ALGORITHMS FOR DEEP PACKET INSPECTION

By

Jignesh D. Patel

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

Computer Science

2012

ABSTRACT

ALGORITHMS FOR DEEP PACKET INSPECTION

By

Jignesh D. Patel

The core operation in network intrusion detection and prevention systems is Deep Packet Inspection (DPI), in which each security threat is represented as a signature, and the payload of each data packet is matched against the set of current security threat signatures. DPI is also used for other networking applications like advanced QoS mechanisms, protocol identification *etc.*. In the past, attack signatures were specified as strings. Today most DPI systems use Regular Expression (RE)s to represent signatures. RE matching for networking applications is difficult for several reasons. First, the DPI application is usually implemented in network devices, which have limited computing resources. Second, as new threats are discovered, the size of the signature set grows over time. Last, the matching needs to be done at network speeds, the growth of which outpaces improvements in computing speed; so there is a need for novel solutions that can deliver higher throughput. As a result, RE matching for DPI is a very important and active research area.

We study existing methods proposed for RE matching, identify their limitations, and propose new methods to overcome these limitations. RE matching remains a fundamentally challenging problem due to the difficulty in compactly encoding Deterministic Finite state Automata (DFA). While the DFA for any one RE is typically small, the DFA that corresponds to the entire set of REs is usually too large to be constructed or deployed.

To address this issue, many alternative automata implementations that compress the size of the final automaton have been proposed. We improve upon previous research in three ways. First, we propose a more efficient "Minimize then Union" framework for constructing compact alternative automata that minimizes smaller automata before combining them. Previously proposed automata construction algorithms employ a "Union then Minimize" framework where the automata for each RE are joined before minimization occurs. This leads to expensive minimization on a large automata and a large intermediate memory footprint. Our minimize then union approach requires much less time and memory, allowing us to handle a much larger RE set. Second, we propose the first hardware-based RE matching approach that uses Ternary Content Addressable Memory (TCAM). Prior hardware based RE matching algorithms typically use FPGA. The main drawback of FPGA is that resynthesizing and updating FPGA circuitry to handle RE updates is slow and difficult. In contrast, TCAM supports easy RE updates, and we show that we can achieve very high throughput. Furthermore, TCAMs are widely used in modern networking devices for tasks such as packet classification, so no major architecture modifications are needed to implement our approach in existing networking devices. Finally, we propose new overlay automata models that effectively address the replication of DFA states that occurs when multiple REs are combined. The idea is to group together the replicated DFA structures instead of repeating them multiple times. The result is that we get a final automata size that is close to that of a NFA (which is linear in the size of the RE set), and simultaneously achieve fast deterministic matching speed of a DFA.

ACKNOWLEDGMENTS

I would like to take this opportunity to thank all the people who have helped me during my graduate career and made this Dissertation possible.

First and foremost, I would like to thank my advisor, Dr. Eric Torng, for his constant guidance, support and encouragement.

I would like to express my earnest gratitude to my thesis committee members Dr. Richard Enbody, Dr. Alex Liu and Dr. Peter Magyar for being there for me whenever I needed.

I would also like to thank the staff of the CSE department for all their help and support.

Finally I would like to thank my friends and family for all their support and encouragement.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

1.1 Problem Statement

Deep Packet Inspection (DPI) is the core component of many networking devices on the Internet such as Network Intrusion Detection (or Prevention) Systems (NIDS/NIPS), firewalls, and layer 7 switches. In DPI, in addition to examining the packet headers, the entire contents of each packet is compared against a set of signatures to check if any signature is found in the packet or not. For instance, for security applications, each individual virus or attack threat is represented using one signature. The payload of each packet passing through the network device is compared against the set of signatures, and a match indicates the corresponding threat is found. Necessary action to neutralize the threat can then be taken. Application level signature analysis is also used for providing advanced QoS mechanisms, detecting peer-to-peer traffic, and in general application

protocol identification.

In the past, DPI typically used *string matching* as the core operation, in which signatures are specified as simple strings. Today, DPI typically uses *Regular Expression (RE) matching* as the core operation, in which signatures are specified as REs. REs are used instead of simple string patterns because REs are fundamentally more expressive and thus are able to describe a wider variety of attack signatures [43]. Most open source and commercial intrusion detection and prevention systems such as Snort [2, 39], Bro [37], HP TippingPoint and Cisco networking appliances use RE matching. Likewise, some operating systems such as Cisco IOS and Linux [1] have built RE matching into their layer 7 filtering functions.

So the problem we are trying to solve is as follows: given a set of REs, $\mathcal{R}$, and an input stream, we want to quickly find all occurrences of each RE from $\mathcal{R}$ in the input stream.

1.2 Research Problems

There are several challenges in implementing RE matching parsers for network applications. First, for many DPI applications, the signature set size rapidly grows over time. For example for security applications, new attack threats are regularly discovered and so the signature set size keeps growing. The current release of the Snort rules has close to 2000 REs in it. So the DPI engine should be able to handle a large RE set and it also needs to be scalable. Second, since each packet needs to be scanned in real time as it is

processed, the DPI engine needs to be able to process the packets at a fast and deterministic rate. As network speed increases, this becomes an increasingly difficult and important problem to solve. Finally, the DPI engine is typically implemented in a network device, like a router, which usually has limited memory and processing power. So the DPI engine needs to achieve the high throughput using limited hardware resources. As both traffic rates and signature set sizes are rapidly growing over time, fast and scalable RE matching is now a core network security issue. As a result, there has been a lot of recent work on implementing high speed RE parsers for network applications.

The straightforward approach to performing RE matching is to convert the RE set into an equivalent automata and use the packet payloads as input strings for the automata. Two standard choices are *Deterministic Finite state Automata (DFA)* and *Nondeterministic Finite state Automata (NFA)*. The DFA has the advantage of maintaining only a single active state at any time. Thus processing each input character requires only a single lookup, so the throughput achieved is fast and deterministic. However, DFAs experience state explosion where the number of states in the DFA can be exponential in terms of the number of REs. Thus, DFAs require too much memory to store them. The NFA has the advantage of small size where the number of states in the NFA is typically linear in the number of REs, hence requiring little memory to store them. However, the NFA has no limits on the number of active states, which means that the number of lookups needed to process each input character is high and unpredictable. So NFAs cannot achieve high and deterministic throughput.

1.3 Research Goals

As high and deterministic throughput is the primary requirement on networking devices, high speed RE matching is typically based on the DFA. But the high memory requirement of DFAs limits the number of REs in the ruleset that can be parsed simultaneously. In this thesis, we propose algorithmic solutions to implement RE matching based on the DFA that simultaneously achieves high throughput and low memory requirement.

Storing a DFA requires a large amount of memory because (1) the number of states grows exponentially with the number of REs, and (2) more states implies more transitions need to be stored since each state needs to store $256 = 2^8$ transitions.

The first research goal was to develop efficient algorithms that reduce the number of transitions of a DFA that need to be stored. The *Delayed Input DFA (D^2FA)* proposed by Kumar *et al.* [26] reduces the number of stored transitions by exploiting redundancy among the transitions. This and other previous techniques employ a "union then minimize" framework, in which they first build a large automata corresponding to all the REs in the ruleset, and then perform an expensive minimization on the large automata. We develop algorithms that use a "minimize then union" framework to build the D^2FA. In this approach we first minimize the automata corresponding to each individual RE in the ruleset, which is an inexpensive step because the automata are very small. We then use a fast algorithm to union the minimized automata together in such a way that the minimization is not lost. The D^2FA can be used for a software implementation of a DPI engine.

The compressed transition table is stored in RAM, and the processor does a RAM lookup for each transition of the automata. The drawback of implementing D^2FA in software is that the throughput is reduced (we explain this in Section 3.3.3.)

The second research goal was to find an efficient implementation of RE matching in networking device hardware. To this end, we develop techniques to implement the D^2FA for RE matching using *Ternary Content Addressable Memory (TCAM)*. TCAMs are already widely used in networking devices for header based packet forwarding, so our techniques can be implemented on current TCAM hardware without requiring major modifications. We also develop techniques to increase throughput by processing more than one input character in each cycle.

While the D^2FA is much smaller than a DFA, the memory requirement is still proportional to the number of DFA states, which grows exponentially with the number of REs. The ultimate goal for RE matching is to develop an automata model for RE matching that achieves throughput close to that of a DFA but only requires space close to that of a NFA. Our final research goal was to develop such an automata model. For this, we have developed two new automata models, Overlay Deterministic Finite state Automata (ODFA) and Overlay Delayed Input DFA (OD^2FA) as well as algorithms to implement OD^2FA automata in both software and hardware. Our hardware OD^2FA implementation achieves the speed of a DFA and the memory requirement of a NFA for many RE sets.

The rest of this proposal is organized as follows. In Chapter 2 we discuss related problems and research. Background about DFA, D^2FA and TCAM is presented in Chapter 3. Our

research related to D^2FA and implementing RE matching in TCAM is presented in Chapters 4 and 5, respectively. Chapter 6 presents our research for the OD^2FA automata model and implementation. Finally, Chapter 7 ends the dissertation with concluding remarks.

Chapter 2

Related Work

In the past, DPI typically used string matching (often called pattern matching) as a core operator; string matching solutions have been extensively studied [4, 5, 44, 46, 48, 49, 52]. Several TCAM-based solutions have been proposed for string matching [5, 12, 46, 52], but they do not generalize to RE matching because they only deal with independent strings and do not use DFAs. Sommer and Paxson [43] first proposed using REs instead of strings to specify attack signatures. Today most DPI engines uses RE matching as a core operator because strings are not adequate to precisely describe attack signatures.

There are two main approaches in previous work to developing RE matching solutions. One is to start with a DFA and compress it. The second is to start with an NFA and develop methods for coping with multiple active states.

We first review DFA compression work. Great work has been done in reducing the number of transitions stored per DFA state such as D^2FA [6, 8, 17, 26, 27]. These techniques exploit

transition redundancy between states to compress the size of the DFA. We present a novel "minimize then union" approach of building the D^2FA incrementally. Our approach can build much larger D^2FAs in fraction of the time compared to the previous solutions. This work is presented in [36]. Recently and independently, Liu *et al.* proposed to construct DFA by hierarchical merging [29]. That is, they essentially propose the "minimize then union" framework for DFA construction. They consider merging multiple DFAs at a time rather than just two. However, they do not consider D^2FA, and they do not prove any properties about their merge algorithm including that it results in minimum state DFAs.

Another approach to reducing the number of transitions stored per DFA state is alphabet encoding. In this approach the input characters are mapped to a new alphabet such that input characters which are always treated identically in the DFA are combined into one new character, thus reducing the size of the alphabet [8,9,13,22]. This work is orthogonal to our techniques, and can be used together to improve the results.

In [32] we present our current RE matching solution using TCAMs. Here we exploit both inter state and intra state transition redundancy to minimize the number of transitions stored per DFA state.

There has been work to increase the throughput by creating multi-stride DFAs and NFAs that scan multiple characters per transition [9,13]. This work primarily applies to FPGA NFA implementations since multiple character SRAM based DFAs have only been evaluated for a small number of REs. The ability to increase stride has been limited by the constraint that all transitions must be increased in stride; this leads to excessive memory

explosion for strides larger than 2. In [32] we present the technique of variable striding, in which we increase stride selectively on a state by state basis while carefully controlling the increase in required space. Alicherry *et al.* have explored variable striding for TCAM-based string matching solutions [5] but not for DFAs that apply to arbitrary RE sets.

Our techniques in [32] achieve very high transition compression; requiring close to just 1 transition per state. However, that might still not be practical if the number of states grows exponentially with the number of REs. Some work has attempted to address state explosion that occurs due to extensive state replication.

