node. At this memory location, all next node pointers (identifiers of the subsequent "next nodes") are stored, along with some auxiliary information. The auxiliary information may be a flag indicating if the node corresponds to a match in a string matching automata or a valid prefix in an IP lookup trie, and an identifier for the string, or the next hop for the matching prefix. The auxiliary information usually

requires only a few bits and is kept once for every node; on the other hand, identifiers of the "next node" use $\lceil \log_2 n \rceil$ bits each and are required once for every next node pointer. Thus, in large graphs (say a million nodes) containing multiple next node pointers per node (say two), the memory required by the identifiers of the "next node" (20-bits per identifier, 2 such identifiers per node) can be much higher than the memory required by the auxiliary information.

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

We begin the description of a new method to store directed graph structures that we call HEXA. While conventional methods use $\lceil \log_2 n \rceil$ bits to identify each of n nodes in a graph, by taking advantage of the graph structure, HEXA employs a novel method that can use a fixed constant number of bits per node for structured graphs such as tries. Thus, when HEXA based identifiers are used to denote the transitions of the graph, the fast memory needed to store these transitions can be dramatically reduced. The total

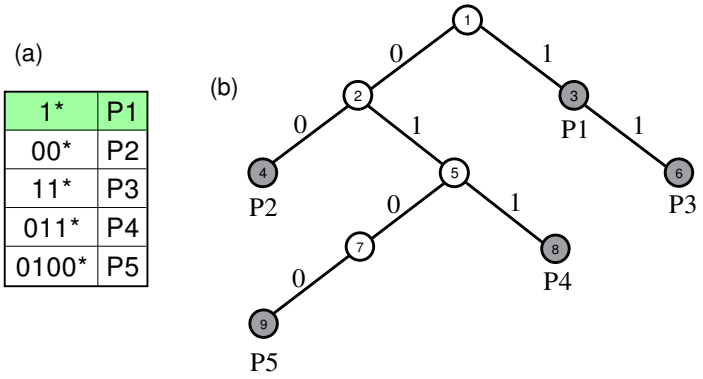


Figure 3.1 (a) Routing table; (b) corresponding binary trie.

memory is also reduced significantly, because auxiliary information often represents a 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 the paths leading to them, the nodes can be identified by properties of these paths. 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 in order to uniquely identify each node. To clarify, we consider a simple trie-based example before formalizing the ideas behind HEXA.

3.1.2 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 3.1. 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-tuples consisting of the prefix flag and the left right children (NULL indicates no child):

1. 0, 2, 3	4. 1, NULL, NULL	7. 0, 9, NULL
2. 0, 4, 5	5. 0, 7, 8	8. 1, NULL, NULL
3. 1, NULL, 6	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^{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 the trie in order to traverse it: the first bit is set if node corresponds to a valid prefix, and the second and third bits are set if the node has a left or 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, have 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 follows; we also list the corresponding next hops:

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

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

This example suggests that we can dramatically reduce the memory requirements used 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.

3.1.3 Devising One-to-one Mappings

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 identifiers 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 the 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 a few additional (say c) bits, which we can alter at our convenience. We call these c-bits the node's discriminator. Thus, the HEXA identifier of a node will be the history of labels used to 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 [Kirsch, and Mitzenmacher 2005] and cuckoo hashing [Pagh, and Rodler 2001], and we use results from these analyses.

Note that when traversing the graph, we need to know a node's discriminator in order to access it. Hence instead of storing a single bit for each left and right child, we store the discriminator if the child exists. In practice, we also typically reserve the all-0 c-bit word to represent NULL, giving us only 2^c-1 memory locations.

This problem can now be viewed as a bipartite graph matching problem. The bipartite graph $G = (V_1+V_2, E)$ consists of the nodes of the original directed graph as the left set of vertices, and the memory locations as the right set of vertices. The edges connecting

Nodes	Input labels	Four choices of HEXA identifiers	Choices of memory locations
1	-	00 -, 01 -, 10 -, 11 -	h() = 0, h() = 4 h() = 1, h() = 5
2	0	00 0, 01 0, 10 0, 11 0	h() = 1, h() = 5 h() = 2, h() = 6
3	1	00 1, 01 1, 10 1, 11 1	h() = 0, h() = 4 h() = 1, h() = 5
4	00	00 00, 01 00, 10 00, 11 00	h() = 2, h() = 6 h() = 3, h() = 7
5	01	00 01, 01 01, 10 01, 11 01	h() = 1, h() = 5 h() = 2, h() = 6
6	11	00 11, 01 11, 10 11, 11 11	h() = 8, h() = 3 h() = 0, h() = 4
7	010	00 010, 01 010, 10 010, 11 010	h() = 1, h() = 5 h() = 2, h() = 6
8	011	00 011, 01 011, 10 011, 11 011	h() = 0, h() = 4 h() = 1, h() = 5
9	0100	00 0100, 01 0100, 10 0100, 11 0100	h() = 0, h() = 3 h() = 4, h() = 6

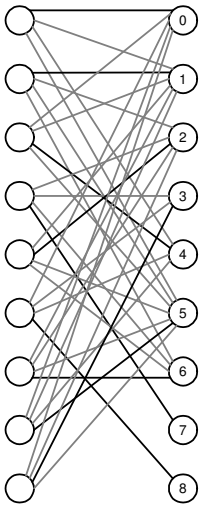


Figure 3.2 Memory mapping graph, bipartite matching.

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 [Pagh, and Rodler 2001], 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 3.1.4.

Continuing with our example of the trie shown in Figure 3.1, 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 3.2, 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.

3.1.4 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 these 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 a 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 with high probability, 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 [Pagh, and Rodler 2001]. 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 [Pagh, and Rodler 2001].

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

We will demonstrate in our experiments in Section 3.4.2 that the number of changes needed to maintain a HEXA representation with node insertions and deletions is quite reasonable in practice. Again, similar results can be found in the setting of cuckoo hashing.

3.1.5 Summarizing HEXA

HEXA is a novel representation for structured graphs such as tries. HEXA uses a unique method to locate the nodes of the graph in memory, which enables it to avoid using any "next node" pointer. Since these pointers often consume most of the memory required by the graph, HEXA based representations are significantly more compact than the standard representations, and extremely valuable in ASIC implementations. We now proceed with the description of CAMP, which is a pipelined implementation of tries to enable high lookup throughput. We will later show that a pipelined trie such as CAMP with HEXA encoded nodes can result in a longest prefix match solution that is

significantly superior to the current state-of-the-art methods in all four performance metrics defined in Chapter 1.

3.2 CAMP – Pipelining a Trie

Recent advances in optical and signaling technology have pushed network link rates beyond 40 Gbps, with 160 Gbps links now appearing. A line card terminating a 160 Gbps IP link needs to forward a minimum-sized packet within 2 ns. To do so, a packet header must be processed within 2 ns. At such speeds, both IP lookup, and packet classification become very challenging. The performance problem with longest prefix match is due to the sequential memory accesses required per match, and the growing global routing tables containing over one hundred thousand prefixes. The dual challenges of serialized access and large datasets have inspired a number of novel specialized hardware architectures.

Memory bandwidth is an important concern in any implementation, whether it is based on off-chip memory or an ASIC. For example, at 160 Gbps rates, a multi-bit trie of stride 4 requires 8 memory accesses every 2 ns. Achieving this bandwidth using a single memory is challenging. A number of researchers have proposed a pipelined trie. Such tries enable high throughput because when there are enough memories in the pipeline, no memory stage is accessed more than once for a search and each stage can service a memory request for a different lookup each cycle.

Most recently, [Baboescu, Tullsen, Rosu, and Singh 2005] 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 much more flexibility in mapping trie nodes to pipeline stages, which in turn maintains uniform memory occupancy. We extend this approach with an architecture called *Circular,*

Adaptive and Monotonic Pipeline (*CAMP*). Our work, while also exploiting a circular pipeline, differs from the previous circular pipeline proposals in several ways.

First, CAMP differs in the way the trie is split into sub-tries. While Baboescu et al. aim at having a large (~4000) number of equally sized sub-tries, our design strives for simplicity. Thus, CAMP splits a trie into one root sub-trie and multiple leaf sub-tries. The root sub-trie handles the first few bits (say *r*) of the IP address, and it is implemented as a table, indexed by the first *r* bits of the IP address. With this, there may be up to 2^r leaf sub-tries; each of which can be independently mapped to the pipeline. By judiciously mappings these, the system maintains near-optimal memory utilization, not only in memory space but also in the number of accesses per pipeline stage.

Second, having a reduced number of sub-tries of different sizes, we propose a different heuristic to map them to the pipeline stages. As a matter of fact, our scheme proves to be much simpler, and also gracefully handles incremental updates.

Finally, our design uses a different mechanism to maximize pipeline utilization and handle out of order lookup conditions. In particular, we aim at having not more than one access per pipeline stage for any lookup. CAMP goes further and decouples the dependence of number of pipeline stages from the number of trie levels. Thus it can employ a large number of compact and fast pipeline stages to enable high throughput while consuming low power. With a large number of stages, pipeline utilization may degrade significantly. To this end, CAMP employs effective schemes to achieve high utilization.

We also present an extensive analysis of the design tradeoffs and their impact on lookup rate and power consumption. For real routing tables storing 150,000 prefixes, CAMP achieves 40 Gbps throughput with a power consumption of 0.3 Watts. Projections on 250,000 prefixes show a power consumption of 0.4 Watts at the same throughput. We begin with a description of the operation of pipelined tries.

3.2.1 Introducing CAMP

Pipelining is an effective way to achieve high lookup rates. Previous pipelined schemes are based on the assumption that the pipeline is linear, and has a unique entry and exit point; moreover, it is assumed that a global mapping is performed on the entire trie. We remove both assumptions, based on the observation that practical prefix-sets present us considerable opportunity to split a trie into multiple sub-tries; thus, different pipeline entry points can be assigned to them. This leads to many mapping opportunities, from which assignments may be chosen to achieve balanced pipeline stages. Moreover, it also eliminates two important limitations faced by any global mapping based scheme, namely, 1) the number of pipeline stages is bound to the maximum prefix length, and 2) adding a memory bank requires a complete remapping (in scenarios of an overflow generated by a sequence of prefix insertions).

We introduce Circular Adaptive and Monotonic Pipeline (CAMP) using a set of 8 small prefixes shown in Figure 3.3 along with the corresponding binary trie. Pipelining this trie will require 6 stages. A level-based mapping [Basu, and Narlikar 2003] will result in 1, 2, 3, 5, 2 and 2 nodes in stages 1 to 6, respectively, while a height-based mapping [Hasan, and Vijaykumar 2005] will result in 6, 4, 2, 1, 1 and 1 nodes. Thus, both of these mappings create unbalanced pipeline and the degree of imbalance is dependent upon the prefix set.

We now consider splitting this trie into four sub-tries. Since prefix P1 is only 1-bit long, we first expand it to 2 bits using controlled prefix expansion (see Figure 3.3). Now, all prefixes in the database are longer than 2-bits; therefore, the upper two levels of the trie can be stored in a direct index table, which leaves us with three sub-tries of at most four levels each. More generally, when a routing database contains prefixes all of which are longer than x-bits (shorter prefixes are expanded to x-bits), then the first x levels of the

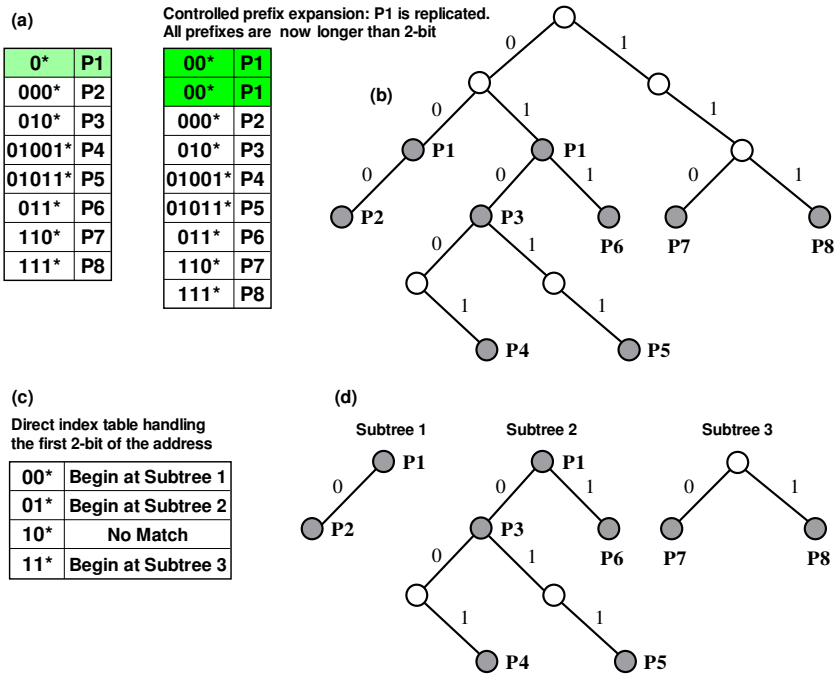


Figure 3.3 (a) Routing table (prefixes shorter than 2-bits are expanded using controlled prefix expansion) (b) unibit trie of six levels; (c) Direct index table for first 2-bits, (d) resulting 4 sub-tries of four levels each.

trie can be replaced by a direct index table containing 2^x entries, each of which points to one of the up to 2^x sub-tries with height at most $32 - x$.

With multiple subtries, we now seek to obtain a balanced mapping of nodes to pipeline stages. We exploit the fact that requests can enter and exit at any stage, thus roots of sub-tries can be mapped to any stage. If we also allow a request to wrap-around through the pipeline (*i.e.*, by taking advantage of the circular pipeline), we can get a high degree of flexibility in mapping. Nodes descended from the root of a sub-trie can be stored at subsequent pipeline stages, wrapping around once the final stage is reached. In the example above, the 3 sub-tries constructed from the 8 prefix table can be mapped to a four stage circular memory pipeline with dynamic entry points as shown in Figure 3.4. Note that the first two bits are used to determine the entry stage into the pipeline and subsequent bits are processed within different pipeline stages.

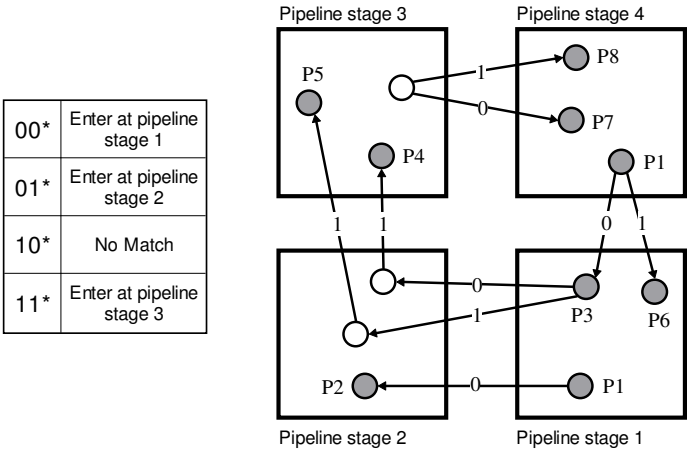


Figure 3.4 A four stage circular pipeline and the way the three subtries in Figure 3 are mapped onto them.

3.2.2 General Dynamic Circular Pipeline

A general circular pipeline may not require a node to be stored in a stage adjacent to the parent node's stage. For example, the two nodes of the first sub-trie in the previous example can be stored at any two distinct stages, because, irrespective of the way they are stored, a lookup request for this sub-trie will access each stage only once. However, this will require the pipeline to insert no-ops when a request traverses a stage where the required node is not present. Supporting no-ops increases the flexibility in storing the nodes of various sub-tries which can lead to more balanced pipeline stages. On the other hand, as will be shown later, it may complicate the update scenario.

A general circular pipeline has three important properties, *i*) it allows dynamic entry and exit points, *ii*) it is circular, thus all neighboring stages are connected in one direction, and *iii*) it supports no-ops for which requests are simply passed over whenever the designated node is not found. The corresponding mapping algorithm maps the root of each sub-trie to some pipeline stage and subsequent nodes are mapped such that, *a*) a

node is stored at a stage which is at least one ahead (accounting for wraparound) of the stage where its parent is stored, and *b*) all lookup paths terminate before making a circle through the pipeline. Thus, nodes along any path are mapped in a monotonically increasing pipeline stage and every lookup is guaranteed to make at most one access to a memory stage.

It can be argued that the lookup throughput of a general circular pipeline matches that of any other pipeline because a lookup request accesses a memory at most once. However, allowing dynamic entry points introduces new problems due to request conflicts. A request contending to enter the i^{th} stage may have to wait until a bubble (idle cycle) arises there. In an extreme case, a request may have to wait for such a bubble indefinitely, if other requests are entering the pipeline every cycle and keeping its entry stage busy. This may lead to non-deterministic performance, low pipeline utilization and out-of-order request processing. However, as we will see next, relatively straightforward techniques coupled with a small speedup in pipeline operating-rate ensure deterministic performance.

3.2.3 Detailed Architecture of CAMP

The schematic block diagram of a CAMP system is shown in Figure 3.5, which consists of a direct lookup table and a circular pipeline of memories. The direct table lookup performs a lookup on the first *x*-bits of the address (*x* being the initial stride in the lookup trie), and determines the stage where the root node of the corresponding sub-trie is stored. Subsequently, a lookup request to traverse through the sub-trie is dispatched into that stage. All requests to a pipeline stage are stored in an ingress FIFO in front the stage. As soon as the stage receives a bubble (idle cycle), the request at the front of the FIFO is issued into the pipeline. The request traverses through the pipeline and as it reaches the stage containing the last leaf node, it comes out with either valid next hop information or a no match.

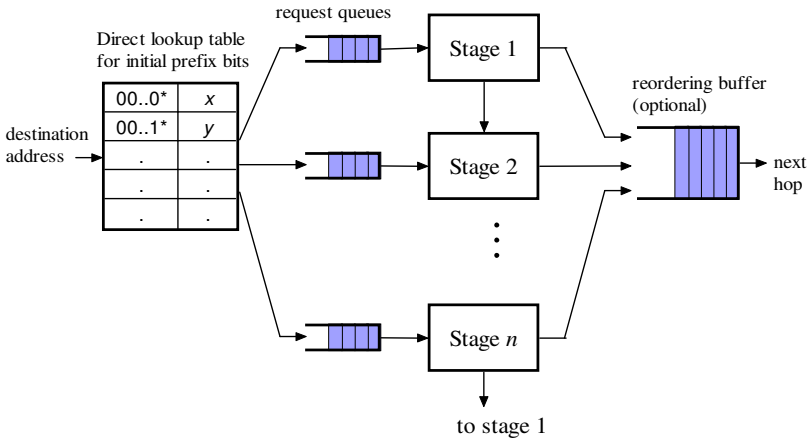


Figure 3.5 Schematic block diagram of a CAMP system.

The ingress FIFO in front of each stage is crucial in improving the efficiency. Consider a system without such queues. It is possible that a stream of n lookup requests enters some stage resulting in a train of n entries in the pipeline, and subsequent requests contending to enter the pipeline waits for up to n cycles. Thus, the efficiency can be as low as 50%, because the pipeline services n requests and then waits for up to n cycles before servicing subsequent requests. In the worst-case, efficiency can be even lower. Consider a situation when a request enters at a pipeline stage i. The next request will wait for 1 cycle if it contents to enter at the pipeline stage i+1. The third will waits for 2 cycles, as it contends to enter at the stage i+2. If this pattern will continue, the i^{th} request will wait for i–1 cycles, which will lead to a very low efficiency.

The ingress FIFO in front of each pipeline stage serves as a reorder buffer, which obviates the above issue of head of line blocking. If a request must wait for a few cycles because its entry stage is busy, it stays in the stage's ingress FIFO instead of blocking the subsequent requests. Quite intuitively, large request queues will improve the efficiency of the pipeline, as they will provide extended immunity against conditions when a request must wait before being dispatched into the pipeline.

A drawback of using these ingress FIFOs is that the requests may leave the pipeline out-of-order. Therefore, a reorder buffer is required at the output to restore the order of requests. Reordering is optional because the problem of out-of-order arises only among the packets destined to different destinations. A single TCP flow will never experience any reordering, as any two packets having the same prefix (thus designated to the same "next hop") always traverse thru the same path in the lookup trie. Hence, these requests will contend to enter the pipeline at the same stage, where they are serviced in a first-in first-out order. We now introduce the metric of pipeline efficiency and characterize it for different pipeline configurations and input traffic patterns.

3.2.4 Characterizing the Pipeline Efficiency

The primary metric to characterize the efficiency of CAMP is pipeline utilization. Pipeline utilization is the fraction of time the pipeline remains busy provided that there is a continuous backlog of lookup requests. Another metric, which more directly reflects the performance, is Lookups per Cycle or LPC, *i.e.* the rate at which lookup requests are dispatched into the pipeline.

A linear pipeline guarantees a LPC of 1 however pipeline utilization can remain low if a majority of prefixes are not 32-bits long (hence they do not use all stages). In a CAMP pipeline, on the other hand, pipeline utilization can be maintained at 1 and requests may be dispatched at rates higher than one per cycle. LPC greater than 1 can be achieved, *i*) when most requests do not make a complete circle through the pipeline, or *ii*) when there are more pipeline stages than there are levels in the trie. Thus, whenever some pipeline stages are not traversed by a request, new requests contending to enter those stages can be issued into the pipeline. Note that, practical IP lookups, where a majority of prefixes are only 24-bits long, will only utilize 75% stages of a 32-stage pipeline.

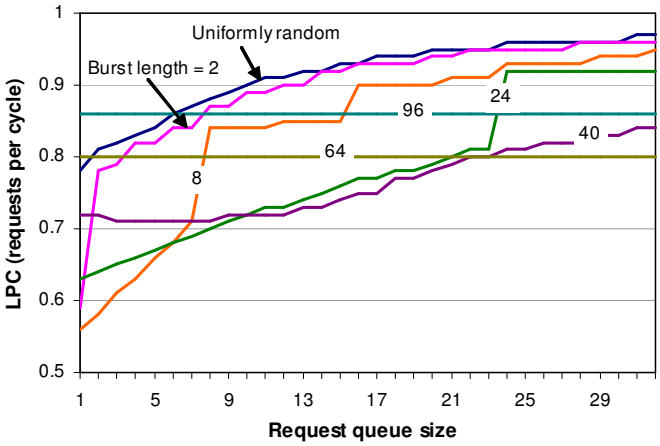


Figure 3.6 LPC of CAMP versus request queue size.

In order to evaluate the efficiency of CAMP, we use software simulations to determine the pipeline utilization and the resulting LPC. In the first set of experiments, we assume that all requests make one complete circle through the pipeline and there are as many pipeline stages as there are levels in the trie (in this case pipeline utilization will be equal to the LPC). Later we consider scenarios, when requests do not make a complete circle. We also assume that the requests that find its ingress FIFO full are discarded; in an actual implementation, to avoid such discards, a large buffer can be allocated which will feed the ingress FIFOs. The only variable in the arriving requests now is the entry point in the pipeline. We consider following four different distributions of the entry points of the arriving requests: *i*) uniformly random request to each stage, *ii*) uniformly random short bursts of requests to each pipeline stage, and *iii*) uniformly random long bursts of request to each stage, and *iv*) weighted random arrivals; thus some stages receive more requests than the others.

Our representative setup has 24 pipeline stages and requests circle through all stages before exiting. In Figure 3.6, we report the LPC for different request queue sizes. It is clear that, a LPC of 0.8 can be achieved for all traffic patterns, once the request queue size is 32. This suggests that CAMP remains 80% efficient for practically all traffic patterns. In another experiment, we fixed the request arrival rate at 0.8 per cycle and the

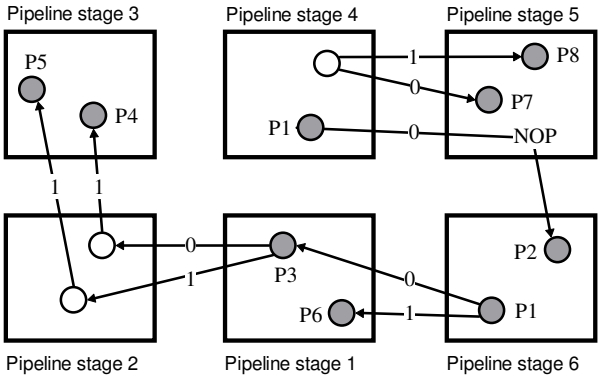


Figure 3.7 A six stage circular pipeline and the way the three sub-tries in Figure 3.6 are mapped onto them.

request queue size at 32 and measured the discard rate and the average delay experienced by a request. After running the experiment for more than 100 million iterations, no requests were discarded and the average delay experienced by a request was only a few tens of cycles.

Not surprisingly, in these experiments, very long bursts of requests to various pipeline stages result in higher utilization because when many requests arrive at a stage one after another, they are all serviced without conflict. On the other hand, when the burst lengths are comparable to the pipeline depth, trains of requests are created and subsequent bursts may have to wait before getting dispatched into the pipeline.

3.2.5 When is LPC greater than one?

While the LPC of a linear pipeline is always one, the LPC of CAMP can be engineered to be greater than one, which can improve the throughput. This is possible because CAMP enables a trie data-structure to be pipelined further, up to a number of stages much larger than the number of levels in the trie. For example, the mapping of the three sub-tries shown in Figure 3.3 to a six stage pipeline is shown in Figure 3.7. As we will soon see, with many sub-tries, it is not difficult to determine an appropriate stage for

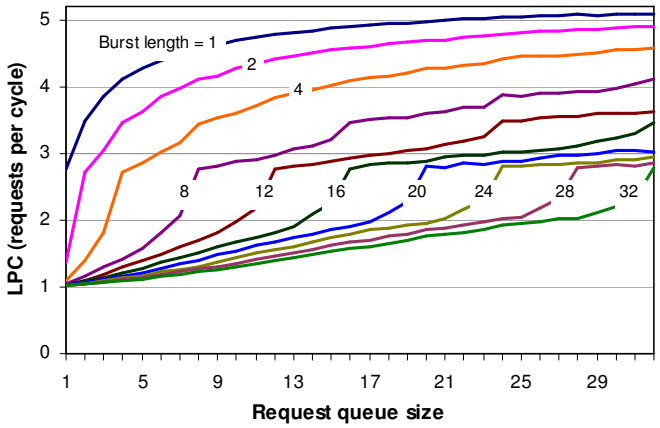


Figure 3.8 LPC of CAMP versus request queue size. Requests arrive at each pipeline stage in random bursts (burst length highlighted in the figure).

the sub-trie's root so that every stage of the pipeline is nearly uniformly populated. When there are many stages in the pipeline, each sub-trie (and its lookup requests) will span only a fraction of all stages. This can lead to a dispatch rate higher than one per cycle, assuming that all arriving requests do not traverse the same sub-trie. In fact, when sub-tries and therefore the associated prefixes are nearly uniformly dispersed all around the stages (because stages are balanced), it is less likely that all lookup requests will contend to enter one stage. Notice that, an orthogonal factor leading to higher LPC is the fact that most prefixes have only 24-bits.

From an implementation perspective, it is neither difficult nor expensive to implement more pipeline stages relative to the trie levels. A multi-bit trie with the appropriate node encoding (tree-bit map or shape shifting trie), will not only reduce the memory but also effectively increase the number of stages in the pipeline per trie level. A stride of k will reduce the number of levels in the trie by a factor of k, which can directly lead to a k-times higher LPC if we keep the same number of pipeline stages.

We now consider evaluating such scenarios. Our setup consists of a 32-stage pipeline, and an initial stride of 8 leading to the sub-tries containing at most 24 leftmost prefix bits. Each sub-trie uses tree-bit map of stride 3, thus a single lookup path spans across

at most 8 pipeline stages. In this experiment, we also assume that the average prefix length is 24-bits, thus a request on average traverses thru only 6 stages. As reported in Figure 3.8, the LPC ranges from 3 to 5, even for long bursts of requests contending to enter the same stage. For smaller bursts, which are more common, LPC is even higher.

3.2.6 Mapping IP Lookup Tries in CAMP

To use CAMP, we need a mapping algorithm which assigns the trie nodes to the pipeline stages. The primary purpose of the algorithm is to achieve a uniform distribution of nodes to stages. In particular, the mapping should minimize the size of the biggest (and bottleneck) stage. This will not only enable high throughout but also reduce the chances of unbalanced pipeline during updates.

Problem Formulation

We can formulate the above problem as a constrained graph coloring problem, where colors represent the pipeline stages, and the graph corresponds to the set of sub-tries. The following two constraints guide the coloring: *i*) every color should be nearly equally used, and *ii*) a relation of order, when traversing a sub-trie from the root to the leaves, must be associated with the color assignment. The first constraint captures the intent of achieving a uniform distribution of nodes across the stages. The second constraint arises due to the fact that nodes must be mapped to the circular pipeline stages in a circular and monotonic order. Thus, all paths from root to leaf must be assigned distinct colors in a monotonic order (including wraparound).

If we represent each color by an integer, the relation of order is the "saturated <" relation. In other words, if we have N colors (1, 2, ..., N) then the following relation will hold: $1<2<...<N<1$. A mapping which doesn't preserve this order relation is exemplified in Figure 3.9(a). Such a mapping can cause a lookup to circle through the

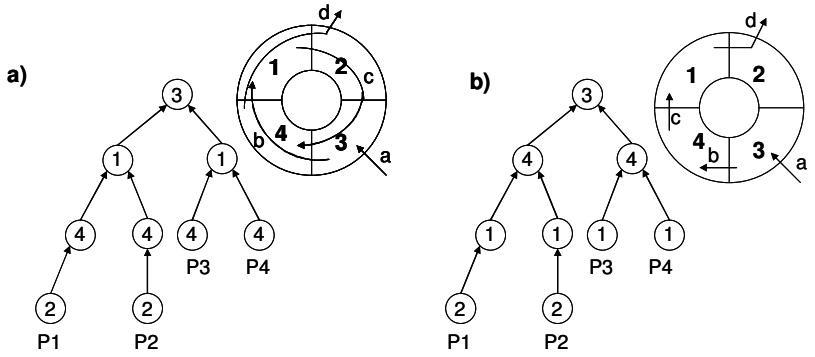


Figure 3.9 a) invalid assignment: matching P1 causes one extra loop of the circular pipeline; b) valid assignment: the circular pipeline is traversed only once.

pipeline more than once, thereby reducing the LPC. A mapping which preserves the order relation is illustrated in Figure 3.9(b), where all paths from root to leaf (*i.e.* any lookup operation) traverse through a color at most once. Naturally there can be several mapping choices which will preserve the order relation and we are interested in those which lead to a nearly uniform usage of different colors. Such constrained graph coloring problem is NP-hard and can be reduced to the well known partition problem therefore we present a heuristic algorithm to obtain a near optimal solution.

