Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection

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

Sarang Dharmapurikar
Washington University
Computer Science and Engineering
St. Louis, MO 63130-4899
+1-314-935-8563
sarang@arl.wustl.edu

Fang Yu
University of California, Berkeley
Department of Computer Science
Berkeley, CA 94720
+1-510-642-8284
fyu@eecs.berkeley.edu

Patrick Crowley
Washington University
Computer Science and Engineering
St. Louis, MO 63130-4899
+1-314-935-9186
pcrowley@wustl.edu

Jonathan Turner
Washington University
Computer Science and Engineering
St. Louis, MO 63130-4899
+1-314-935-8552
jon.turner@wustl.edu

ABSTRACT

There is a growing demand for network devices capable of examining the content of data packets in order to improve network security and provide application-specific services. Most high performance systems that perform deep packet inspection implement simple string matching algorithms to match packets against a large, but finite set of strings. However, there is growing interest in the use of regular expression-based pattern matching, since regular expressions offer superior expressive power and flexibility. Deterministic finite automata (DFA) representations are typically used to implement regular expressions. However, DFA representations of regular expression sets arising in network applications require large amounts of memory, limiting their practical application.

In this paper, we introduce a new representation for regular expressions, called the *Delayed Input DFA* (D^2FA), which substantially reduces space requirements as compared to a DFA. A D^2FA is constructed by transforming a DFA via incrementally replacing several transitions of the automaton with a single default transition. Our approach dramatically reduces the number of distinct transitions between states. For a collection of regular expressions drawn from current commercial and academic systems, a D^2FA representation reduces transitions by more than 95%. Given the substantially reduced space requirements, we describe an efficient architecture that can perform deep packet inspection at multi-gigabit rates. Our architecture uses multiple on-chip memories in such a way that each remains uniformly occupied and accessed over a short duration, thus effectively distributing the load and enabling high throughput. Our architecture can provide cost-effective packet content scanning at OC-192 rates with memory requirements that are consistent with current ASIC technology.

SIGCOMM'06, September 11-15, 2006, Pisa, Italy.

Categories and Subject Descriptors

C.2.0 [**Computer Communication Networks**]: General – Security and protection (e.g., firewalls)

General Terms

Algorithms, Design, Security.

Keywords

DFA, regular expressions, deep packet inspection.

1. INTRODUCTION

Many critical network services handle packets based on payload content, in addition to the structured information found in packet headers. 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 [5], Bro [4], 3Com's TippingPoint X505 [20], and various network security appliances from Cisco Systems [21]. Cisco, in fact, has integrated the regular expression based content inspection capabilities into its Internetworking Operating System (IOS) [21]. Additionally, layer 7 filters based on regular expressions [30] 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 [2, 3]. 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. Table 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.

In this paper, we introduce a highly compact DFA representation. Our approach reduces the number of transitions associated with each state. The main observation is that groups of states in a DFA often have identical 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 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 on-chip memory, organized in multiple independent modules. Modern VLSI technology easily enables this sort of integration of several embedded memories on a single die; for example, IBM's ASIC fabrication technology [23] can integrate up to 300 Mbits of embedded memory on one chip. 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, our contributions are *a*) the D^2FA representation of regular expressions which significantly reduces the amount of memory required, *b*) a single-chip architecture that uses the D^2FA representation, and *c*) a load balancing algorithm which ensures that on-chip resources are uniformly used, thereby enabling worst-case performance guarantees.

The remainder of the paper is organized as follows. Background on regular expressions and related work are presented in Section 2. Section 3 describes the D^2FA representation. Details of our construction algorithm and the compression results are presented in Section 4. Section 5 presents the system architecture, load balancing algorithms and throughput results. The paper ends with concluding remarks in Section 6.