One approach is to simply partition REs into groups building an automata for each group [7,42,51]. With this approach, at run time, each automata must process all packet payloads; that is, similar to an NFA, multiple active states must be maintained. The one advantage this approach has compared to an NFA is that the number of active states at any given time is known in advance, so a system can be designed to accommodate the increased bandwidth requirements for processing packet payloads. This approach is usually used with any of the RE matching techniques when all REs cannot be compiled into a single automata. Our goal is to conquer state explosion so that such partitioning is not needed. If we cannot fully achieve our goal, our work should at least reduce the number of partitions required. In particular, because our techniques achieve greater compression of DFAs than previous software-based techniques, less partitioning of REs will be required.

A second approach is to use "scratch memory" to manage state replication and avoid state

explosion [10, 25, 41]. However, there are several issues with this approach. First, the size of the required scratch memory may itself be significant. Second, the processing required to update the scratch memory after each transition may be significant. Finally, many of these approaches are not fully automated. For example, as Yang *et al.* write in [50] about XFA, "... prior work on improved signature representations has required manual analysis of REs (e.g., to identify and eliminate ambiguity [41]) ...".

Liu *et al.* developed a new method for RE matching that was the first to introduce relative state addressing through the use of offset transitions [28]. In their work, they significantly reduce the number of stored transitions by exploiting state replication and transition sharing without using TCAM. However, they do require the use of bitmaps for each DFA state which means they still require at least one bit per DFA state which means they ultimately do not address the state explosion problem. The current best approach for coping with state explosion is that of Peng *et al.* [38], though they do not offer an automata model. We propose new automata model, ODFA, which facilitates reasoning about state replication and provides a systematic way of handling state replication. Some preliminary results indicate that our technique require significantly fewer TCAM entries than the technique in [38].

Much of the NFA work has exploited the parallel processing capabilities of FPGA technology to cope with the multiple active states that arise from NFA [7, 9, 14, 15, 33, 34, 40, 45]. However, it is not clear that FPGA's can cope with the large number of active states required when processing large signature sets. Furthermore, FPGA's cannot be quickly

reconfigured when the RE sets change and they have relatively slow clock speeds. Also, FPGAs are not commonly embedded in network processors as TCAMs commonly are. One recent work in this direction is that of Yang *et al.* [50] where they use ordered binary decision diagrams to facilitate updating a set of active states in one operation. This is an intriguing idea that merits further study and comparison with DFA compression approaches.

Chapter 3

Background

In this section, we first discuss the background material for the research presented in the later sections.

3.1 DFA for RE Matching

Most RE parsers use some variant of the Deterministic Finite state Automata (DFA) representation of REs. Any set of REs can be converted into an equivalent DFA with the minimum number of states [19, 20]. Traditionally, a DFA is defined as a 5-tuple $D = (Q, \Sigma, q_0, A, \delta)$, where

Q is the set of states,

Σ is the alphabet,

$q_0 \in Q$ is the start state, and

$A \subseteq Q$ is the set of accepting states.

δ: $Q \times \Sigma \rightarrow Q$ is the transition function,

DFAs have the property of needing constant memory access per input symbol, and hence result in predictable and fast bandwidth. The main problem with DFAs is space explosion: a huge amount of memory is needed to store the transition function δ which has $|Q| \times |\Sigma|$ entries. Specifically, the number of states can be very large (state explosion), and the number of transitions per state is large ($|\Sigma| = 256$).

A straightforward approach to implement DFAs is to store the transition function δ in a two dimensional ($|Q|$ by $|\Sigma|$) array. However, $|Q|$ is very large (typically ten thousand or larger) and $|\Sigma| = 2^{8*k}$, where $k \geq 1$, for k-stride DFAs that process k 8 bit characters per transition. Thus, although a $|Q|$ by $|\Sigma|$ array is fast in theory, it is not in reality because it consumes so much memory (hundreds of megabytes) that it has to be stored in DRAM instead of SRAM and DRAM is an order of magnitude slower than SRAM.

In a standard DFA, each state is only marked as either accepting or non-accepting. Given the set of REs $\mathcal{R}$, reaching an accepting state only tells us that some RE in $\mathcal{R}$ matched, but does not tell specifically which RE in $\mathcal{R}$ matched. However, in DPI applications we must keep track of which REs in $\mathcal{R}$ have been matched. For example, each RE may correspond to a unique security threat that requires its own processing routine.

This leads us to define Pattern Matching Deterministic Finite State Automata (PMDFA). The key difference between a PMDFA and a DFA is that for each state q in a PMDFA, we cannot simply mark it as accepting or rejecting; instead, we must record which REs from $\mathcal{R}$ are matched when we reach q.

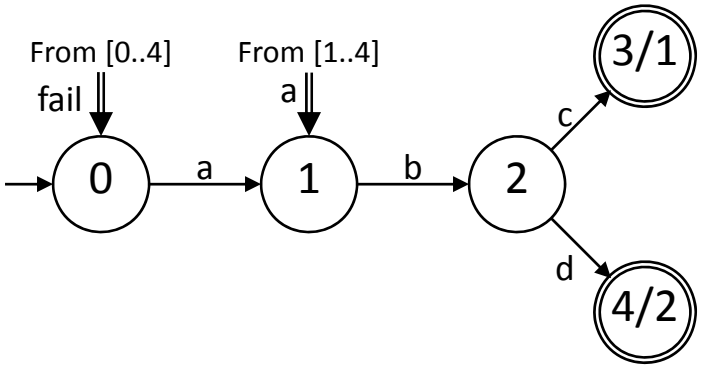
Definition 1 (Pattern Matching DFA (PMDFA)). *Given as input a set of REs $\mathcal{R}$, a PMDFA is a 5-tuple $(Q, \Sigma, q_0, M, \delta)$ where the term M is defined as $M: Q \rightarrow 2^{\mathcal{R}}$. For each state q in the DFA, M gives the set of REs from $\mathcal{R}$ that are matched when we reach q. All the other terms are defined in the same way as in a DFA.*

In a PMDFA, there can be many pairs of states that are equivalent except for the set of REs accepted by the two states. In a DFA, such a pair of states will be merged since they would be completely equivalent. Because of this, the resulting minimum state PMDFA is typically larger than the minimum state DFA. Since we always use a PMDFA, in the rest of the report we just use the term DFA to mean a PMDFA.

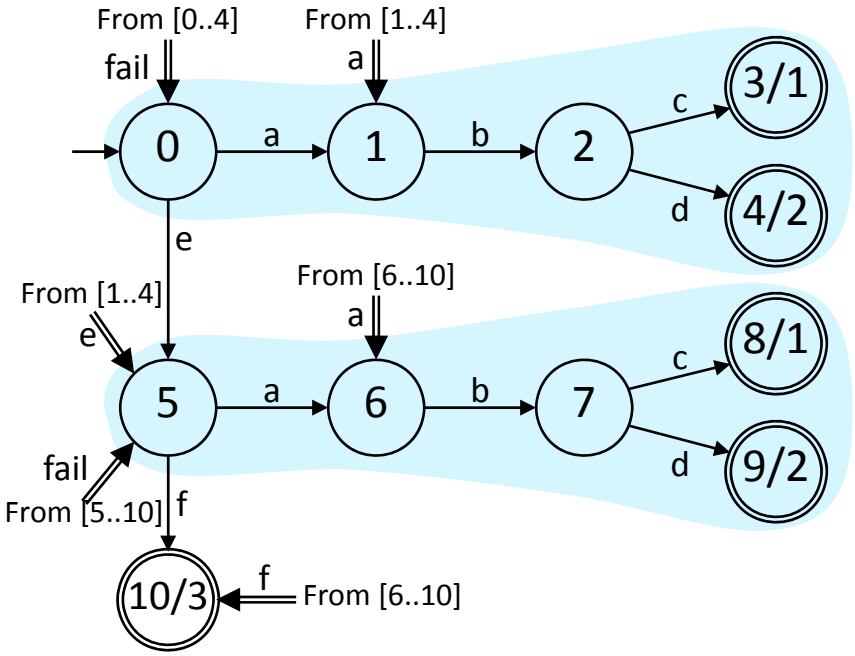
3.2 Understanding DFA space explosion

DFAs suffer from space explosion due to two reasons, which we call *transitions sharing* and *state replication*. We explain these reasons using the DFAs shown in Figure 3.1.

We first define some of our notation for the DFAs in Figure 3.1 for the RE sets {`/abc/`, `/abd/`} and {`/abc/`, `/abd/`, `/e.*f/`}. Note that any RE that is not anchored (*i.e.* does not begin with a '`^`') has an implicit '`.*`' in the beginning, since the RE match can begin anywhere in the input stream. To simplify the diagram, we condense many transitions that have a common destination state on common input characters as follows. These transitions are denoted with double arrows with their character labels next to the double arrow. The source states for these transitions are denoted as "From [x..y]" which represents the set



(a) DFA for RE set {`/abc/`, `/abd/`}.



(b) DFA for RE set {`/abc/`, `/abd/`, `/e.*f/`}.

Figure 3.1: Example of DFA and state replication. (For interpretation of the references to color in this and all other figures, the reader is referred to the electronic version of this dissertation.)

of states with state IDs in the range [x..y]. For example, we represent four transitions starting in states 1 through 4 that end in state 1 on character '`a`' using double arrows beneath "From [1..4]" and an '`a`' next to the double arrow. When the text next to a double arrow is "fail", this represents all character transitions not explicitly shown in the

figure. For example, the "fail" transition in Figure 3.1(a) represents all transitions out of state 0 for characters that are not 'a', all transitions out of state 1 for characters that are not 'b', and so on. Finally, in an accepting state, the number(s) following the '/' represents the ID(s) of the RE matched by that accepting state. We also use the notation $s_1 \xrightarrow{\sigma} s_2$ to denote the transition $\delta(s_1, \sigma) = s_2$.

We define a *self-looping state* as a state which has more than $\Sigma/2$ ($= 128$) of its outgoing transitions going back to itself. Self-looping states are the "failure states" on which the DFA stays when the current input character does not advance the (partial) matching of any of the REs in the RE set. For example in Figure 3.1(b) states 0 and 5 are self-looping states. The transitions in a DFA can be categorized into three types:

1. *Failure transitions* are those that go to the self-looping states. It indicated that the current input character does not advance (or start) the matching of any RE. In Figure 3.1(a), all the incoming transitions of state 0 are failure transitions.

2. *Restartable transitions* are those that go to a state at a lower level than the current state, usually a non self-looping state. It indicates that the current partial matches are lost but there is a new partial match of another (possibly the same) RE. In Figure 3.1(b), the incoming transitions of state 5 on character 'e' from states [1..4] are restartable transitions. For instance the transitions $2 \xrightarrow{e} 5$ means that we had a partial match (ab) of REs/abc/ and /abd/ (since the current state is 2), and the current input 'e' does not advance the match of either of these REs, but it starts the matching of a new RE /e.*f/.

3. *Forward transitions* are the those that go from one state to the next in a chain of states that identify a RE. These transitions advance the current partial match of the RE by one character. In Figure 3.1(b), the outgoing transition of state 0 on characters '`a`' and '`e`' are forward transitions.

3.2.1 Transition Sharing

We say *two transitions are shared* when, out of the three values in a transition (source state, input character, destination state), they differ in only one value. Two shared transitions can only possibly differ in either the input character or the source state (since a DFA has only one transition per source state and input character pair). This gives us two causes of transition sharing: character redundancy and state redundancy.

Character redundancy is when two shared transitions differ in only the input character value. That is, for a state $q \in Q$, we often have $\delta(q, \sigma_1) = \delta(q, \sigma_2)$ for characters σ_1 and σ_2 in Σ. A DFA has a lot of character redundancy since for most states, most of their transitions are failure transitions going to the same self-looping state. Only a few of transitions for most states are either restartable or forward transitions. In addition, if a RE has a chracter range (like '`[a-z]`') in it, then it leads to character redundant forward transitions. For example in Figure 3.1(a), 254 of the 256 transitions for state 1 go to the same state 0.