The Largest First Coloring Algorithm

Several simple heuristics can be obtained to perform the coloring which preserves the order relation. For instance, each sub-trie can be colored by first randomly selecting a color for the root node and then incrementing the color when proceeding towards the leaves. While such a randomized scheme may lead to fairly balanced color distributions in the case of a large number of sub-tries, it may not be satisfactory when there are not that many sub-tries or when some sub-tries are significantly larger than the others. We therefore introduce a more effective coloring heuristic. In particular, if we color sub-tries sequentially, at each coloring step we want to exploit the information about the current status of the color distribution.

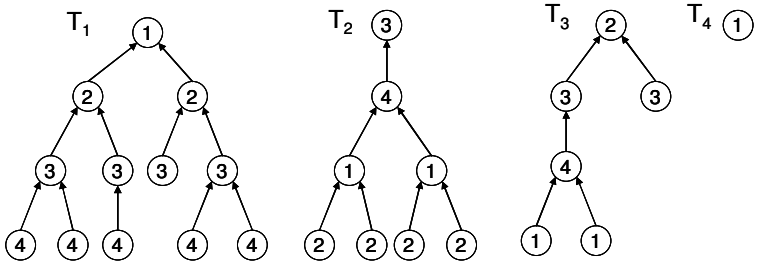


Figure 3.10 Example coloring with largest first heuristic.

A largest first coloring heuristic seeks to obtain uniform color usage by coloring the sub-tries in a sequence such that the larger sub-tries are colored before the smaller ones. Such a sequence is motivated by the well-known bin-packing heuristic and can be attributed to the fact that if tries are colored in a decreasing size sequence then the coloring of smaller tries can effectively correct the unbalances caused by the already colored bigger tries. Thus, the largest first heuristic first sorts all sub-tries according to their size and then in a decreasing order, assigns colors to the nodes of the sub-tries. For the currently selected sub-trie, the coloring needs to restore any discrepancy in the color usage until now. Since, the choice of a color for the root determines the colors for all subsequent nodes, the largest first algorithm tries all possible colors for the root node (subsequent nodes are colored with increasing color values) and finds the color usage for each choice. Finally, it picks the one which results in the most uniform color usage and moves on to the next sub-trie. Figure 3.10 illustrates the application of the largest first heuristic on a set of four sub-tries.

Additional Considerations

The above coloring heuristic only considers unitary increment between colors of a node and those of its children. It would be possible to add further flexibility in color assignment by removing this constraint, without affecting the correctness of the system. The shaded area in Figure 3.11(a) illustrates this possibility. The added flexibility may lead to more uniform usage of colors however it complicates the coloring. The

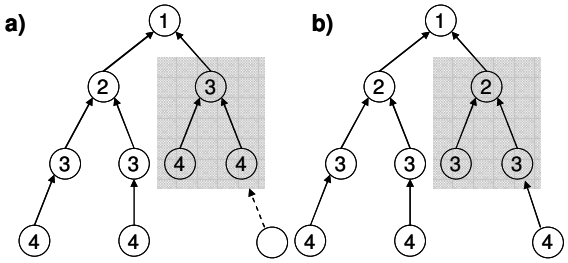


Figure 3.11 An insertion operation causes a sub-trie remapping in case of skip-level assignment.

complexity may be acceptable if the mapping were static; however in practical systems, updates often add and remove nodes from the tries, in which case, remapping a large part of the trie may be needed if the unitary increment constraint is not applied. As an example, let us add a bottom right node to the trie shown in Figure 3.11(a). Since color 4 is already used at the leaf, the colors of the nodes in the shaded area must be reassigned, as illustrated in Figure 3.11(b). For this reason, we do not consider the possibility of skipping colors between adjacent levels.

3.3 Coupling HEXA with CAMP

In this section we describe how the HEXA and CAMP techniques can be brought together to create an efficient and high performance longest prefix match architecture. HEXA is a compression method that encodes the "next node" pointers in structured graphs such as tries in a novel way that substantially reduces the amount of memory required. Memory compression is extremely valuable in ASIC implementations, where expensive and limited embedded memories are used, however HEXA does not aid in improving the lookup performance of the trie. In fact, in an optimal setting, HEXA encoded tries use much smaller stride values than competing approaches, which increases the number of memory accesses required to lookup any given address. Thus, if the entire trie is stored using a single embedded memory, HEXA encoding can reduce the lookup throughput and may not be desirable. Fortunately, it is possible in ASIC based systems to employ a reasonably large number of compact but independent

memory modules, thereby substantially increasing the amount of memory bandwidth available. CAMP perfectly complements HEXA in such settings: HEXA provides memory compression so that larger lookup data structures can be accommodated by the on-chip memory; CAMP enables bandwidth efficiency by properly utilizing the available memory bandwidth to enable high lookup throughput.

While HEXA is crucial in reducing the memory required to store a given trie, CAMP also plays an important role in improving the space efficiency. In CAMP, memory modules are configured in a novel circular pipeline, which unlike a traditional linear pipeline, allows multiple entry and exit points. Such a structure provides much greater flexibility in mapping the trie nodes to the pipeline stages and keeps the pipeline stages balanced, which helps reduce the amount of memory required to store a given number of prefixes. In our experiments, we will report that when HEXA is coupled with CAMP, a longest prefix match engine can be devised which is not only efficient in space, but also enables high lookup rate and low power dissipation. There are, however some complications in coupling HEXA with CAMP, which we will discuss before proceeding with the experimental evaluation.

When we couple CAMP with HEXA, we will obtain nodes from a collection of sub-tries rather than from a single trie for mapping to memory. This apparent complication does not pose any serious threat to the mapping process, because nodes within each sub-trie will maintain unique HEXA identifiers. The real complication arises due to the use of multiple memory modules. In our description presented in Section 3.1.3, we map the HEXA encoded nodes to memory words within a single address space. CAMP however uses multiple memory modules (pipeline stages), thus HEXA encoded trie nodes are required to be mapped to both "a memory module" and "a word within the module". Additionally, when we couple CAMP with HEXA, we will map nodes from a collection of sub-tries rather than from a single trie. A logical first step is divide the trie into multiple sub-tries and map sub-trie nodes to memory modules, which can be accomplished with the algorithm presented in our description of CAMP in Section

3.2.6. The next step is to determine the HEXA identifier of nodes within each module, and then map these identifiers to memory words within the module. Since the HEXA identifier of every node is unique, the HEXA identifiers of the sub-trie nodes within each memory module will also be unique. Thus, within each memory module, HEXA identifiers to memory words mapping can be accomplished, with the construction of a bipartite graph of the sub-trie nodes to be stored in the module and its memory words, and subsequently finding a perfect matching in the graph (the procedure is described in Section 3.1).

Once both – node to pipeline stage, and node to memory word location – mappings are complete and next node pointers of each node (discriminators of the next nodes) are stored, the execution of the pipeline will require a slight modification. In a standalone CAMP system, IP addresses are inserted in the entry stage; subsequently, lookup of a node in any pipeline stage provides the location of the next node (if it exists) stored in the next stage. This location is passed over to the next stage along with the IP address and the current lookup bit position within the IP address. When coupled with HEXA, a node will not explicitly store the location of its next nodes; rather their discriminator values will be stored. Therefore, in such a system, a discriminator, an IP address, and the current lookup bit position are passed between the pipeline stages. Recall that the discriminator of a node is hashed along with the node's HEXA identifier to compute its memory locations, thus each pipeline stage will require a hash function circuit. Other components of the system will remain unaffected. To summarize, there are two key differences between a standalone CAMP system, and an integrated HEXA/CAMP system: the integrated system requires a hash function within each pipeline stage to map HEXA identifiers to memory locations, and discriminators instead of explicit memory locations are passed between pipeline stages.

3.4 Experimental Evaluation

In this section we evaluate the performance of HEXA and CAMP and compare them with state-of-the art methods such as Eatherton-Dittia tries, and linear pipelines. We first consider unibit tries, and show how HEXA can lead to dramatic memory reduction and how the selection of the initial stride in CAMP keeps the pipeline stages balanced. Subsequently, we analyze the impact of route updates, and show how both HEXA and CAMP can gracefully handle them. Thereafter, we extend these analyses to multi-bit trie implementations and show how our solutions remain effective for such representations. We conclude with a brief analysis of power dissipation and die area. Our study focuses mainly on practical databases: we therefore begin with a brief discussion of the IPv4 address allocation process and trends in BGP routing table growth.

3.4.1 Datasets - BGP Routing Tables and Trends

BGP tables have grown steadily over the past two decades from less than 5000 entries in the early 1990s to nearly 75,000 entries in 2000 to up to 135,000 entries today. The trends in the growth are well studied in [BGP Table Data 2006][Huston 2001], which highlight that 16 to 24-bit long prefixes makes up the bulk of the BGP table. It has been shown that a small fraction (<1%) of prefixes are longer than 24-bits and are likely to remain so in the near future due to address aggregation and route aggregation techniques. The use of prefix length filtering also limits the propagation of longer prefixes throughout the global BGP routing domain.

Another important trend concerns updates in BGP tables. A majority of updates are linked to network link failure and recovery which removes a set of neighboring prefixes from the trie and quickly adds them back either due to the link recovery or due to the discovery of an alternative path.

To summarize the BGP trends: *i*) the number of prefixes in BGP tables has grown nearly exponentially and is likely continue to grow; *ii*) prefixes smaller than 26-bits make

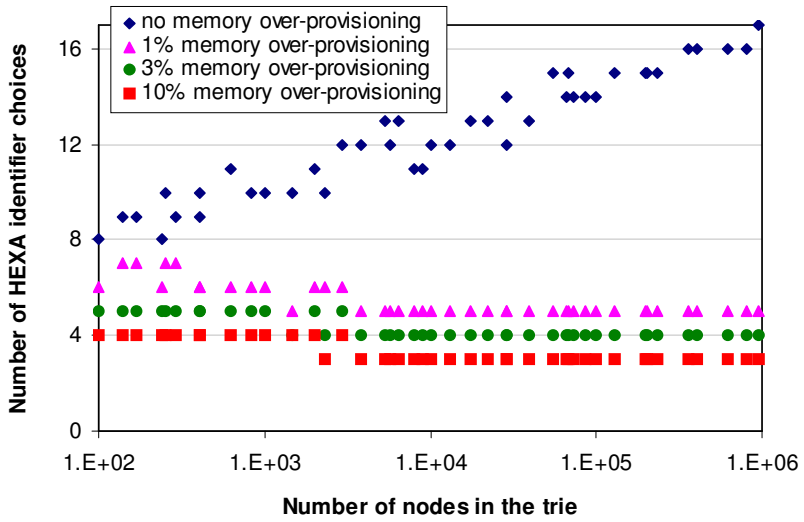


Figure 3.12 For different memory over-provisioning values and trie sizes, the number of choices of HEXA identifier that is needed to successfully perform the memory mapping.

up the bulk of the BGP table and this is likely to remain true in the near future; *iii*) route updates can be concentrated in short periods of time; however, updates rarely change the shape of the trie, even after extended periods of time.

We now discuss the memory requirements of pipelined tries. Unless otherwise specified, the experiments reported in this section are based on a dataset consisting of more than fifty BGP tables obtained from [BGP Table Data 2006] and [Routing Information Service], containing from 50,000 to 135,000 prefixes.

3.4.2 Experimental Evaluation of HEXA

We have performed a thorough experimental evaluation of the HEXA representations of lookup trie. The results shown here demonstrate that, HEXA can dramatically reduce the memory required by a binary trie; at the same time it can also reduce the memory in more sophisticated trie implementations like multi-bit trie and tree bit-map.

Binary Tries

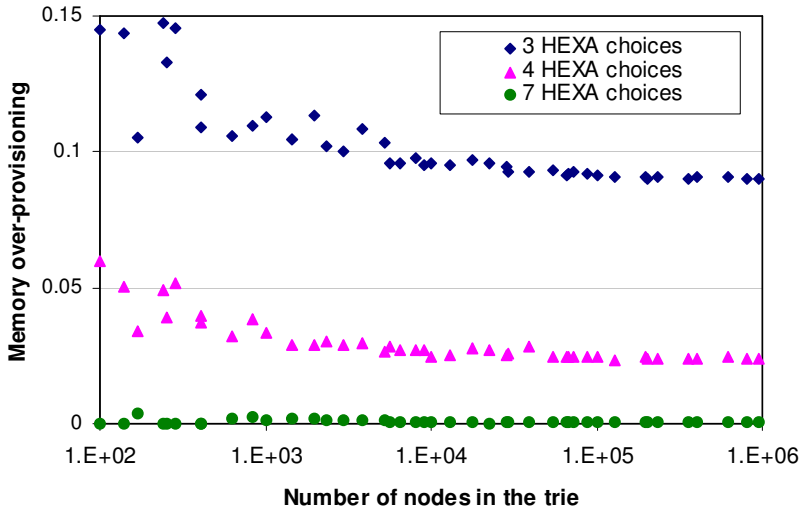


Figure 3.13 For different number of choices of HEXA identifiers and trie sizes, the memory over-provisioning that is needed to successfully perform the memory mapping.

In Figure 3.12, for varying trie sizes, we plot the number of choices of HEXA identifiers that are needed to find a perfect matching in the memory mapping graph. As expected, more choices of HEXA identifiers or increased memory over-provisioning $((m-n)/m)$ helps in finding a perfect matching. In agreement with the theoretical analysis, for $m=n$, the required number of HEXA identifier choices remains $O(\log n)$. However, when m is slightly greater than n (results for 1, 3 and 10% are reported here), the required number of choices becomes constant, independent of the trie size. Recall that the number of HEXA identifier choices determines the number of discriminator bits that are needed for a node, thus a small memory over-provisioning is desirable in order to keep the discriminators constant in size.

From a practical point, we would like to keep the number of choices of HEXA identifiers a power of two minus one, so that one discriminator value will be used to indicate a null child node and all remaining permutations of discriminator values will be used in finding better matching. Thus, we are interested in such choices as 1, 3, 7, etc. Therefore, we fix the number of HEXA choices at these values, and plot the memory

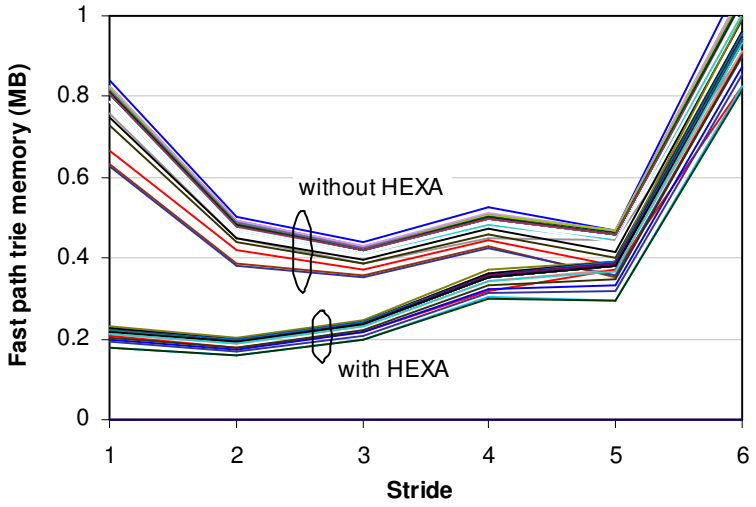


Figure 3.14 Memory needed to represent the fast path portion of the trie with and without HEXA. 32 tries are used, each containing between 100-120k prefixes.

over-provisioning needed to successfully perform a one-to-one memory mapping (Figure 3.13). It is clear that that for 3 HEXA identifier choices, the required memory over-provisioning is 10%. Thus, 2.2 bits are enough to represent each node identifier.

Multi-bit Tries

We now extend our evaluation of HEXA to multi-bit tries where tree bit-maps are used to represent the multi-bit nodes. Notice that when HEXA is used for such tries, the bit-masks used for the tree bitmap nodes are not affected; only the pointers to the child nodes are replaced with the child's discriminator. The first design issue in such tries is to determine a stride which will minimize the total memory. We accomplish this experimentally by applying different strides to our datasets and measuring the total fast path memory. The results are reported in Figure 3.14. Clearly, strides of 3, 4 and 5 are the most appropriate choices, when HEXA is not used. When HEXA is employed, large strides no longer remain effective in reducing the memory. This happens because a uni-bit HEXA trie requires just 2-bits of discriminator to represent a node, thus there is little room for further memory reductions by representing a subset of nodes with a

bitmap. In fact, with increasing stride, the bitmaps grow exponentially and quickly surpass any memory savings achieved with the tree bitmap based multi-bit nodes.

Note that smaller strides may not be acceptable in off-chip memory based implementations. However, in an embedded implementation such as pipelined trie [Basu, and G. Narlikar 2003], small stride may enable higher throughput, as reported in [Baboescu, Tullsen, Rosu, and Singh 2005]. This happens because with small stride, one can employ much deeper pipelines and each pipeline stage can be kept compact and fast.

Incremental Updates

We now present the results of incremental updates on tries represented with HEXA. In our experiments, we remove a trie node and add another to the trie, and then attempt to find a mapping for the newly added node. The general objective of triggering minimum changes in the existing mapping is achieved by finding the shortest augmenting path in the memory mapping graph, between the newly added node and some free memory location (as described in Section 3.1.4). We find that the shortest augmenting path indeed remains small, thus a small number of existing nodes are remapped. In Figure 3.15, we plot the probability distribution of the number of nodes that are remapped during an update of a single trie node. It is clear that no update is likely to take more than 19 memory operations and a large majority of updates require less than 10 memory operations. Thus, when prefixes are added or removed, update operations in a HEXA encoded trie can be carried out very quickly, irrespective of the trie shape and the update patterns.

3.4.3 Experimental Evaluation of CAMP

Before reporting our experimental results, we discuss some practical considerations that arise in a CAMP system.

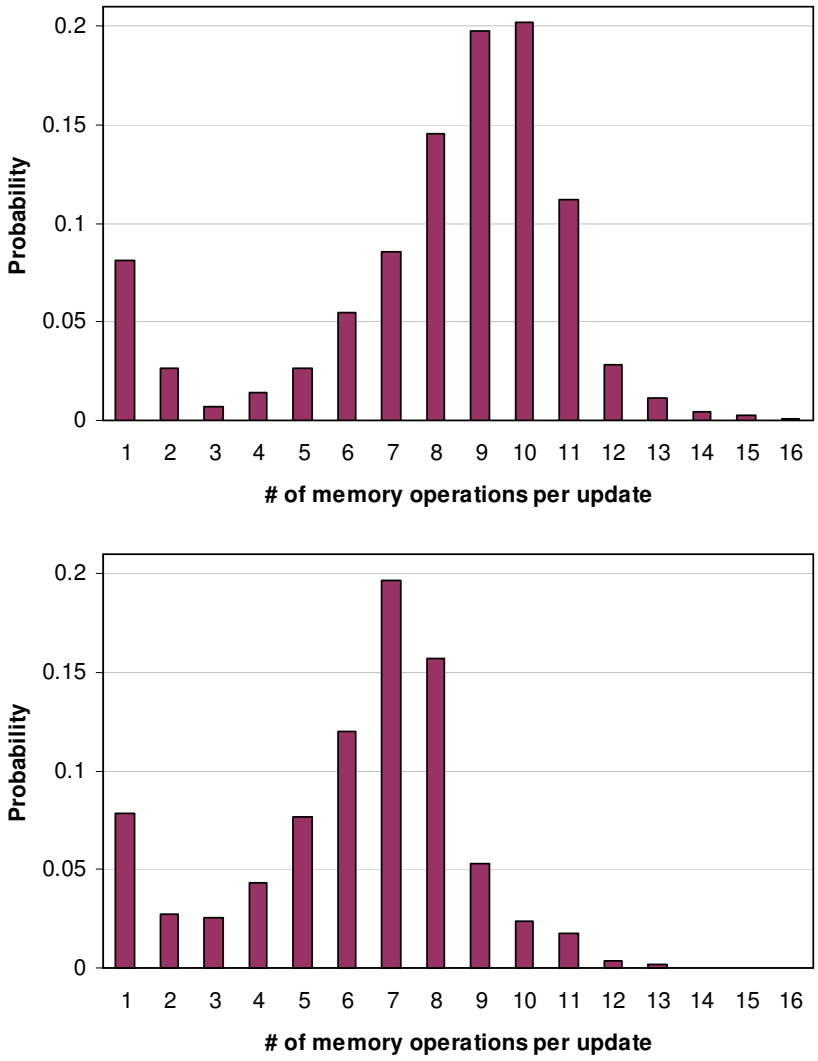


Figure 3.15 Probability distribution of the number of memory operations required to perform a single trie update. Upper trie size = 100,000 nodes, Lower trie size = 10,000 nodes.

Practical Considerations

Two important issues must be addressed when designing a CAMP pipeline: *i*) choice of the number of stages and *ii*) selection of the initial stride, which divides a trie into multiple sub-tries. We postpone the discussion of these3e issues to subsequent sections, and concentrate on another important design issue. For a given number of stages and

initial stride, how to dimension each stage and how does this compare with a linear pipeline?

To answer these questions, we determine the memory requirement of every pipeline stage for an array of routing tables in our dataset. Thereafter, from among all these data points, we compute the maximum memory requirement of every stage. Since some tables contain fewer prefixes than others, it is likely that they will require relatively less memory at each stage and hence may not contribute to the maximum computation. Therefore, we normalize the memory requirement of a stage for a given prefix set before considering it for the maximum computation. Thus the impact of a prefix set's size is eliminated but that of the prefix trends and length distribution are preserved. This gives us a first order estimate of the memory required at each pipeline stage for the today's prefix sets.

In Figure 3.16(a) we plot the normalized size of each stage of a CAMP pipeline for all routing tables (normalized with respect to the average pipeline stage size). The initial stride is set to 8, thus all subsequent uni-bit sub-tries require 25 pipeline stages. A "dot" represents the size of the corresponding pipeline stage for a prefix-set. The maximum size of each pipeline stage from among all dots is shown as an envelope in solid line. In Figure 3.16(b) and (c), we draw similar plots for a linear pipeline using a level-to-stage and height-to-stage mapping, respectively. We then add up the maximum size of each stage, represented by the envelope. This provides us the total memory overhead of each scheme (printed in the same plots). It can be noted that CAMP has a total memory overhead of 2.4% as compared to 23% in the height-to-stage mapping and 31% in level-to-stage mapping. Thus, not only does CAMP allow a more balanced distribution of nodes to stages (highlighted by Figure 3.16), but it also reduces the total memory.

Initial Stride and Number of Sub-tries

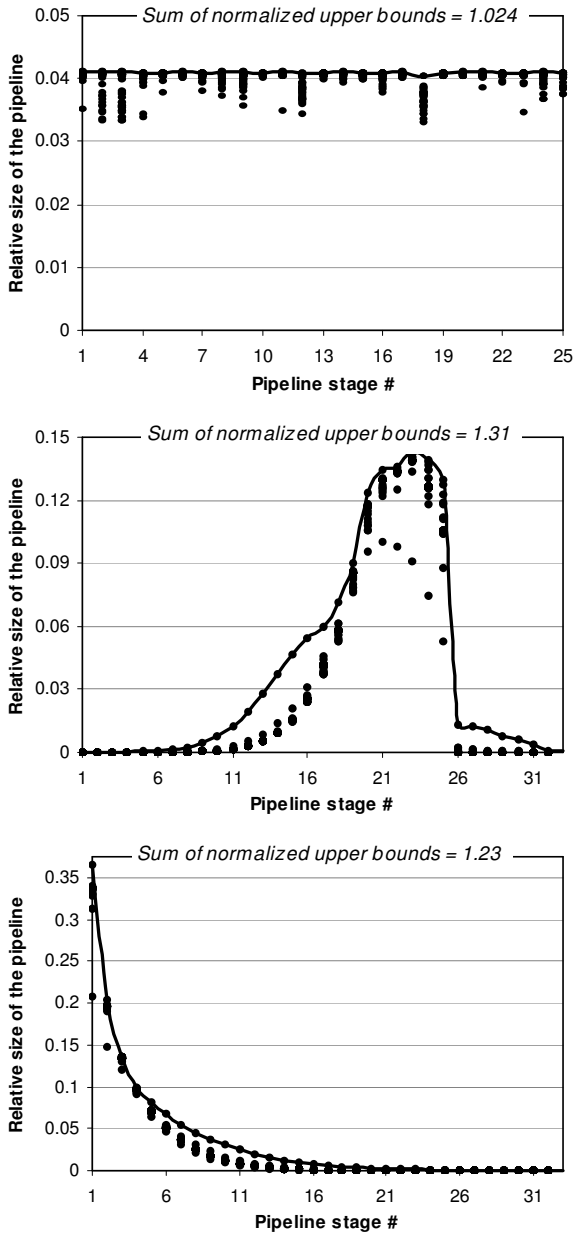


Figure 3.16 Normalized memory requirements of each pipeline stage in a binary trie a) CAMP using largest first heuristic, b) level to pipeline stage mapping, c) height to stage mapping. Leaf pushing was not done in these experiments.

The selection of the initial stride determines the number of sub-tries a trie will be split into. Specifically, an initial stride of k will lead to up to 2^k sub-tries. A large number of sub-tries generally leads to more balanced pipeline stages. On the same

dataset used in the previous analysis, we verified that the 2.4% memory overhead reported for an initial stride of 8 reduces to 0.02% and 0.01% for initial strides of 12 and 16, respectively. Larger initial strides, however, come at a cost. The direct indexed array which processes the initial k-bits and selects a sub-trie has 2^k entries. Therefore, an initial stride of 12, which requires a 4K entry table, is preferable over 16, which requires a 64K entry table.

Incremental Updates

From the previous discussion it is clear that the CAMP mapping algorithm leads to uniform pipeline utilization once an appropriate initial stride is chosen. We now study the effect of updates, which may disturb a balanced system. The goal of the discussion is twofold: first, we seek to evaluate the degree of imbalance that can be introduced by incremental updates in extreme scenarios; second, we seek to determine a bound on the extra memory needed to compensate for the imbalance.

An extreme (and unlikely) scenario is created by considering a subset of BGP tables, each containing nearly 105,000 prefixes, and simulating a sequence of migrations from one table to the other. The system begins in a balanced state (an initial stride of 12 is assumed) and each successive migration incrementally removes all prefixes belonging to the previous table and adds the ones present in the new table. During migration, the node to stage assignment of already existing sub-tries is preserved (and extended to the newly added nodes of the same sub-tries), while the roots of the newly added sub-tries are assigned a random stage. The results of these experiments are reported in Figure 3.17, where several distinct simulations have been run starting from a different routing table. The sizes of the smallest and the largest pipeline stage, normalized with respect to the total table size, are shown by a sequence of min-max data points in black-gray shade. The upper and lower envelope of all min-max data points is drawn in the same plot. It is clear that, even in this extreme case, the imbalance leads to only a 4% increase in the occupancy of the largest stage.

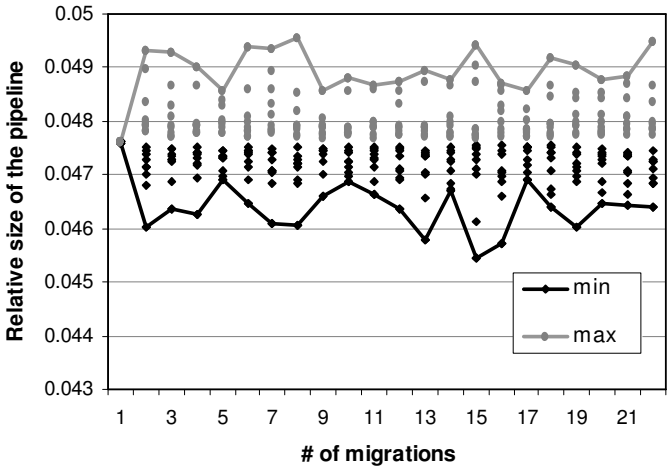


Figure 3.17 Successive migrations between a set of 22 distinct BGP tables. The upper and lower bound of the relative pipeline size are highlighted.

A more realistic scenario has been created by considering monthly snapshots of the rrc00 routing table over time, from 2002 till 2006 [Routing Information Service], during which, the table grew from 90126 prefixes to 135520 prefixes. Two cases are considered: in the first one, a balanced node to stage assignment is performed at the beginning (2002) and incremental updates are carried out until 2006 without any intermediate rebalancing. In the second case, the system is rebalanced, once every year, with a new (and balanced) node to stage assignment. Figure 3.18, reporting the result of this experiment, can be read as Figure 3.17 with the difference that the x-axis now reports the timestamp of each table snapshot. Without rebalancing, the maximum variation in the occupancy of the largest memory stage is 6%, while with rebalancing it is 4%. Note that such variation decreases every year; in particular, it is limited to less than 1% after 2006. In fact, as the routing table grows and the trie becomes relatively denser, it becomes more difficult to disturb a balanced system.