2. BACKGROUND AND RELATED WORK

Deep packet inspection has recently gained popularity as it provides the 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" [19]. Some applications requiring deep packet inspection are listed below:

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

Deep packet inspection often involves scanning every byte of the packet payload and identifying a set of matching predefined patterns. Traditionally, rules have been represented as exact match strings consisting of known patterns of interest. Naturally, due to their wide adoption and importance, several high speed and efficient string matching algorithms have been proposed recently. Some of the standard string matching algorithms such as Aho-Corasick [7] Commentz-Walter [8], and Wu-Manber [9], use a preprocessed data-structure to perform high-performance matching. A large body of research literature has concentrated on enhancing these algorithms for use in networking. In [11], Tuck et al. presents 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 pattern matching hardware architectures. In [12] Tan et al. propose an efficient algorithm that converts an Aho-Corasick automaton into multiple binary state machines, thereby reducing the space requirements. In [13], the authors present an FPGA-based design which uses character pre-decoding coupled with CAM-based pattern matching. In [14], Yusuf et al. use hardware sharing at the bit level to exploit logic design optimizations, thereby reducing the area by a further 30%. Other work [25, 26, 27, 28, 29] presents several efficient string matching architectures; their performance and space efficiency are well summarized in [14].

In [1], Sommer and Paxson note that regular expressions might prove to be fundamentally more efficient and flexible as compared to exact-match strings when specifying attack

signatures. The flexibility is due to the high degree of expressiveness achieved by using character classes, union, optional elements, and closures, while the efficiency is due to the effective schemes to perform pattern matching. Open source NIDS systems, such as Snort and Bro, use regular expressions to specify rules. Regular expressions are also the language of choice in several commercial security products, such as TippingPoint X505 [20] from 3Com and a family of security appliances from Cisco Systems [21]. Although some specialized engines such as RegEx from Tarari [22] report packet scan rates up to 4 Gbps, the throughput of most such devices remains limited to sub-gigabit rates. There is great interest in and incentive for enabling multi-gigabit performance on regular expressions based rules.

Consequently, several researchers have recently proposed specialized hardware-based architectures which implement finite automata using fast on-chip logic. Sindhu et al. [15] and Clark et al. [16] have implemented nondeterministic finite automata (NFAs) on FPGA devices to perform regular expression matching and were able to achieve very good space efficiency. Implementing regular expressions in custom hardware was first explored by Floyd and Ullman [18], who showed that an NFA can be efficiently implemented using a programmable logic array. Moscola et al. [17] have used DFAs instead of NFAs and demonstrated significant improvement in throughput although their datasets were limited in terms of the number of expressions.

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

Commercial content inspection engines like Tarari's RegEx already emphasize the ease of programmability provided by a dense multiprocessor architecture coupled to a memory. Content inspection engines from other vendors [33, 34], also use memory-based architectures. In this context, Yu et al. [10] have proposed an efficient algorithm to partition a large set of regular expressions into multiple groups, such that overall space needed by the automata is reduced dramatically. They also propose architectures to implement the grouped regular expressions on both general-purpose processor and multi-core processor systems, and demonstrate an improvement in throughput of up to 4 times. In this paper, we extend these memory-based architectures and propose algorithms which can enable the efficient implementation of regular expressions at multi-gigabit rates while preserving the flexibility provided by programmability.

3. DELAYED INPUT DFAS

It is well-known that for any regular expression set, there exists a DFA with the minimum number of states [3]. 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 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 [7]. We extend their technique and apply it to DFAs obtained from regular expressions, rather than simple string sets.

3.1 Motivating Example

We introduce our approach using an example. The left side of Figure 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 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 12123141135. Notice that the sequence of match states is the same, so if the second 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 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 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 a more compact representation, at the cost of some reduction in the worst-case performance.