State redundancy is when two shared transitions differ in only the source state value. That is, for a character $\sigma \in \Sigma$, we have $\delta(p, \sigma) = \delta(q, \sigma)$ for states p and q in Q. The cause for

the large amount of state redundancy is failure and restartable transitions, because both of these types of transitions go to the same next state for many different states in the DFA. For example in Figure 3.1(a), for all the states in the DFA, their failure transitions go to state 0, and their transition on input character '`a`' goes to state 1.

3.2.2 State Replication

When the NFA is converted to an equivalent DFA, the number of states typically increases exponentially. This happens because most of the states in the NFA are replicated many times in the DFA. To understand this, consider the DFAs in Figure 3.1. Figure 3.1(a) shows the DFA for the RE set {`/abc/`, `/abd/`}, and Figure 3.1(b) shows the DFA after the RE `/e.*f/` is added to this RE set. As we can see, the entire DFA in Figure 3.1(a) is repeated twice in the DFA in Figure 3.1(b). Each state is replicated twice because of the wildcard closure '`.*`' in the new RE that is added.

In general when building the DFA for an RE set where some REs contains $\star$'s, the states in the DFAs that corresponds to individual REs are replicated multiple times. And when a state is replicated, we automatically get replication of the transitions of that state, causing transitions replication.

3.3 D^2FA

The Delayed Input DFA (D^2FA) was proposed by Kumar *et al.* [26] to compress the size of the DFA transition function δ by exploiting state redundancy. The basic idea of D^2FA is that in a typical DFA for real world RE set, given two states u and v, $\delta(u, \sigma) = \delta(v, \sigma)$ for many symbols $\sigma \in \Sigma$. We can remove all the transitions for v from δ for which $\delta(u, \sigma) = \delta(v, \sigma)$ and make a note that v's transitions were removed based on u's transitions. When the D^2FA is later processing input and is in state v and encounters input symbol σ, if $\delta(v, \sigma)$ is missing, the D^2FA can use $\delta(u, \sigma)$ to determine the next state. We can do the same thing for most states in the DFA, and it results in tremendous transition compression. Kumar *et al.* observe an average decrease of 97.6% in the amount of memory required to store a D^2FA when compared to its corresponding DFA.

In more detail, to build a D^2FA from a DFA, we just do the following two steps:

1. For each state $u \in Q$, pick a *deferred* state, denoted by $F(u)$. (We can have $F(u) = u$.)

2. For each state $u \in Q$ for which $F(u) \neq u$, remove all the transitions for u for which $\delta(u, \sigma) = \delta(F(u), \sigma)$.

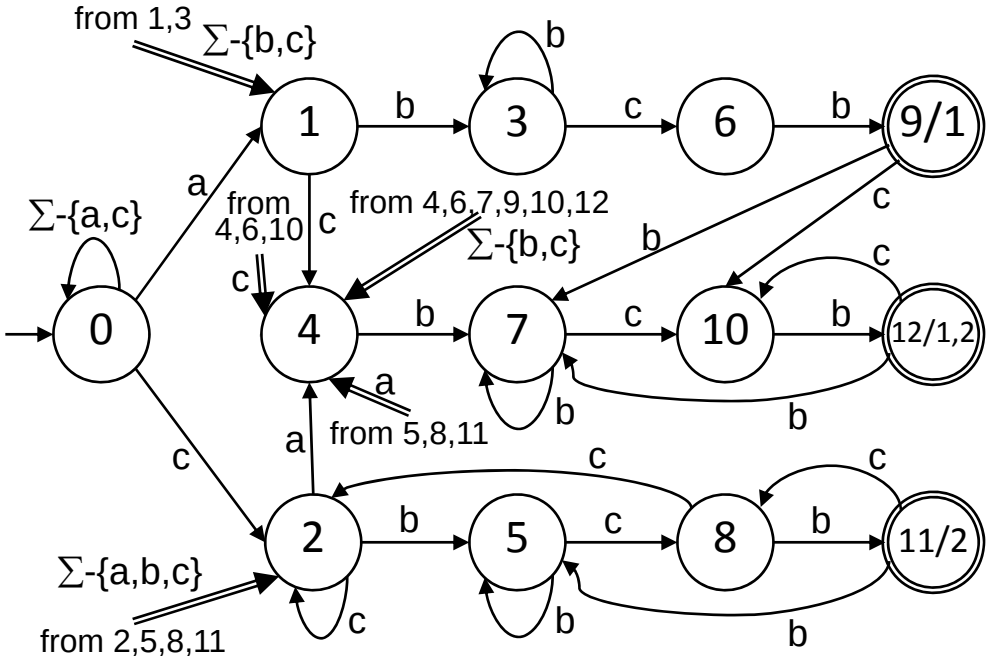
When traversing the D^2FA, if on current state u and current input symbol σ, if $\delta(u, \sigma)$ is missing (*i.e.* has been removed), we can use $\delta(F(u), \sigma)$ to get the next state. Of course, $\delta(F(u), \sigma)$ might be missing too, in which case we then use $\delta(F(F(u)), \sigma)$ to get the next state, and so on.

Figure 3.2(a) shows a DFA for the REs set {/.⋆a.⋆bcb/, /.⋆c.⋆bcb/}, and Figure 3.2(c) shows the D^{2}FA built from the DFA. The dashed lines represent deferred states. The DFA has $13 \times 256 = 3328$ transitions, whereas the D^{2}FA only has 1030 actual transitions and 9 deferred transitions.

3.3.1 D^{2}FA Definition

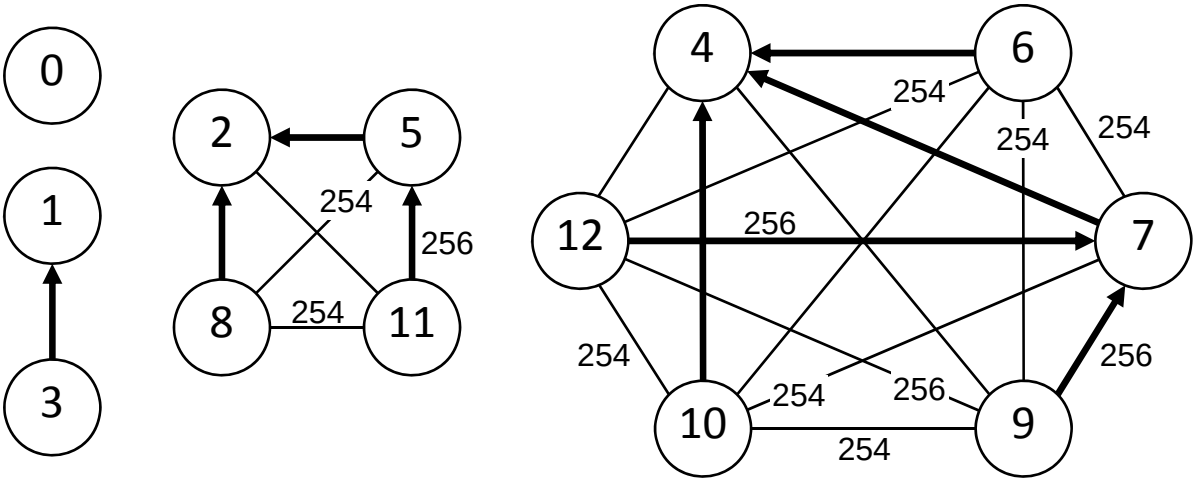
We formally define a D^{2}FA and introduce some notation here.

Definition 2 (D^{2}FA). *Let* $D = (Q, \Sigma, q_0, M, \delta)$ *be a DFA. A corresponding D^{2}FA* D' *is defined as a 6-tuple* $D' = (Q, \Sigma, q_0, M, \rho, F)$. *The first four terms here are defined the same way as in the DFA. The function* $F: Q \to Q$ *defines a unique deferred state*

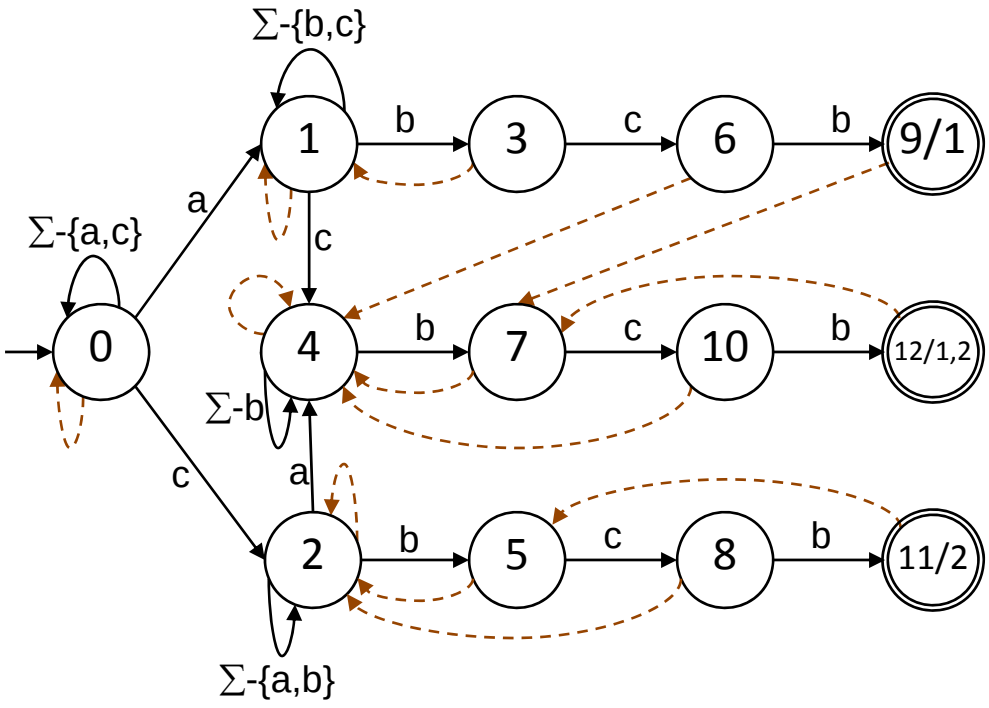


(a) DFA for RE set {/.⋆a.⋆bcb/, /.⋆c.⋆bcb/}

Figure 3.2: D^{2}FA example.



(b) SRG for the DFA. Edges with weight ≤ 1 are not shown. Unlabeled edges have weight 255



(c) The corresponding D^{2}FA. Dashed edges represent deferment.

Figure 3.2: D^{2}FA example (cont'd).

for each state in Q*, and the partial function* $\rho\colon Q \times \Sigma \to Q$ *is a partially defined transition function. Together, the deferment function* F *and the partial transition function* ρ *are equivalent to DFA transition function* δ*. We use* $\mathrm{dom}(\rho)$ *to denote the domain of* ρ*,* i.e. *the values for which* ρ *is defined. The key property of the* D^2FA D' *that corresponds to DFA* D *is as follows:*

$$\forall \langle s, \sigma \rangle \in Q \times \Sigma, \langle s, \sigma \rangle \in \mathrm{dom}(\rho) \iff (F(s) = s \vee \delta(s, \sigma) \neq \delta(F(s), \sigma))$$

That is, for each state, ρ *only has those transitions that are different from that of its deferred state in the underlying DFA. When defined,* $\rho(s, \sigma) = \delta(s, \sigma)$*.*

The function F *defines a directed graph on the states of* Q*, which we call the* deferment forest*. A* D^2FA *is well defined if and only if there are no cycles of length* > 1 *in the deferment forest (*i.e. *there are no cycles except self-loops.)*

The total transition function for the D^2FA *(derived from* ρ*) is defined as*

$$\delta'(s, \sigma) = \begin{cases} \rho(s, \sigma) & \text{if } \langle s, \sigma \rangle \in \mathrm{dom}(\rho) \\ \delta'(F(s), \sigma) & \text{else} \end{cases}$$

It is easy to see that δ' *is well defined and equal to* δ *if the* D^2FA *is well defined.*

We need the restriction that the deferment forest cannot have a cycle other than a self-loop on the states because otherwise all states on the cycle might have their transitions on some $\sigma \in \Sigma$ removed, and there would no way of finding the next state.