We conclude that, even in extreme update scenarios, the occupancy of a CAMP pipeline stage can increase only marginally. Hence, small memory over-provisioning should be adequate. Although there are effective methods to rebalance a CAMP system in face of

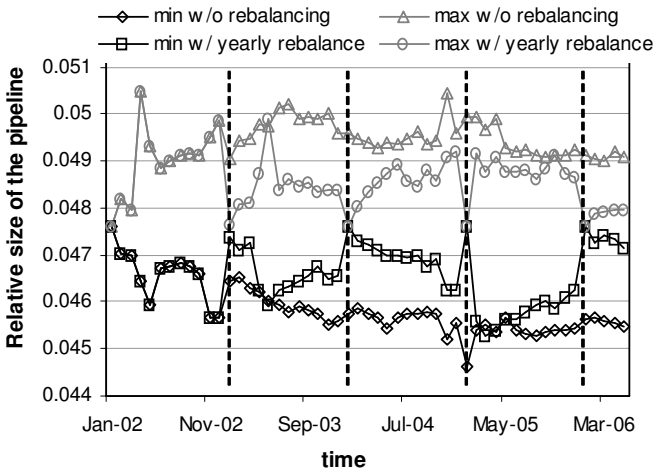


Figure 3.18 Effect of incremental updates over time; two scenarios are represented: once without and one with yearly rebalancing.

real-time incremental updates, the limited amount of imbalance and the infrequent need of rebalancing renders them not worthwhile.

Multi-bit Tries

Until now, we have only considered a uni-bit trie lookup. We now extend our evaluation to multi-bit tries where tree-bit maps are used to represent multi-bit nodes. The first design issue is to determine a stride which minimizes the total memory. We accomplish this experimentally by applying different strides on our datasets and measuring the total memory. The results are reported in Figure 3.19. It is obvious that strides of 3, 4 and 5 are the most appropriate choices.

When selecting the stride from among the three choices above, CAMP has relatively higher flexibility than a linear pipeline. In the case of a linear pipeline, a higher stride will reduce the number of memory stages, which may increase the size of each stage. A linear pipelined trie will therefore generally prefer conservative strides (*e.g.*: 3) so as to keep the bottleneck stage smaller, even though this may lead to non optimal total memory. CAMP, on the other hand, may choose a relatively larger stride due to the fact that pipeline stages are uniformly sized and no single stage is the bottleneck.

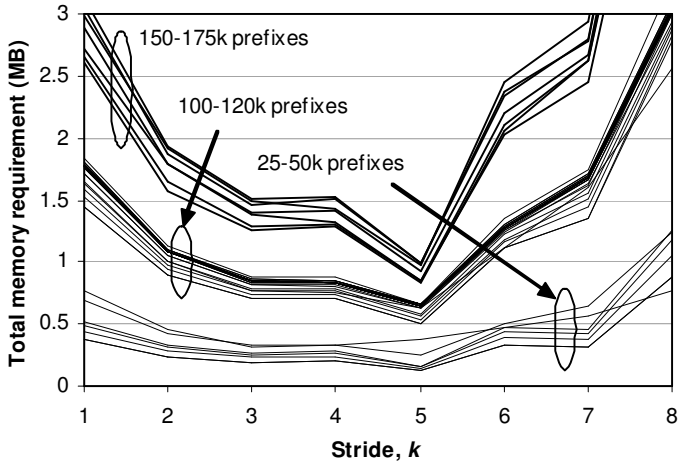


Figure 3.19 Total memory requirements of a tree-bit mapped multi-bit trie with different stride values (to highlight the properties of CAMP, we do not use HEXA in this experiment).

Additionally, as we will show in the next subsection, CAMP exhibits more flexibility in selecting the number of pipeline stages which can reduce their size independent of the adopted stride.

Number of Pipeline Stages

A key property of CAMP is that the number of pipeline stages can be different from the number of trie levels. This enables a trie data-structure to be pipelined to many more stages. Besides reducing the size of each pipeline stage (thus enabling them to run faster), more stages also improves the overall LPC, leading to a higher throughput. On the other hand, a large number of stages may lead to a less balanced distribution of nodes across different stages.

We experimentally quantify the impact of the number of stages on the node distribution. We keep an initial stride at 9 and the stride of each sub-trie is 5. In Figure 3.20, we vary the number of pipeline stages from 6 thru 30 and plot the excessive nodes allocated to the largest pipeline stage (percentage of the average number of nodes in a stage). Clearly, more stages results in higher imbalance as the largest stage is relatively

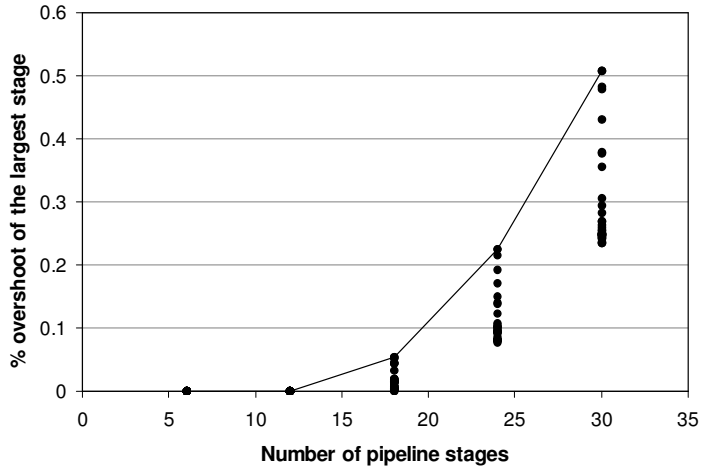


Figure 3.20 Percentage overshoot of size of the largest pipeline stage from the average pipeline stage size.

more occupied. However, note that, even for 30 pipeline stages, the largest stage is less than 1% bigger than the average stage. Therefore, we can conclude that the overall impact of higher number of pipeline stages on the node distribution is very nominal.

Power Dissipation and Area Estimates

We now characterize the power dissipation and die area of a CAMP and HEXA system. The analysis is carried out assuming a 0.09μm CMOS process and using CACTI3.2 [Shivakumar, Jouppi 2001]. The evaluation considers large synthetic prefix sets, besides our original dataset. We allocate an additional 25% memory to account for pathological conditions which may arise in the future. Wherever there is choice, we pick an optimum memory configuration (number of banks and clock frequency), which meets a given throughput objective. Finally, throughout the experiments, we use a tree-bit mapped multi-bit trie of stride between 2 and 4, whichever.

In Figure 3.21(a), we plot the power dissipation of the system for different link rates. As shown, the power dissipation for 1 million prefixes is 7 Watts when a 5-stage pipeline is used, and drops down to 3.4 Watts when a 10-stage pipeline is used. This can be

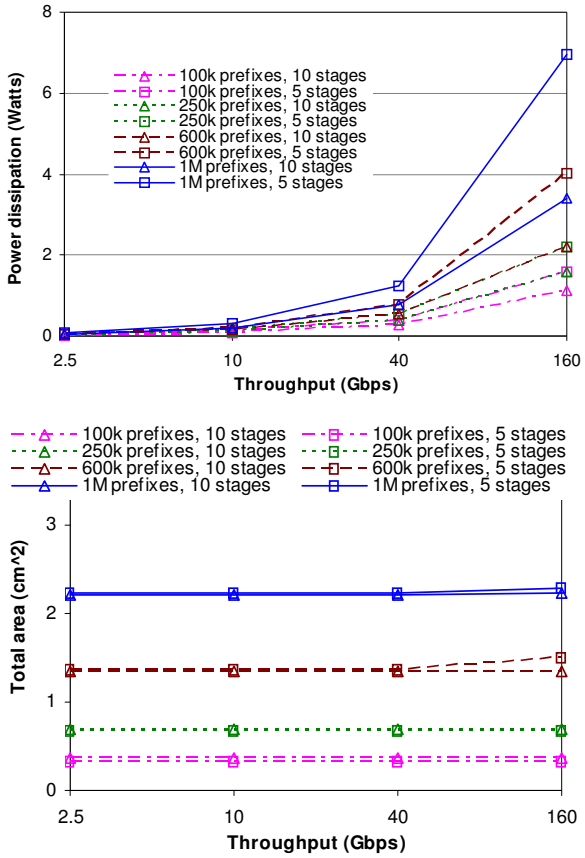


Figure 3.21 Power consumption and area estimates of different configurations.

explained as follow. The size of each stage is halved (from 1.6 MB to 0.8 MB) when doubling the number of stages. A single memory bank of these sizes has an access time of 4.2 ns and 2.2 ns, respectively. Therefore, achieving a 160 Gbps throughput requires a 4-bank and 2-bank memory, respectively, the former consuming 33% more energy per clock cycle. A smaller number of stages also lead to a lower LPC, thus requiring clocking the memory at higher rates.

Another interesting observation is that 1 million prefixes on a 10 stage pipeline dissipates less power than 600k prefixes on a 5 stage pipeline. In order to obtain an optimum number of pipeline stages which minimizes the power dissipation, we measure the power dissipation while varying the number of stages. In Figure 3.22 we plot the

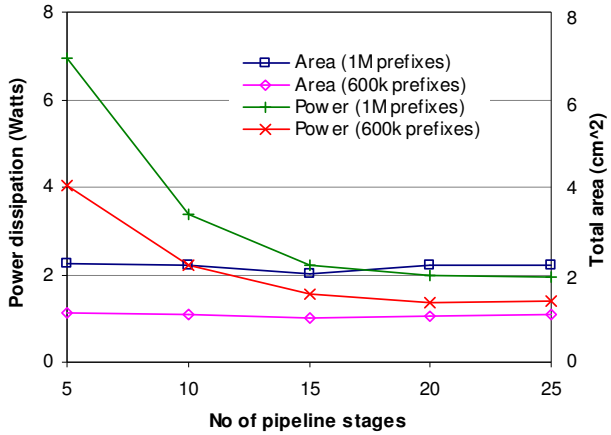


Figure 3.22 Power consumption of different configurations.

power dissipation of a 1 million and 600k prefix system providing 160 Gbps throughput. Power dissipation clearly drops as we increase the number of stages, however, beyond 15 stages, the reductions are nominal. It happens because every stage is 0.5 MB in a 15-stage pipeline, and a single memory bank of this size has access time of less than 2 ns, sufficient to provide 160 Gbps. Beyond 25 stages, the increase in power due to the increased number of individual components exceeds the reductions due to higher LPC. Hence, the overall power dissipation begins to increase.

The cost and yield of an ASIC depends very much on the die size; therefore, we also quantify the die size of the system. In Figure 3.21(b), we plot the area in cm^2, required by a 5- and 10-stage pipeline for different link rates. As expected, a larger number of prefixes results in proportionally larger area. We report the area requirements as a function of the number of pipeline stages in Figure 3.22, which suggests that as the number of stages increases, area first decreases and then increases after a certain point. However, the area sensitivity is small, because area is mostly independent of the LPC and clock frequency and only loosely coupled to the number of banks.

3.5 Worst-case Scenarios and Discussion

Since both CAMP and HEXA solutions are probabilistic – HEXA relies upon the randomness of the hash function; CAMP relies upon the trie shape – we discuss some worst-case scenarios and the likelihood of their occurrences in order to evaluate the vulnerability of these solutions. A key distinction between the vulnerability of HEXA and CAMP is that in HEXA, due to the use of hash function, the likelihood of finding a mapping of nodes to memory locations becomes independent of the prefix database, while in CAMP the imbalance in the memory utilization resulting from the node to pipeline stage mapping depends entirely upon the prefixes (trie shape). Consequently, memory efficiency in CAMP is vulnerable to the worst-case datasets; to tackle this problem, we describe an extension of CAMP, which enables robust and balanced memory mapping independent of the trie shape. In HEXA, the mapping performance remains robust and independent of the prefix database; therefore, we limit our evaluation to a brief analysis, which we present first.

The memory mapping process in HEXA uses a (pseudo)-random hash function to map HEXA identifiers to memory locations; therefore it can be reduced to the well studied and understood balls and bins problem. For n trie nodes, there are n balls, and for m memory locations, there are m bins. For a c-bit discriminator, each ball has 2^c pseudo-random choices of bins to pick from, and the goal is to insert each ball in some bin. For $2^c = 3$, and $m = 1.1n$, it has been shown that the likelihood that some balls do not find a bin is a negative exponential in n [Pagh, and Rodler 2001]. (for a million nodes trie, this probability is lower than the probability that there is a power failure across the entire globe.) A catch in this analysis is that it assumes a uniformly random hash function; practical hash functions are however (pseudo)-random. While several well known pseudo-random hash function exhibit a high degree of randomness, they also present a security threat. If the hash function is known, an attacker can propagate such IP prefixes that create a trie whose nodes and their HEXA identifiers lead to too many hash collisions, and the memory mapping will fail. A standard solution is to use a secret

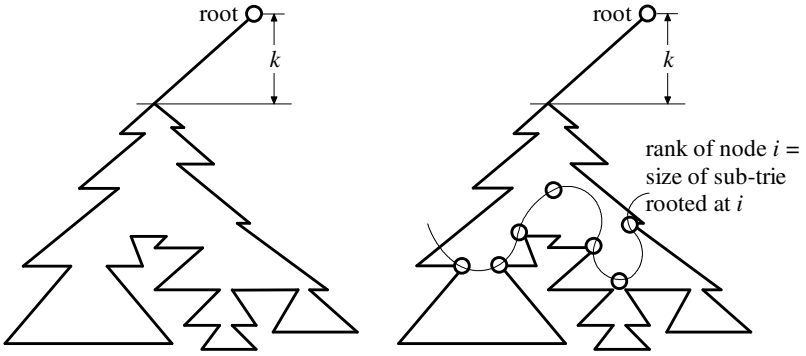


Figure 3.23 a) a worst-case prefix set, b) the way adaptive CAMP splits a trie into a parent and multiple child sub-tries.

key as an auxiliary input to the hash function. If the secret is sufficiently long, then it will become difficult for an attacker to create such anomalous conditions.

With this straightforward solution to make HEXA secure and robust, we now shift our attention to CAMP in which the mapping efficiency relies on the shape of the trie, leaving it more vulnerable to anomalous conditions. In our experiments, we have solely considered practical routing tables; we now consider such datasets that present a potential threat in mapping, and can cause severe imbalance in the memory utilization. Such worse-case conditions clearly arise when it is not trivial to split a trie into multiple sub-tries and uniformly map them to different stages. Recall that we divided a single trie into up to 2^k sub-tries by separately considering the initial k-bits of the address. As shown in Figure 3.23(a), any trie which begins with a long skinny section is difficult to be split with this mechanism. If we attempt to split such a trie, it will require a large initial stride, which can make the direct index table ($2^{\text{initial stride}}$ entries) prohibitively large.

In order to handle these worst-case conditions, we propose an extension called adaptive CAMP, which allows a trie to be split into a parent sub-trie and multiple child sub-tries. This way, not only can we directly control the number and size of the sub-tries generated, but we can also ensure that the resulting sub-tries are equal in size. The process begins with assigning rank (total number of its descendents) to each node. We then distinguish all nodes whose rank is equal to the size of the sub-tries we want to

generate. These nodes form a sub-trie of which they are the root. We then remove these sub-tries from the original trie, and iteratively apply the process to this remaining trie. The procedure is illustrated on the above trie in Figure 3.23(b), where the trie is split into a parent sub-trie and multiple child sub-tries (the root node of each sub-trie is highlighted).

Once these sub-tries are generated, their nodes are mapped (both the child and parent sub-tries) to the pipeline stages. Pipeline stages are expected to be more balanced, as the sub-tries are of roughly equal size. A direct index table will no longer be required. Requests will be first dispatched into the pipeline to parse the parent sub-trie, and then another request will be dispatched to parse one of the child sub-tries. Such multiple request dispatches will clearly reduce the LPC and more pipeline stages will be required to mitigate this issue. Extended CAMP, thus, trades-off performance with robustness and security.

3.6 Concluding Remarks

In this chapter, we described the design of high performance architecture to implement longest prefix match. A unique characteristic of ASIC is that they can pack a limited number of memory bits on-chip; however, these bits can be configured in multiple memory modules to provide enormous amounts of bandwidth. To exploit this limited but fast pool of embedded memories and enable high performance, we combine two novel architectures, which we dub HEXA and CAMP. HEXA is a novel representation for structured graphs such as tries, and uses a unique method to locate the nodes of the graph in memory, which enables it to avoid using any "next node" pointer. Since these pointers often consume much of the memory required by the graph, HEXA based representations are significantly more compact, making them desirable in an ASIC. CAMP perfectly complements HEXA and utilizes the bandwidth provided by multiple embedded memories. To do so efficiently, CAMP uses a multi-point access circular

pipeline of memories; each stage stores a single or set of levels of the lookup trie and a stream of lookup requests are issued into the pipeline, one every cycle, in order to achieve high throughput. Circular structure provides much more flexibility in mapping nodes of the lookup trie to the stages, which in turn, improves the memory utilization and also reduces the total memory and power consumption.

Chapter 4

Packet Content Inspection I

In this chapter, we continue our endeavor to devise architectures that enable high performance by developing a novel approach to packet content inspection. Packet content inspection has recently gained popularity as it provides the capability to accurately classify and control traffic in terms of content, applications, and individual subscribers. Forwarding packets based on content (either for the purpose of application-level load-balancing in a web switch or security-oriented filtering based on content signatures) requires new levels of support in networking equipment. Traditionally, this deep packet inspection has been limited to comparing packet content to sets of strings. State-of-the-art systems, however, are replacing string sets with regular expressions, due to their increased expressiveness. Several content inspection engines have recently migrated to regular expressions, including: Snort, Bro, 3Com's TippingPoint X505, and various network security appliances from Cisco Systems. Additionally, layer 7 filters based on regular expressions are available for the Linux operating system. While flexible and expressive, regular expressions have traditionally required substantial amounts of memory, which severely limits performance in the networking context.

To see why, we must consider how regular expressions are implemented. A regular expression is typically represented by a deterministic finite automaton (DFA). For any regular expression, it is possible to construct a DFA with the minimum number of states. The memory needed to represent a DFA is, in turn, determined by the product of the number of states and the number of transitions from each state. For an ASCII alphabet, each state will have 256 outgoing edges. Typical sets of regular expressions

containing hundreds of patterns for use in networking yield DFAs with tens of thousands of states, resulting in storage requirements in the hundreds of megabytes. Standard compression techniques are not effective for these tables due to the relatively high number of unique 'next-states' from a given state. Consequently, traditional approaches quickly become infeasible as rule sets grow.

We introduce a compact representation for DFAs, which that enables packet content inspection to be implemented using an ASIC on-chip memory. Our approach reduces the number of transitions associated with each state. The main observation is that groups of states in a DFA often have very similar outgoing transitions and we can use this duplicate information to reduce memory requirements. For example, suppose there are two states s_1 and s_2 that make transitions to the same set of states, $\{S\}$, for some set of input characters, $\{C\}$. We can eliminate these transitions from one state, say s_1, by introducing a default transition from s_1 to s_2 that is followed for all the characters in $\{C\}$. Essentially, s_1 now only maintains unique next states for those transitions not common to s_1 and s_2 and uses the default transition to s_2 for the common transitions. We refer to a DFA augmented with such default transitions as a Delayed Input DFA (D^2FA).

In practice, the proper and effective construction of the default transitions leads to a tradeoff between the size of the DFA representation and the memory bandwidth required to traverse it. In a standard DFA, an input character leads to a single transition between states; in a D^2FA, an input character can lead to multiple default transitions before it is consumed along a normal transition.

Our approach achieves a compression ratio of more than 95% on typical sets of regular expressions used in networking applications. Although each input character potentially requires multiple memory accesses, the high compression ratio enables us to keep the data structure in the on-chip memory modules in an ASIC, where the increased bandwidth can be provided efficiently. We describe an ASIC architecture that employs a

modest of amount on-chip memory, organized in multiple independent modules. We use multiple embedded memories to provide ample bandwidth. However, in order to deterministically execute the compressed automata at high rates, it is important that the memory modules are uniformly populated and accessed over short periods of time. To this end, we develop load balancing algorithms to map our automata to the memory modules in such a way that deterministic worst-case performance can be guaranteed. Our algorithms can maintain throughput at 10 Gbps while matching thousands of regular expressions.

To summarize, we propose *a*) the D^2FA representation of regular expressions which significantly reduces the amount of memory required, *b*) an ASIC architecture that uses the D^2FA representation, and *c*) a load balancing algorithm which ensures that on-chip resources on the ASIC are uniformly used, thereby enabling worst-case performance guarantees.

The remainder of the chapter is organized as follows. Section 4.1 describes the D^2FA representation. Details of our construction algorithm and the compression results are presented in Section 4.2. Section 4.3 presents the system architecture, load balancing algorithms and throughput results. We summarize the architecture in Section 4.4. Finally, in Section 4.5, we present preliminary research results of applying HEXA to finite automata.

4.1 Delayed Input DFAs

It is well-known that for any regular expression set, there exists a DFA with the minimum number of states [Hopcroft 1971]. The memory needed to represent a DFA is determined by the number of transitions from one state to another, or equivalently, the number of edges in the graph representation. For an ASCII alphabet, there can be up to 256 edges leaving each state, making the space requirements excessive. Table

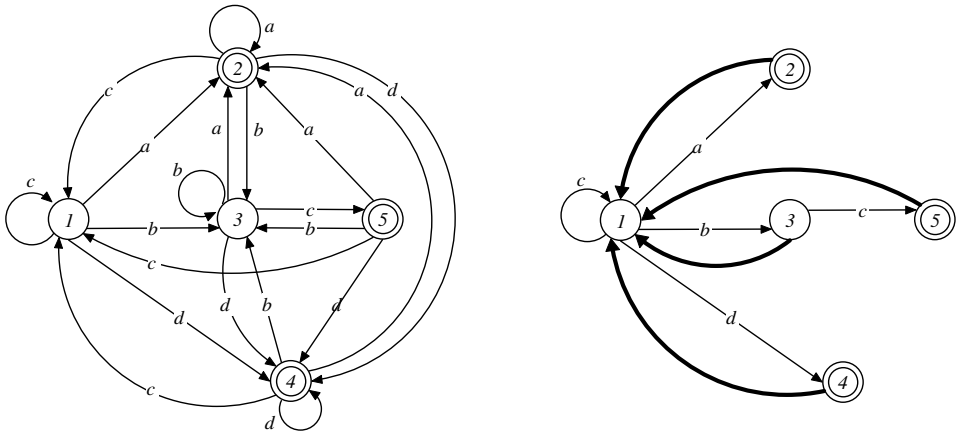


Figure 4.1 Example of automata which recognize the expressions a+, b+c, and c*d+

compression techniques can be applied to reduce the space in situations when the number of distinct "next-states" from a given state is small. However, in DFAs that arise in network applications, these methods are typically not very effective because on average, there are more than 50 distinct "next-states" from various states of the automaton.

We introduce a modification to the standard DFA that can be represented much more compactly. Our modifications are based on a technique used in the Aho-Corasick string matching algorithm [Aho, and Corasick 1975]. We extend their technique and apply it to DFAs obtained from regular expressions, rather than simple string sets.

4.1.1 Motivating Example

We introduce our approach using an example. The left side of Figure 4.1 shows a standard DFA defined on the alphabet {a,b,c,d} that recognizes the three patterns, p_1=a$^+$, p_2=b$^+$c, and p_3=c*d$^+$ (in these expressions, the asterisk represents 0 or more repetitions of the immediately preceding sub-expression, while the plus sign represents one or more repetitions). In this DFA, state 1 is the initial state, and states 2, 5 and 4 are match states for the three patterns p_1, p_2 and p_3, respectively.

The right side of Figure 4.1 shows an alternate type of DFA, which includes unlabeled edges that are referred to as default transitions. When matching an input string, a default transition is used to determine the next state, whenever the current state has no outgoing edge labeled with the current input character. When following a default transition the current input character is retained. Consider the operation of the two automata on the input string `aabdbc`. For this input, the sequence of states visited by the left-hand automaton is 1223435, where the underlined states are the match states that determine the output value for this input string. The right-hand automaton visits states 1212314135. Notice that the sequence of match states is the same, so if the second automaton output associates these states with the same three patterns, it produces the same output as the first one. Indeed, it is not difficult to show that the two automata visit the same sequence of match states for any input string. That is, they produce the same output, for all inputs and are hence equivalent.

Note that the right-hand automaton in Figure 4.1 has just nine edges, while the one on the left has 20. We find that for the more complex DFAs that arise in network applications, we can generally reduce the number of edges by more than 95%, dramatically reducing the space needed to represent the DFA. There is a price for this reduction of course, since no input is consumed when default edges are followed. In the example in Figure 4.1, no state with an incoming default transition also has an outgoing default transition, meaning that for every two edges traversed, we are guaranteed to consume at least one input character. Allowing states to have both incoming and outgoing default transitions leads to a more compact representation, at the cost of some reduction in the worst-case performance.

4.1.2 Problem Statement

We refer to an automaton with default transitions as a Delayed Input DFA (D^2FA). We represent a D^2FA by a directed graph, whose vertices are called states and whose edges are called transitions. Transitions may be labeled with symbols from a finite alphabet Σ. Each state may have at most one unlabeled outgoing transition, called its default transition. One state is designated as the initial state and for every state s, there is a (possibly empty) set of matching patterns, $\mu(s)$.

For any input string $x \in \Sigma^*$, we define the destination state, $\delta(x)$ to be the last state reached by starting at the initial state and following transitions labeled by the characters of x, using default transitions whenever there is no outgoing transition that matches the next character of x (so, for the D^2FA on the right side of Figure 4.1, $\delta(\texttt{abcb})=3$ and $\delta(\texttt{dcbac})=1$). We generalize δ to accept an arbitrary starting state as a second argument; so for the D^2FA on the right side of Figure 4.1, $\delta(\texttt{abcb},2) =3$.

Consider two D^2FAs with destination state functions δ_1 and δ_2, and matching pattern functions μ_1 and μ_2. We say that the two automata are equivalent if for all strings x, $\mu_1(\delta_1(x))=\mu_2(\delta_2(x))$. In general, given a DFA that recognizes some given set of regular expressions, our objective is to find an equivalent D^2FA that is substantially more memory-efficient.

We can bound the worst-case performance of a D^2FA in terms of the length of its longest default path (that is, a path comprising only default transitions). In particular, if the longest default path has k transitions, then for all input strings, the D^2FA will consume at least one character for every k transitions followed. To ensure that a D^2FA meets a throughput objective, we can place a limit on the length of the longest default path. This leads to a more refined version of the problem, in which we seek the smallest equivalent D^2FA that satisfies a specified bound on default path length.

4.1.3 Converting DFAs to D^{2}FAs

Although, we are in general interested in any equivalent D^{2}FA, for a given DFA, we have no general procedure for synthesizing a D^{2}FA directly. Consequently, our procedure for constructing a D^{2}FA proceeds by transforming an ordinary DFA, by introducing default transitions in a systematic way, while maintaining equivalence. Our procedure does not change the state set, or the set of matching patterns for a given state. Hence, we can maintain equivalence by ensuring that the destination state function $\delta(x)$, does not change.

Consider two states u and v, where both u and v have a transition labeled by the symbol a to a common third state w, and no default transition. If we introduce a default transition from u to v, we can eliminate the a-transition from u without affecting the destination state function $\delta(x)$. A slightly more general version of this observation is stated below.

Lemma 1. Consider a D^{2}FA with distinct states u and v, where u has a transition labeled by the symbol a, and no outgoing default transition. If $\delta(a,u)=\delta(a,v)$, then the D^{2}FA obtained by introducing a default transition from u to v and removing the transition from u to $\delta(a,u)$ is equivalent to the original DFA.

Note that by the same reasoning, if there are multiple symbols a, for which u has a labeled outgoing edge and for which $\delta(a,u)=\delta(a,v)$, the introduction of a default edge from u to v allows us to eliminate all these edges. Our procedure for converting a DFA to a smaller D^{2}FA applies this transformation repeatedly. Hence, the equivalence of the initial and final D^{2}FAs follows by induction. The D^{2}FA on the right side of Figure 4.1 was obtained from the DFA on the left, by applying this transformation to state pairs (2,1), (3,1), (5,1) and (4,1).

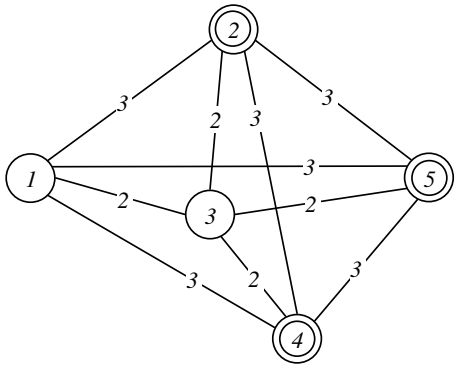


Figure 4.2 Space reduction graph for DFA in Figure 4.1.

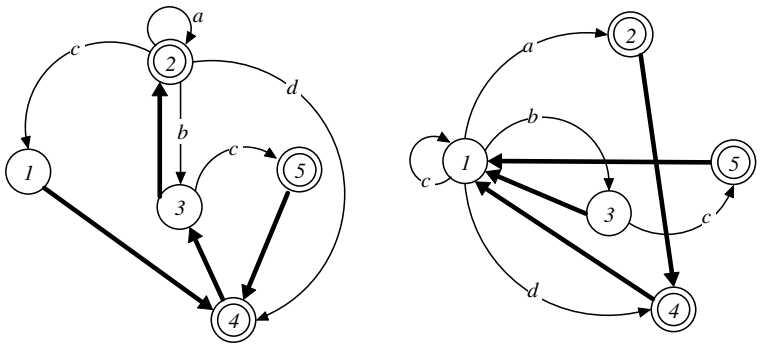


Figure 4.3 D2FAs corresponding to two different maximum weight spanning trees.

For each state, we can have only one default transition, so it's important to choose our default transitions carefully to allow us to get the largest possible reduction. We also restrict the choice of default transitions to ensure that there is no cycle defined by default transitions. With this restriction, the default transitions define a collection of trees with the transitions directed towards the tree roots and we can identify the set of transitions that gives the largest space reduction by solving a maximum weight spanning tree problem in an undirected graph which we refer to as the space reduction graph.