3.2 Problem Statement

We refer to an automaton with default transitions as a *Delayed Input DFA* (D^{2}FA). We represent a D^{2}FA by a directed graph, whose vertices are called *states* and whose edges are called

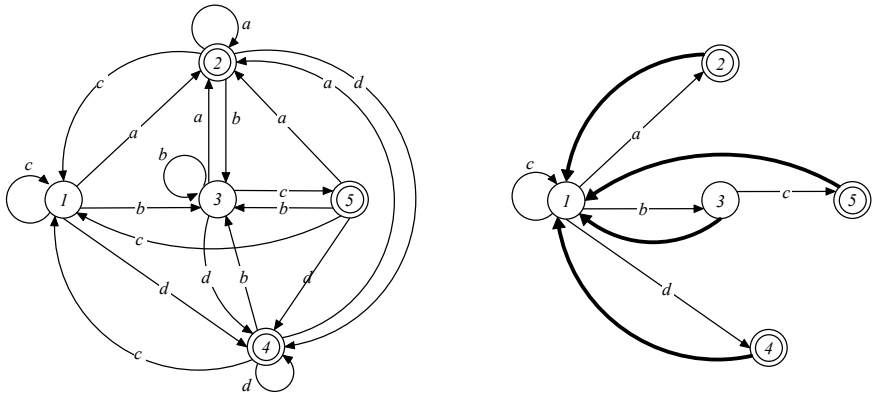

Figure 1. Example of automata which recognize the expressions a^+, b^+c, and c^*d^+

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*, μ(*s*).

For any input string $x \in \Sigma^*$, we define the *destination state*, δ(*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 D²FA on the right side of Figure 1, δ(abcb)=3 and δ(dcbac)=1). We generalize δ to accept an arbitrary starting state as a second argument; so for the D²FA on the right side of Figure 1, δ(abcb,2) =3.

Consider two D²FAs 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 D²FA that is substantially more memory-efficient.

We can bound the worst-case performance of a D²FA 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 D²FA will consume at least one character for every *k* transitions followed. To ensure that a D²FA 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 D²FA that satisfies a specified bound on default path length.

3.3 Converting DFAs to D²FAs

Although, we are in general interested in any equivalent D²FA, for a given DFA, we have no general procedure for synthesizing a D²FA directly. Consequently, our procedure for constructing a D²FA 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 δ(*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 δ(*x*). A slightly more general version of this observation is stated below.

Lemma 1. Consider a D²FA with distinct states *u* and *v*, where *u* has a transition labeled by the symbol a, and no outgoing default transition. If δ(a,*u*)=δ(a,*v*), then the D²FA obtained by introducing a default transition from *u* to *v* and removing the transition from *u* to δ(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 δ(a,*u*)=δ(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 D²FA applies this transformation repeatedly. Hence, the equivalence of the initial and final D²FAs follows by induction. The D²FA on the right side of Figure 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).

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 δ(a,*u*)=δ(a,*v*). The space reduction graph for the DFA on the left side of Figure 1 is shown in Figure 2. Notice that the spanning tree of the space reduction graph that corresponds to the default transitions for the D²FA in Figure 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 3 shows D²FAs 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 1, they have default paths of length 3 and 2, respectively, meaning that their worst-case performance will not be as good.

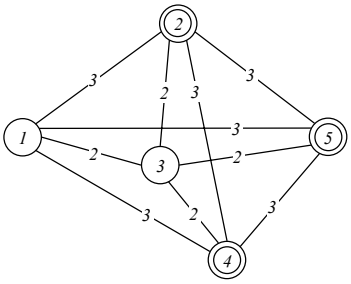

Figure 2. Space reduction graph for DFA in Figure 1.

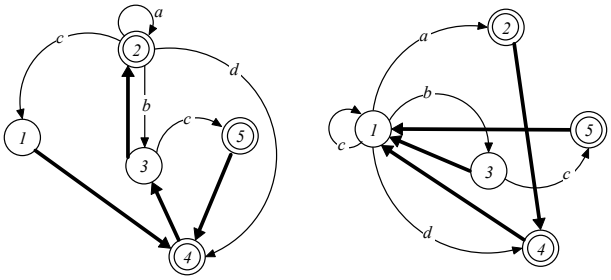

Figure 3. D²FAs corresponding to two different maximum weight spanning trees

4. BOUNDING DEFAULT PATHS

If our only objective was minimizing the space used by a D²FA, 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 D²FA 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 [39]. It's also not clear that such a spanning tree leads to the smallest D²FA. 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 D²FA construction. One conceptually straight-forward method builds a collection of trees incrementally. The method (which is based on Kruskal's algorithm [36]) 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 *default* 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 about the diameter buildup of the trees until the diameter bound is reached. In a setup, where the diameter bound is not applied, refined spanning tree algorithm creates default transition trees of equal weight but relatively smaller diameter. When diameter bound is applied, the