We also use the term *deferment pointer* to refer to the deferred state of a state. That is, if $F(u) = v \wedge u \neq v$, we say the deferment pointer of state u is set to state v. If $F(u) = u$, we say the deferment pointer for state u is not set.

States that defer to themselves (*i.e.* deferment pointer is not set), which we call *root states*, must have all their transitions defined. Each connected component of the deferment forest is called a *deferment tree*. It is easy to see that each deferment tree has exactly one root state in it, and the deferment pointer of all the other states in the deferment tree are set towards the root state.

We use $u \rightarrow v$ to denote $F(u) = v$, *i.e.* u directly defers to v. In this case, we say state u is a *child* of state v, and state v is the *parent* of state u, in the deferment forest. We use $u \twoheadrightarrow v$ to denote that there is a path from u to v in the deferment forest defined by F. In this case we say state u is a *descendant* of state v, and state v is the *ancestor* of state u, in the deferment forest.

The *deferment depth of state* u, denoted $\psi(u)$, is the distance, in the deferment tree containing u, of state u from the root state of that deferment tree. The (maximum) *deferment depth of* D^2FA D', denoted $\Psi(D')$, is the maximum deferment depth among all the states in D'. We use $\overline{\psi}(D')$ to denote the average deferment depth among all the states in D'.

We use $u \sqcap v$ to denote the number of transitions in common between states u and v; *i.e.* $u \sqcap v = |\{\sigma \mid \sigma \in \Sigma \wedge \delta(u, \sigma) = \delta(v, \sigma)\}|$.

We only consider D^2FA that correspond to minimum state DFA, though the definition applies to all DFA.

3.3.2 Original D^2FA Algorithm

In this section we explain the original D^2FA construction algorithm proposed by Kumar *et al.* [26]. They first build a DFA for the given RE set.

The amount of transition compression achieved by the D^2FA depends on the number of common transition between each (non-root) state and its deferred state. So next, in order to maximize transition compression, they essentially solve a maximum weight spanning tree problem on the following weighted graph which they call a *Space Reduction Graph (SRG)*. The SRG is a complete graph with the DFA states, Q, as its vertices. The weight of any edge (u, v) in the SRG is equal to the number of common transitions between DFA states u and v. They use the the Kruskal's algorithm [23] to construct the maximum weight spanning tree. Edges with weight ≤ 1 are not considered (selecting an edge with weight 1 does not reduce the transition function, since it will result in removal of one actual transition and addition of the deferment pointer transition.) For this reason the maximum weight spanning tree construction might result in a forest.

Once the spanning forest is constructed, (one of) the state(s) in the center of each tree is selected as the root for that tree, and all edges are directed towards the root. These directed edges give the deferred state for each state.

Figure 3.2(b) shows the SRG built for the DFA in Figure 3.2(a), along with the maximum weight spanning forest with roots selected and the edges directed.

3.3.3 Limiting Deferment Depth in Original D^{2}FA Algorithm

A D^{2}FA has the drawback that while parsing the input string, the current input character is not advanced when a deferment transition is followed (hence the name *delayed input* DFA.) In the worst case for a given state u and current input character σ, we might have to do $\psi(u) + 1$ lookups to find the next state $\delta'(u, \sigma)$; that is $\psi(u)$ lookups to get to the root state following deferment transitions and 1 more lookup to get the actual next state.

This is a problem since we no longer get deterministic throughput, which was the main reason for using the DFA. So, in general, it is better to have low deferment depth for all states. If we set an upper bound on Ψ, then we achieve deterministic throughput, since we would have a constant bound on the number of lookups per input character.

Recall that during the maximum weight spanning tree construction, Kruskal's algorithm considers edges in decreasing edge weight order. At any time during the construction, many edges will have the current largest edge weight (since there are only 257 possible edge weights.) In order to reduce the deferment depth of the resulting D^{2}FA, Kumar *et al.* propose the following tie breaking heuristic: among all edges with the current maximum weight, pick the one that will result in the least increase in the diameter when added to the spanning forest.

Also, given an upper bound, Ω, on the D^2FA deferment depth Ψ, Kumar *et al.* propose the following method to generate D^2FA with deferment depth within the bound: during the maximum weighted spanning tree construction, an edge is only added to the spanning tree if it does not cause the tree diameter to go over $2 \times \Omega$. Since the tree center is chosen as the root state, this guarantees that $\Psi(D') \leq \Omega$.

3.3.4 Backpointer D^2FA Algorithm

The *level* of a state u in a DFA is the length of the shortest string that takes the DFA from the start state to state u. Becchi and Crowley [8] propose an algorithm to build the D^2FA based on the following idea: each state in the DFA should defer to a state that is at a lower level than itself. Because of this, every deferred transition followed will decrease the level of the current state by at least 1. Any actual transition taken can only increase the level of the current state by 1. Therefore, when processing any input string of length n, at most $n-1$ deferred transitions will be followed. So this method guarantees an amortized cost of at most 2 lookups per input character.

To build the D^2FA, they build the DFA for the given RE set first. Next, for each state u, among all the states at a lower level than u, they set $F(u)$ to be the state which shares the most transitions with u. Since each state defers to a state at a lower level than itself, the deferment forest can never have a cycle, so the D^2FA is well defined.

The resulting D^2FA is typically a bit larger in size than the D^2FA built using the algorithm proposed by Kumar *et al.*.

3.4 Classifiers

In this section we define a classifier, related terminology and describe a classifier minimization problem. A classifier is essentially a mapping function from the source domain to the target domain. In a d-dimensional classifier, the input value is composed of d fields.

A classifier is traditionally defined for the (header based) packet classification problem. The input value is the packet header, which has five fields: Protocol type, Source IP address, Source port number, Destination IP address and Destination port number. The output is the decision or action to be taken for the packet, which typically has values like accept, discard, accept and log, discard and log *etc.*. So the classifier is defined as a 5-dimensional classifier, with the set of possible packet headers as the source domain, and set of possible actions as the target domain. For each possible packet header, the classifier gives the action to be taken.

3.4.1 Classifier definition

We now formally define a d-*dimensional classifier* and related terminology.

A *field* F_i is a finite width variable. The domain of field F_i of w bit width is $dom(F_i) = [0..2^w - 1]$. The domain of a d-dimensional classifier, f, defined over the d fields $F_1, \ldots, F_d$ is $dom(f) = dom(F_1) \times \cdots \times dom(F_d)$. A *packet* is a d-tuple $(p_1, \ldots, p_d)$, where, for $1 \leq i \leq d$, $p_i \in dom(F_i)$.

A *rule* has the form $\langle predicate \rangle \rightarrow \langle decision \rangle$. A rule $\langle predicate \rangle$ is a d-tuple $(S_1, \ldots, S_d)$,

where, for $1 \leq i \leq d$, $S_i \subseteq \mathrm{dom}(F_i)$; and it *covers* the set of packets $S_1 \times \cdots \times S_d \subseteq \mathrm{dom}(f)$. A packet p matches rule r if and only if the predicate of r covers p. The set of possible rule decisions is denoted by H.

The classifier $f = \langle r_1, \ldots, r_n \rangle$ is specified as a sequence of rules. For packet p, the first rule in the sequence that p matches is said to be the *binding rule* for p. If p does not match any rule in f, then p does not have any binding rule (or is unbound). For a bound packet p, the output of the classifier, $f(p)$, is given by the decision of the binding rule for p. For unbound packets, p, $f(p)$ is undefined. The cost of a classifier f, denoted $\mathrm{Cost}(f)$, is the number of rules in f.

The *Cover* of a classifier f, denoted $\mathrm{Cover}(f)$, is defined as the set of packets in $\mathrm{dom}(f)$ that have a binding rule in f (*i.e.* set of packets that match at least one rule in f.) A classifier f, is said to be a *complete classifier* if $\mathrm{Cover}(f) = \mathrm{dom}(f)$, otherwise f is said to be an *incomplete classifier*.

Clearly, two rules in a classifier can be overlapping (*i.e.* at least one packet matches both rules), as well as conflicting (*i.e.* overlapping and having different decisions). But that is ok, since the classifier output for a bound packet is uniquely defined by its binding rule.

3.4.1.1 Prefix Classifier

A *prefix* $\{0,1\}^k\{*\}^{w-k}$ with k leading bits (*i.e.* 0s or 1s), for a field of width w, denotes the range of values $[\{0,1\}^k\{0\}^{w-k}, \{0,1\}^k\{1\}^{w-k}]$. A rule is said to be a *prefix rule* if and

only if every S_i in the rule predicate $(S_1, \ldots, S_d)$ is represented as a prefix. A classifier f is said to be a *prefix classifier* if and only if every rule in f is a prefix rule.

3.4.1.2 Ternary Classifier

A *ternary value* for a field of width w is of the form $\{0, 1, *\}^w$, and denotes the set of values obtained by replacing the $*$'s with 0's and 1's in all possible combinations (if there are k $*$'s, there are 2^k ways to replace the $*$'s with 0's and 1's.) A rule is said to be a *ternary rule* if and only if every S_i in the rule predicate $(S_1, \ldots, S_d)$ is represented as a ternary value. A classifier f is said to be a *ternary classifier* if and only if every rule in f is a ternary rule.

A prefix classifier is a special case of a ternary classifier, since every prefix is also a ternary value.

3.4.1.3 Weighted Classifier

In a weighted classifier, each decision in H has a weight associated with it. The cost of a classifier f is then equal to the sum of the weights of decisions of all the rules in f. The unweighted classifier is a special case of weighted classifier with weights of all the decisions set to 1.

3.4.2 Classifier Minimization

Two classifiers f_1 and f_2 are *equivalent*, denoted $f_1 \equiv f_2$, if and only if $Cover(f_1) = Cover(f_2)$ and $\forall p \in Cover(f_1), f_1(p) = f_2(p)$. For a classifier f, we use $\{f\}$ to denote the set of all classifiers that are equivalent to f.

The classifier minimization problem is then defined as follows.

Definition 3 (Classifier Minimization Problem). *Given a classifier* f_1, *find a prefix classifier* $f_2 \in \{f_1\}$ *such that for any prefix classifier* $f \in \{f_1\}$, $Cost(f_2) \leq Cost(f)$.

Multi-dimensional classifier minimization has been shown to be NP-hard. An optimal solution for 1-dimensional complete classifier minimization was proposed by Suri *et al.* [47]. Meiners *et al.* [30, 31] proposed algorithms for 1-dimensional complete weighted classifier minimization and 1-dimensional incomplete weighted classifier minimization.

3.5 TCAM Introduction

In any regular memory, the input is the memory address location, and the output is the contents of the memory at that location. In a Ternary Content Addressable Memory (TCAM), as the name suggests, it is the exact opposite. The input to a TCAM is binary value, and the output of the TCAM is the address of the location, if any, at which the given value occurs. The ternary refers to the fact that the contents of the memory are ternary bits, *i.e.* 0, 1 or $*$ (don't care). The $*$ matches both a 0 and a 1.

If more than one location matches the given (binary) value, then the address of the first location that matches the value is returned. We call this the *first match semantics* of TCAM.

The key thing about TCAMs is that the output is returned in constant time. TCAMs internally have a massively parallel hardware that searches the given input against all the entries stored in the TCAM at once, and returns the address of the first match. For this reason, TCAM memory chips have very limited size. The largest available chip is about 72Mb, and typical sizes are around 1Mb to 8Mb. TCAM chips also consume a lot of energy compared to regular memory.

The TCAM chip is usually paired with a corresponding SRAM that stores output values. The matching address from the TCAM is used as input to the SRAM to get the output value.

TCAM chips are widely used in networking devices for packet classification. A ternary classifier for packet classification can be naturally implemented in a TCAM. All the rules predicates are stored, in order, in the TCAM, and the corresponding rule decisions are stored in the SRAM. The packet header is then used as a lookup key for the TCAM, and the matching SRAM values gives the decision for the packet.