The space reduction graph for a given DFA is a complete, undirected graph, defined on the same vertex set as the DFA. The edge joining a pair of vertices (states) u and v is assigned a weight $w(u,v)$ that is one less than the number of symbols `a` for which

$\delta(a,u)=\delta(a,v)$. The space reduction graph for the DFA on the left side of Figure 4.1 is shown in Figure 4.2. Notice that the spanning tree of the space reduction graph that corresponds to the default transitions for the D^{2}FA in Figure 4.1 has a total weight of 3+3+3+2=11, which is the difference in the number of transitions in the two automata. Also, note that this is a maximum weight spanning tree for this graph. Figure 4.3 shows D^{2}FAs corresponding to two different maximum weight spanning trees. Note that while these two automata use the same number of edges as the one in Figure 4.1, they have default paths of length 3 and 2, respectively, meaning that their worst-case performance will not be as good.

4.2 Bounding Default Paths

If our only objective was minimizing the space used by a D^{2}FA, it would suffice to find a maximum weight spanning tree in the space reduction graph. The tree edges correspond to the state pairs between which we create default transitions. The only remaining issue is to determine the orientation of the default transitions. Since each vertex can have only one outgoing default transition, it suffices to pick some arbitrary state to be the root of the default transition tree and direct all default transitions towards this state.

Unfortunately, when this procedure is applied to DFAs arising in typical network applications, the resulting default transition tree has many long paths, implying that the D^{2}FA may need to make many transitions for each input character consumed. We can improve the performance somewhat, by selecting a tree root that is centrally located within the spanning tree. However, this still leaves us with many long default paths. The natural way to avoid long default paths is to construct a maximum weight spanning tree with a specified bounded diameter. Unfortunately, the construction of such spanning trees is NP-hard [Garey, and Johnson 1979]. It's also not clear that such a spanning tree leads to the smallest D^{2}FA. What we actually require is a collection of bounded diameter

trees of maximum weight. While this problem can be solved in polynomial time if the diameter bound is 1 (this is simply maximum weight matching), the problem remains NP-hard for larger diameters.

Fortunately, we have found that fairly simple methods, based on classical maximum spanning tree algorithms, yield good results for D^2FA construction. One conceptually straight-forward method builds a collection of trees incrementally. The method (which is based on Kruskal's algorithm [Kruskal 1956]) examines the edges in decreasing order of their weight. An edge $\{u,v\}$ is selected as a "tree-edge" so long as u and v do not already belong to the same tree, and so long as the addition of the edge will not create a tree whose diameter exceeds a specified bound. Once all the edges have been considered, the tree edges define default transitions. We orient the default transitions in each tree by directing them towards a selected root for that tree, where the roots are selected so as to minimize the distance to the root from any leaf.

The one complication with this method is checking the diameter bounds. We can do this efficiently by maintaining for each vertex u a value $d(u)$ which specifies the number of edges in the longest tree path from u to a vertex in the same tree. These values can be used to check that the addition of a new edge will not violate the diameter bound. When a new tree edge is added, the distance values must be updated for vertices in the tree formed by the addition of the new edge. This can be done in linear time for each update. Consequently, the total time needed to maintain the distance values is $O(n^2)$. Since Kruskal's algorithm, on which our algorithm is based, requires $O(n^2 \log n)$ time on complete graphs, the diameter checking does not increase the asymptotic running time of the algorithm.

One refinement to this fairly simple algorithm is shown below. While examining the edges in decreasing order of their weights, we also look for an edge among all equal weight edges, which results in the minimum expansion in the diameter of the trees joined. In practice, since there are only 255 different weight values, at any point in time,

there will often be plenty of equal weight edges to choose from. The resulting refined algorithm begins with the weighted undirected space reduction graph $G=(V,W)$ and modifies an edge set *tree_edges* which form the default transition trees. First it considers all edges of weight 255, and incrementally constructs default trees of small diameters. Then it repeatedly considers smaller weight edges and adds them to the default transition trees.

It turns out that the refinement generally leads to default transition trees with significantly smaller diameter as compared to a normal spanning tree, which remains oblivious to the diameter of the trees until the diameter bound is reached. In a setup, where the diameter bound is not applied, the refined spanning tree algorithm creates default transition trees of equal weight but relatively smaller diameter. When the

```
procedure refinedmaxspantree (graph G=(V, W), modifies set edge tree_edges);
(1)  vertex u, v;
(2)  set edges;
(3)  set weight-set[255];
(4)  tree_edges := { }; edges := W;
(5)
(6)  for edge (u, v) ∈ edges ⇒
(7)     if weight(u, v) > 0 ⇒
(8)           add (u, v) to weight-set[weight(u, v)];
(9)     fi
(10)
(11) for integer i = 255 to 1 ⇒
(12)    do weight-set[i] ≠ [ ] ⇒
(13)          Select (u, v) from weight-set[i] which leads to the
(14)          smallest growth in the diameter of the tree_edges trees
(15)          if vertices u and v belongs to different default trees ⇒
(16)                if tree_edges ∪ (u, v) maintains the diameter bound ⇒
(17)                      tree_edges := tree_edges ∪ (u, v);
(18)                fi
(19)          fi
(20)    od
(21) rof
end;
```

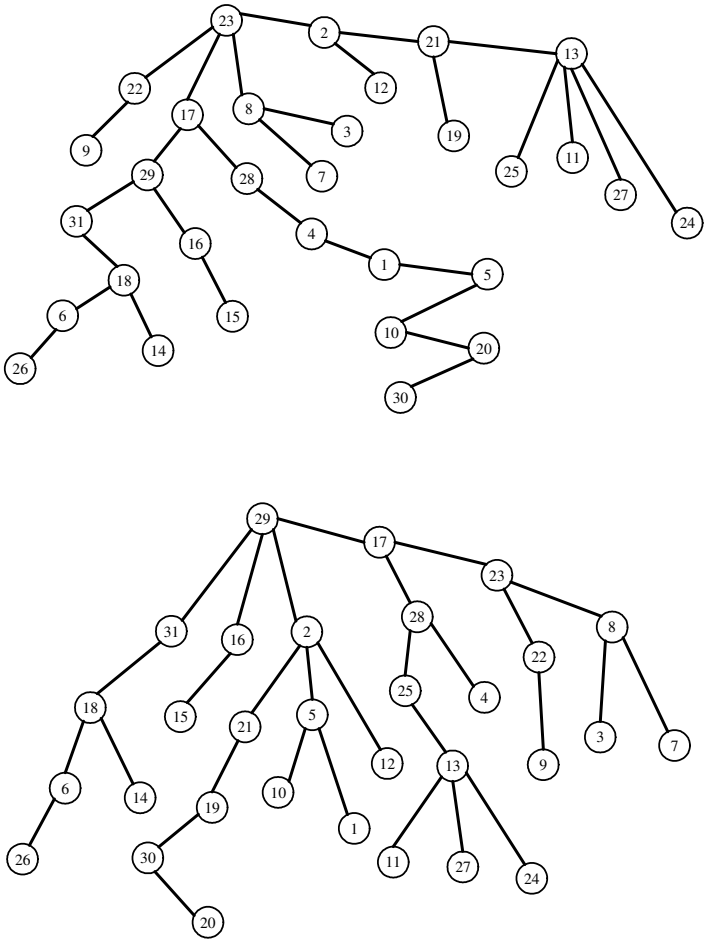


Figure 4.4 Default transition trees formed by the spanning tree algorithm and by the refined version.

diameter bound is applied, the refined algorithm creates trees with higher weight too. This happens, because a normal spanning tree, in its process, quickly creates several trees whose diameter is "too large" and hence can not be further linked to any tree. The refined version ensures that tree diameter remains small; hence more trees can be linked, resulting in higher weight.

In order to illustrate the effect of this refinement, we take a synthetic DFA, which consists of 31 states. All pairs of states *u* and *v* were assigned transitions on a random number (drawn from a geometric distribution with success probability, $\pi = 0.05$, thus

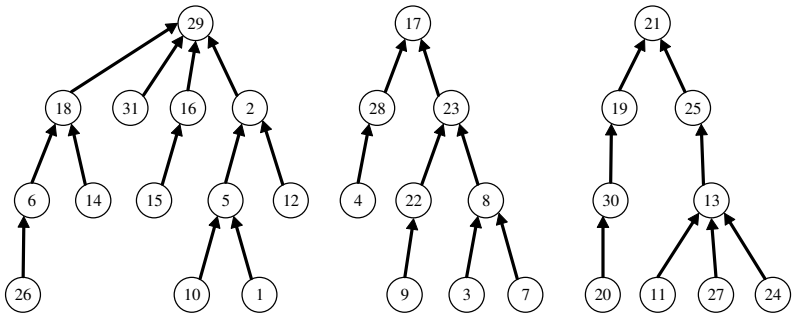


Figure 4.5 Default transition trees (forest) formed by the refined spanning tree with the tree diameter bounded to 6.

mean, $E(X) = 19$) of symbols a such that $\delta(a,u)=\delta(a,v)$. Thus the weight of the edges in the space reduction graph was geometrically distributed. When we ran the normal and refined versions of the default trees construction algorithms without any diameter bound, they created spanning trees of weight 1771, as shown in Figure 4.4. While the weights of both trees are maximum, their diameters are 13 and 10 respectively. If we choose nodes 28 and 29, respectively, as the roots of these two trees, the longest default paths contain 7 and 5 edges, while the average length of default paths are 3.8 and 2.8, respectively.

Clearly, the refinement in the spanning tree algorithm reduces the memory accesses needed by a D^2FA for every character. We will later see that when diameter bounds are applied, the refined algorithm creates more compact D^2FAs as well.

When we bounded the diameter of the trees to 6, and ran our algorithm on the same synthetic DFA, it created three default transition trees, as shown in Figure 4.5. The total weight of all three trees was 1653, which suggests that the resulting D^2FA will require slightly more space as compared to the one with no diameter restraint. However, bounding the diameter to 6 ensures an important property that the length of all default paths can be easily limited to 4 and hence the D^2FA will require at most 4 memory accesses per character.

4.2.1 Results on Some Regular Expression Sets

In order to evaluate the space reductions achieved by a delayed input DFA, or D^2FA, we performed experiments on regular expression sets used in a wide variety of networking applications. Our most important datasets are the regular expression sets used in deep packet inspection appliances from Cisco Systems [Eatherton, and Williams 2005]. This set contains more than 750 moderately complex expressions, which are used to detect the anomalies in the traffic. It is widely used across several Cisco security appliances and Cisco commonly employs general purpose processors with a gigabyte or more of memory to implement them. In addition to this set, we also considered the regular expressions used in the open source Snort and Bro NIDS, and in the Linux layer-7 application protocol classifier. The Linux layer-7 protocol classifier consists of 70 expressions. Snort contains more than 1500 expressions, although, they don't need to be matched simultaneously. An effective way to implement the Snort rules is to identify the expressions for each header rule and then group the expressions corresponding to the overlapping rules (the set of header rules a single packet can match to). We use this approach. For the Bro NIDS, we present results for the HTTP signatures, which consist of 648 regular expressions.

Given these regular expression sets, as the first step in constructing DFAs with a small number of states, we used the set splitting techniques proposed in [Yu, et al. 2005]. This approach splits the regular expressions into multiple sets so that each set creates a small DFA. We created 10 sets of rules from the Cisco regular expressions, and were able to reduce the total memory footprint to 92 MB, as there were a total of 180138 states, and each individual DFA had less than 64K states, (thus 2 bytes encodes a state). Clearly, such an efficient grouping resulted in a significant space reduction over more than a gigabyte space required otherwise. We split the Linux layer-7 expressions into three sets, such that the total number of states was 28889. For the Snort set, we present results for the header rule "`tcp $EXTERNAL_NET any -> $HTTP_SERVERS`

Table 4.1 Our representative regular expression groups.

Source	# of regular expressions	Avg. ASCII length of expressions	% expressions using wildcards (*, +, ?)	% expressions length restrictions {,k,+}
Cisco	590	36.5	5.42	1.13
Cisco	103	58.7	11.65	7.92
Cisco	7	143.0	100	14.23
Linux	56	64.1	53.57	0
Linux	10	80.1	70	0
Snort	11	43.7	100	9.09
Snort	7	49.57	100	28.57
Bro	648	23.6	0	0

`$HTTP_PORTS,`" which consists of 22 complex expressions. Since Snort rules were complex, with long length restriction on various character classes, we applied rewriting techniques proposed in [Yu, et al. 2005] to some rules and split them further into four sets. Bro regular expressions were generally simple and efficient therefore we were able to compile all of them in a single automaton. The key properties of our representative regular expression groups are summarized in Table 4.1.

In order to estimate the reduction objectives of D^2FA, we introduce a term redundancy. There is redundancy if there exist multiple transitions from different states leading to the same "next state" for the same input character. For example in Figure 4.1, there are transitions from state 1, 2, 3, 4 and 5 all leading to the same next state on input *b*. So, there are 4 redundant states. Even though, it may not be possible to eliminate all redundant transitions, it still gives a good estimate on the upper bound of the number of transitions that can be eliminated by constructing a D^2FA from the DFA.

After constructing the minimum state DFAs from these regular expressions, we used both normal and refined versions of the default trees construction algorithms to construct the corresponding D^2FAs. The reduction in the number of transitions is shown in Table 4.2 with no diameter bounds applied. The length of default paths are

Table 4.2 Original DFA and the D2FA constructed using the basic and the refined default tree construction algorithm, without any diameter bound.

	Original DFA				
rules	*Total # of states*	*Total # of transitions*	*Total # of distinct transitions*	*Total # of redundant transitions*	*% duplicates*
Cisco590	17,713	4,534,528	1,537,238	4,509,852	99.45
Cisco103	21,050	5,388,800	1,236,587	5,346,595	99.21
Cisco7	4,260	1,090,560	312,082	1,063,896	97.55
Linux56	13,953	3,571,968	590,917	3,517,044	98.46
Linux10	13,003	3,328,768	962,299	3,052,433	91.69
Snort11	41,949	10,738,944	540,259	10,569,778	98.42
Bro648	6,216	1,591,296	149,002	1,584,357	99.56

	Delayed input DFA, D^2FA							
	Basic Algorithm				**Refined Algorithm**			
rules	*Total # of transitions*	*% reduction*	*Avg. default length*	*Max. default length*	*Total # of transitions*	*% reduction*	*Avg. default length*	*Max. default length*
Cisco590	36,519	99.2	18.32	57	36,519	99.2	8.47	17
Cisco103	53,068	99.0	16.65	54	53,068	99.0	7.82	19
Cisco7	28,094	97.4	19.61	61	28,094	97.4	10.91	23
Linux56	58,571	98.3	7.68	30	58,571	98.3	5.62	21
Linux10	285,991	91.3	5.14	20	285,991	91.3	4.64	17
Snort11	168,569	98.4	5.86	9	168,569	98.4	3.43	6
Bro648	7,082	99.5	6.45	17	7,082	99.5	2.59	8

also shown. It is clear that, D^2FAs eliminates nearly all redundancy from the DFAs. It is also apparent that refined version of algorithm creates substantially smaller default paths as compared to the basic algorithm. In order to get a sense of the distribution of the number of labeled transitions per state of a D^2FA, we plot this quantity in Figure 4.6, for the Cisco regular expression group containing 590 expressions. The majority of states have 2 or fewer labeled transitions. Note that most states have 2 transitions because most rules are case insensitive, like `[d-eD-E0-9\-_][/\\][^/\\\r\n?\x26\s\t:]*[.][Nn][Uu].`

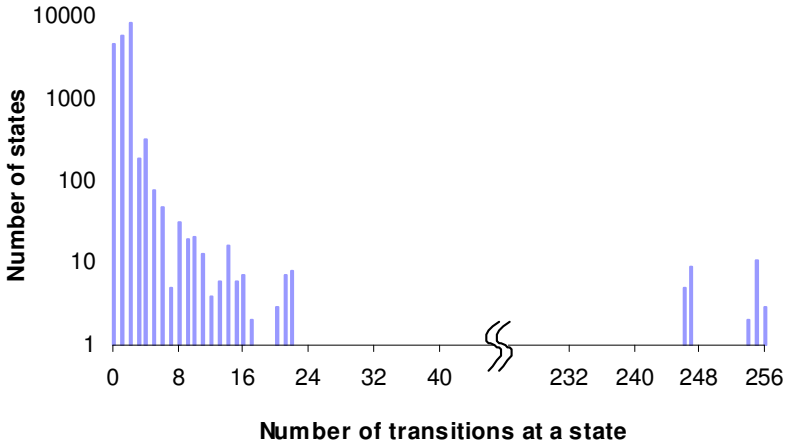


Figure 4.6 Distribution of number of transitions per state in the D2FA constructed from the Cisco590 expression set.

Since the above results are with no diameter restrictions, default transition paths are quite long. In order to achieve smaller default paths, we ran our algorithm with the diameter restricted to a small constant. In this case, we first compare the reductions achieved by both versions of default tree construction algorithm. In Table 4.3, we report the number of transitions in the resulting D^2FA, with the length of default paths bounded to 4 edges. Clearly, refined version of spanning tree yields relatively more compact D^2FA.

In Figure 4.7, we plot the reduction in the number of transitions of a DFA, as a ratio of number transitions in the D^2FA and the number of distinct transitions (transitions leading to distinct "next states") in the original DFA, by applying the refined version of spanning tree and bounding the default paths at different values. It is obvious that smaller default path restrictions produce D^2FAs with a higher number of labeled transitions. Note that, the reduction numbers plotted are with respect to the total number of distinct transitions (leading to different "next states") at various states in the original DFA, and not all transitions. Clearly this metric is conservative and suggests the space reduction by D^2FA over a DFA using the best (possibly hypothetical) table compression scheme which enables it to store only the distinct transitions. If we would use the total transitions in a DFA as our metric, D^2FA will result in even higher reduction.

4.2.2 Summarizing the Results

The results suggest that a delayed input DFA or D^2FA can substantially reduce the space requirements of regular expression sets used in many networking applications. For example, using a D^2FA, we were able to reduce the space requirements of regular expressions used in deep packet inspection appliances of Cisco Systems to less than

Table 4.3 Number of transitions in D2FA with default path length bounded to 4.

DFA	Basic algorithm	Refined algorithm
Cisco590	97,873	70,793
Cisco103	115,654	82,879
Cisco7	37,520	36,091
Linux56	69,437	66,739
Linux10	314,915	302,112
Snort11	180,545	178,354
Bro648	11,906	8,078

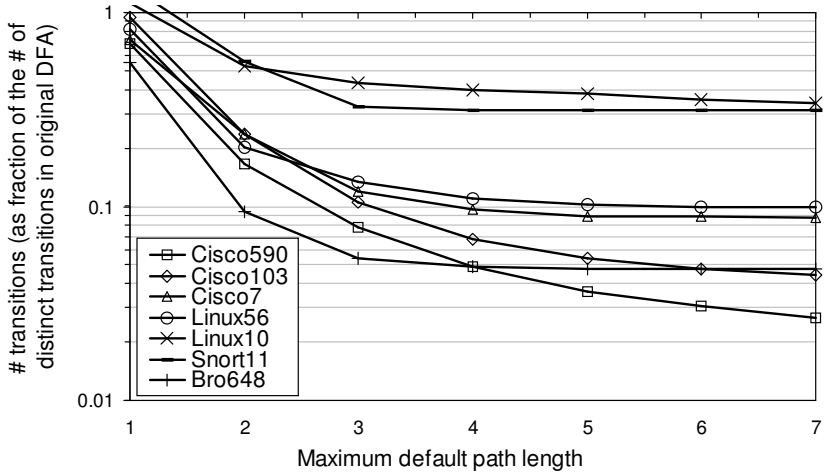


Figure 4.7 Plotting total number of labeled transitions in D2FAs for various maximum default path length bounds.

2MB. We also saw significant reduction in the Bro and Linux layer-7 expressions. Snort expressions resulted in moderate improvements (according to our conservative metric) as there were fewer distinct transitions per state.

The D^{2}FA reduces the space requirements at the cost of multiple memory accesses per character. In fact, splitting an expression set into multiple groups adds to the number of memory accesses as it creates multiple D^{2}FAs, all of which need to be processed. Although, D^{2}FAs perform equally well on expression sets which are not split, we decided to split, in order to reduce the total number of states in the DFA to begin with (*e.g.* 92 MB for 9 partitions of the Cisco rules versus >1 GB without rule partitioning). Such a design choice makes sense in our context, because we use multiple embedded memories available in an ASIC, which provides us with ample bandwidth, but limited capacity. We now present our architecture and algorithms to map the D^{2}FAs onto them.

4.3 Regex ASIC Architecture

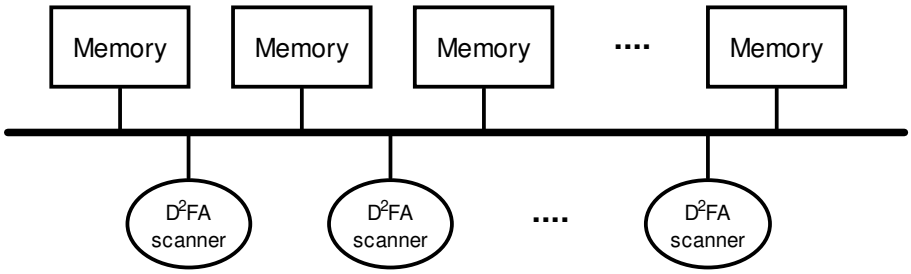


Figure 4.8 Logical structure of the memory subsystem.

In this section, we describe an ASIC architecture and an algorithm that maps the D^2FA nodes onto embedded memory modules. One of our design objectives is flexibility, so we predominantly use embedded memories in order to store the automata rather than synthesizing them in logic gates [Floyd, and Ullman 1982]. Using memory rather than logic allows the architecture to remain flexible in the face of frequently updated regular expressions. In addition to dense ASIC embedded memory technologies like IBM's, modern FPGAs such as the Xilinx Virtex-4 contain several hundreds of 18Kbit memory blocks providing several megabytes in aggregate. The embedded memories in FPGAs have multiple ports and clock rates of up to 300 MHz. Of course, ASIC technologies provide a higher degree of flexibility, with the number of ports, the size of each memory, and the clock rate all being design specific. Thus, a memory-based scalable design in an ASIC setting is eminently practical. Given this, we architect our embedded memory based ASIC with the following points in mind.

- While small memories often clock at higher rates, every additional memory adds to the overhead of the control circuitry. Therefore, we intend to use an adequate number of reasonably sized memories, so that the overall bandwidth remains appropriate while maintaining reasonable control complexity.

- Using multiple, equally-sized embedded memories will enable the architecture to scale capacity and bandwidth linearly with increasing on-chip transistor density.

- A die with several equally sized memories can achieve efficient placement and routing, resulting in minimal wasted die area.

Therefore, our design will use memories of equal size, independent of the characteristics of any particular data set. In fact, using several small equally sized memories is a natural choice given that the kind of expressions and the resulting automata are likely to change very often.

The resulting architecture consists of a symmetric tile of equally sized embedded memories; the logical organization of this system is shown in Figure 4.8. Note that FPGAs, with hundreds of fix-sized memory modules, fall within the scope of this architecture. As can be seen, there are multiple memories, each accessible by an array of regular expression engines. Each engine is capable of scanning one packet at a time. Multiple engines are present to exploit the packet- and flow-level parallelism available in most packet processing contexts. While throughput for an individual packet will be limited to that of a single memory, overall system throughput can approach the aggregate memory bandwidth.

To do so, we must map the D^2FA to these memories in such a way that, *a*) there is minimal fragmentation of the memory space, so that every memory remains uniformly occupied; and *b*) each memory receives a nearly equal number of accesses, so that none of them becomes a throughput bottleneck. We now propose algorithms to achieve these objectives.

4.3.1 Randomized Mapping

A straightforward uniformly random mapping of states to memory modules can provide scalable average-case performance. The expectation is that over a long period of time, each memory will receive a nearly equal fraction of all references. Thus, with a reasonable number of concurrent packets, average throughput can remain high. Consider a case of m memory modules and p concurrently scanned packets. If each

packet generates a read request at an interval of l cycles (*i.e.*, the memory read latency), we need to scan $m \times l$ packets concurrently in order to keep the m memories busy. In practice, we need more packets due to random conflicts. The problem can be modeled as a balls and bins problem. There are m bins (memory modules) and balls (memory requests) arrive to them randomly. Only one can be serviced at each bin per cycle, so any remaining balls must wait for subsequent memory cycles. If m balls arrive randomly, $1\text{-}e^{-1}$ will be served and rest has to wait for next cycle. Thus only 65% of the memories will be busy. As more balls arrive, more memories will remain busy. Thus, scanning many packets concurrently improves the overall throughput, while individual packets are served relatively slowly.

We report the throughput of such a randomized architecture in Figure 4.9, assuming a dual-port embedded memory running at 300 MHz and a read access latency of 4 cycles. In this experiment, we have limited the longest default paths in the D^2FA to 7. The input data was generated from the MIT DARPA Intrusion Detection Data Sets [MIT DARPA dataset]. We inserted additional data into these sets so that the automaton will detect approximately 1% matches. It is evident from the plots that as we increase the number of concurrently scanned packets, the overall throughput scales up. Moreover, as the number of embedded memories increases, the throughput scales almost linearly up to 8 memories, beyond which there is little improvement. This saturation is due to significant spatial locality in the automata traversal in which some states are visited more often than the others. In fact, in some cases, we found that a single state is visited almost 30% of the time. If such a state resides in memory module k, it is likely that memory module k will limit the overall performance irrespective of the number of modules. However, such situations are rare, and the average performance remains excellent.

A randomized system is also likely to have a very low worst-case throughput as evident from Figure 4.9. This can be explained as follows. A D^2FA often needs to traverse multiple default transitions for a character; if the maximum default path length is limited

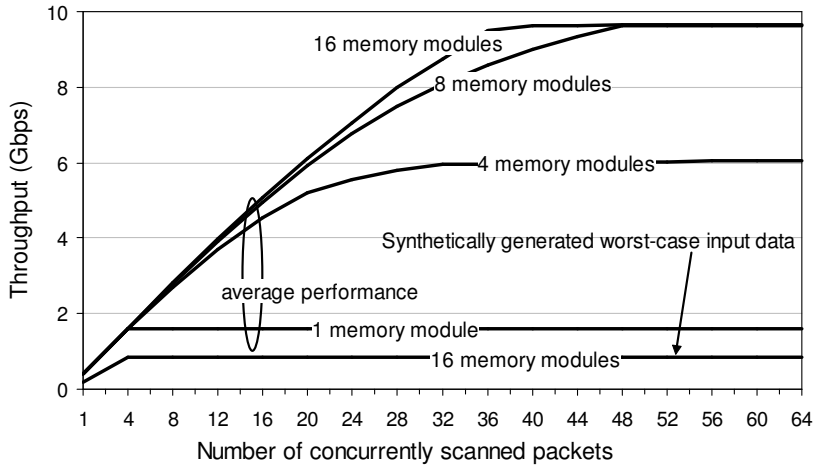


Figure 4.9 Throughput with default path length bounded to 7 and using the randomized mapping.

to 7, then 8 state traversals might be needed for a character. Since the state to memory mapping is random, there may exist default paths along which all states reside in the same memory module (or in a small number of modules). If the input data is such that the automaton repeatedly traverses such default paths, then throughput will degrade.

Moreover, when we map multiple automata (one for each regular expression group) onto memory modules randomly, default paths of different automata may map to the same memory module. In this case, packets traversing those paths will be processed serially, and overall system throughput could diminish even further. Since this randomized approach is subject to these pathological worst-case conditions, we now propose deterministic mapping algorithms capable of maintaining worst-case guarantees.

4.3.2 Deterministic and Robust Mapping

The first goal of a robust and deterministic mapping is to ensure that all automata, which are executed simultaneously, are stored in different memory modules. This will

ensure that each executes in parallel without any memory conflicts. Achieving this goal is straight-forward, provided that there are more memory modules than automata. The second goal is to ensure that all states along any default path map to different memory modules. Thus, no pathological condition can arise for long default paths as a memory module will be referred at most once. Another benefit is that we will need fewer concurrent packets to achieve a given level of throughput, due to the better utilization of the bandwidth.

Problem Formulation: We can formulate the above problem as a graph coloring problem, where colors represent memory modules and default paths of the D^2FAs represent the graph. As we have seen, these paths form a forest, where vertices represent states and directed edges represent default transitions. Our goal is to color the vertices of the forest so that all vertices along any path from a leaf to the root are colored with different colors. Moreover, we need to ensure that every color is nearly equally used, so that memories remains uniformly occupied. Clearly, if d is the longest default path, i.e. the depth of the deepest tree, then we need at least d+1 colors[1]. We present two heuristic algorithms, to color the trees in the forest.

Deterministic and Robust Mapping

The largest first algorithm is similar to the first-fit, decreasing bin-packing heuristic [Liang 1980], one of the best known heuristics for solving the NP-complete bin packing problem. The algorithm is formally described above, where the directed graph D represents the default transitions and C the set of all colors. The algorithm proceeds by ordering the default transition trees according to their size (i.e., the number of vertices times the size of each vertex). Then, in decreasing order of size, it colors each tree such that all vertices at different depths are colored with one of the d+1 colors. Since there are a total of d+1 colors and the maximum depth of a tree is d, vertices along all default

[1] A natural way to construct a D^2FA is to limit the default path length to the number of memory modules (colors) available to it

```
procedure largest-first-coloring (dgraph D(V, W), set color C);
(1)  heap h, c, l;
(2)  for tree t ∈ D ⇒
(3)     for vertex u ∈ t ⇒     size(t) := size(t) + size(u); rof
(4)     h.insert(t, size(t));
(5)  rof
(6)  for color j ∈ C ⇒         c.insert(j, 0);   rof
(7)  do h ≠ [ ] ⇒
(8)     t := h.findmax();      h.remove(t);
(9)     for all depth values i ∈ t ⇒
(10)            l.insert(i, size of all vertices at depth i);
(11)    rof
(12)    color j := c.findmax();
(13)    do l ≠ [ ] ⇒
(14)            depth i := l.findmin();        size s := l.key(i);      l.remove();
(15)            Color vertices at depth i in tree t with color j;
(16)            c.changekey(j, c.key(j) + s);
(17)            j := c.findnextmax();
(18)    od
(19) od
end;
```

paths are guaranteed to get different colors. In order to ensure that colors are nearly equally used, largest first heuristics are used. For a currently selected tree, it groups the vertices at different depths and sorts the group with respect to the size of all vertices in the group. Then, it assigns the most used color to the smallest group and the least used color to the largest group.