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

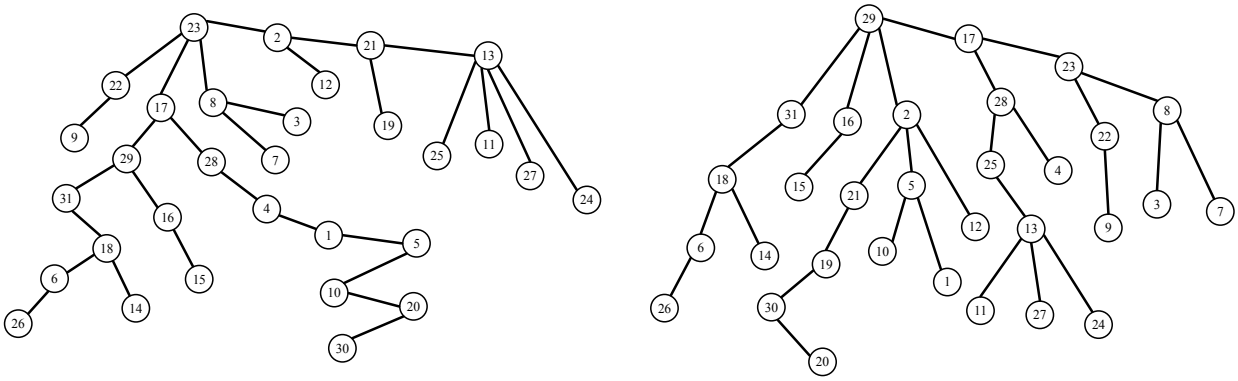

Figure 4. Default transition trees formed by the spanning tree algorithm and by our refined version

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 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 spanning tree algorithms without any diameter bound, they created spanning trees of weight 1771, as shown in Figure 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 root of these two trees, the longest default paths will contain 7 and 5 edges, while the average length of default paths will be 3.8 and 2.8, respectively.

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

When we bounded the diameter of the spanning tree to 7, and ran our algorithm on the same synthetic DFA, it created three default transition trees, as shown in Figure 5. The total weight of all three trees was 1653, which suggests that the resulting DF^{2}A will require slightly more space as compared to the one with no diameter restraint. However, bounding the diameter to 7 ensures an important property that the length of all default paths can be easily limited to 4 and hence the D^{2}FA will require at most 4 memory accesses per character.

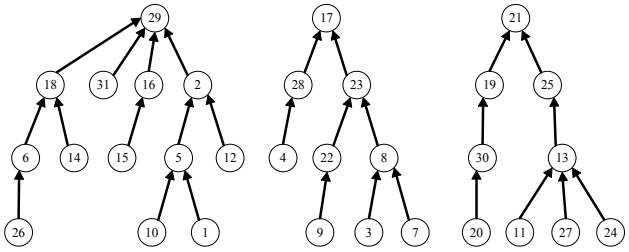

Figure 5. Default transition trees (forest) formed by the refined spanning tree with the tree diameter bounded to 7

4.1 Results on some regular expression sets

In order to evaluate the space reductions achieved by a delayed input DFA, or D^{2}FA, we performed experiments on regular expression sets used in a wide variety of networking applications. Our most important dataset are the regular expression sets used in deep packet inspection appliances from Cisco Systems [38]. 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 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. Linux layer-7 protocol classifier consists of 70 expressions. Snort contains more than a thousand and half 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 to construct DFAs with a small number of states, we used the set splitting techniques proposed by Yu et al. in [10]. It 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 efficient grouping resulted in 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 $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 [10] 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 1.