Chapter 4

Software Implementation

In this section we present our work on the software implementation of RE matching. A software solution typically uses a DFA to achieve deterministic throughput. The software solution can be implemented on general purpose processors, or on customized ASIC chips.

4.1 Introduction/Motivation

The straightforward way to implement a DFA in software is to store the DFA transition table, δ, in a two dimensional $Q \times \Sigma$ array. But DFAs suffer from space explosion when multiple REs are combined, making them impractical even for moderately sized RE set. D^2FA are very effective at dealing with the space explosion problem of the DFA. In particular, D^2FA exhibit tremendous transition compression reducing the size of the DFA by a huge factor. This makes D^2FA much more practical for a software implementation of RE matching than DFAs. In our work we focus on the D^2FA.

4.1.1 Solution Goals

For software implementation of RE matching, given as input a set of REs $\mathcal{R}$, we need to be able to build a compact D^{2}FA as efficiently as possible that also supports frequent updates. Efficiency is important because RE matching solutions are typically implemented in networking devices, which usually have very limited computing resources. Current methods for constructing D^{2}FA may be so expensive in both time and space that they may not be able to construct the final D^{2}FA even if the D^{2}FA is small enough to be deployed in networking devices that have limited computing resources. Such issues become doubly important when we consider the issue of the frequent updates (typically additions) to $\mathcal{R}$ that occur as new security threats are identified.

4.1.2 Summary and Limitations of Prior Art

Given the input RE set $\mathcal{R}$, any solution that builds a D^{2}FA for $\mathcal{R}$ will have to do the following two operations: (a) union the automata corresponding to each RE in $\mathcal{R}$ and (b) minimize the automata, both in terms of the number of states and the number of edges. Previous solutions [8,26] (discussed in Section 3.3) employ a "Union then Minimize" framework in which: (1) they first build automata for each RE within $\mathcal{R}$, and perform union operations on these automata to arrive at one combined automaton for all the REs in $\mathcal{R}$, and (2) next they minimize the resulting combined automaton. In particular, previous solutions first construct the combined NFA for the RE set. Then they perform a computationally expensive NFA to DFA subset construction on the large combined NFA,

followed by or composed with DFA minimization (for states). And last they perform the D^2FA minimization (for edges).

There are three fundamental limitations with prior solutions, due to which they do not meet our goals. First, they perform the minimization on the large combined automata which is expensive in both time and space. Second, prior methods build the corresponding minimum state DFA before constructing the final D^2FA. This is very costly in both space and time. The D^2FA is typically 50 to 100 times smaller than the DFA, so even if the D^2FA would fit in available memory, the intermediate DFA might be too large, making it impractical to build the D^2FA. This is exacerbated in the case of the Kumar *et al.* algorithm which needs the SRG which ranges from about the size of the DFA itself to over 50 times the size of the DFA. The resulting space and time required to build the DFA and SRG impose serious limits on the D^2FA that can be practically constructed. We do observe that the method proposed in [8] does not need to create the SRG. Furthermore, as the authors have noted, there is a way to go from the NFA directly to the D^2FA, but implementing such an approach is still very costly in time as many transition tables need to be repeatedly recreated in order to realize these space savings. In addition, this direct NFA to D^2FA construction would still need to perform the expensive subset construction on the large combined NFA. Third, none of the previous methods support updating the D^2FA when a new RE is added to $\mathcal{R}$. The whole D^2FA would have to be rebuilt when the RE set is updated.

4.1.3 Summary of Our Approach

To address the limitations of prior solutions, we propose a "Minimize then Union framework". Specifically, we first minimize the small automata corresponding to each RE from $\mathcal{R}$, and then union the minimized automata together. In particular, given $\mathcal{R}$, we first build a DFA and D^{2}FA for each individual RE in $\mathcal{R}$. The heart of our technique is the D^{2}FA merge algorithm that performs the union. It merges two smaller D^{2}FAs into one larger D^{2}FA such that the merged D^{2}FA is equivalent to the union of REs that the D^{2}FAs being merged were equivalent to. Starting from the the initial D^{2}FAs for each RE, using this D^{2}FA merge subroutine, we merge two D^{2}FAs at a time until we are left with just one final D^{2}FA. The initial D^{2}FAs are each equivalent to their respective REs, so the final D^{2}FA will be equivalent to the union of all the REs in $\mathcal{R}$. A key property of our D^{2}FA merge algorithm is that it automatically produces a minimum state D^{2}FA without explicit state minimization. Likewise, it creates efficient state deferment in the merged D^{2}FA using state deferment information from the input D^{2}FAs. Together, these optimizations lead to a vastly more efficient D^{2}FA construction algorithm in both time and space.

The D^{2}FA produced by our merge algorithm can be larger than the minimal D^{2}FA produced by the Kumar *et al.* algorithm. This is because the Kumar *et al.* algorithm does a global optimization over the whole DFA (using the SRG), whereas our merge algorithm efficiently computes state deferment in the merged D^{2}FA based on state deferment in the two input D^{2}FAs. In most cases, the D^{2}FA produced by our approach is sufficiently small to be deployed. However, in situations where more compression is needed, we offer an

efficient final compression algorithm that produces a D^2FA very similar in size to that produced by the Kumar *et al.* algorithm. This final compression algorithm uses an SRG; we improve efficiency by using the deferment already computed in the merged D^2FA to greatly reduce the size of this SRG and thus significantly reduce the time and memory required to do this compression.

4.1.3.1 Advantages of our algorithm

One of the main advantages of our algorithm is a dramatic increase in time and space efficiency. These efficiency gains are partly due to our use of the Minimize then Union framework instead of the Union then Minimize framework. More specifically, our improved efficiency comes about from the following four factors. First, other than for the initial DFAs that correspond to individual REs in $\mathcal{R}$, we build D^2FA bypassing DFAs. Those initial DFAs are very small (typically < 50 states), so the memory and time required to build the initial DFAs and D^2FAs is negligible. The D^2FA merge algorithm directly merges the two input D^2FAs to get the output D^2FA without creating the DFA first. Second, other than for the initial DFAs, we never have to perform the NFA to DFA subset construction. Third, other than for the initial DFAs, we never have to perform DFA state minimization. Fourth, when setting deferred states in the D^2FA merge algorithm, we use deferment information from the two input D^2FA. This typically involves performing only a constant number of comparisons per state rather than a linear in the number of states comparison per state as is required by previous techniques. All told, our algorithm has a practical time

complexity of $O(n|\Sigma|)$ where n is the number of states in the final D^2FA and $|\Sigma|$ is the size of the input alphabet. In contrast, Kumar *et al.*'s algorithm [26] has a time complexity of $O(n^2(\log(n) + |\Sigma|))$ and Becchi and Crowley's algorithm [8] has a time complexity of $O(n^2|\Sigma|)$ just for setting the deferment state for each state and ignoring the cost of the NFA subset construction and DFA state minimization. Section 4.4.4 has a more detailed complexity analysis.

Because of these efficiency advantages in time and space complexity, given the same limited resources, our algorithm can build much larger D^2FAs than are possible with previous methods. Besides being much more efficient in constructing D^2FA from scratch, our algorithm is very well suited for frequent RE updates. When an RE needs to be added to the current set, we just need to merge the D^2FA for the RE to the current D^2FA using our merge routine which is a very fast operation.

4.2 Minimum State PMDFA construction

Before we present our algorithm for efficient D^2FA construction, we consider the problem of constructing minimum state DFA for a given RE set.

Given a set of REs $\mathcal{R}$, we can build the corresponding minimum state DFA using the standard Union then Minimize framework: first build a combined NFA for all the REs in $\mathcal{R}$, then convert the NFA to a DFA, and finally minimize the DFA. This method can be very slow, mainly due to subset construction in the NFA to DFA conversion, which often

results in an exponential growth in the number of states. Instead, we propose a more efficient Minimize then Union framework.

Let R_1 and R_2 denote any two disjoint subsets of $\mathcal{R}$, and let D_1 and D_2 be their corresponding minimum state DFAs. We use the standard *union cross product* construction for DFAs to construct a minimum state DFA D_3 that corresponds to $R_3 = R_1 \cup R_2$. Specifically, suppose we are given the two DFAs $D_1 = (Q_1, \Sigma, q_{01}, M_1, \delta_1)$ and $D_2 = (Q_2, \Sigma, q_{02}, M_2, \delta_2)$. The union cross product DFA of D_1 and D_2, denoted as $\mathrm{UCP}(D_1, D_2)$, is given by

$$D_3 = \mathrm{UCP}(D_1, D_2) = (Q_3, \Sigma, q_{03}, M_3, \delta_3)$$

where

$$Q_3 = Q_1 \times Q_2$$

$$q_{03} = \langle q_{01}, q_{02} \rangle$$

$$\forall q_i \in Q_1, \forall q_j \in Q_2,\ M_3(\langle q_i, q_j \rangle) = M_1(q_i) \cup M_2(q_j)$$

$$\forall \sigma \in \Sigma, \forall q_i \in Q_1, \forall q_j \in Q_2,\ \delta_3(\langle q_i, q_j \rangle, \sigma) = \langle \delta_1(q_i, \sigma), \delta_2(q_j, \sigma) \rangle$$

Each state in D_3 corresponds to a pair of states, one from D_1 and one from D_2. For notational clarity, we use $\langle$ and $\rangle$ to enclose an ordered pair of states. Transition function δ_3 just simulates both δ_1 and δ_2 in parallel. Many states in Q_3 might not be reachable from the start state q_{03}. Thus, while constructing D_3, we only create states that are reachable from the start state q_{03}.

We now argue that this construction is correct. This is a standard construction, so the fact that D_3 is a DFA for $R_3 = R_1 \cup R_2$ is straightforward and covered in standard automata theory textbooks (*e.g.* [20]). We now show that D_3 is also a minimum state DFA for R_3 assuming $R_1 \cap R_2 = \emptyset$. Recall that we are using DFA to mean a PMDFA (see Section 3.1.) For a traditionally defined DFAs, the UCP construction is not guaranteed to produce a minimum state DFA.

Theorem 1. *Given two RE sets,* R_1 *and* R_2*, and equivalent minimum state DFAs,* D_1 *and* D_2*, the union cross product DFA* $D_3 = UCP(D_1, D_2)$*, with only reachable states constructed, is the minimum state DFA equivalent to* $R_3 = R_1 \cup R_2$ *if* $R_1 \cap R_2 = \emptyset$*.*

Proof. First since only reachable states are constructed, D_3 cannot be trivially reduced. Now assume D_3 is not minimum. That would mean there are two different states in D_3, say $\langle p_1, p_2 \rangle$ and $\langle q_1, q_2 \rangle$, that are indistinguishable. This implies that

$$\forall x \in \Sigma^\star,\ M_3(\delta_3(\langle p_1, p_2 \rangle, x)) = M_3(\delta_3(\langle q_1, q_2 \rangle, x)).$$

Working on both sides of this equality, we get,

$$\begin{aligned} \forall x \in \Sigma^\star,\ M_3(\delta_3(\langle p_1, p_2 \rangle, x)) &= M_3(\langle \delta_1(p_1, x), \delta_2(p_2, x) \rangle) \\ &= M_1(\delta_1(p_1, x)) \cup M_2(\delta_2(p_2, x)) \end{aligned}$$

as well as,

$$\forall x \in \Sigma^\star,\ M_3(\delta_3(\langle q_1, q_2\rangle, x)) = M_3(\langle \delta_1(q_1, x), \delta_2(q_2, x)\rangle)$$
$$= M_1(\delta_1(q_1, x)) \cup M_2(\delta_2(q_2, x))$$

This implies that

$$\forall x \in \Sigma^\star,\ M_1(\delta_1(p_1, x)) \cup M_2(\delta_2(p_2, x)) = M_1(\delta_1(q_1, x)) \cup M_2(\delta_2(q_2, x))$$

Now since $R_1 \cap R_2 = \emptyset$, this gives us

$$\forall x \in \Sigma^\star,\ M_1(\delta_1(p_1, x)) = M_1(\delta_1(q_1, x)) \quad \text{and}$$
$$\forall x \in \Sigma^\star,\ M_2(\delta_1(p_2, x)) = M_2(\delta_1(q_2, x))$$

This implies that p_1 and q_1 are indistinguishable in D_1 and p_2 and q_2 are indistinguishable in D_2. Since $\langle p_1, p_2\rangle \neq \langle q_1, q_2\rangle$, we have that $p_1 \neq p_2 \vee q_1 \neq q_2$, implying that at least one of D_1 or D_2 is not a minimum state DFA, which is a contradiction and the result follows. □

Our efficient construction algorithm works as follows. First, for each RE $r \in \mathcal{R}$, we build an equivalent minimum state DFA D for r using the standard method, resulting in a set of DFAs $\mathcal{D}$. Then we merge two DFAs from $\mathcal{D}$ at a time using the above UCP construction until there is just one DFA left in $\mathcal{D}$. The merging in done in a greedy manner: in each step,

the two DFAs with the fewest states are merged together. Note the condition $R_1 \cap R_2 = \emptyset$ is always satisfied in all the merges, so Theorem 1 ensures that we always have a minimized DFA.