When the forest consists of a large number of trees, largest first coloring ensures that colors are nearly equally used; thereby ensuring that different memory modules will remain uniformly occupied. However, when there are a small number of trees, the largest first algorithm often leads to uneven memory usage. A simple example is shown on the left hand side of Figure 4.10, where there are two trees which are colored with 4 colors. With the largest first algorithm, color 3 is used to color 7 vertices, while colors 1, 2 and 4 are each used to color only 3 vertices. An alternative coloring, which uses each

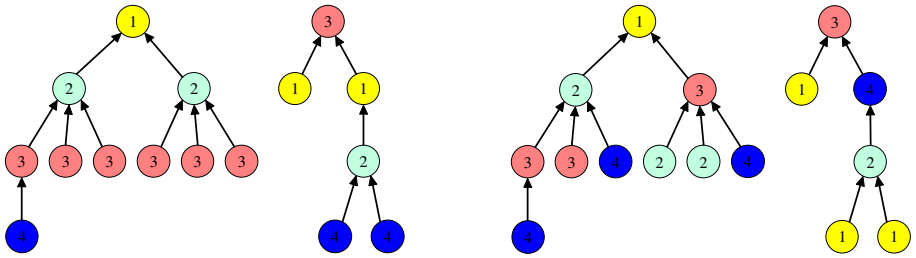


Figure 4.10 Left diagram shows two trees colored by largest first algorithm. Right diagram shows a better coloring.

color uniformly and also ensures that vertices along a default path uses different colors, is shown on the right hand side in the same figure. We now propose an algorithm which produces such coloring.

Adaptive Coloring Algorithm

The largest first algorithm performs poorly because it does not exploit situations when multiple colors are available to color a vertex. For instance, in the example shown in Figure 4.10, the largest first algorithm assigned color 3 to all vertices at depth 3, although five of these six vertices can be colored with either color 3 or 4. In practice, a D^2FA creates default trees with many such opportunities. This adaptive algorithm exploits this power of multiple choices and results in a more uniform color usage.

It begins by assigning a set of all C colors to all vertices and then removes colors from each set until every vertex is fully colored (*i.e.* a single color left in their set). In order to remove appropriate colors, it keeps track of two variables for every color. The first variable *used* tracks the total number of vertices colored by each color, and the second variable *deprived* tracks the future choices of colors that remain in the sets of those vertices not yet fully colored. More specifically, for every color, *deprived* maintains the number of the vertices, which are deprived of using it, as it has been removed from their color set and *used* maintains the number of vertices colored with it. Clearly, the

```
procedure adaptive-coloring (dgraph D(V, W), set color C);
(1)  heap h;
(2)  for color c ∈ C ⇒        used[c] := 0;   deprived[c] := 0;        rof
(3)  for vertex u ∈ V ⇒
(4)     set color colors[u] := C;
(5)     h.insert(u, depth(u));
(6)  rof
(7)  do h ≠ [ ] ⇒
(8)     u := h.findmax();       h.remove(u);
(9)     if |colors[u]| > 1 ⇒ assign-color(u, D, C); fi
(10) od
end;
procedure assign-color (vertex u, dgraph D(V, W) , set color C);
(1)  color c;
(2)  Pick c from colors[u] with min used[c] and max deprived[c];
(3)  colors[u] := c;
(4)  used[c] := used[c] + size(u);
(5)  for v ∈ descendents(u) ⇒ colors[v] := colors[v] − c;   rof
(6)  for v ∈ ancestors(u) ⇒ colors[v] := colors[v] − c;      rof
(7)  calculate-deprived(D, C);
(8)  if def-trans(u) ≠ NULL ⇒ assign-color(def-trans(u), D, C); fi
end;
procedure calculate-deprived (dgraph D(V, W) , set color C);
(1)  for color c ∈ C ⇒        deprived[c] := 0; rof
(2)  for vertex u ∈ V ⇒
(3)     if |colors[u]| = 1 ⇒
(4)             color c := colors[u];
(5)             for v ∈ descendents(u) ⇒
(6)                     if |colors[v]| > 1 ⇒    deprived[c] += size(v);        fi
(7)             rof
(8)     fi
(9)  rof
end;
```

goal is to more often use colors *a*) which most of the vertices are deprived of and *b*) with which fewest vertices are fully colored with.

After initializing the color sets of each vertex, the next step is to decide an ordering of the vertices, in which colors will be removed from their color set. An effective ordering

is to first choose vertices which do not have a high degree of freedom in choosing colors. Since vertices along longer default paths have fewer choices (*e.g.* vertices along x deep default paths can pick one of $d-x+1$ colors), they should be colored first. Therefore, adaptive algorithm processes vertices of all trees simultaneously, in a decreasing order of the depth values. It chooses a vertex, and removes all but one color from its color set, thus effectively coloring it. Whenever a vertex u is colored with color c, color c is removed from the color set of all ancestors and descendents of u, since it can't be used to color any of them. Then, all ancestor vertices of u are recursively colored. The algorithm is formally presented above. A set *colors* is kept for every vertex and initially it contains all C colors. Once all but one color is removed from this set, the vertex gets colored. The steps involved in the coloring of two trees by the adaptive algorithm using four colors are illustrated in Figure 4.11.

Coloring Results

In order to evaluate, how uniformly the largest first and adaptive algorithms utilize various colors, we generated D^2FA such that they have different numbers of default transition trees in the corresponding forest. This was achieved by limiting the default path length to different values. We also limited ourselves to use only $d+1$ colors (where d is the longest default path), as allowing the use of more colors makes the coloring far easier. Our principal metric of coloring efficiency is the maximum discrepancy in color usage. If $used(i)$ is the size (number of vertices times the number of transitions it has) of all vertices using the i-th color, then the maximum color usage discrepancy will be,

$$(\max_i used(i) - \min_i used(i)) \Big/ \max_i used(i)$$

Clearly, smaller values of discrepancy reflect more uniform usage of various colors. We plot the maximum discrepancy in color usage in Figure 4.12, for different number of default transition trees in the forest. It is apparent that adaptive algorithm uses colors more uniformly. Using the adaptive coloring algorithm, once we limited the default

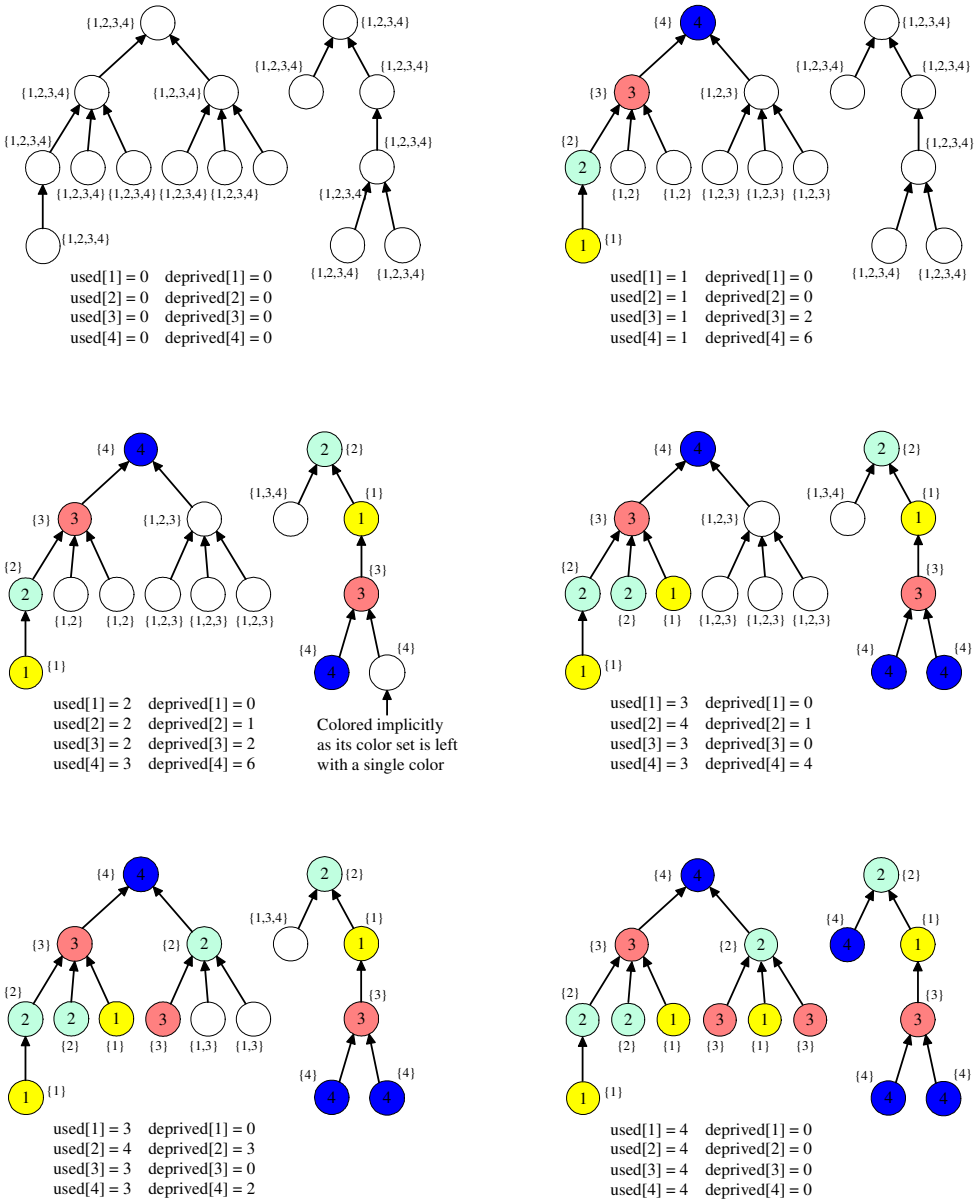


Figure 4.11 Various steps involved in the coloring of two trees with adaptive algorithm (assuming equally sized vertices).

paths to 7 or less, we were able to map all of our D^{2}FA to memory modules such that there was a maximum discrepancy of less than 7 bytes in the memory occupancy.

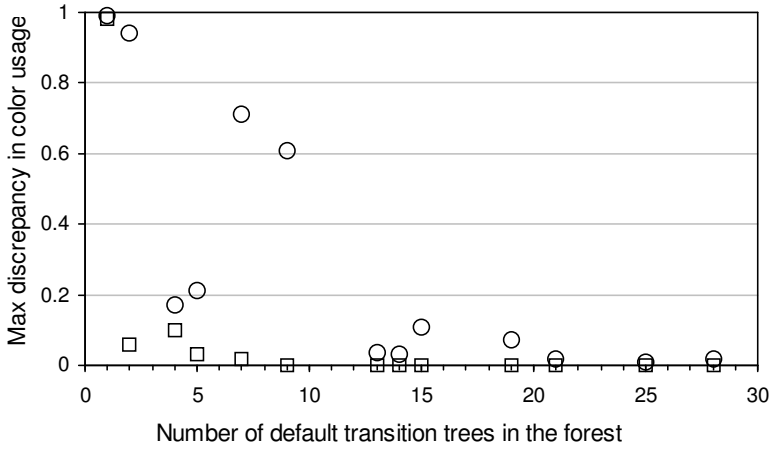


Figure 4.12 Plotting maximum discrepancy in color usage, circles for max-min and squares for adaptive algorithm.

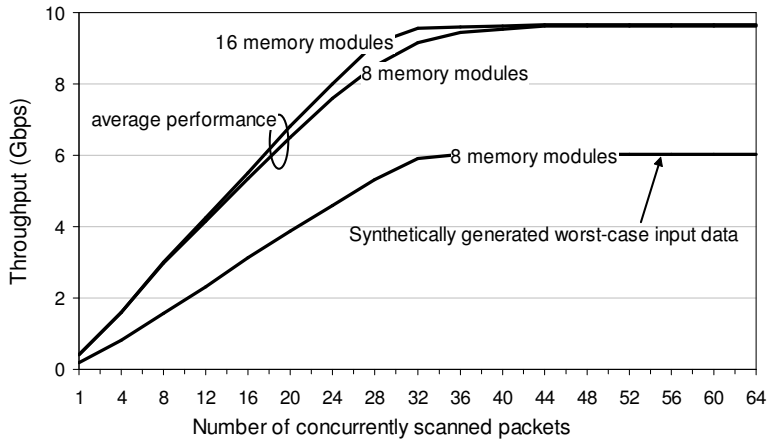


Figure 4.13 Throughput with default path length bounded to 7 and using adaptive-coloring based deterministic mapping.

We finally report the throughput of the D^2FAs generated from the Cisco rules, with the default path length limited to 7, in Figure 4.13. Note that since we are using coloring, we need at least 8 memory modules. We assume a dual-port embedded memory running at 300 MHz, read access latency of 4 cycles and the previous MIT intrusion detection data set. The performance achieved by deterministic mapping is clearly superior to the

randomized mapping, as *a*) it ensures good worst-case throughput, and *b*) it requires fewer concurrent packets to achieve high average throughput.

4.4 Summarizing the D^{2}FA based ASIC

We have introduced a new representation for regular expressions, called the delayed input DFA or D^{2}FA, which significantly reduces the space requirements of a DFA by replacing its multiple transitions with a single default transition. By reduction, we show that the construction of an efficient D^{2}FA from a DFA is NP-hard. We therefore present heuristics for D^{2}FA construction that provide deterministic performance guarantees. Our results suggest that a D^{2}FA constructed from a DFA can reduce memory space requirements by more than 95%. Thus, the entire automaton can fit in the on-chip memories in an ASIC. Since embedded memories provide ample bandwidth, further space reductions are possible by splitting the regular expressions into multiple groups and creating a D^{2}FA for each of them.

As a side effect, a D^{2}FA introduces a cost of possibly several memory accesses per input character, since D^{2}FAs require multiple default transitions to consume a single character. Therefore, a careful implementation is required to ensure good, deterministic performance. We present a memory-based ASIC architecture, which uses multiple embedded memories, and show how to map the D^{2}FAs onto them in such a way that each character is effectively processed in a single memory cycle. As a proof of concept, we were able to construct D^{2}FAs from regular expression sets used in many widely used systems, including those employed in the widely used security appliances from Cisco Systems that required less than 2 MB of embedded memory and provided up to 10 Gbps throughput at a modest clock rate of 300 MHz. The proposed architecture can provide deterministic performance guarantees with today's VLSI technology, and a worst-case throughput of OC192 can be achieved while simultaneously executing several thousands of regular expressions.

4.5 Future Direction (Bounded HEXA)

In Chapter 3, we presented HEXA, which is a novel method to encode "next nodes" in a directed acyclic graph. We now extend HEXA to represent nodes of finite automata to facilitate further memory compression desired in ASICs. Our current extension is applicable only to automata that recognize exact match strings, such as Aho-Corasick. Although HEXA can be extended further to represent general finite automata; we leave it to the future work, with the exception of a brief description of some components of the design and the challenges involved.

One potential problem with exact match string automata is that, due to the presence of cycles, HEXA identifiers of nodes may become unbounded if we continue traversing a loop and receiving input symbols. One way to enable bounded HEXA identifiers is to restrict each identifier to say previous k symbols, where k may vary for different nodes. This, however, requires that all incoming k-long paths into all nodes of the graph have identical sequences of labels. Clearly, nodes of a general graph will not meet this requirement even for k=1 as there may be multiple incoming transitions into a node labeled with different symbols. Fortunately string based automata such as Aho-Corasick, Wu-Manber and Commentz-Walter do not exhibit this property and all incoming transitions into a node are labeled with identical symbols. In fact, all incoming k-long paths into a node are labeled with identical sequence of symbols, thus potentially creating long unique identifiers; notice that here k is different for different nodes. Even high performance networking specific variants of these well known automata such as the bit-split Aho-Corasick [Tan, and Sherwood 2005] exhibit similar characteristics.

For such graphs, we introduce an extension called *bounded HEXA* (bHEXA) which uses a variable but finite number of symbols in the history to identify a node, instead of examining the entire history. Since the number of history symbols that we examine may

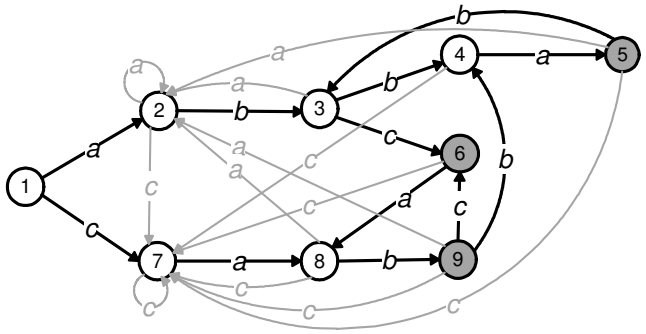


Figure 4.14 Aho-Corasick automaton for the three strings abc, cab and abba. Gray indicates accepting node.

be different for different nodes, bHEXA identifiers require additional bits to indicate this length. While these bits add to the memory needed, having variable length identifiers also opens up another dimension of multiple choices of identifiers for the nodes, which helps in finding a one-to-one mapping and reduces the dependence on discriminator bits or even avoid using them. To clarify, we consider a simple string-based example.

4.5.1 Motivating Example

Let us consider the Aho-Corasick automaton for the 3 strings: *abc*, *cab* and *abba*, defined over the alphabet {a, b, c}. The automaton (shown in Figure 4.14) consists of 9 nodes (all symbols for which a transition is not shown in the figure are assumed to lead to state 1). A standard implementation of this automaton will use 4-bit node identifiers. These identifiers will determine the memory location where the transitions of the node will be stored. There are three transitions per node (over symbols a, b and c, respectively) and assuming that a match flag is required for every node, the fast path memory will store four entries for each of the nine nodes, as shown below:

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

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

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

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

enforces these constraints and ensures that the bHEXA identifier of each node is unique.

4.5.2 Memory Mapping

The next step is to select a bHEXA identifier for every node to ensure that each identifier is mapped to a unique memory location. A large fraction of nodes, being away from the root node, are likely to have several choices of bHEXA identifiers, which will improve the probability of a one-to-one mapping. These choices however come at a cost; if a node has k choices (can have up to $k-1$ symbols long bHEXA identifiers) then up to $\lceil \log_2 k \rceil$ additional bits may be needed to indicate the length of its identifier. During the graph traversal, these bits will be required to determine the exact number of history symbols that forms the bHEXA identifier of the node. In our example automaton, node 5 has 5 choices; hence 3-bits may be needed to indicate the length of its bHEXA identifier. We can however omit the last choice from its set of legitimate identifiers, thereby keeping the bHEXA identifiers within four symbols and requiring only 2-bits. For completeness, we also keep c-bit discriminators (c may be zero, if we do not need them). Notice that instead of storing the complete bHEXA identifier, only $c+\lceil \log_2 k \rceil$ bits worth of information is required to be stored; this information along with the history of input symbols is sufficient to re-generate the complete bHEXA identifier of any given node.

Continuing with our example, we construct a memory mapping graph (as described in Chapter 3), which is shown in Figure 4.15. In the graph we use $m=10$, thus an extra memory cell is available for the nine nodes. We also limit the bHEXA identifiers to contain up to three history symbols and do not use discriminators. The edges of the graph are determined by the hash function h, which is:

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

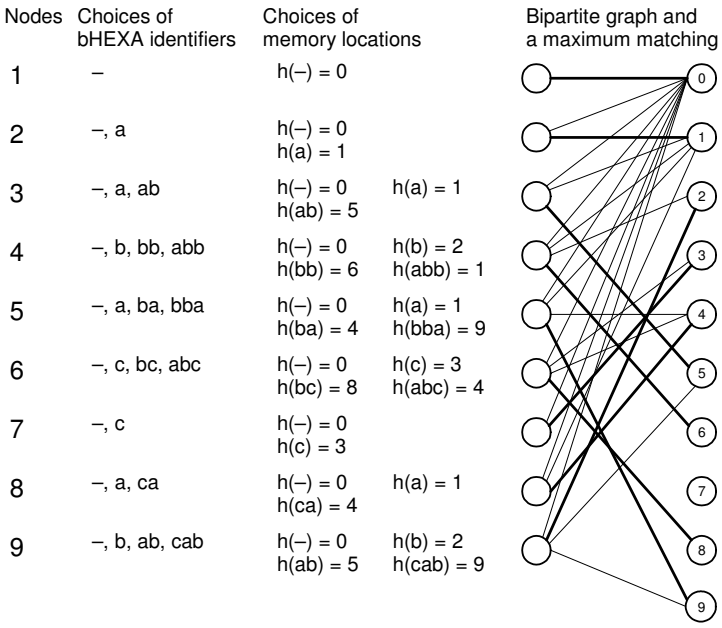


Figure 4.15 Memory mapping graph, bipartite matching.

In this formula, the input symbols are assumed to take these numerical values: –=0, a=1, b=2, c=3. In the same figure, a maximum matching in the memory mapping graph is highlighted, which assigns a unique memory location to each node of the automaton. According to this matching, the bHEXA identifiers of the nodes are chosen as:

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

Notice again that we only store the length of bHEXA identifiers in the memory (and discriminators, if they are used). During the graph traversal, the length and the history of input symbols are sufficient to reconstruct the complete bHEXA identifier. Since the length can be encoded with 2-bits in this case and there are no discriminators, the fast

path will require total 7 bits per node: a match flag and 2-bits each to indicate the length of the bHEXA identifiers of the three "next nodes" for the symbols `a`, `b` and `c`, respectively. The resulting programming of the fast path memory is shown below:

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

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

A memory efficient way of finding one-to-one mapping should iterate between two phases. In the first phase, upward expansion will be applied until the added memory exceeds the memory needed by a single bit of sideways expansion. If one-to-one mapping is not yet found then the second phase will begin, which will reset the previous upward expansion and perform a bit of sideways expansion. If a one-to-one mapping is

still not found, the first phase is repeated (without resetting the sideways expansion). This method is expected to find a one-to-one mapping while also minimizing the memory.

4.5.3 Practical Considerations

The challenges that may appear during the implementation of bHEXA are likely to depend primarily on the characteristics of the directed graph. The first challenge may arise when the directed graph contains long paths, all of whose edges have identical labels. Consider the Aho-Corasick automaton for l characters long string such as `aaaaa`… There will be l+1 nodes in the automaton and the legitimate bHEXA identifier for the i^{th} node will be any such string (`aaa`…) of length less than i. In this case, if we attempt to find a one-to-one mapping without using any discriminator then the bHEXA identifier of any i^{th} node will be i–1 characters long. Since there are l+1 nodes, the longest bHEXA identifier will contain l symbols and $\lceil \log_2 l \rceil$ bits will be required to store its length. If we employ c discriminator bits then the longest bHEXA identifiers can be reduced by a factor of 2^c, nevertheless the total number of bits that will be stored per bHEXA identifier will remain the same. Clearly, large l will undermine the memory savings achieved by using bHEXA. While such strings are not common, we would still like to decouple the performance of bHEXA from the characteristics of the strings sets.

One way to tackle the problem is to allow the length bits to indicate superlinear increments in bHEXA identifier length. For instance, if there are three length bits available then they may be used to represent the bHEXA lengths of 0, 1, 2, 3, 5, 7, 12, and 16, thereby covering a much larger range of bHEXA lengths. Of course, the exact values that the length bits will represent will depend upon the strings database. A second way to tackle the problem is to employ a small on-chip CAM to store those nodes of the automaton that could not be mapped to a unique memory location due to

the limited number of length and discriminator bits. In our previous example, if *l* is 9, and the bHEXA lengths are represented with 3-bits, then at least 2 nodes of the automaton cannot be mapped to any unique memory location. These nodes can be stored in the CAM and can be quickly looked up during parsing. We refer to the fraction of total nodes that can not be mapped to unique memory location as the spill fraction. In our experiments, we find that for real world string sets, the spill fractions remains low, hence a small CAM will suffice.

4.5.4 Some Results on String Sets

We now report the results obtained from experiments in which we use bHEXA to implement string based pattern matchers, in which we find that bHEXA representations achieve between 2-5 fold reductions in the memory. We use string sets obtained from a collection of sources: peptide protein signatures [Comprehensive Peptide Signature Database], Bro signatures, and string components of the Cisco security signatures. We have also used randomly generated signatures whose lengths were kept comparable to the real world security signatures. These strings were implemented with Aho-Corasick automata; in most experiments we did not use failure pointers as they reduce the throughput. Without failure pointers, an automaton has 256 outgoing transitions per node, and may require large amounts of memory. In order to cope up with such high fan-out issue, we have considered the recently proposed bit-split version of Aho-Corasick, wherein multiple state machines are used, each handling a subset of the 8-bits in each input symbol. For example, one can use eight binary state machines, with each machine looking at a single bit of the 8-bit input symbols, thereby reducing the total number of per node transitions to 16.

First, we report the results on randomly generated sets of strings consisting of a total 64,887 ASCII characters. In Figure 4.16(a), we plot the spill fraction (number of automaton nodes that could not be mapped to a memory location) as we vary the

memory over-provisioning. It is clear from the plot that it is difficult to achieve zero spill without using discriminators. With a single bit of discriminator and less than 10% memory over-provisioning, spill fraction becomes zero, even when the bHEXA lengths are limited to 4. Thus, total 3-bits are needed in this case, to identify any given node: one for its discriminator and two to indicate the length of its bHEXA identifier. This represents more than five fold reduction in the memory when compared to a standard implementation, which will require 16-bits to represent a node.

Next we report similar results for real world string sets. In Figure 4.16(b), we plot the spill fraction for the set of protein strings, and the strings extracted from the Bro signatures, and Cisco security signatures. We only report results of those bHEXA configurations (number of discriminator bits and maximum bHEXA length) that keep the spill fraction at an acceptably low value. For the Bro strings, about 10% memory over-provisioning is needed in order to keep the spill fraction below 0.2%. The spill level corresponds to 11 nodes which remain unmapped in the automaton consisting of total 5,853 nodes. The bHEXA configuration in this case does not use any discriminator and limits the length to 8, thus total of 3-bits are needed to identify any given node. For the protein patterns, again a 10% memory over-provisioning is needed in a configuration that uses 1-bit discriminator and up to 8 characters long bHEXA identifiers. Thus, in this case, 4-bits are needed to represent a node.

In the Cisco string set containing total 622 strings, there was one string that consisted of the \x04 ASCII symbol repeated 50 times, which creates up to 50 states with identical bHEXA identifiers. This is precisely the issue that we have described in Section 4.5.3. With restricted bHEXA length and limited discriminator bits, it is impossible to uniquely identify each of the resulting 51 nodes. Consequently, in a configuration where we employ 4-bits per bHEXA identifier, 35 nodes remain unmapped even if we arbitrarily increase the memory over-provisioning (refer to third set of vertical columns in Figure 4.16(b)). As we remove this string from the database, we were able to reduce

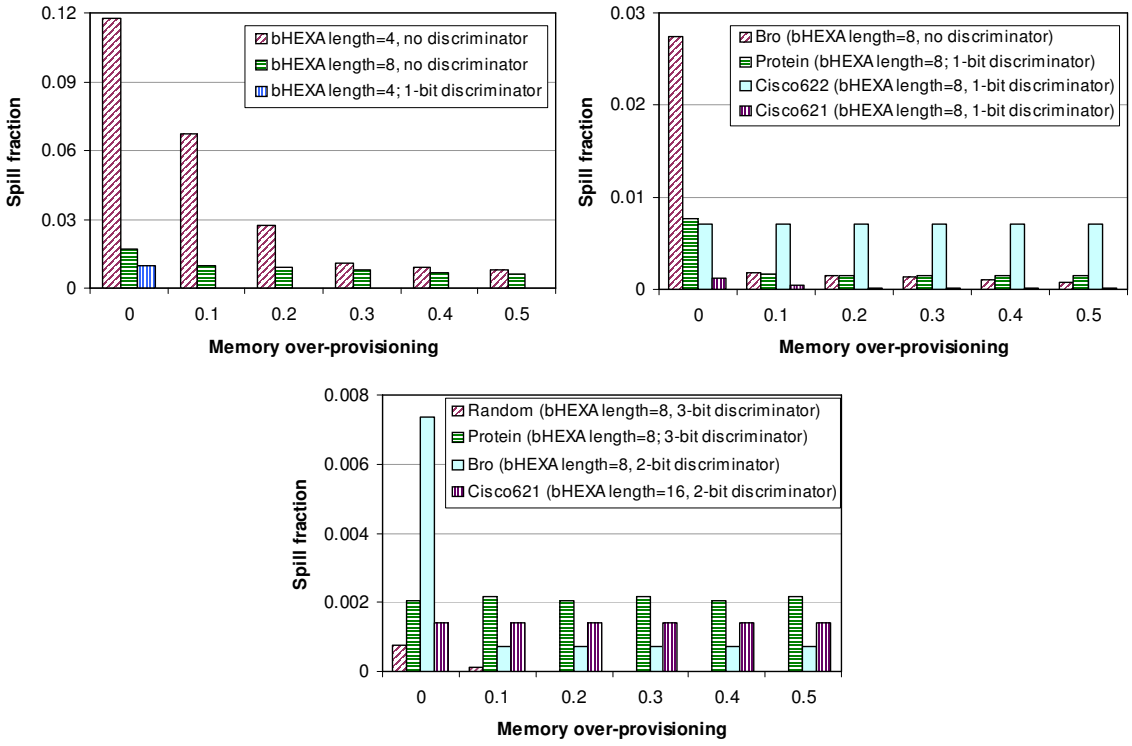


Figure 4.16 Plotting spill fraction: a) Aho-Coroasick automaton for random strings sets, b) Aho-Coroasick automaton for real world string sets, and c) random and real world strings with bit-split version of Aho-Corasick.