In order to estimate the reduction objectives of D^{2}FA, we introduce a term *duplicate transition*. Transitions are *duplicate* if

Table 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

there exists more than one of them leading to the same "next state" for the same input character. For example in Figure 1, if the transitions on input *b* from states 1 is termed original then the ones from state 2, 3, 4 and 5 are duplicates. Even though, it may not be possible to eliminate all duplicate 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 spanning tree to construct the corresponding D^2FAs. The reduction in the number of transitions is shown in Table 2 with no diameter bounds applied. The length of default paths, which gives an estimate of the added memory bandwidth a D^2FA will need over a DFA, are also shown. It is clear that, D^2FAs eliminates nearly all duplicate transitions from the DFAs. It is also apparent that refined version of spanning tree creates substantially smaller default paths as compared to a normal spanning tree. In order to get a sense of the distribution of the number of labeled transitions per states of a D^2FA, we plot it in Figure 6, for the Cisco regular expression group containing 590 expressions. Majority of states has 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].`

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 normal spanning tree and by our refined version. In Table 3, we

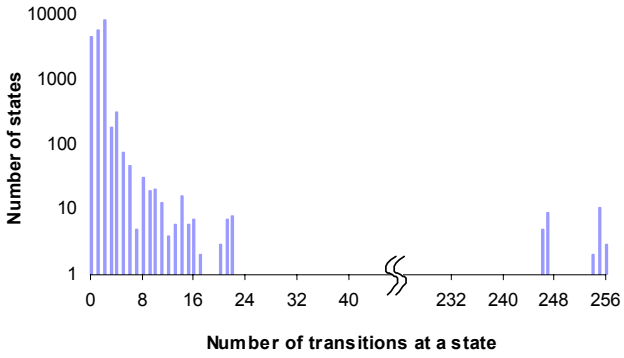

Figure 6. Distribution of number of transitions per state in the D^2FA constructed from the Cisco590 expression set

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 7, we plot the reduction in the number of transitions of a DFA, as ratio of number transitions in the D^2FA and 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 relatively 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

Table 3. Number of transitions in D^2FA with default path length bounded to 4

DFA	Normal spanning tree	Refined spanning tree
Cisco590	97873	70793
Cisco103	115654	82879
Cisco7	37520	36091
Linux56	69437	66739
Linux10	314915	302112
Snort11	180545	178354
Bro648	11906	8078

Table 2. Original DFA and the D^2FA constructed using the normal and the refined spanning tree, without any diameter bound

	Original DFA					Delayed input DFA, D^2FA							
						Normal spanning tree				Refined spanning tree			
DFA	*Total # of states*	*Total # of transitions*	*Total # of distinct transitions*	*Total # of duplicate transitions*	*% duplicates*	*Total # of transitions*	*% reduction*	*Avg. default length*	*Max. default length*	*Total # of transitions*	*% reduction*	*Avg. default length*	*Max. default length*
Cisco590	17713	4534528	1537238	4509852	99.45	36519	99.2	18.32	57	36519	99.2	8.47	17
Cisco103	21050	5388800	1236587	5346595	99.21	53068	99.0	16.65	54	53068	99.0	7.82	19
Cisco7	4260	1090560	312082	1063896	97.55	28094	97.4	19.61	61	28094	97.4	10.91	23
Linux56	13953	3571968	590917	3517044	98.46	58571	98.3	7.68	30	58571	98.3	5.62	21
Linux10	13003	3328768	962299	3052433	91.69	285991	91.3	5.14	20	285991	91.3	4.64	17
Snort11	41949	10738944	540259	10569778	98.42	168569	98.4	5.86	9	168569	98.4	3.43	6
Bro648	6216	1591296	149002	1584357	99.56	7082	99.5	6.45	17	7082	99.5	2.59	8