In our experiments, our Minimize then Union technique runs exponentially faster than the standard Union then Minimize technique because we only apply the NFA to DFA subset construction step on the NFAs that correspond to each individual RE rather than on the combined NFA for all the REs. This makes a significant difference even when we have a relatively small number of REs. For example, for the C7 RE set which contains 7 REs, the standard technique requires 385.5 seconds to build the DFA, but our technique builds the DFA in only 0.66 seconds.

4.3 Efficient D^{2}FA Construction

In this section, we describe how we can extend the Minimize then Union technique to directly build the D^{2}FA bypassing the DFA construction. We first build the D^{2}FA for each individual RE in the RE set, and then merge these D^{2}FAs together to get the combined D^{2}FA for the entire RE set.

4.3.1 Improved D^{2}FA Construction for One RE

To build the initial D^{2}FA for each RE in $\mathcal{R}$, we can use the original D^{2}FA algorithm proposed in [26]. However, we propose several improvements to original algorithm that

facilitate our D^{2}FA merge algorithm, our techniques for hardware implementation of RE matching presented in Chapter 5 and the overlay automata approach presented in Chapter 6.

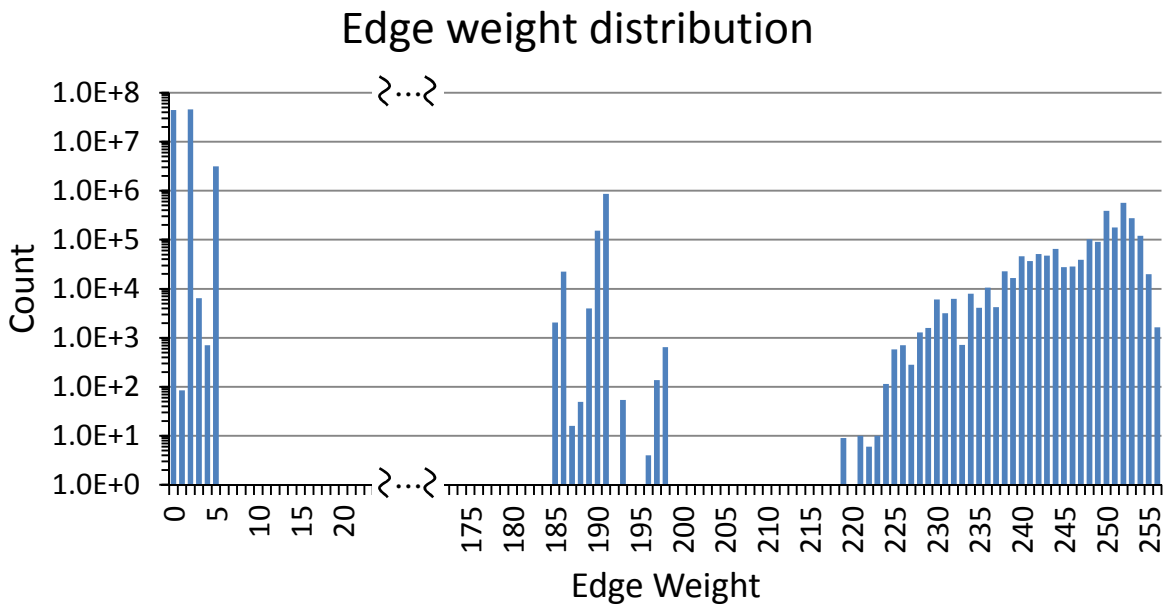


Figure 4.1: Edge weights distribution in a typical SRG.

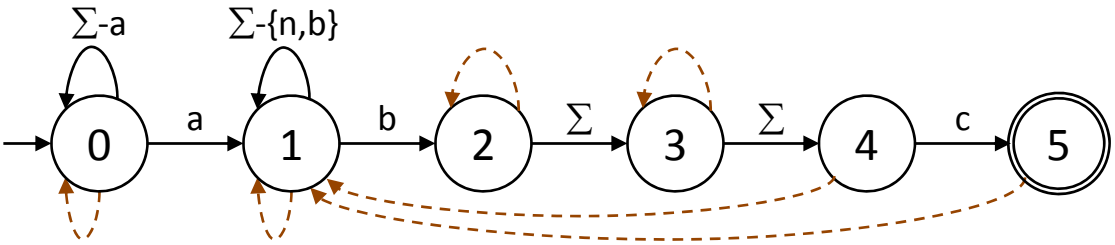
Figure 4.1 shows the typical distribution of the weights of the edges in the SRG. The distribution is typically bimodal. The weights of the edges are very high (> 128) or very low (< 20). The reason behind this is that, for all state pairs for which both states have their failure transitions going to the same self-looping state, the two states will have most of their transitions in common, and hence result in a very high weight edge in the SRG. Likewise, for all state pairs for which both states have their failure transitions going to different self-looping states, the two states will have none (or very few) of their transitions in common, and hence result in a very low weight edge in the SRG. If we remove the low weight edges from the SRG, we get a natural partitioning of the states based on the

self-looping state they fail to. Let us call this partitioning of states $\mathcal{P}$. Each partition in $\mathcal{P}$ will have at most one self-looping state.

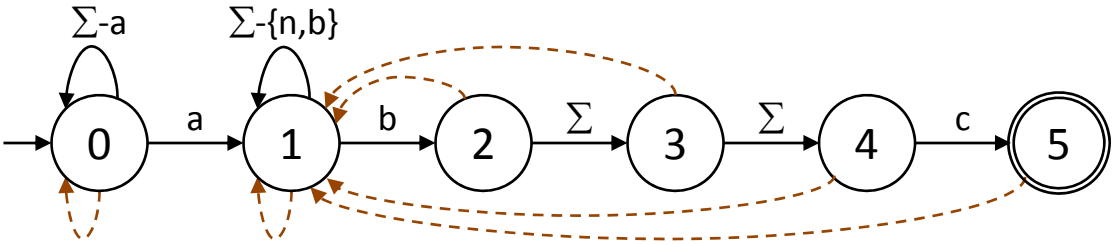
Multiple deferment trees: We remove the low weight (< 20) edges from the SRG before building the maximum spanning tree. The result of this is that the deferment forest has multiple deferment trees, one tree for each partition in $\mathcal{P}$. This only results in a small increase in the number of transitions in the resulting D^{2}FA, since edges removed from the SRG have very low weight. For each partition in $\mathcal{P}$, the unique self-looping state (if any) within the partition is chosen as the root of the corresponding deferment tree.

Handling non-self-looping roots: We can have a partition in $\mathcal{P}$ which does not have any self-looping state. In such cases we will have a non self-looping state selected for the partition. This will happen for REs that have a '`.`' (or a large range like `[^a]`) without the closure '$\star$'. For example consider that D^{2}FA shown in Figure 4.2(a) for the RE `/a.*b..c/`. The deferment forest will have 4 root states, 0, 1, 2 and 3. States 0 and 1 are self-looping. However, states 2 and 3 are not self-looping and are only roots states because they have no transition in common with other states.

In such cases, we make these states non root states and set their deferment as follows. We look at the deferment of the next state where the transition on the '`.`' goes to. If we have more than one consecutive '`.`', we note the state where the last '`.`' transitions to. In our example, the next state of the last '`.`' is state 4. We follow the deferment of this state until we reach its root, and select that root as the deferred state of the non self-looping



(a) D^{2}FA for RE `/a.*b..c/` with non self-looping roots



(b) D^{2}FA after setting deferment for non self-looping roots

Figure 4.2: Example showing D^{2}FA with non self-looping root states.

roots. In our example, the deferment chain of state 4 ends in state 1, so state 1 is chosen as the deferred state for both states 2 and 3. Figure 4.2(b) shows the resulting D^{2}FA.

Setting the deferment of non self-looping roots in this manner does not reduce the size of the D^{2}FA since these states will not have any transitions (or very few transitions) in common with their deferred states. However, this results in a better structure of the deferment forest. It also ensures we have the condition that all roots states are self-looping states and vice versa.

Improved edge weight tie breaking: Recall that during the construction of the maximum spanning tree using Kruskal's algorithm, at any time there are usually many edges with the current maximum weight. We use the following tie breaking strategy.

For each state u, we store a value, $deg'(u)$, which is initially set to 0. During Kruskal's

algorithm, when an edge $e = (u, v)$ is added to the current spanning tree, $deg'(u)$ is incremented by 2 if $level(u) \leq level(v)$; otherwise it is incremented by 1. Recall that $level(u)$ is the length of the shortest string that takes the DFA from the start state to state u. We similarly update $deg'(v)$. Then we use the following tie breaking order among edges having the current maximum weight.

1. Edges that have a self-looping state as one of their end points are given the highest priority.

2. Next, priority is given to edges with higher sum of deg' of their end vertices.

3. Next, priority is given to edges with higher difference between the levels of their end vertices.

The sum of degrees of end vertices is used for tie breaking in order to prioritize states that are already highly connected. However, we also want to prioritize connecting to states at lower levels, so we use deg' instead of just the degree. Using the difference between levels of end points for tie breaking also prioritizes states at a lower level. This helps reduce the deferment depth and the D^2FA size for RE sets whose D^2FAs have a higher average deferment depth.

There are several benefits of these improvements.

1. Having the self-looping states in the center helps to minimize the average height of the deferment tree. Also, prioritizing edges with well connected endpoints increases the fanout, which again reduces tree height. The result is that we get a D^2FA that

has a much lower deferment depth.

2. The state partitioning $\mathcal{P}$ identifies a natural partitioning of states, such that all replications of one NFA state are in different partitions. So typically all partitions in $\mathcal{P}$ have sizes close to each other; and because of our tie breaking strategy, all the deferment trees have very similar structure. This property helps to improve the effectiveness of our D^{2}FA merge algorithm explained in the next section, and of our table consolidation technique explained in Section 5.3.

3. Having self-looping states as roots helps to improve the effectiveness of our variable striding technique which we describe in Section 5.4. And the condition that all roots states are self-looping states and vice versa is needed for our overlay automata approach described in Chapter 6.

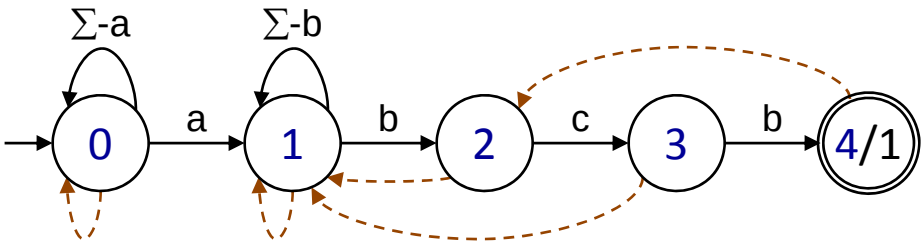
4.3.2 D^{2}FA Merge Algorithm

The UCP construction merges two DFAs together. We extend the UCP construction to merge two D^{2}FAs together as follows. To build a D^{2}FA from a DFA, we basically just need to set the deferment pointer, $F(u)$, for each state. During the UCP construction, as each new state u is created, we define $F(u)$ at the same time. We then define ρ to only include transitions for u that differ from $F(u)$.