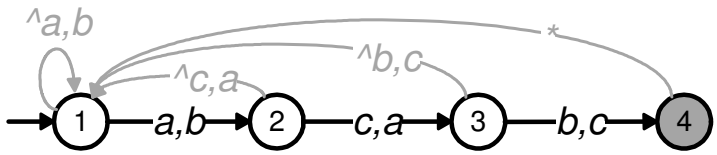
the spill fraction to 0.1% with no memory over-provisioning and for an identical bHEXA configuration (last set of vertical columns in Figure 4.16(b)).

These results suggest that bHEXA based representations reduces the memory by between 3 to 5 times, when compared to standard representations. In our final set of experiments, we attempted to represent bit-split Aho-Corasick automaton with bHEXA. We employed four state-machines, each handling two bits of the 8-bit input character. To our surprise, we found that bit-split versions were more difficult to map to the memory, and required longer discriminators and bHEXA identifiers, which increases the number of bits per node. In spite of employing the techniques we have discussed in Section 4.5.3 (*e.g.* using superlinear increase in the bHEXA length), we generally require 5 bits to represent each node of a bit-split automaton. This represents

approximately 2-3 fold reduction in memory as compared to a standard implementation. The results are plotted in Figure 4.16(c).

4.5.5 Challenges with General Finite Automaton

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



In this automaton, none of the nodes have all of its incoming paths labeled with unique sequence of symbols. Thus, it is difficult to use bHEXA identifiers to identify them. One may add new symbols in the alphabet, which will represent those character classes that are present in the regular expressions, thereby enabling paths with unique sequences of symbol. This however is likely to significantly expand the alphabet size, which will significantly increase the number of outgoing transitions from every node[2]. For instance, we find that, the regular expressions sets used in modern security

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

appliance from Cisco Systems have several thousand different character classes. Other sets of regular expressions in Snort and Bro exhibit similar characteristics. This is likely to offset any memory reduction achieved with the bHEXA identifiers.

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

To conclude, it appears plausible to employ bHEXA for the general finite automata used to represent regular expressions rules used in modern networking equipments. However, we believe that it will not lead to significant memory saving due for complex patterns, due to the added complexity in parsing and symbol resolution to the character classes. Nevertheless, we leave further investigation of the issue for the future research.

Chapter 5

Packet Content Inspection II

In Chapter 4, we focused on algorithms and architectures to implement packet content inspection functions in an ASIC setting. ASICs are custom tailored devices designed specifically for a given function which results in an unparalleled efficiency and performance. However, at low unit volumes, they are becoming unattractive as NRE (Non-Recurring Engineering) costs are skyrocketing today, turnaround times are getting longer and yield is a major challenge. Besides, ASICs lack in programmability and the practice of extensive use of embedded memory keeps a design from reaping the benefits of continuously growing density and declining per-bit cost in commodity off-chip memory components. Consequently, network processors (NP) have emerged to provide the required programmability while maintaining an acceptable level of performance. Systems, where cost and economics are more important than raw performance, are therefore increasingly using NPs as the platform of choice.

An NP is a software programmable device whose feature set is specifically customized for network specific operations. A typical architecture consists of a dense array of simple and efficient micro-processors connected to a number of specialized hardware units, a limited number of embedded memory blocks, and external memory modules. A variety of external memories are supported, and each processor supports multiple hardware thread contexts to tolerate the potentially long memory access latencies. In such setting, memory bandwidth is a precious resource, which often limits the throughput; therefore, it becomes critical for any implementation to be thrifty in memory bandwidth usage. We introduce two novel methods to efficiently implement

packet content inspection functions in such memory bandwidth constrained environments.

Our first method is called *Content Addressed Delayed input DFA* (CD^{2}FA), which provides a compact representation of regular expressions yet requires equal amount of memory bandwidth as a traditional uncompressed DFA for its execution. A CD^{2}FA builds upon D^{2}FA and addresses successive states of D^{2}FA using their content, rather than a "content-less" identifier. This makes selected information available earlier in the state traversal process, which makes it possible to avoid unnecessary memory accesses. We demonstrate that such content-addressing can be effectively used to obtain automata that are very compact and can achieve high throughput. Specifically, we show that CD^{2}FAs use as little as 10% of the space required by a conventional compressed DFA, and match the throughput of an uncompressed DFA.

Our second solution is a novel machine called a *History based Finite Automata* (H-FA), which can recognize complex regular expressions at a given parse speed using memory bandwidth comparable to that of a DFA and CD^{2}FA, but only a fraction of memory space. In fact, the memory space reduction can be so dramatic that we find that the addition of a small data cache (such as 4 KB) can significantly improve the packet throughput by keeping high cache hit rate. We begin with the description of CD^{2}FAs.

5.1 Introduction to CD^{2}FAs

5.1.1 Content Addressing

In a conventional DFA implementation, states are identified by numbers and these numbers are used to locate a table entry that contains information defining the given state. Content-addressed D^{2}FAs replace state identifiers with *content labels* that include part of the information that would normally be stored in the table entry for the state.

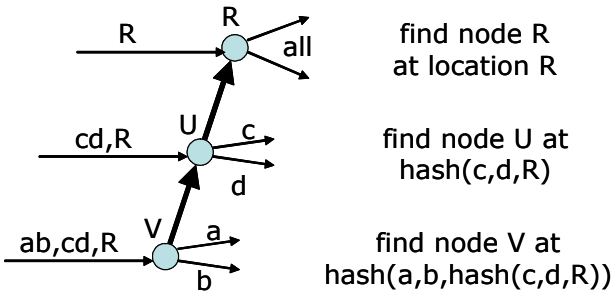


Figure 5.1 Content-Addressing.

The content labels can be used to skip past default transitions that would otherwise need to be traversed before reaching a labeled transition that matches the current input character. Using hashing, we can also use the content labels to locate the table entry for the next state.

We illustrate the idea of content addressing with the example shown in Figure 5.1. This figure shows three states of a D^2FA, *R*, *U* and *V*. The heavy-weight edges in the figure represent default transitions and *R* is the root of one of the trees defined by the default transitions. State *U* has labeled transitions for characters `c` and `d`, in addition to its default transition to *R*. State *V* has labeled transitions for characters `a` and `b`, in addition to its default transition to *U*. The arrows coming in from the left represent transitions from other states to states *R*, *U* and *V*. For each such predecessor state, we store a content label that includes the information shown in the figure, in addition to some auxiliary information that will be discussed later. The content label for transitions entering state *U* is `cd`,*R*. This label tells us that state *U* has outgoing transitions labeled by the characters `c` and `d`, and that its parent is *R*, which is the root of a default transition tree. The content label for transitions entering state *V* is `ab,cd`,*R*. This tells us that state *V* has outgoing transitions labeled by the characters `a` and `b`, and that its parent (in the default transition tree) has outgoing transitions labeled by the characters `c` and `d`, and that its parent's parent is *R*, which is the root of a default transition tree.

Suppose that the current state of the D^{2}FA is one of the predecessors of state *V* and that the current input character selects a content label for a transition to state *V* and that the next input character is *x*. While *V* is the next state, since *V* has no labeled transition for *x*, we would like to avoid visiting state *V* so that we can skip the associated memory access. Similarly, we would like to avoid visiting state *U*, since it also has no labeled transition for *x*. Assume that we have a hash function *h* for which *h*(cd,*R*)=*U* and for which *h*(ab,*U*)=*V*. Given the content label ab,cd,*R* (which is stored at the predecessor state), we can determine that neither our immediate next state (*V*) nor its parent (*U*) has an outgoing transition for x. Hence, we can proceed directly to *R*. If on the other hand, the next input character is c or d, then we can proceed directly to *U* by computing *h*(cd,*R*). Similarly, if the next input character is a or b, we can proceed directly to *V* by computing *h*(ab,*h*(cd,*R*)).

Summarizing, we associate a content label with every state in a D^{2}FA. Each label includes a character set for the state and each of its ancestors in the default transition tree, plus a number identifying the state at the root of the tree. We augment the content label with a bit string that indicates which of the states on the path from the given state to the root of its tree are matching states for the automaton. In our examples, we use underlining of the character set for a given state to denote that the state is a matching state. So, if state *U* in our example matched an input pattern of interest, we would write the content label for *U* as cd,*R* and the content label for *V* as ab,cd,*R*. Content labels are stored at predecessor states, and hashing is used to map the labels to the next state that we need to visit.

5.1.2 Complete Example

We now turn to a more complete example. Figure 5.2*a* shows a DFA that matches the patterns $a[a\underline{A}]^+b^+$, $[a\underline{A}]^+c^+$, $b^+[a\underline{A}]^+$, $b^+[c\underline{C}]$ and dd^+. Part *b* of the figure shows a corresponding space reduction graph and part *c* shows a D^{2}FA constructed using this

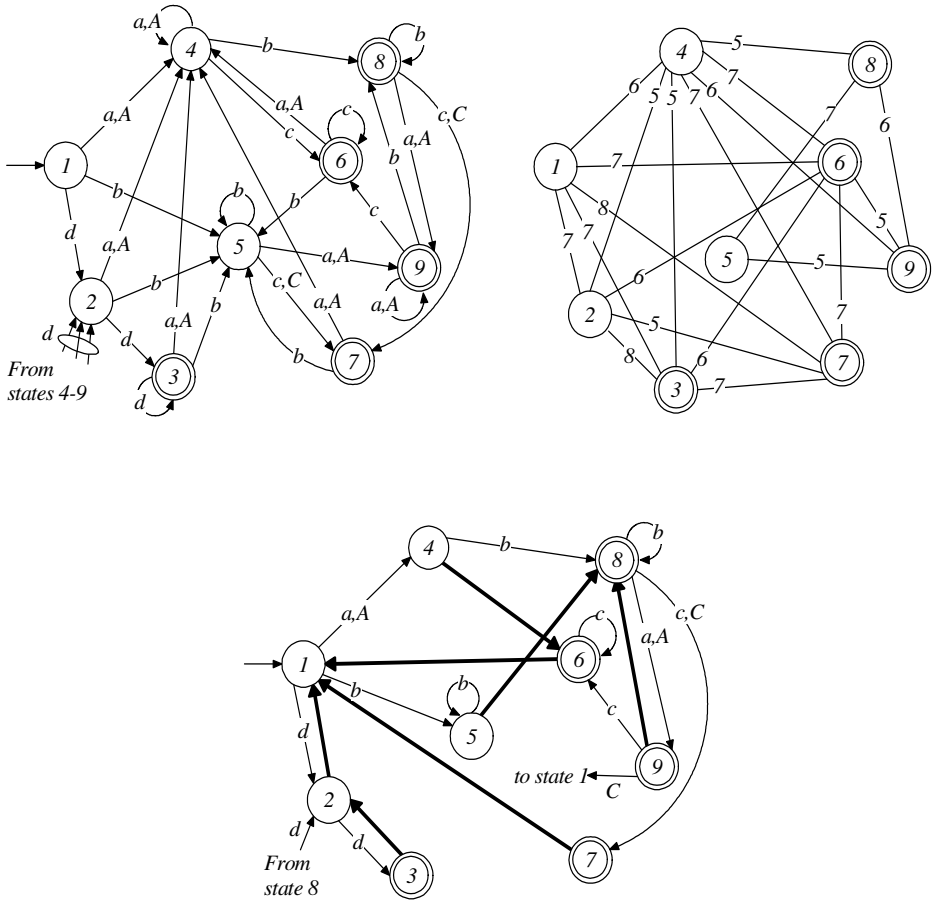


Figure 5.2 a) DFA recognizing patterns [aA]+b+, [aA]+c+, b+[aA]+, b+[cC], and dd+ over alphabet {a, b, c, d, A, B, C, D} (transitions for characters not shown in the figure leads to state 1). b) Corresponding space reduction graph (only edges of weight greater than 4 are shown). c) A set of default transition trees (tree diameter bounded to 4 edges) and the resulting D²FA.

space reduction graph. The default transitions are shown as bold edges. Note that states 1 and 8 are roots of their default transition trees and that the longest sequence of default transitions that can be followed without consuming an input character is 2. If we use the D^{2}FA to parse an input string, the number of memory accesses can be as large as three times the number of characters in the input string. Consider a parse of the string `aAcba`. Using the original DFA, we can write this in the form

$$1 \xrightarrow{a} 4 \xrightarrow{A} 4 \xrightarrow{c} \underline{6} \xrightarrow{b} 5 \xrightarrow{a} \underline{9}$$

Here, the underlined state numbers indicate matching states. Using the D^{2}FA, we the parse of the string will be

$$1 \xrightarrow{a} 4 \xrightarrow{A} 614 \xrightarrow{c} 6\underline{6} \xrightarrow{b} 15 \xrightarrow{a} 8\underline{9}$$

Here, we are showing the intermediate states traversed by the D^{2}FA. To specify the CD^{2}FA, we first need to write the content labels for each of the states. These are listed below.

1. 1
2. *d*,1
3. $\underline{-}$, *d*,1
4. *b*,$\underline{c}$,1
5. *b*,8
6. $\underline{c}$,1
7. –,1
8. 8
9. $\underline{cC}$,8

Note that since states 3 and 7 have no labeled outgoing transitions in the D^{2}FA, their content labels include empty character sets that are indicated by dashes. The dash in the content label for state 3 is underlined to indicate that state 3 is a matching state. The complete representation of the CD^{2}FA is shown below.

1. *a*: *b*,$\underline{c}$,1
   b: *b*,8
   c: 1
   d: *d*,1
   A: *b*,$\underline{c}$,1
   B: 1
   C: 1
   D: 1
2. *d*: $\underline{-}$,*d*,1
3.
4. *b*: 8
5. *b*: b,8
6. *c*: $\underline{c}$,1
7.
8. *a*: $\underline{cC}$,8
   b: 8
   c: –,1
   d: 1
   A: $\underline{cC}$,8
   B: 1
   C: –,1
   D: 1
9. *c*: $\underline{c}$,1
   C: 1

Here, for each state, we list the content labels associated with the character for which there is an outgoing labeled transition from the state. Note that only states 1 and 8 have labeled outgoing transitions for every character and states 3 and 7 have no labeled transitions.

Let's use this representation to parse the input string `aAcba`. In state 1, we find that the content label for the first input character (*a*) is *b*,*ε*,1. This tells us that the next state is *h*(*b*,*h*(*c*,1))=4, where *h* is an assumed hash function that maps content labels to the original state numbers. Since state 4 has no defined transition for the next input character (*A*), we proceed directly to state 1, skipping intermediate states *h*(*b*,*h*(*c*,1))=4 and *h*(*c*,1)=6. We are now ready to process *A*. We see that its content label is also *b*,*ε*,1. In this case however, the parent of the next state does have a defined transition for the next character (*c*), so we proceed to that state, which we find by computing *h*(*c*,1)=6. In state 6, we process character *c* using the content label *ε*,1. Since the label indicates that the next state *h*(*c*,1)=6 is a match state, we make note of the match, but since state 6 has no labeled transition for the next input character (*b*), we proceed to state 1. Continuing in this way produces the parse

$$1 \xrightarrow{a} 461 \xrightarrow{A} 46 \xrightarrow{c} \underline{6}1 \xrightarrow{b} 58 \xrightarrow{c} \underline{9}$$

If we compare this parse with the parse for the D^{2}FA, we see that the transitional states are simply shifted to the left, reflecting the fact that the CD^{2}FA skips past these states as it processes each input character.

5.1.3 Details and Refinements

In our examples, we have assumed the existence of a hash function that we could use to map content labels to state numbers. Since the numbers used to identify states are arbitrary, any hash function that produces distinct state numbers for each content label

can be used. Note that hash values are only needed for those states that are not roots of their default transition trees. The root states can simply be numbered sequentially and since there are only a few such states; the number of bits needed to represent these states can also be small.

There are some tricks that can be used to ensure uniqueness of the hash values computed for each content label. Specifically, for each character set in a content label, the order in which the characters are listed is arbitrary. Consequently, we can change the order of the characters in order to avoid conflicting hash values. If content labels are packed into words of fixed size, we can sometimes pad short label by repeating some characters, thus changing the hash value without changing the label's meaning. In some cases it may be necessary to augment the hash values with additional bits to ensure uniqueness. We refer to such extra bits as *discriminators*. As we report later, we have found that very few discriminator bits are needed in practice.

To find the content label for the current input character, we need to know where it appears in the list of content labels for the current state. States that are at the roots of their default transition trees have content labels for every symbol in the alphabet, so we can use simple indexing to find the appropriate label in this case. For states that are not roots, we have content labels only for those characters for which there is a labeled outgoing transition. The content label used to reach the state tells us which characters the state has outgoing transitions for. If our next character is the i-th one in the list of characters found in the content label at the predecessor state, then the next content label we need to consult will be at position i in the list of content labels for the current state. So, given the starting location of the list of content labels for the state, we can use indexing to find the specific content label of interest, without having to scan past the other content labels defined for the current state.

5.1.4 Memory Requirements of a CD^{2}FA

The memory required for a CD^{2}FA depends directly upon the D^{2}FA from which it is constructed and the size of the resulting content labels. If we let $a(s)$ denote the number of ancestors of state s in its default transition tree (including s) and $c(s)$ denote the number of characters for which s and its non-root ancestors have labeled transitions, then we need at least $c(s)b + a(s) + r$ bits to represent the content label for state s, where b is the number of bits needed to represent a character and r is the number of bits needed to represent the identifier for a root. In addition, to identify which characters in the content label correspond to transitions from which ancestors of state s, we can use an additional bit per character, giving $c(s)(b+1) + a(s) + r$. Additional bits may be needed for discriminators, which we ignore for now. Note that if we require that content labels be packed into a fixed size word, then the depth of the default transition trees will be limited by the word size, since both $c(s)$ and $a(s)$ get large for states that are far from the roots of their trees.

If we allow the content labels to have variable lengths, then we can potentially reduce the overall space requirement, since nodes close to the roots of their trees will have smaller content labels. If the nodes with larger content labels have relatively few incoming transitions, then the impact of these larger content labels on the overall memory requirements will be limited. Of course, allowing variable length content labels also means that we have to include length information in content labels, adding to the space needed to represent each label. In our experimental results, we allow content labels of two sizes: 32 bits and 64 bits. This adds approximately $c(s)$ bits to each content label (details in section 5.2), but does lead to a significant overall space savings.

This discussion makes it clear that the problem of constructing D^{2}FAs that lead to small CD^{2}FA is non-trivial. As shown in Chapter 4, bounding the default paths to a small constant in general leads to larger D^{2}FAs than if we allow the depth to become large. However, small depth D^{2}FAs will have relatively small content labels. The use of variable length content labels adds another dimension to the problem, since it makes it desirable for states with many incoming transitions to have small content labels. Hence,

it makes sense to position these states close to the roots of their default transition trees. Unfortunately, we don't know in advance, which states will have large numbers of incoming transitions, since the introduction of default transitions can dramatically change the number of labeled transitions entering a state. In the next section, we focus on a simple heuristic approach, which we have found produces good results experimentally.

5.2 Construction of Good D^{2}FAs

In this section, we attempt to construct D^{2}FAs, which lead to compact CD^{2}FAs. We need to ensure the size of content labels is bounded so that they can be fetched in a single memory access (in our experiments, we limit them to 64-bits), hence we only consider edges of the space reduction graph, whose weights are sufficiently large (in our case larger than 252). Our general objective is to create compact CD^{2}FAs and not compact D^{2}FAs (which can be created by solving a maximum weight spanning tree problem as described in Chapter 4), therefore we take proper care that default paths do not grow too deep and that content labels do not become too big.

To meet these objectives, we have developed a simple yet effective heuristic called *CRO*, which runs in three phases, called *creation*, *reduction* and *optimization*. The creation phase creates a set of initial default transition trees whose default paths are limited to one default transition. The reduction phase reduces the number of trees by iteratively *dissolving* and *merging* trees, while maintaining the default length bound of one transition. The optimization phase attempts to reduce the total space requirement further, by allowing some default paths to grow longer than just one default transition.

5.2.1 Creation Phase

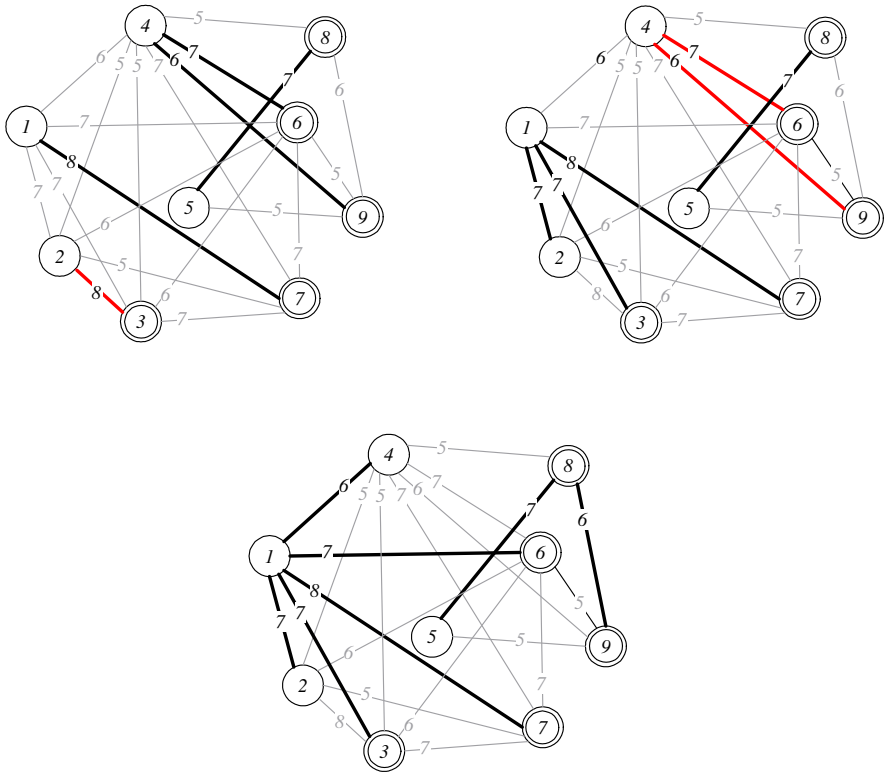


Figure 5.3 a) A set of default transition trees created by Kruskal's algorithm with tree diameter bounded to 2. b) After dissolving tree 2-3 and joining its vertices to root vertex 1. c) After dissolving tree 9-4-6 and joining its vertices to root vertices 8, 1 and 1.

During the creation phase, a forest on the space reduction graph which consists of trees whose diameter is limited to 2. The algorithm is based on Kruskal's algorithm [Kruskal 1956], thus it examines all edges in the space reduction graph in a decreasing order of weight. An edge is chosen to be part of the forest if adding the edge to the forest neither creates a cycle nor violates the diameter bound of two. In order to achieve maximum reduction in this phase, the selections made by the algorithm also aims to ensure that those states, where there is higher number of incoming labeled transitions, are more likely to become tree roots. Therefore, an *in-degree*, equal to the total number of incoming labeled transitions to a state, is assigned to all states. As the algorithm proceeds, from among all unexamined edges of equal weights, the ones, whose joining vertices have higher cumulative *in-degree* are examined earlier than those edges whose joining vertices have lower *in-degree*.

In Figure 5.3*a*, we illustrate the outcome of the creation phase, when applied to the space reduction graph shown in Figure 5.2*b*. Four default transition trees form, three of which contain a single edge. In general, the creation phase forms a large number of trees which contain a single edge because, once such a tree forms, it can not be linked to other trees containing one or more edges (because diameter bound of 2). Consequently, the weight of the forest can be increased further by reducing the number of trees. For instance, if, instead of selecting the edge 2-3 in Figure 5.3*a*, we select edges 1-2 and 1-3, then the weight can be increased by 6, while maintaining the diameter bound. Therefore, the creation phase is followed by a reduction phase, which reduces the number of trees.

5.2.2 Reduction Phase

During the reduction phase, the number of trees is reduced in an attempt to increase the weight of the spanning forest. Trees in the current forest are repeatedly examined in an order of decreasing weight (sum of weight of all edges in the tree). For any tree under examination, it is first dissolved and all its edges are removed from the forest. Afterwards, each vertex u of the dissolved tree is joined to the root vertex r of one of the tree among all trees in the forest, for which edge (u, r) has the highest weight. If the result of dissolving and reconnecting the vertices does not lead to an increase in the weight of the forest then the dissolved tree is restored. The reduction phase stops when the forest remains unaffected after a pass in which all trees are examined.

The outcome of the reduction phase is illustrated in Figure 5.3*b* and 3*c*. Initially, tree 5-8 is examined, however it is not dissolved because dissolving it and connecting its vertices to one of the tree roots doesn't increase the weight of the forest. Afterwards, tree 2-3 is dissolved and vertices 2 and 3 are joined to the root vertex 1. This increases the weight of the forest by 6. Thereafter, tree 4-6-9 is dissolved, and its vertices 4, 6, and 9 are

joined to root vertices 8, 1, and 1 respectively. The weight again increases by 6. None of the two remaining trees can be dissolved further, therefore the reduction phase stops.

The reduction phase, in this instance, has reduced the number of trees from 4 to 2 and increased the weight of the forest from 36 to 48. Thus, the total number of labeled transitions in the D^2FA has been reduced from 36 to 24. In large automata, reduction phase is much more effective in reducing the number of default transition trees and therefore the total number labeled transitions in a D^2FA.

The CD^2FA synthesized immediately after the reduction phase is generally compact as *i*) there is a reduced number of labeled transitions in the D^2FA and *ii*) all default paths are limited to a single default transition leading to compact content labels. However, even more compact CD^2FA can be created by allowing longer default paths for certain states, specifically the states where not many labeled transitions enter. The optimization phase carries out these optimizations, where the diameter of the certain parts of the trees is expanded.

5.2.3 Optimization Phase

Prior to the optimization phase, a CD^2FA is constructed and content labels are associated with all labeled transitions. At this point, some states may have many labeled outgoing transitions because their default paths are limited to a single default transition. We may reduce the number of labeled transitions at these states by allowing them to have longer default paths. This, however, may increase the size of the content label of transitions entering those states, as labels associated with those transitions will store all characters for which transitions are defined at all states along the default path. Therefore, it is important to selectively increase the default path of only those states which results in an overall space reduction.

To accomplish this, the optimization phase proceeds with an assignment of *in-degree* (size of content label of transitions entering into the state) and *out-degree* (size content label of transitions leaving the state) to all states. The eligible candidates for the default path expansion are those states which have high *out-degree* and low *in-degree*. Therefore, states are repeatedly examined in decreasing order of their (*out-degree* – *in-degree*) values. For a state under examination, a new default state from among all states, whose default path contains a single default transition, is evaluated. If one such default state results in an overall reduction in the memory, then it becomes the new default transition of the examined state. The time needed to examine a state is $O(n)$, thus the time to once examine all n states is $O(n^2)$.

After all states are examined once, the default paths contain up to two default transitions. The procedure is repeated until a pass doesn't result in any reduction in the total memory. Note that during a pass, default paths grow by at most one default transition. In practice, we found that the algorithm stops after 1-2 passes, thus the resulting default paths contain at most 2-3 default transitions and the asymptotic run time of optimization phase remains $O(n^2)$.

5.3 Optimizing Content Labels

In this section, we present optimizations to compactly encode CD^{2}FA content labels. We begin these with an attempt to reduce the number of symbols in the alphabet.

5.3.1 Alphabet Reduction

A CD^{2}FA consists of root states, which do not have a default transition, and non-root states, which have a valid default transition. Even though, root states have labeled transitions defined for all characters in the alphabet, a large fraction of these lead to the same next state. We refer to the most frequently occurring "next state" from any given

state as its *common transition*. If we let A denote the original alphabet, let $C(s)$ denote the characters, for which common transitions are present at a root state s, then its alphabet can be reduced to A-$C(s)$, if we explicitly store the common transition of the state. In general, alphabet of the root states can be reduced to $A - \bigcup_{s \in \text{root states}} C(s)$. For example, root states 1 and 8 of the D^{2}FA in Figure 5.2*c*, have common transitions (over characters *B* and *D*) leading to state 1. Note that these transitions are not explicitly shown in the figure, assuming that all transitions which are not explicitly shown in the figure leads to state 1. Once we remove the characters for these common transitions from the alphabet, it can be reduced to {*a*,*b*,c,d,*A*,*C*}.

It turns out that in all CD^{2}FAs we consider in our experiments, the reduced alphabet of the root states contains less than 128 characters. Moreover, even though there are up to a thousand root states, there are less than 64 distinct common transitions at these states; thus, we only need a 64 entry table to store the content labels associated with the common transitions. We also need to associate each root to one of the table entries, which can be done efficiently by appropriately numbering the root states. For instance if all root states with identical common transition are assigned a series of contiguous integers, then we only need to associate the first and last integer value to the common transition.

The second step is to reduce the alphabet of non-root states, which have a small number of labeled transitions and a default transition. If $L(s)$ is the set of characters for which labeled transitions are present at a non-root state s, then the alphabet of non-root states can become $\bigcup_{s \in non-\text{root states}} L(s)$. Using this procedure, alphabet of the non-root states of the D^{2}FA shown in Figure 5.2*c* can be reduced to {*b*,*c*,*d*,*C*}.

If we take the union of the reduced alphabet of root and non-root states, the resulting set (referred as A_r) still contains much fewer characters than the ASCII alphabet, thus characters of the reduced alphabet may require fewer bits to represent. For instance we were able to represent them by 7-bits in our experiments, as A_r contained between 64

and 128 characters. In order to translate a character from ASCII alphabet to the reduced alphabet, an *alphabet translation table* is needed, which contains 256 entries and is indexed by the ASCII character. Entries, which correspond to the characters in the reduced alphabet, contain a 7-bit translated character, while entries, for which a character is not present in the reduced alphabet, contain a special symbol. This table requires less than 256 bytes and therefore can be easily stored either in the data cache or in the instruction cache via a function call.