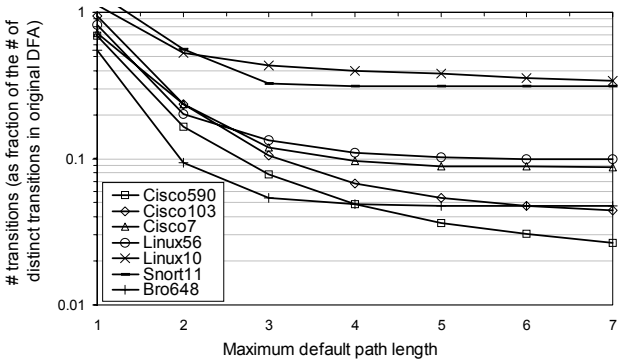

Figure 7. Plotting total number of labeled transitions in D²FAs for various maximum default path length bounds

compression scheme which enables it to store only the distinct transitions. If we would use the total transitions in a DFA as our metric, D²FA will result in even higher reduction.

4.2 Summarizing the results

The results suggest that a delayed input DFA or D²FA can substantially reduce the space requirements of regular expression sets used in many networking applications. For example, using D²FA, we were able to reduce the space requirements of regular expressions used in deep packet inspection appliances of Cisco Systems to less than 2 MB. 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.

D²FA 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 D²FAs, all of which needs to be executed in parallel. Although, D²FA performs 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 clever rule partitioning). Such a design choice makes sense in our context, because we use multiple embedded memories, which provides us with ample bandwidth, but limited capacity. We now present our architecture and algorithms to map the D²FAs onto them.

5. REGEX SYSTEM ARCHITECTURE

In this section, we propose an efficient regular expression engine consisting of multiple embedded memories and processors. We also propose algorithms to efficiently map the D²FA onto the architecture.

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 [18]. Using memory rather than logic allows the architecture to remain flexible in the face of frequently updating regular expressions. In addition to dense ASIC embedded memory technologies like IBM's [23], modern FPGAs such as the Xilinx Virtex-4 contain several hundreds of 18Kbit memory blocks [24] 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 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 design is eminently practical. Given this, we design our embedded memory architecture 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 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 D²FA 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.

5.1 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

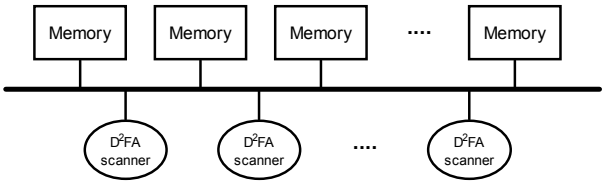

Figure 8. Logical structure of the memory subsystem

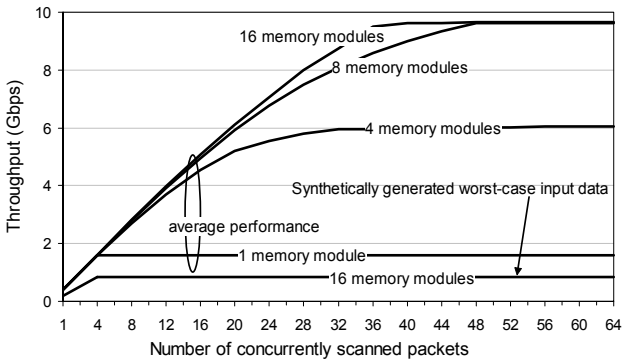

Figure 9. Throughput with default path length bounded to 7 and using the randomized mapping

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, $1-e^{-1}$ will 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 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^{2}FA to 7. The input data was generated from the MIT DARPA Intrusion Detection Data Sets [31]. 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 9. This can be explained as follows. A D^{2}FA often needs to traverse multiple default transitions for a character; if the maximum default path length is limited 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.

```
procedure max-min-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;
```

5.2 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 D^{2}FA 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.