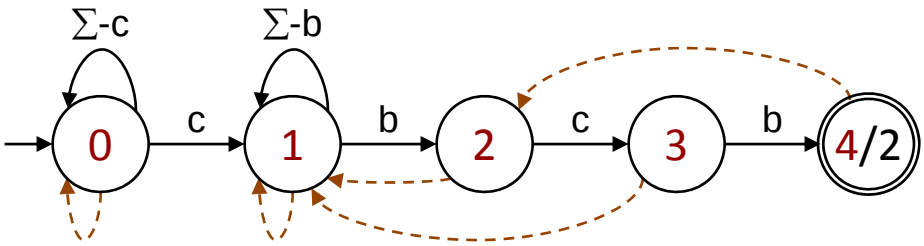
To help explain our algorithm, Figure 4.3 shows an example execution of the D^{2}FA merge algorithm. Figures 4.3(a) and 4.3(b) show the D^{2}FAs for the REs `/.*a.*bcb/`

and /.⋆c.⋆bcb/. Figure 4.3(c) shows the merged D^{2}FA for the D^{2}FAs in Figures 4.3(a) and 4.3(b). We use the following conventions when depicting a D^{2}FA. The dashed lines correspond to the deferred state for a given state. For each state in the merged D^{2}FA, the pair of numbers above the line refers to the states in the original D^{2}FAs that correspond to the state in the merged D^{2}FA. The number below the line is the state in the merged D^{2}FA. The number(s) after the '/' in accepting states gives the id(s) of the pattern(s) matched. Figure 4.3(d) shows how the deferred state is set for a few states in the merged D^{2}FA D_3. We explain the notation in this figure as we give our algorithm description.

For each state $u \in D_3$, we set the deferred state $F(u)$ as follows. While merging D^{2}FAs D_1 and D_2, let state $u = \langle p_0, q_0 \rangle$ be the new state currently being added to the merged D^{2}FA D_3. Let $p_0 \rightarrow p_1 \rightarrow \cdots \rightarrow p_l$ be the maximal deferment chain DC_1 (*i.e.* p_l defers to itself) in D_1 starting at p_0, and $q_0 \rightarrow q_1 \rightarrow \cdots \rightarrow q_m$ be the maximal deferment chain

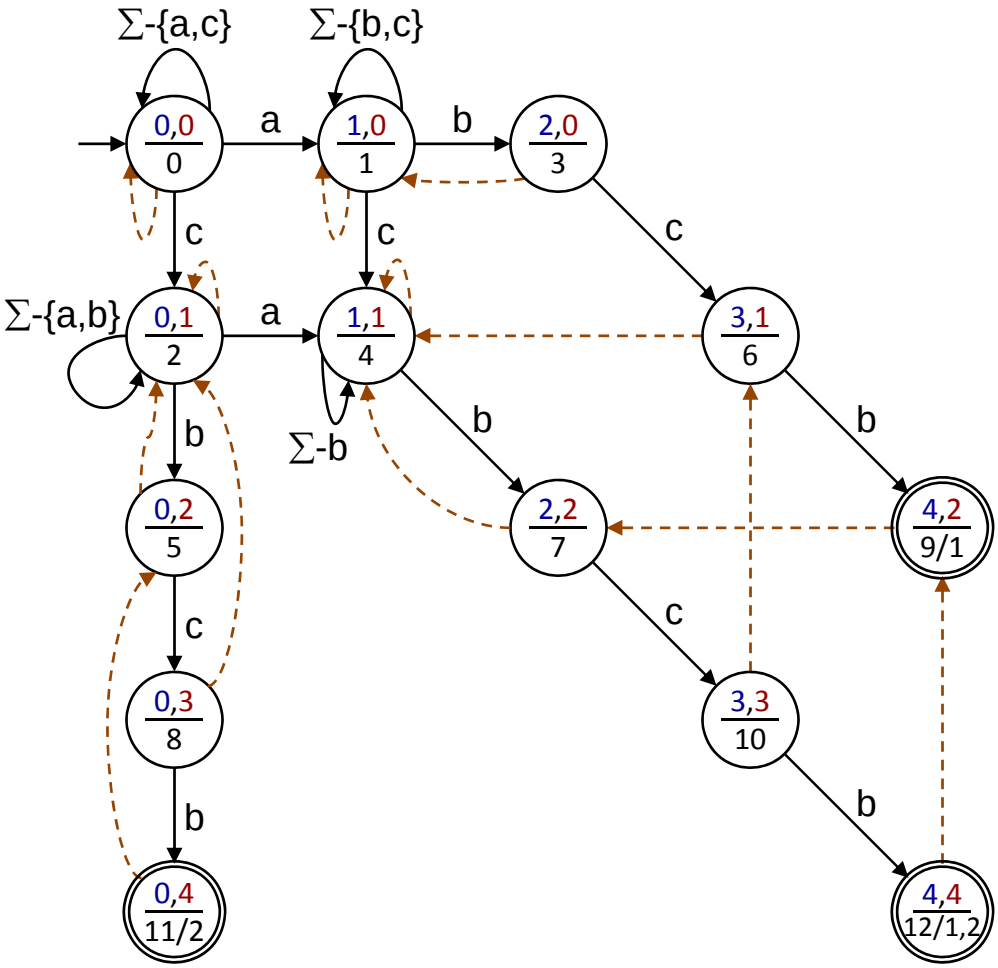


(a) D_1, the D^{2}FA for RE /.⋆a.⋆bcb/.

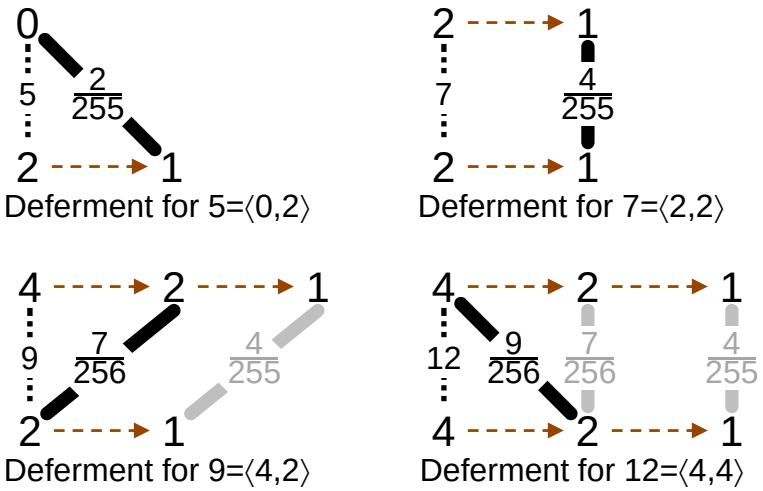


(b) D_2, the D^{2}FA for RE /.⋆c.⋆bcb/.

Figure 4.3: D^{2}FA merge example.



(c) D_3, the merged D^2FA.



(d) Illustration of setting deferment for some states in D_3.

Figure 4.3: D^2FA merge example (cont'd).

DC_2 in D_2 starting at q_0. For example, in Figure 4.3(d), we see the maximal deferment chains for $u = 5 = \langle 0,2\rangle$, $u = 7 = \langle 2,2\rangle$, $u = 9 = \langle 4,2\rangle$, and $u = 12 = \langle 4,4\rangle$. For $u = 9 = \langle 4,2\rangle$, the top row is the deferment chain of state 4 in D_1 and the bottom row is the deferment chain of state 2 in D_2. We will choose some state $\langle p_i, q_j\rangle$ where $0 \leq i \leq l$ and $0 \leq j \leq m$ to be $F(u)$. In Figure 4.3(d), we represent these candidate $F(u)$ pairs with edges between the nodes of the deferment chains. For each candidate pair, the number on the top is the corresponding state number in D_3 and the number on the bottom is the number of common transitions in D_3 between that pair and state u. For example, for $u = 9 = \langle 4,2\rangle$, the two candidate pairs represented are state 7 ($\langle 2,2\rangle$) which shares 256 transitions in common with state 9 and state 4 ($\langle 1,1\rangle$) which shares 255 transitions in common with state 9. Note that a candidate state pair is only considered if it is reachable in D_3. In Figure 4.3(d) with $u = 9 = \langle 4,2\rangle$, three of the candidate pairs corresponding to $\langle 4,1\rangle$, $\langle 2,1\rangle$, and $\langle 1,2\rangle$ are not reachable, so no edge is included for these candidate pairs. Ideally, we want i and j to be as small as possible though not both 0. For example, our best choices are typically $\langle p_0, q_1\rangle$ or $\langle p_1, q_0\rangle$. In the first case, $p_0 \sqcap p_1 = \langle p_0, q_0\rangle \sqcap \langle p_1, q_0\rangle$, and we already have $p_0 \rightarrow p_1$ in D_1. In the second case, $q_0 \sqcap q_1 = \langle p_0, q_0\rangle \sqcap \langle p_0, q_1\rangle$, and we already have $q_0 \rightarrow q_1$ in D_2. In Figure 4.3(d), we set $F(u)$ to be $\langle p_0, q_1\rangle$ for $u = 5 = \langle 0,2\rangle$ and $u = 12 = \langle 4,4\rangle$, and we use $\langle p_1, q_0\rangle$ for $u = 9 = \langle 4,2\rangle$. However, it is possible that both states are not reachable from the start state in D_3. This leads us to consider other possible $\langle p_i, q_j\rangle$. For example, in Figure 4.3(d), both $\langle 2,1\rangle$ and $\langle 1,2\rangle$ are not reachable in D_3, so we use reachable state $\langle 1,1\rangle$ as $F(u)$ for $u = 7 = \langle 2,2\rangle$.

We consider a few different algorithms for choosing $\langle p_i, q_j \rangle$. The first algorithm which we call the *first match method* is to find a pair of states $(p_i,\ q_j)$ for which $\langle p_i, q_j \rangle \in Q_3$ and $i + j$ is minimum. Stated another way, we find the minimum $z \geq 1$ such that the set of states $Z = \{\langle p_i, q_{z-i} \rangle \mid (\max(0, z - m) \leq i \leq \min(l, z)) \wedge (\langle p_i, q_{z-i} \rangle \in Q_3)\} \neq \emptyset$. From the set of states Z, we choose the state that has the most transitions in common with $\langle p_0, q_0 \rangle$ breaking ties arbitrarily. If Z is empty for all $z > 1$, then we just pick $\langle p_0, q_0 \rangle$, *i.e.* the deferment pointer is not set (or the state defers to itself). The idea behind the first match method is that $\langle p_0, q_0 \rangle \sqcap \langle p_i, q_j \rangle$ decreases as $i + j$ increases. In Figure 4.3(d), all the selected $F(u)$ correspond to the first match method.

A second more complete algorithm for setting $F(u)$ is the *best match method* where we always consider all $(l+1) \times (m+1) - 1$ pairs and pick the pair that is in Q_3 and has the most transitions in common with $\langle p_0, q_0 \rangle$. The idea behind the best match method is that it is not always true that $\langle p_0, q_0 \rangle \sqcap \langle p_x, q_y \rangle \geq \langle p_0, q_0 \rangle \sqcap \langle p_{x+i}, q_{y+j} \rangle$ for $i + j > 0$. For instance we can have $p_0 \sqcap p_2 < p_0 \sqcap p_3$, which would mean $\langle p_0, q_0 \rangle \sqcap \langle p_2, q_0 \rangle < \langle p_0, q_0 \rangle \sqcap \langle p_3, q_0 \rangle$. In such cases, the first match method will not find the pair along the deferment chains with the most transitions in common with $\langle p_0, q_0 \rangle$. In Figure 4.3(d), all the selected $F(u)$ also correspond to the best match method. It is difficult to create a small example where first match and best match differ.