5.3.2 Optimizing Content Label of Non-root States

The content labels of non-root states may have variable lengths, which depend upon the number of labeled transitions leaving the state and its non-root predecessors. At present, we intend to restrict the content labels to two words (8-bytes), so that they can be fetched in a single memory access[3]. Thus, a content label may require 3 additional bits to store its length. We can perform an optimization by considering that memories often allow addressing at 4-byte boundaries; in other words, memory is organized as 4-byte words, in which case content labels will either be one or two words long, and a single bit will be sufficient to store its length.

When content labels have variable length, a complication may arise, as we need to know, where the content label for an input character appears in the list of content labels for the current state. In order to appropriately index this list, with the content label of each state, we need to store the length of the content labels of the states where its labeled transition enters. Thus, the content label of a state s with $c(s)$ labeled transitions requires $c(s)$ additional bits. As we have already mentioned, we need two additional bits per content label in the list, one to indicate whether it associates with a match and another to indicate the depth of the associated next state, in its default transition tree.

[3] These schemes can be easily extended to memory technologies, where minimum access size is different from 8-bytes.

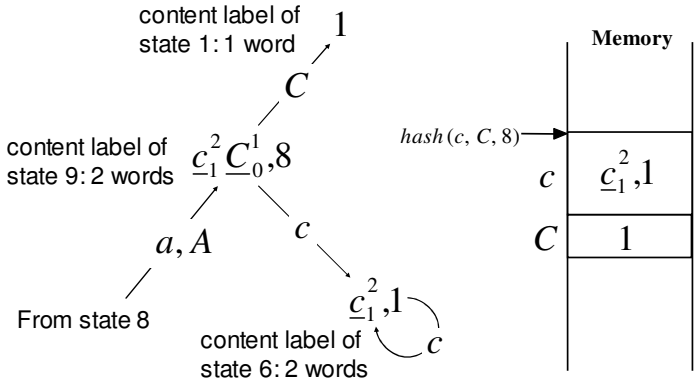


Figure 5.4 Storing list of content labels for state 9 in memory.

Consider an example, where we seek to store the list of content labels for state 9 of the CD^{2}FA in Figure 5.2*c*. State 9 has 2 labeled transitions, one leading to state 1 on input *C* and another leading to state 6 on input *c*. If we assume that that content label of the first transition is 1 word long, while the second is 2 words, then the content label of state 9 (more specifically of labeled transitions entering state 9), will be $\underline{c}_1^2\underline{C}_0^1,8$; here we indicate length of the content label as a superscript and the depth (in its default transition tree) of the next state as a subscript. The resulting memory structure is shown in Figure 5.4; state 9 requires total 3 memory words at an address determined by applying a hash on its content label (we discuss more about hashing in section 6).

With an ASCII alphabet, (8-bit characters), the content label for a state will require 11-bits per labeled transition, plus $\lceil \log_2 t \rceil$ bits to represent the root of its default transition tree (t is the number of default transition trees). In our experiments, we reduce the alphabet to fewer than 128 characters, thus, 7-bits represent a character, which enables us to use only 10-bits per labeled transition. Also there are fewer than a thousand root states, thus 10-bits are enough to represent them. Therefore, the content label of a state, which has up to 2 labeled transitions, fits in a 4-byte word, while states with between 3 and 5 labeled transitions require two words. Note that, we only allow a state to have up to five labeled transitions, thus, we only consider those edges of the space reduction graph, whose weight is higher than 251.

5.3.3 Numbering Root States

With only $\lceil \log_2 t \rceil$ bits to represent root states, transitions leaving root states are stored in a two dimensional table with t rows and $|A_r|$ columns. The table is indexed using the root state number as row index and the input character as the column index. Each cell of the table is two words long (even though content label of many transitions may be just 1 word long); thus a root state requires $2|A_r|$ memory words.

5.4 Memory Packing

When we introduced CD^{2}FA, we assumed that there exists a hash function that maps content labels to the required state numbers. In this section, we present algorithms to devise such mapping. While associating state numbers to content labels, we are interested in not only associating unique numbers but also such numbers that can be directly used as an index into the memory. Thus, we would like to associate a unique memory address to the content label of each state, so that the list of content labels for all labeled transitions leaving the state is stored at that address. This will truly enable us to require a single memory access per input character. Throughout this section, we refer to the state number as the memory address where it is stored and storing a state means storing the content labels for its labeled transitions. We focus on storing non-root states, as root states are simply stored as a two dimensional table.

The size of the list of content labels for a state depends both upon the number of labeled transitions leaving the state as well as length of their content labels (1 or 2 words). Traditional table compression schemes [Hopcroft, and Ullman 1979] may be applied to associate a unique address to each state's content label, however these schemes are known to be NP-hard, and they also incur sizeable overheads as they

require *i*) additional pointer per state, and *ii*) a marker for every content label. They also require an additional memory access per character, which may reduce the throughput.

We present a novel method which enables, *i*) an optimal memory utilization with zero space overhead, and *ii*) single memory access per input character. It is based on classical bipartite graph matching, with running time of $O(n^{3/2})$, where n is the number of states. Our method proceeds by forming groups of states so that states with identical memory requirement belong to the same group. Since we allow a non-root state to have at most 5 labeled transitions, the memory requirement of a non-root state can vary from one word to up to ten words; hence there can be up to 10 groups of states. Afterwards, memory is partitioned in 10 regions and states of each group are stored in different regions. Note that, in a CD^{2}FA, states can be easily mapped to their memory regions as the memory requirements of a state can be directly inferred from the states' content label.

Afterwards, our algorithm handles a group at a time and, as described below, stores its states into its memory region.

5.4.1 Packing Problem Formulation

Let there are n states in a group and each state requires s memory words to store its labeled outgoing transitions. Clearly, the group's memory region must contain at least ns words. We consider a slight memory over-provisioning, so the memory region consists of ms words (where $m = n+\Delta$, and Δ/n is the over-provisioning). The content labels of all states of the group needs to be uniquely mapped to one of the m memory locations (which become the content labels' state number). We apply a hash function (with codomain = $[1, m]$) to the content labels to compute this mapping. As traditional hashing is subject to collisions, multiple content labels may be mapped to a single state number. Collision resolution policies can be applied however they are likely to degrade

the performance by requiring additional memory accesses. They will also incur space overheads by unnecessarily storing the content labels (as the hash keys).

Our algorithm eliminates both these deficiencies by enabling a collision free hashing, *i.e.* content labels are mapped to unique state number. This is achieved by exploiting the possibility of renaming a content label, without changing its meaning, thus effectively changing its hash value. There are three ways to rename content labels without changing their meanings. *a*) The simplest way is to modify the value of *discriminator*. *b*) An alternative is to change the order in which characters appear in the content label; thus a content label with t characters can have $t!$ different possible names. *c*) In fixed size word length restricted content labels, yet another possibility is to pad short label by repeating some characters already present in the content label, or by modifying the unused bits. With these facilities to modify the name of a content label without changing its meaning, a naïve mapping may arbitrarily rename them whenever a collision occurs. We develop a more systematic approach to select the appropriate names.

Our approach progresses by evaluating all possible names (called candidate names) that can be assigned to a content label by employing the three mentioned methods. A hash is then applied to the candidate names, and the result is a set of candidate state numbers for the content label. Once all candidate state numbers are known, a bipartite graph $G = (V_1+V_2, E)$ is constructed, where vertex set V_1 corresponds to the n content labels and V_2 the m state numbers. Edge set E contains all edges (u, v) such that $u \in V_1$, $v \in V_2$ and v is a candidate state number for u.

After constructing the bipartite graph G, the next step is to seek a *perfect matching*, *i.e.* match each content label to a unique state number. It is likely that no perfect matching exists. A *maximum matching* M in G, which is the largest set of pairwise non-adjacent edges, may not contain n edges, in which case some content labels will not be assigned any state number. However using theoretical analysis, it has been shown that, when the number of candidate names per content label is $\Omega(\log n)$, then a perfect matching will

exist with high probability, even if $\Delta = 0$. As Δ increases slightly, probability of perfect matching grows very quickly, which guarantees that little over-provisioning will always result in a perfect matching.

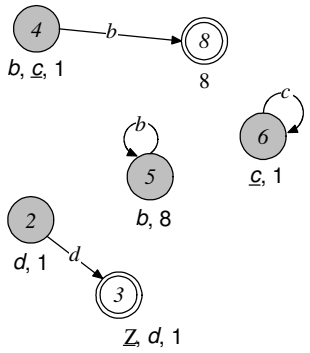
Once a perfect matching is found, for each content label, we fix its name to the one, for which its state number corresponds to a matching edge. These content labels are guaranteed to enable a collision free hashing during lookup.

5.4.2 An Illustrating Example

Before presenting the analysis of our memory packing, we consider a simple example to illustrate the basic ideas. We consider the CD^{2}FA shown in Figure 2*c*. There are 9 states, and the content labels of labeled transitions entering these states are shown in Figure 5.5*a*. There are 7 non-root states. States 3 and 7 do not require any memory, as they do not have any labeled outgoing transition (their content labels, however, may be stored at other states, from where a labeled transition enters these states). State 9 is the only state in its group, thus its packing is trivial. States 2, 4, 5 and 6, as shown in Figure 5.5*b*, each requires one word; therefore these are packed in a memory region containing 4 or more words.

First, we consider no memory over-provisioning ($m = n = 4$), and a single bit discriminator. We limit ourselves to using discriminators to rename content labels and do not use other methods. Thus, there are two candidate names for each state's content label, and the candidate state numbers by applying hash over these are shown in Figure 5.5*c*. The resulting bipartite graph is shown in Figure 5.5*d*; there are two perfect matching in this graph, one containing edges, 4-2, 2-1, 5-4 and 6-3 and another containing edges, 4-4, 2-2, 5-1 and 6-3. Either of these will suffice in mapping unique state numbers to the content labels. Note that, in this case, we have not used memory over-provisioning; indeed, we find that, we can generally avoid memory over-

State	Content labels of transitions entering the state	size of content label
1	1	1
2	*d*, 1	1
3	Z, *d*, 1	1
4	*b*, *c*, 1	1
5	*b*, 8	1
6	*c*, 1	1
7	Z, 1	1
8	8	1
9	*cC*, 8	1



Using 1-bit discriminator in a content label

State	*hash* (discriminator, content label)
4	*hash* (0, *b*, *c*, 1) = 2
	hash (1, *b*, *c*, 1) = 4
2	*hash* (0, *d*, 1) = 1
	hash (1, *d*, 1) = 2
5	*hash* (0, *b*, 8) = 1
	hash (1, *b*, 8) = 4
6	*hash* (0, *c*, 1) = 2
	hash (1, *c*, 1) = 3

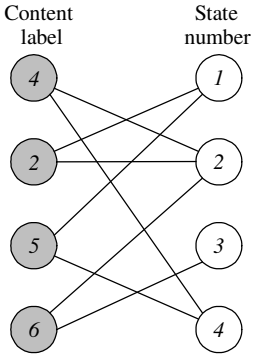


Figure 5.5 a) Content labels of states of the CD2FA shown in Figure 2. b) Non-root states requiring one word to store the content labels associated with their labeled transitions. c) Candidate content labels (using 1-bit discriminators) and the resulting candidate state numbers. d) Corresponding bipartite graph.

provisioning and also avoid discriminators because the other two methods of renaming content labels creates enough edges in the bipartite graph so that a perfect matching most likely exists.

5.4.3 Analysis of the Packing Problem

The possibility of an optimal packing depends on the likelihood of finding a perfect matching on the above bipartite graph. A necessary and sufficient condition that a perfect matching exists is given by Hall's Matching Theorem [Hall 1936].

Hall's Matching Theorem: Given a set of n items, and a set of identifiers for each item (called its candidate set), each item can be assigned a unique identifier from its candidate set if, and only if, for every $k \in [1, n]$, the union of candidate sets of any k items, contains at least k identifiers.

Thus, we have to show that, for every k content labels, the union of their candidate state numbers contains k or more distinct numbers. For k=1, this is obvious, as the candidate set of any content label is non-empty. For k>1, Hall's theorem can be unsatisfied. This is due to the use of hashing in determining the state numbers. Even though a content label can have many (say l) names, its candidate set may still contain a single state number, due to collisions. In general, k content labels will have a total of kl random state numbers in the union of their candidate set. Thus, in order to compute the likelihood of a perfect matching, we compute the probability with which a set of kl randomly chosen numbers $\in [1, m]$ contains k or more distinct numbers.

The problem of finding perfect matchings in such bipartite graphs is well studied. In [Motwani 1994], author shows that a perfect matching in a symmetric bipartite graph with n left and right vertices and with random edges, exists with high probability when the number of edges are $O(n \log n)$. In fact, this threshold is sharp, which means that the probability of perfect matching increases very quickly, as we add slightly more edges after threshold. In an asymmetric case, (when $m > n$), [Fotakis, et al. 2003] shows that the probability of a perfect matching again increases quickly, as m is greater than n. For instance, when $m/n = 1.01$, (implies 1% memory over-provisioning), a perfect matching exists with high probability, if there are more than $7n$ edges in the bipartite graph.

With these results we can conclude that if we have the flexibility to assign $O(\log n)$ different names to each content label, then we will most likely find a perfect matching without any memory over-provisioning. $O(\log n)$ corresponds to approximately 16 choices of names for each content label in a 64K state CD^2FA; this can be easily achieved even without using discriminators. As expected, in our experiments, we found

Table 5.1 Our representative regular expression groups.

Source	# of regular expressions	Avg. ASCII length of expressions	% expressions using wildcards (*, +, ?)	% expressions length restrictions {,k,+}
Cisco	590	36.5	5.42	1.13
Cisco	103	58.7	11.65	7.92
Cisco	7	143.0	100	14.23
Linux	56	64.1	53.57	0
Linux	10	80.1	70	0
Snort	11	43.7	100	9.09
Bro	648	23.6	0	0

a perfect matching in all CD^{2}FAs without using memory over-provisioning or employing the discriminators.

5.5 Experimental Evaluation of CD^{2}FA

In order to evaluate the effectiveness of a CD^{2}FA, we perform experiments using the same datasets that we used in our evaluation of D^{2}FA (described in Chapter 4). The properties of the representative regular expression groups drawn from this dataset are summarized in Table 5.1.

We applied CRO algorithm on these regular expression groups to create CD^{2}FAs. In Table 5.2 and 5.3, we report the properties of the original DFA and the outcome of the CD^{2}FA construction algorithm after every phase; we report the number of trees in the CD^{2}FA, total number of labeled transitions, and memory needed by the CD^{2}FA. We also report the size of the reduced alphabet. While reduction phase is most effective in reducing memory, alphabet reduction also reduces memory by nearly two times. It is clear that, memory reduction achieved by CD^{2}FA, constructed from the CRO algorithm, is between 2.5 to 20 times, when compared to a table compressed DFA. If we compare CD^{2}FA to uncompressed DFA (which is a fair comparison because a

Table 5.2 Properties of the original DFA from out dataset.

Dataset	Original DFA			
	# of states	*# distinct transitions*	*Memory (MB)*	
			No compression	*With table compression*
Cisco590	17,713	1,537,238	9.07	6.23
Cisco103	21,050	1,236,587	10.77	9.56
Cisco7	4,260	312,082	2.18	1.14
Linux56	13,953	590,917	7.14	3.62
Linux10	13,003	962,299	6.65	3.35
Snort11	37,167	441,414	19.03	3.55
Bro648	6,216	149,002	3.18	1.26

Table 5.3 CD^{2}FA constructed after each phase of the CRO algorithm. Last column is the ratio of memory size of a CD2FA and that of a table compressed DFA (DFATC)

CD^{2}FA									
After creation phase			After reduction phase			After optimization phase and alphabet reduction			
# of trees	*# of transitions*	*Memory (MB)*	*# of trees*	*# of transitions*	*Memory (MB)*	*# of trees*	*# of transitions*	*Memory (MB)*	*Alphabet size*
4,227	1,099,809	8.87	243	117,743	0.80	243	62,043	0.39	98
4,617	1,205,978	9.72	684	253,239	1.87	684	122,679	0.86	106
838	220,705	1.76	194	59,077	0.44	194	32,842	0.23	126
1,741	459,215	3.73	266	156,485	1.17	266	85,444	0.61	123
3,361	870,623	7.27	994	382,464	3.01	994	183,237	1.48	118
3,024	806,790	6.31	257	188,913	1.28	257	65,629	0.36	37
370	100,341	0.77	24	15,183	0.08	24	9,779	0.05	83

CD^{2}FA matches an uncompressed DFA in throughput), the memory space reductions are much higher, between 5 to 60 times.

While CD^{2}FAs match uncompressed DFAs in terms of throughput, in a practical system with an on-chip cache, a CD^{2}FA may surpass a DFA by achieving higher cache hits due to its smaller memory footprint. In Figure 5.6, we report the throughput results,

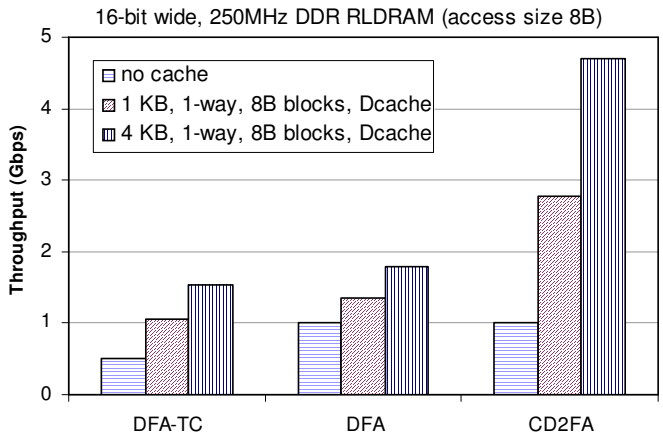


Figure 5.6 Throughput results on Cisco rules, without and with data cache. Table compressed DFA (DFA-TC), uncompressed DFA and CD2FA are considered and the Input data stream results in a very high matching rate (~10%).

where we have performed a trace driven cache based memory model simulation using Dinero IV simulator [Hill, and Elder 1998]. In order to create near worst-case conditions for a cache, the input data stream contained a high concentration of matching patterns (around 10% matches), which resulted in very low spatial locality in automata traversal. Even under these conditions, we found that cache hit rates were moderately good (25-50%), enough to improve the throughput. Hit rates of CD^2FA were noticeably higher (>60%) as it had much smaller memory footprint. Hence its throughput is also much higher. Note that the throughput of a table compressed DFA is much lower as it requires more than one memory access per input character.

5.6 H-FA: Compact yet Fast Machines

In this section, we propose a novel machine that deals with the problem of DFA state explosion in a unique way. State explosion in a DFA occurs because the DFA states must encode partial match information for many constituent patterns. The regular expressions that are typically used in networking comprise simple patterns with one or many closures over characters classes embedded in between (*e.g.* `abc.*cde` or `ab[a-`

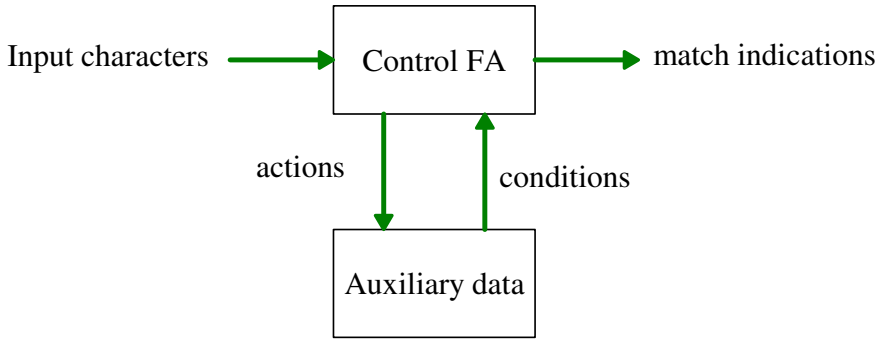


Figure 5.7 History based Finite Automaton.

`z]*ef`). The prefix portion in these patterns can be matched with a stream of suitable characters and the subsequent characters can be consumed without moving beyond 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 combination of multiple partial matches has to be followed. A DFA represents each such combination with a separate state due to its inability to remember anything other than its current state of execution. With multiple closures, the number of combinations 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 a single execution state and yet avoid the memory blowup that normally accompanies state explosion, is to equip the machine with a small and fast auxiliary data which we call *history*, that registers the key events which occurs during the parse history, such as encountering a closure. Recall that the state explosion occurs because parsing get stuck at a single or multiple closures; thus if the history buffer registers these events then the automaton may avoid the need to explicitly represent these states in the lookup table. We call this class of

machines *History based Finite Automaton* (H-FA). It is illustrated in Figure 5.7. Overall state of the HFA includes state of the control DFA and the values of the history data.

The execution of H-FA is augmented with the conditions within the history buffer. The control automaton is similar to a traditional DFA and consists of a set of states and transitions. Multiple transitions on a single character may leave from a state (like in a NFA), however, only one of these transitions is taken during the execution, which is determined by the contents of the history buffer. Thus, certain transitions in an H-FA have an associated condition. The contents of the history buffer may be updated during the machine execution. The challenge is deciding what to place in the auxiliary data. We want a small amount of data stored in the history buffer as well as a small number of distinct conditions and actions, such that the control automaton does not explode in the number of states.

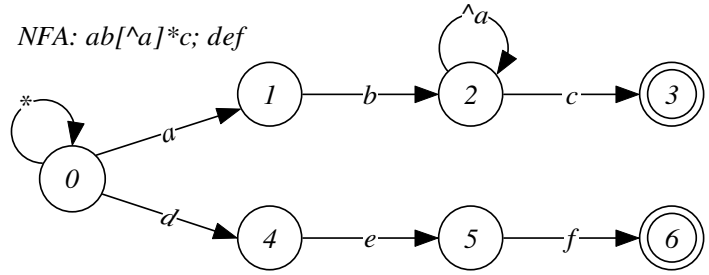
Fortunately there is direct connection between the size of an H-FA control automaton (number of states and transitions) and the partial matches in the regular expressions pattern that are registered in the history buffer. If we judiciously choose these partial matches then the H-FA states can be limited. The next obvious questions are: *i*) how to determine these key 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 now proceed with comprehensive description of H-FA where we attempt to answer these questions.

5.6.1 Motivating Example

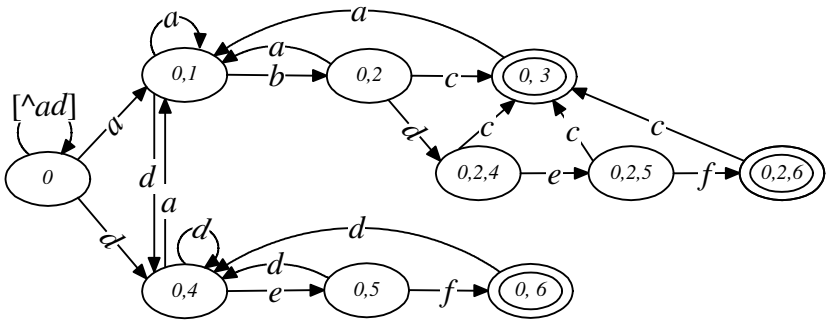
We introduce the construction and executing of H-FA with a simple example. Consider two reg-ex patterns listed below:

$r_1 = .^*ab[\hat{}a]^*c$; $r_2 = .^*def$;

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

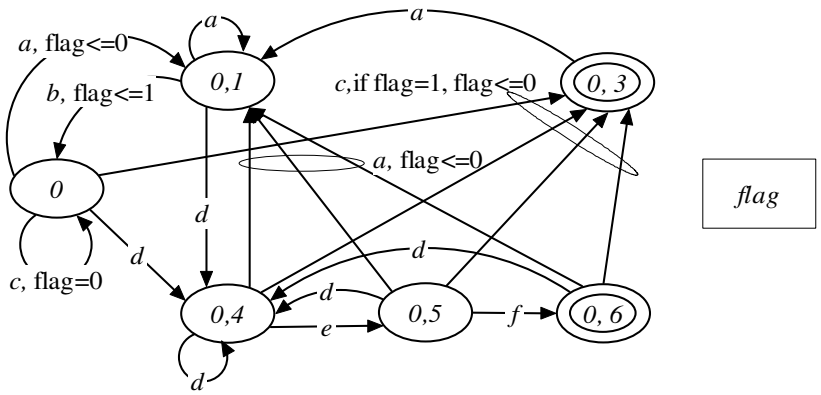


Let us examine the corresponding DFA, which is shown below (some transitions are omitted from the figure to keep it readable; missing transitions usually lead to state 0):



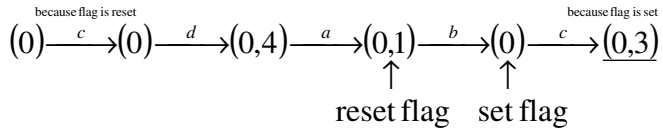
The DFA has 10 states; each DFA state corresponds to a subset of NFA state numbers, as shown above. There is a small blowup in the number of states, which occurs due to the presence of the Kleene closure $[\hat{}a]^*$ in the expression r_1. Once the parsing reaches the Kleene closure (NFA state 2), subsequent 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 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 (meaning that NFA state 2 is also active).

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. Such an exponential explosion clearly occurs because the DFA needs to remember that it has reached these closure NFA states during the parsing. Intuitively, the simplest way to avoid the explosion is to equip the DFA to remember those closures which have been reached during the parsing. In the above example, if the machine can maintain an additional flag which indicates 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; is addition it maintains a flag, which is either set or reset (indicated by <=1, and <=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 to multiple flags; each flag indicating a Kleene closure. The transitions will be made depending upon the state of all flags and the flags will be updated during certain transitions. As illustrated by the above example, augmenting an automaton with these flags can avoid state explosion. However, we need a more systematic way to construct these H-FAs, which we describe now.

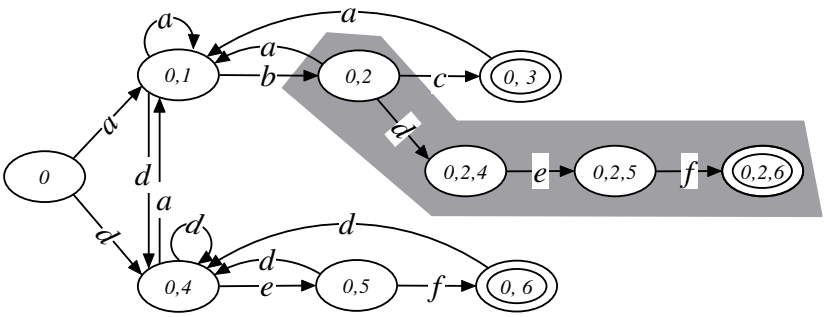
5.6.2 Formal Description of H-FA

History based Finite Automata (*H-FA*) consists of an automaton and a set called history buffer. The transition of the automaton has *i*) an associated *condition* based upon the contents of the *history*, and *ii*) an associated *action* which either inserts into the history set, removes from it, or both. *H-FA* can thus be represented by a 6-tuple $M = (Q, q_0, \Sigma, A, \delta, H)$, where Q is the set of states, q_0 is the start state, Σ is the alphabet, A is the set of accepting states, δ is the transition function, and H is the history set. The transition

function δ takes in a character, a state, and a history value as its input and returns a new state and a new history value.

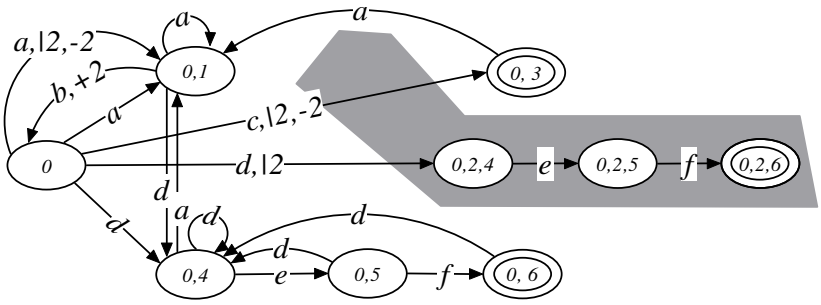
$$\delta: Q \times \Sigma \times H \rightarrow Q \times H$$

H-FAs can be synthesized either directly from a NFA or from a DFA. For clarity, we explain the construction from a combination of NFA and DFA. We consider our previous example of two reg-exes. First, we determine those NFA states of the reg-exes, which we will register in the history buffer. We defer the formal method to pick such NFA states, and at present just pick the closure states. Since, the first reg-ex, r_1 contains a closure represented by the NFA state 2; we keep a flag in the history for this state. Next, we identify those DFA states, which contain this closure NFA state number, and call these DFA states *fading states*, which are highlighted below.

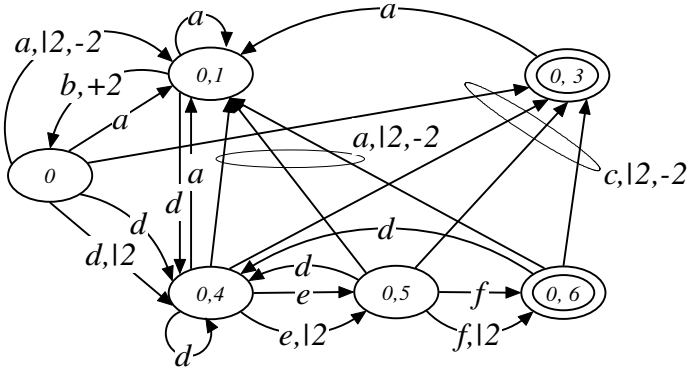


Afterwards, we attempt to remove the NFA state 2 from the fading DFA states. Notice that, if we will make a note that the NFA state 2 has been reached by setting the history flag, then we can remove the NFA state 2 from the fading states subset. The consequence of removing the NFA state 2 from the fading states is that these fading states may overlap with some DFA states in the non-fading region, thus they can be removed. Transitions which originated from a non-fading state and led to a fading state and vice-versa will now set and reset the history flag, respectively. Furthermore, all transitions that remain in the fading region will have an associated condition that the

flag is set. Let us illustrate the removal of the NFA state 2 from the fading state (0, 2). After removal, this state will overlap with the DFA state (0); the resulting conditional transitions are shown below:



Here a transition with "|*s*" means that the transition is taken when history flag for the state *s* is set; "+*s*" implies that, when this transition is taken, the flag for *s* is set, and "-*s*" implies that, with this transition, the flag for *s* is reset. Notice that all outgoing transitions of the fading state (0,2) now originate from state (0) and have the associated condition that the flag is set. Also those transitions which led to a non-fading state reset the flag and incoming transitions into state (0,2) originating from a non-fading state now has an action to set the flag. Once we remove all states in the fading region, we have the following H-FA:



Notice that several transitions in this machine can be pruned. For example the two transitions on character *d* from state (0) to state (0,4) can be reduced to a single unconditional transition (the pruning process is later described in greater detail). Once we completely prune the transitions, the H-FA will have total 4 conditional transitions; remaining transitions will be unconditional. When there are multiple closures, then multiple flags can be employed in the history buffer and the above procedure can be repeatedly applied to synthesize the H-FA.