5.2.1 Deterministic and Robust Mapping

The max-min algorithm is similar to the first-fit, decreasing bin-packing heuristic [37], 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

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

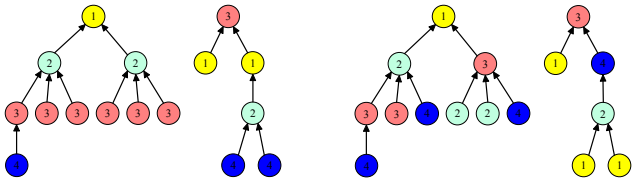

Figure 10. Left diagram shows two trees colored by max-min algorithm. Right diagram shows a better coloring

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 paths are guaranteed to get different colors. In order to ensure that colors are nearly equally used, max-min 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, max-min 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 max-min algorithm often leads to uneven memory usage. A simple example is shown on the left hand side of Figure 10, where there are two trees which are colored with 4 colors. With the max-min 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 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.

5.2.2 Adaptive coloring algorithm

The max-min 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 10, the max-min 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^{2}FA 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 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

```
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;
```

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 11.

5.2.3 Coloring results

In order to evaluate, how uniformly the min-max and adaptive algorithms utilize various colors, we generated D^{2}FA 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

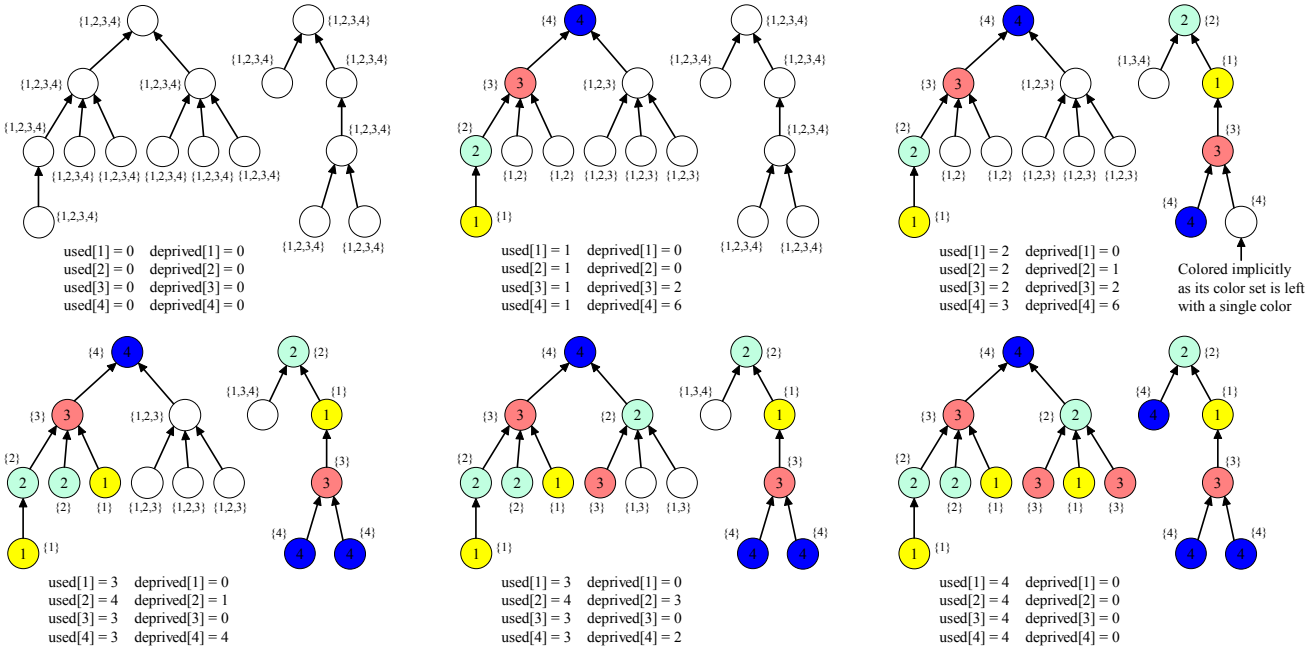

Figure 11. Various steps involved in the coloring of two trees with adaptive algorithm (assuming equally sized vertices)