The above example demonstrates a general method for the H-FA construction from a DFA. In order to achieve the maximum space reduction, the algorithm should only register those NFA state numbers in the history buffer which appear most frequently in the DFA states. Thus, if the history buffer has room for say 16 flags, then those 16 NFA states should be identified which appear most often in the DFA states. Afterwards, the above procedure can be repeatedly applied. With multiple flags in the history buffer, some transitions may have conditions over multiple history flags. Some transitions may also set or reset multiple flags. If there are k flags in the history buffer and h represents this vector, then a condition C will be a k-bit vector, which becomes true if all those bits of h are found set whose corresponding bits in C are also set.

The representation of conditions as vectors eases the pruning process, which is carried out immediately after the construction. The pruning process eliminates any transition with condition C_1, if another transition on condition C_2 exists between the same pair of states, over the same character such that the condition C_1 is a subset of the condition C_2 (*i.e.* C_2 is true whenever C_1 is true) and the actions associated with both the transitions are identical. In general, the pruning process eliminates a large number of transitions, and it is essential in reducing the memory requirements of the H-FA. After pruning, there may remain a small amount of blowup in the number of transitions. In the worst-case, if we eliminate k NFA states from the DFA by employing k history flags then there can be up to 2^k additional conditional transitions in the resulting H-FA. However,

such worst-case conditions are rare; normally there is only a small blowup in the number of transitions. We now present a brief analysis of these blowups.

5.6.3 Analysis of the Transition Blowup

Consider a set k of regular expressions each containing a closure. Let the i^{th} expression be denoted by $r_1^i c_0^i [c_1^i]^* c_2^i r_2^i$, where $r_1 c_0$ and $c_2 r_2$ are prefix and suffix parts of the expression; here the closure is over set of characters denoted by c_1, c_0 denotes the set of characters preceding the closure and c_2 denotes the set of characters following the closure. For such an expression, if c_1 contains a large number of characters, then there is likely to be a state blowup in the DFA. On the other hand, if we construct an H-FA, and allow each of the k closures to be represented by flags in the history buffer, then the blowup in the number of conditional transitions will depend directly upon c_2.

First, if none of the c_2's overlaps with each other, then there will be at most one conditional transition per character per state and in total there will be up to k additional conditional transitions per state. On the other hand, when c_2's are overlapping then there may be an exponential blowup in the number of conditional transitions.

To better understand the nature of the transition blowup, let us consider the transitions leaving DFA state (i,j,k), which comprises three NFA states. We assume that the NFA state i corresponds to a closure and needs to be represented by a history flag. Let the closure be over a character set c_1, and the character set which moves the parsing past the closure is c_2. If we remove the NFA state i from all DFA states then the state (i,j,k) may be merged with a pre-existing DFA state (j,k). Let the transition on a character c from state (i,j,k) lead to state (p,q,r). For $c \in c_1$, p must be i; p may differ from i only when $c \in c_2$ or $c \notin c_1$. Hence, after i is removed from the DFA states, the newly added conditional transitions from the state (i,j) over characters $c \in c_1$ will be identical to the transitions leaving state (i,j); hence they will be removed during pruning. Only those conditional

transitions will remain, which are over the characters $c \in c_2$ or $c \notin c_1$. In situations when there are multiple closures, there may be multiple permutations of the conditional transitions only if character sets c_2^i, over which parsing progresses ahead of the closures are overlapping. For instance, if each c_2^i is $\{a\}$ then there can be up to 2^k conditional transitions over the character *a*, and the conditions will be every possible combination of the *k* flags in the *history buffer*.

The actions (insert/remove from *history*) associated with the conditional transitions will depend upon the characteristics of c_0 and c_1. Flags will be set by the transitions over character c_0, and reset by the transitions on characters not from the set c_1. Thus, if c_0 is small and c_1 is large, then only a small number of transitions will have an associated action. If we examine the regular expressions used in practical signatures, the sets c_0 and c_2 are usually small, thus the H-FA will be extremely effective is reducing the number of state. On the other hand, the set c_1 is large; hence, there will be minimal blowup in the number of conditional transitions. We present detailed results of the nature of H-FA constructed from current reg-ex signatures in the next section; here we resume with the discussion of certain concerns with the implementations of H-FA's history buffer and the associated conditional transitions.

5.6.4 Implementing History and Conditional Transitions

It is clear that, if there is no overlap between the sets of the characters which moves the parsing past the closure, then a state will have at most two transitions on any character, one unconditional, and another conditional. When certain characters of these sets are overlapping, say *t*-times then there may be up to 2^t conditional transitions per state over that character. In most of our experiments, *t* remains smaller than 3. Thus, there are at most 8 conditional transitions per state. In rare situations, where *t* is greater than 3, we split the reg-ex sets into multiple sets, so that *t* becomes smaller than 3, which keeps the maximum number of conditional transitions at 8.

With up to 8 transitions per character per state, they can be stored together at some memory location, and can be fetched in a single memory access. For 16K states, 16-bits will represent a transition, and for 16-bit history buffer, conditions and actions can be represented with 32-bits, thus 6-bytes will represent a conditional transition, and 48-bit wide logical memory will be sufficient. Such logical bus widths can be achieved in NPs as well as in an ASIC/FPGA based system.

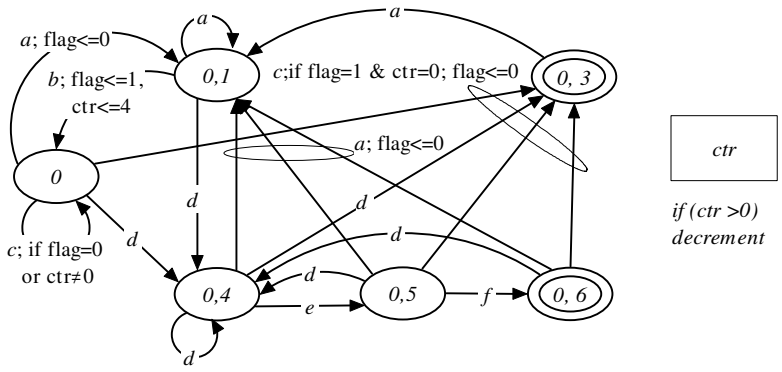
Once the conditional transitions are fetched from the memory, the next step involves the selection of appropriate transitions. This selection will depend upon the contents of the history buffer. First those transitions are filtered out whose condition do not satisfy (a condition is false if some flags which are set in the condition, are not set in the history); unconditional transitions are never filtered. Afterwards, from among remaining transitions, the one whose condition required most set flags is selected. Note that there can never be a tie (multiple conditional transitions with equal number of set flags). In terms of the hardware cost, the logic to compute if the conditions are met or not will require k gates per condition, and the logic to decide among the chosen transitions will require k adders, $\log_2 k$ priority encoders, and a few gates to glue them together. In total, the circuitry will require less than 1000 gates for a 16-bit history buffer; and will be able to make decisions in a few nano-seconds (there will be roughly $2\log_2 k+3$ gates in the critical path).

5.6.5 H-cFA: Handling Length Restrictions

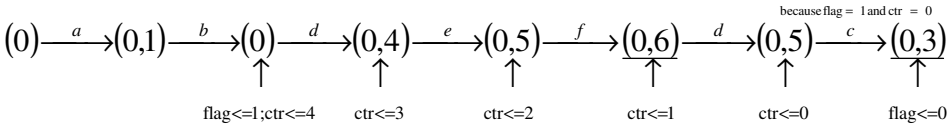
We now propose an extension called *History based counting finite Automata* (H-cFA), which efficiently solves the limitations of finite automata in efficiently implementing length restriction on sub-expressions within the regular expression. We begin with an example, in which we consider the same set of two reg-exes in our previous example with the closure in the first reg-ex replaced with a length restriction of 4, as shown below:

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

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 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 solution 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. We now present our detailed experimental results, where we highlight the effectiveness of our solutions.

5.6.6 Experimental Results

In this section, we report the effectiveness of H-FA and H-cFA in reducing memory on a selected set of reg-exes datasets. In Table 5.4 and 5.5, we report the results from our representative set of experiments, and highlight the number of flags and counters that we employ in the history buffer. Snort rules comprise of several long length restrictions; we find that, for these datasets, H-cFAs are extremely effective in keeping the memory small. Without employing the counting capability, a composite automaton for Snort reg-

Table 5.4 Properties of the DFA constructed from our key reg-ex datasets.

Source	Avg. ASCII length	# of closures, # of length restriction	DFA	
			# of automata	total # of states
Cisco64	19.8	14, 1	1	132,784
Snort rule 1	36.9	6, 6	3	62,589
Snort rule 2	16	1, 2	1	12,703
Snort rule 3	13.8	5, 1	2	4,737
Linux70	21.4	11, 0	2	20,662

Table 5.5 Results of the H-FA and H-cFA construction

Source	Composite H-FA / H-cFA				% space reduction with H-FA	H-FA parsing rate speedup
	# of (flags, counters) in history	Total # of states	Max # of transitions / character	Total # of transitions		
Cisco64	6, 0	3,597	2	1,215,450	94.69	1x
Cisco64	13, 0	1,861	8	682,718	96.77	1x
Snort rule 1	5, 6	583	8	238,107	97.40	3x
Snort rule 2	1, 2	71	2	27,498	98.58	1x
Snort rule 3	5, 1	116	4	46,124	93.48	2x
Linux70	9, 0	1,304	8	546,378	81.63	2x

exes explodes in size. For Cisco rules, we show how varying the number of flags affects the H-FA size (first two rows of the Table 5.5 use different number of flags). In general, with more history flags, the H-FA is more compact.

Notice that the traditional DFA compression techniques including the D^2FA can also be applied to H-FA, thereby further reducing the memory. The results also demonstrate that with H-FAs, we always require a single composite automaton as opposed to the

DFA approach, where we may require multiple automata, leading to a reduced performance. Thus, H-FA approach also helps in improving the parsing speed.

The table highlights another important result: the blowup in the number of conditional transitions in the H-FA generally remains very small. In a DFA there are 256 outgoing transitions, while in most of the H-FAs these are less than 500. Thus, there is less than 2-fold blowup in the number of transitions; on the other hand reduction in the number of states is generally a few orders of magnitude, thus the net effect is a significant memory reduction.

5.7 Summarizing CD^{2}FA and H-FA

In this chapter, we introduce the Content Addressed Delayed Input DFA (CD^{2}FA), which provides compact representation of regular expressions. A CD^{2}FA is built upon the recently proposed delayed input DFA (D^{2}FA), whose state numbers are replaced with content labels. The content labels compactly contain information which are sufficient for the CD^{2}FA to avoid any default traversal, thus avoiding unnecessary memory accesses and hence achieving higher throughput in a network processor setting. While a CD^{2}FA requires equal number of memory accesses to those required by an uncompressed DFA, in systems with a small data cache, CD^{2}FA surpasses uncompressed DFA in throughput, due to their small memory footprint and high cache hit rate. We find that with a modest 1 KB data cache, that can be easily provided in today's NPs, CD^{2}FA achieves two times increased throughput as compared to an uncompressed DFA, and at the same time requires only 10% of the memory required by a table compressed DFA. Thus, CD^{2}FAs can implement regular expressions much more economically and improve throughput and scalability in the number of rules. A recent development [Becchi, and Crowley 2007] constructs compact D^{2}FA with an amortized parsing complexity of less than 2 memory accesses per input character. Such structures can be coupled with content addressing to enable further space compression.

Our second contribution is H-FA, which is a novel machine that solves the problem of DFA state explosions while maintaining a high parsing performance. In one way, H-FA is similar to a NFA, in that the per character execution complexity of the machine is $O(k)$, where k is the maximum number of concurrent partial matches, say due to the presence of k Kleene closures in the regular expressions. H-FA, however, achieves high parsing performance by partitioning itself into two components, a finite automaton and a history buffer containing a set of flags. The automaton requires a single state traversal per character, thus can be stored in off-chip memory and still be executed at high rates; the history buffer on the other hand requires k parallel examinations, however it is extremely compact (requires only k-bits) and can be stored in fast on-chip registers. Thus, even though the theoretical execution complexity of H-FA is $O(k)$ for k closures in the regular expressions, it can run as fast as a DFA in practice, while avoiding the state explosions of the DFA.

Chapter 6

Summary

IP header lookup and packet content inspection are two important and well-studied topics. Despite the enormous amount attention of the research community, there remain a number of ripe areas for contribution. In this thesis, we describe a number of novel algorithms and architectures for realizing these two functions that aim at advancing the current state-of-the art in a number of different ways. While our focus is throughput, primarily due to the persistent concern over rapidly increasing data rates in the Internet, we also give a fair amount of attention to the other important aspects of implementation such as power consumption, silicon die area, scalability, and robustness. It is often critical that the desired level of throughput is achieved with a reasonably low power consumption, and die area. Scalability and robustness of the architecture, on the other hand, decides how long the architecture will remain relevant and adaptable to meet the newly emerging performance pressures, and how well it will cope with the unforeseeable security threats and changing usage patterns. In today's competitive networking device marketplace, and due to a plethora of existing solutions, these issues become a decisive factor in the success and wide adoption of any architecture.

Centered on the aforementioned four performance aspects, the platforms that we use to evaluate our algorithms are ASICs and network processors. These two platforms are the most commonly used by current network equipment vendors, and each has their own benefits and drawbacks. What differentiates these platforms is a tradeoff between performance and programmability. Network processors provide an unparalleled level of programmability, which enables quicker implementation, and upgrades of any given function. ASICs, on the other hand, are known to provide a high degree of parallel

computation capability, which can be used to achieve high performance and efficiency. Consequently, there are marked differences in the implementation approach employed in ASICs and network processors. Algorithms that require ample parallel computation are preferable in an ASIC. Current ASICs also provide a moderate amount of high bandwidth on-chip memory. Thus, such algorithms can be practically implemented that require compact data-structures but demands high memory bandwidth to enable high performance. Unlike ASICs, current network processors have very limited or no on-chip memory for data storage and off-chip memory is used as the primary storage for all data-structures. Even though off-chip memory size has grown substantially recently, its bandwidth has remained a premium, hence the algorithms used in network processors often seek to keep the number of memory accesses small, even though it requires an increased memory size. While developing our algorithms, we keep these implementation tradeoffs in mind. In addition to these platform specific evaluations, we also conduct a preliminary evaluation of our algorithms in an abstract sense, where our primary metrics are computation complexity, and memory bandwidth and space required.

In IP lookup, which is one of our primary topics, we introduce two novel architectures that complement each other and can be used to design an ASIC can enable high lookup throughput at low levels of power consumption and memory size. The first architecture is called CAMP, which is a pipelined IP lookup architecture based on multi-point access circular pipeline of traditional memories. A key feature of the architecture is that the number of stages in the pipeline is decoupled from the number of levels in the trie. Hence, a large number of smaller memory stages can be employed, leading to a higher throughput at lower die area and power dissipation. The architecture also allows near optimal memory utilization and also ensures that all pipeline stages are equal in size. CAMP also ensures fast incremental updates, which has been validated on a collection of real and synthetic prefix sets. To complement the high lookup throughput made possible with CAMP, we develop HEXA, which is a novel representation for structured graphs such as IP lookup tries that substantially reduces the memory required. HEXA uses a unique method to locate the nodes of the trie in memory, which enables it to

avoid using any "next node" pointer. Since these pointers often consume most of the memory required by the lookup trie, HEXA based representations are significantly more compact than the standard representations. We validate HEXA with a number of well known IP lookup databases which results in a memory reduction of up to 3-times over the state-of-the art methods. Such memory reductions are essential in an ASIC setting, where fast but limited amount of embedded memories are available, and can be used to dramatically improve the packet throughput and reduce the power dissipation.

As part of the experimental evaluation, we thoroughly present and discuss the die area, power consumption, and lookup throughput achieved when a combination of HEXA and CAMP is used to implement IP lookup function in an ASIC setting. Additionally, we also present a preliminary theoretical analysis of the algorithms used in the two architectures. More specifically, in HEXA, we derive an approximate bound on the memory requirements, by borrowing analyses from the Cuckoo hashing. The analysis establishes that the memory requirements in HEXA is $O(n)$, for n node IP lookup trie, as compared to the $O(n \log n)$ memory required in standard solutions. The analysis of the CAMP has been limited to a number of experiments carried over a set of synthetic IP lookup tables, which establishes an approximate experimental bound on the memory requirement and pipeline imbalance.

In the area of deep packet inspection using regular expressions signatures, we introduce a number of new representations for regular expressions. The first representation is delayed input DFA (D^2FA), which significantly reduces the memory requirements of a DFA by replacing its multiple transitions with a single default transition. By reduction, we show that the construction of an efficient D^2FA from a DFA is NP-hard. We therefore present heuristics for D^2FA construction that provide deterministic performance guarantees. Our results suggest that a D^2FA constructed from a DFA can reduce memory space requirements by more than 95%. Thus, the entire automaton can possibly fit in on-chip memories of an ASIC. Since embedded memories provide ample bandwidth, further space reductions are possible by splitting the regular expressions into

multiple groups and creating a D^2FA for each of them, and storing and executing them independently.

As a side effect, a D^2FA introduces a cost of possibly several memory accesses per input character, since it requires multiple default transitions to consume a single character. Therefore, a careful implementation is required to ensure deterministic, and good performance. We present a memory-based architecture, which uses multiple embedded memories, and show how to map the D^2FAs onto them in such a way that each character is effectively processed in a single memory cycle. As a proof of concept, we construct D^2FAs from regular expression sets used in many widely used systems, including those employed in the widely used security appliances from Cisco Systems, that required less than 2 MB of embedded memory and provided up to 10 Gbps throughput at a modest clock rate of 300 MHz. Our ASIC architecture provides deterministic performance guarantees and suggests that with today's VLSI technology, a worst-case throughput of OC192 can be achieved while simultaneously executing several thousands of regular expressions.

Our second contribution is Content Addressed Delayed Input DFA (CD^2FA), which provides a compact representation of regular expressions suitable for implementation in network processor platforms. A CD^2FA is built upon the D^2FA, whose state numbers are replaced with content labels. The content labels compactly contain information which are sufficient for the CD^2FA to avoid any default traversal, thus avoiding unnecessary memory accesses and hence achieving higher throughput. While a CD^2FA requires number of memory accesses equal to those required by an uncompressed DFA, with the addition of small data caches that are increasingly becoming available in current network processors, CD^2FA surpasses uncompressed DFA's in throughput, due to its small memory footprint and therefore higher cache hit rate. We find that with a modest 1 KB data cache, CD^2FA achieves almost two times higher throughput as compared to an uncompressed DFA, and also requires less than 10% of the memory required by a DFA compressed with standard table compression. Consequently, CD^2FAs can

implement regular expressions much more economically and improve the throughput and scalability in the number of rules in a network processor based implementation environment. Given the anticipated effect of Internet growth, increasing number of security threats and diversification of the regular expression signature sets, both D^2FA and CD^2FA based solutions are expected to gain wide adoption and popularity in the immediate future.

References

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

[2] S. Antonatos, et al. *Generating realistic workloads for network intrusion detection systems*. ACM Workshop on Software and Performance, 2004.

[3] F. Baboescu, D. M. Tullsen, G. Rosu, and S. Singh. *A Tree Based Router Search Engine Architecture with Single Port Memories*. Proc. ISCA, 2005.

[4] Z. K. Baker, and V. K. Prasanna. *Automatic Synthesis of Efficient Intrusion Detection Systems on FPGAs*. Proc. Field Prog. Logic and Applications, pp. 311-321, Aug. 2004.

[5] A. Basu and G. Narlikar. *Fast Incremental Updates for Pipelined Forwarding Engines*. Proc. IEEE INFOCOM, 2003.

[6] M. Becchi, and P. Crowley: An improved algorithm to accelerate regular expression evaluation. ANCS 2000.

[7] Y. H. Cho, and W. H. Mangione-Smith. *Deep Packet Filter with Dedicated Logic and Read Only Memories*. Proc. Field Prog. Logic and Applications, pp. 125-134, Aug. 2004.

[8] C. R. Clark and D. E. Schimmel. *Efficient reconfigurable logic circuit for matching complex network intrusion detection patterns*. Proc. 13th International Conference on Field Program, 2003.

[9] M. Degermark, A. Brodnik, S. Carlsson and S. Pink. Small Forwarding Tables for Fast Routing Lookups. Proc. ACM SIGCOMM 1997.

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

[11] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor. *Longest prefix matching using Bloom filters*. Proc. ACM SIGCOMM, 2003.

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

[13] W. Eatherton, and J. Williams. *An encoded version of reg-ex database from cisco systems provided for research purposes*. 2005.

[14] 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.

[15] D. Fotakis, et al. *Space efficient hash tables with worst case constant access time*. Proc. STACS, 2003.

[16] M. R. Garey, and D. S. Johnson. *Bounded Component Spanning Forest*. Computers and Intractability: A Guide to the Theory of NP-Completeness, pp. 208, 1979.

[17] M. Gokhale, et al. *Granidt: Towards Gigabit Rate Network Intrusion Detection Technology*. Proc. Field Programmable Logic and Applications, pp. 404-413, Sept. 2002.

[18] R. Graham, F. Graham, and G. Varghese. *Parallelism versus Memory Allocation in Pipelined Router Forwarding Engines*. Proc. SPAA'04, Barcelona, Spain, 2004.

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

[20] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 1979.

[21] J. E. Hopcroft. *An nlogn algorithm for minimizing states in a finite automaton*. Theory of Machines and Computation, J. Kohavi, Ed. New York: Academic, 1971, pp. 189-196.

[22] G. Huston. *Analyzing the Internet's BGP routing table*. Internet Protocol Journal, 4(1), 2001.

[23] A. Kirsch, and M. Mitzenmacher. *Simple Summaries for Hashing with Multiple Choices*. Proc. Forty-Third Annual Allerton Conference on Communication, Control, and Computing, 2005.

[24] J. B. Kruskal. *On the shortest spanning subtree of a graph and the traveling salesman problem*. Proc. American Mathematical Society, Vol. 7, pp. 48-50, 1956.

[25] S. Kumar, et al. *Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection*. Proc. ACM SIGCOMM, Pisa, Italy, September 12-15, 2006.

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

[27] C. Labovitz, G. R. Malan, and F. Jahanian. *Origins of Internet Routing Instability*. Proc. Infocom, New York, NY, March 1999.

[28] N. J. Larsson. *Structures of string matching and data compression*. PhD thesis, Dept. of Computer Science, Lund University, 1999.

[29] J. Levandoski, E. Sommer, and M. Strait. *Application Layer Packet Classifier for Linux*. http://l7-filter.sourceforge.net/.

[30] F. M. Liang. *A lower bound for on-line bin packing*. Information Processing letters, pp. 76-79, 1980.

[31] A. J. McAuley, and P. Francis. *Fast Routing Table Lookup Using CAMs*. Proc. INFOCOM, 1993.

[32] 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.

[33] S. Nilsson and G. Karlsson. *Fast Address Lookup for Internet Routers*. Proc. IEEE Conf. on BroadBand Communications Tech., 1998.

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

[35] V. Paxson, et al. *Flex: A fast scanner generator*. http://www.gnu.org/software/flex/.

[36] R. Prim. *Shortest connection networks and some generalizations*. Bell System Technical Journal, Vol. 36, pp. 1389-1401, 1957.

[37] M. Roesch. *Snort: Lightweight intrusion detection for networks*. Proc. 13th Systems Administration Conference (LISA), USENIX Association, pp. 229-238, Nov 1999.

[38] S. T. Shafer, and Mark Jones. *Network edge courts apps*. http://infoworld.com/article/02/05/27/020527newebdev_1.html.

[39] R. Sidhu, and V. K. Prasanna. *Fast regular expression matching using FPGAs*. Proc. IEEE Symposium on Field- Programmable Custom Computing Machines, Rohnert Park, CA, USA, April 2001.

[40] R. Sommer, and V. Paxson. *Enhancing Byte-Level Network Intrusion Detection Signatures with Context*. Proc. ACM conf. on Computer and Communication Security, 2003, pp. 262-271.

[41] H. Song, J. Turner, and J. Lockwood. *Shape Shifting Tries for Faster IP Route Lookup*. Proc. IEEE ICNP, 2005.

[42] H. Song, and J. Turner. *Fast Filter Updates in TCAMs for Packet Classification*. Proc. IEEE Globecom, San Francisco, CA, Nov 27 – Dec 1, 2006.

[43] 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, pp. 258–267, Apr. 2004.

[44] E. Spitznagel, D. Taylor and J. Turner. *Packet Classification Using Extended TCAMS*. Proc. ICNP, November 2003.

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

[46] S. Suri, G. Varghese, and P. Warkhede. *Multiway range trees: Scalable IP lookup with fast updates*. Proc. GLOBECOM, 2001.

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

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

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

[50] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. *Scalable High Speed IP Routing Lookups*. Proc. ACM SIGCOMM, pp. 25-37, 1997.

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

[52] F. Yu, et al. *Fast and Memory-Efficient Regular Expression Matching for Deep Packet Inspection*. UCB tech. report, EECS-2005-8.

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

[54] F. Zane, G. Narlikar, and A. Basu. *CoolCAMs: Power-Efficient TCAMs for Forwarding Engines*. Proc. INFOCOM, 2003.

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

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

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

[58] Cisco IOS IPS Deployment Guide. www.cisco.com.

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

[60] Cu-11 standard cell/gate array ASIC. IBM. www.ibm.com.

[61] MIT DARPA Intrusion Detection Data Sets. http://www.ll.mit.edu/IST/ideval/data/2000/2000_data_index.html.

[62] Network Services Processor. OCTEON CN31XX. CN30XX Family.

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

[64] SafeXcel Content Inspection Engine. Hardware regex acceleration IP.

[65] Tarari RegEx. www.tarari.com/PDF/RegEx_FACT_SHEET.pdf.

[66] TippingPoint X505. www.tippingpoint.com/products_ips.html.

[67] Virtex-4 FPGA. Xilinx. www.xilinx.com.

Vita

Sailesh Kumar

Date of Birth April 15, 1980

Place of Birth Bokaro Steel City, India

Degrees B.Tech. IIT Kanpur, Electrical Engineering, May 2000
D.Sc. Computer Science, December 2007

Publications

S. Kumar, J. Turner, and P. Crowley. Peacock Hash: Fast and Updatable Hashing for High Performance Packet Processing Algorithms. *Proc.* ***IEEE INFOCOM***, Arizona, 2008.

S. Kumar, B. Chandrasekaran, J. Turner, and G. Varghese. Curing Regular Expressions Matching Algorithms from Insomnia, Amnesia, and Acalulia. *Proc.* ***IEEE/ACM ANCS***, Orlando, Florida, December, 2007.

S. Kumar, J. Turner, P. Crowley, and M. Mitzenmacher. HEXA: Compact Data Structures for Faster Packet Processing. *Proc.* ***IEEE ICNP***, Beijing, China, October, 2007.

J. Turner, P. Crowley, J. Dehart, A. Freestone, B. Heller, F. Kuhns, S. Kumar, J. Lockwood, J. Lu, M.Wilson, C. Wiseman, and D. Zar. Supercharging PlanetLab - High Performance, Multi-Application, Overlay Network Platform. *Proc.* ***ACM SIGCOMM***, Kyoto, Japan, August, 2007.

S. Kumar, J. Turner, and J. Williams. Advanced Algorithms for Fast and Scalable Deep Packet Inspection. *Proc.* ***IEEE/ACM ANCS***, San Jose, California, December, 2006.

S. Kumar, M. Becchi, P. Crowley, and J. Turner. CAMP: Fast and Efficient IP Lookup Architecture. *Proc.* ***IEEE/ACM ANCS***, San Jose, California, December, 2006.

S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner. Algorithms to Accelerate Multiple Regular Expressions

Matching for Deep Packet Inspection. *Proc.* ***ACM SIGCOMM***, Pisa, Italy, September 12-15, 2006.

S. Kumar, J. Maschmeyer, and P. Crowley. Queuing Cache: Exploiting Locality to Ameliorate Packet Queue Contention and Serialization. *Proc.* ***ACM ICCF***, Ischia, Italy, May 2-5, 2006.

S. Kumar, and P. Crowley. Segmented Hash: An Efficient Hash Table Implementation for High Performance Networking Subsystems. *Proc.* ***IEEE/ACM ANCS***, Princeton, October, 2005.

S. Kumar, P. Crowley, and J. Turner. Buffer Aggregation: Addressing Queuing Subsystem Bottlenecks at High Speeds. *Proc.* ***IEEE Hot-Interconnects-13***, Stanford, August 17-19, 2005.

S. Kumar, P. Crowley, and J. Turner. Design of Randomized Multichannel Packet Storage for High Performance Routers. *Proc.* ***IEEE Hot-Interconnects-13***, Stanford, August 17-19, 2005.

Jan 2008