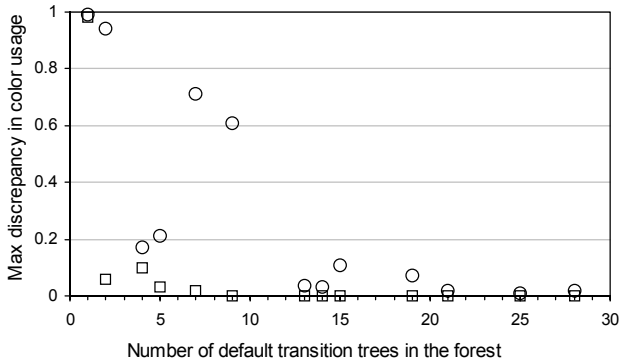

Figure 12. Plotting maximum discrepancy in color usage, circles for max-min and squares for adaptive algorithm

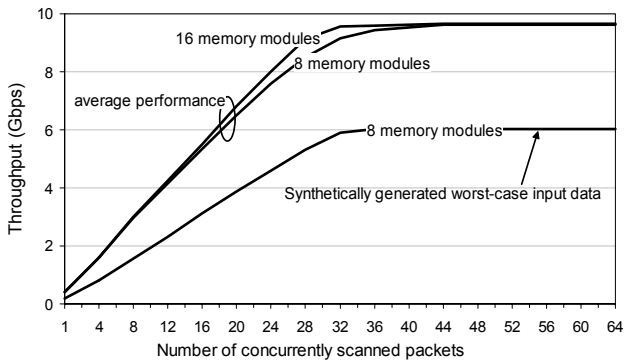

Figure 13. Throughput with default path length bounded to 7 and using adaptive-coloring based deterministic mapping

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 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 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.

We finally report the throughput of the D^{2}FAs generated from the Cisco rules, with the default path length limited to 7, in Figure 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 data set [31]. 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.

6. CONCLUDING REMARKS

In this paper, we introduce a new representation for regular expressions, called the delayed input DFA (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 on-chip memories. 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^2FA introduces a cost of possibly several memory accesses per input character, since D^2FAs may 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 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 were able to 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 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 thousand regular expressions.

7. ACKNOWLEDGMENTS

We are grateful to Will Eatherton and John Williams Jr. for providing us the regular expression sets used in Cisco security appliances. This work was supported by the NSF Grants CNS-0325298 and CCF-0430012 and URP grant from Cisco Systems.

8. REFERENCES

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[16] C. R. Clark and D. E. Schimmel, "Efficient reconfigurable logic circuit for matching complex network intrusion detection patterns," In Proceedings of 13th International Conference on Field Program.

[17] 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.

[18] 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.

[19] Scott Tyler Shafer, Mark Jones, "Network edge courts apps," http://infoworld.com/article/02/05/27/020527newebdev_1.html

[20] TippingPoint X505, www.tippingpoint.com/products_ips.html

[21] Cisco IOS IPS Deployment Guide, www.cisco.com

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

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

[24] Virtex-4 FPGA, Xilinx. www.xilinx.com

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

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

[27] Z. K. Baker, V. K. Prasanna, "Automatic Synthesis of Efficient Intrusion Detection Systems on FPGAs," in Field Prog. Logic and Applications, Aug. 2004, pp. 311–321.

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

[29] M. Gokhale, et al., "Granidt: Towards Gigabit Rate Network Intrusion Detection Technology," Field Programmable Logic and Applications, Sept. 2002, pp. 404–413.

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

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

[32] Vern Paxson et al., "Flex: A fast scanner generator," http://www.gnu.org/software/flex/

[33] SafeXcel Content Inspection Engine, hardware regex acceleration IP.

[34] Network Services Processor, OCTEON CN31XX, CN30XX Family.

[35] R. Prim, "Shortest connection networks and some generalizations," Bell System Technical Journal, 36:1389-1401, 1957.

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

[37] F. M. Liang. A lower bound for on-line bin packing. In Information Processing letters, pages 76-79, 1980.

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

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