When adding the new state u to D_3, it is possible that some state pairs along the deferment chains that were not in Q_3 while finding the deferred state for u will later on be added to Q_3. This means that after all the states have been added to Q_3, the deferment for u

can potentially be improved. Thus, after all the states have been added, for each state we again find a deferred state. If the new deferred state is better than the old one, we reset the deferment to the new deferred state. Algorithm 4.4 shows the pseudocode for the D^2FA merge algorithm with the first match method for choosing a deferred state. Note that we use u and $\langle u_1, u_2 \rangle$ interchangeably to indicate a state in the merged D^2FA D_3 where u is a state in Q_3, and u_1 and u_2 are the states in Q_1 and Q_2, respectively, that state u corresponds to.

4.3.3 Direct D^2FA construction for RE set

Similar to efficient DFA construction, we first build the D^2FA for each RE in $\mathcal{R}$. We now need to merge the D^2FAs together using the `D2FAMerge` algorithm from the previous section. We consider a variety of methods for merging the D^2FAs together including a greedy "Huffman" approach, where in each step, the two smallest D^2FA are merged together. The best approach, we have found experimentally, is to merge all the D^2FAs in a balanced binary tree fashion. This is because a binary tree minimizes the worst-case number of merges that any RE experiences.

We use two different variations of our D2FAMerge algorithm while merging D^2FAs. For all merges except the final merge, we use the first match method for setting $F(u)$. When doing the final merge to get the final D^2FA, we use the best match method for setting $F(u)$. It turns out that using the first match method results in a better deferment forest structure in the D^2FA, which helps when the D^2FA is further merged with other D^2FAs.

Input: A pair of D^2FAs, $D_1 = (Q_1, \Sigma, \rho_1, q_{01}, M_1, F_1)$ and $D_2 = (Q_2, \Sigma, \rho_2, q_{02}, M_2, F_2)$, corresponding to RE sets, say R_1 and R_2, with $R_1 \cap R_2 = \emptyset$.
Output: A D^2FA corresponding to the RE set $R_1 \cup R_2$

```
1  Initialize D3 to an empty D²FA;
2  Initialize queue as an empty queue;
3  queue.push (⟨q01, q02⟩);
4  while queue not empty do
5      ⟨u, u1⟩u2 ← queue.pop();
6      Q3 ← Q3 ∪ {u};
7      foreach c ∈ Σ do
8          nxt ← ⟨δ1'(u1, c), δ2'(u2, c)⟩;
9          if nxt ∉ Q3 ∧ nxt ∉ queue then queue.push (nxt);
10         Add (u, c) → nxt transition to ρ3;
11     M3(u) ← M1(u1) ∪ M2(u2);
12     F3(u) ← FindDefState(u);
13     Remove transitions for u from ρ3 that are in common with F3(u);
14 foreach u ∈ Q3 do
15     newDptr ← FindDefState(u);
16     if (newDptr ≠ F3(u)) ∧ (newDptr ⊓ u > F3(u) ⊓ u) then
17         F3(u) ← newDptr;
18         Reset all transitions for u in ρ3 and then remove ones that are in common with F3(u);
19 return D3;

20 Function FindDefState(⟨ν1, ν2⟩)
21   Let ⟨p0 = ν1, p1, ..., pl⟩ be the list of states on the deferment chain from ν1 to the root in D1;
22   Let ⟨q0 = ν2, q1, ..., qm⟩ be the list of states on the deferment chain from ν2 to the root in D2;
23   for z = 1 to (l + m) do
24       S ← {⟨pi, q(z−i)⟩ | (max(0, z − m) ≤ i ≤ min(l, z)) ∧ (⟨pi, q(z−i)⟩ ∈ Q3)};
25       if S ≠ ∅ then return argmax(ν∈S)(⟨ν1, ν2⟩ ⊓ ν);
26   return ⟨ν1, ν2⟩;
```

Figure 4.4: Algorithm $D2FAMerge(D_1, D_2)$ for merging two D^2FAs.

The local optimization achieved by using the best match method only helps when used in the final merge.

4.3.4 Optional Final Compression Algorithm

When there is no bound on the deferment depth (see Section 4.4.2), the original D^{2}FA algorithm proposed in [26] results in a D^{2}FA with smallest possible size because it runs Kruskal's algorithm on a large SRG. Our D^{2}FA merge algorithm results in a slightly larger D^{2}FA because it uses a greedy approach to determine deferment. We can further reduce the size of the D^{2}FA produced by our algorithm by running the following compression algorithm on the D^{2}FA produced by the D^{2}FA merge algorithm.

We construct an SRG and perform a maximum weight spanning tree construction on the SRG, but we only add edges to the SRG that have the potential to reduce the size of the D^{2}FA. More specifically, let u and v be any two states in the current D^{2}FA. We only add the edge $e = (u, v)$ in the SRG if its weight $w(e)$ is $\geq \min(u \sqcap F(u), v \sqcap F(v))$. Here, $F(u)$ is the deferred state of u in the current D^{2}FA. As a result, very few edges are added to the SRG, so we only need to run Kruskal's algorithm on a small SRG. This saves both space and time compared to previous D^{2}FA construction methods. However, this compression step does require more time and space than the D^{2}FA merge algorithm because it does construct an SRG and then runs Kruskal's algorithm on the SRG.

4.4 D^{2}FA Merge Algorithm Properties

We now discuss some properties of the D^{2}FA merge algorithm itself and the resulting D^{2}FA.

4.4.1 Proof of Correctness

The D^2FA merge algorithm exactly follows the UCP construction to create the states. So the correctness of the underlying DFA follows from the the correctness of the UCP construction.

Theorem 2 shows that the merged D^2FA is also well defined (no cycles in deferment forest).

Lemma 1. *In the D^2FA $D_3 = \text{D2FAMerge}(D_1, D_2)$, $\langle u_1, u_2\rangle \twoheadrightarrow \langle v_1, v_2\rangle \Rightarrow u_1 \twoheadrightarrow v_1 \wedge u_2 \twoheadrightarrow v_2$.*

Proof. If $\langle u_1, u_2\rangle = \langle v_1, v_2\rangle$ then the lemma is trivially true. Otherwise, let $\langle u_1, u_2\rangle \rightarrow \langle w_1, w_2\rangle \twoheadrightarrow \langle v_1, v_2\rangle$ be the deferment chain in D_3. When selecting the deferred state for $\langle u_1, u_2\rangle$, D2FA Merge always choose a state that corresponds to a pair of states along deferment chains for u_1 and u_2 in D_1 and D_2, respectively. Therefore, we have that $\langle u_1, u_2\rangle \rightarrow \langle w_1, w_2\rangle \Rightarrow u_1 \twoheadrightarrow w_1 \wedge u_2 \twoheadrightarrow w_2$. By induction on the length of the deferment chain and the fact that the $\twoheadrightarrow$ relation is transitive, we get our result. □

Theorem 2. *If D^2FAs D_1 and D_2 are well defined, then the D^2FA $D_3 = \text{D2FAMerge}(D_1, D_2)$ is also well defined.*

Proof. Since D_1 and D_2 are well defined, there are no cycles in their deferment forests. Now assume that D_3 is not well defined, *i.e.* there is a cycle in its deferment forest. Let

$\langle u_1, u_2\rangle$ and $\langle v_1, v_2\rangle$ be two distinct states on the cycle. Then, we have that

$$\langle u_1, u_2\rangle \twoheadrightarrow \langle v_1, v_2\rangle \wedge \langle v_1, v_2\rangle \twoheadrightarrow \langle u_1, u_2\rangle$$

Using Lemma 1 we get

$$(u_1 \twoheadrightarrow v_1 \wedge u_2 \twoheadrightarrow v_2) \wedge (v_1 \twoheadrightarrow u_1 \wedge v_2 \twoheadrightarrow u_2)$$

$$i.e.\ (u_1 \twoheadrightarrow v_1 \wedge v_1 \twoheadrightarrow u_1) \wedge (u_2 \twoheadrightarrow v_2 \wedge v_2 \twoheadrightarrow u_2)$$

Since $\langle u_1, u_2\rangle \neq \langle v_1, v_2\rangle$, we have $u_1 \neq v_1 \vee u_2 \neq v_2$ which implies that at least one of D_1 or D_2 has a cycle in its deferment forest, which is a contradiction. □

4.4.2 Limiting Deferment Depth

Since no input is consumed while traversing a deferred transition, in the worst case, the number of lookups needed to process one input character is given by the deferment depth of the D^{2}FA. As proposed in [26], we can guarantee a worst case performance by limiting the deferment depth of the D^{2}FA.

Recall that $\psi(u)$ denoted the deferment depth of state u, and $\Psi(D)$ denoted the deferment depth of the D^{2}FA D.

Lemma 2. *In the D^{2}FA* $D_3 = \text{D2FAMerge}(D_1, D_2)$, $\forall \langle u_1, u_2\rangle \in Q_3$, $\psi(\langle u_1, u_2\rangle) \leq \psi(u_1) + \psi(u_2)$.

Proof. Let $\psi(\langle u_1, u_2\rangle) = d$. If $\psi(\langle u_1, u_2\rangle) = 0$, then $\langle u_1, u_2\rangle$ is a root and the lemma is trivially true. So, we consider $d \geq 1$ and assume the lemma is true for all states with $\psi < d$. Let $\langle u_1, u_2\rangle \rightarrow \langle w_1, w_2\rangle \twoheadrightarrow \langle v_1, v_2\rangle$ be the deferment chain in D_3. Using the inductive hypothesis, we have

$$\psi(\langle w_1, w_2\rangle) \leq \psi(w_1) + \psi(w_2)$$

Given $\langle u_1, u_2\rangle \neq \langle w_1, w_2\rangle$, we assume without loss of generality that $u_1 \neq w_1$. Using Lemma 1 we get that $u_1 \twoheadrightarrow w_1$. Therefore $\psi(w_1) \leq \psi(u_1) - 1$. Combining the above, we get

$$\begin{aligned}
\psi(\langle u_1, u_2\rangle) &= \psi(\langle w_1, w_2\rangle) + 1 \\
&\leq \psi(w_1) + \psi(w_2) + 1 \\
&\leq (\psi(u_1) - 1) + \psi(u_2) + 1 \\
&\leq \psi(u_1) + \psi(u_2)
\end{aligned}$$

□

Lemma 2 directly gives us the following Theorem.

Theorem 3. *If* $D_3 = \text{D2FAMerge}(D_1, D_2)$, *then* $\Psi(D_3) \leq \Psi(D_1) + \Psi(D_2)$.

For an RE set $\mathcal{R}$, if the initial D^2FAs have $\Psi = d$, in the worst case, the final merged D^2FA corresponding to $\mathcal{R}$ can have $\Psi = d \times |\mathcal{R}|$. Although Theorem 3 gives the value of Ψ in the worst case, in practical cases, $\Psi(D_3)$ is very close to $\max(\Psi(D_1), \Psi(D_2))$. Thus

the deferment depth of the final merged D^2FA is usually not much higher than d.

Let Ω denote the desired upper bound on Ψ. To guarantee $\Psi(D_3) \leq \Omega$, we modify the `FindDefState` subroutine in Algorithm 4.4 as follows: When selecting candidate pairs for the deferred state, we only consider states with $\psi < \Omega$. Specifically, we replace line 24 with the following

$$S := \{\langle p_i, q_{z-i}\rangle \mid (\max(0, z-m) \leq i \leq \min(l, z)) \wedge \langle p_i, q_{z-i}\rangle \in Q_3) \wedge (\psi(\langle p_i, q_{z-i}\rangle) < \Omega)\}$$

When we do the second pass (lines 14-18), we may increase the deferment depth of nodes that defer to nodes that we readjust. We record the affected nodes and then do a third pass to reset their deferment states so that the maximum depth bound is satisfied. In practice, this happens very rarely.

When constructing a D^2FA with a given bound Ω, we first build D^2FAs without this bound. We only apply the bound Ω when performing the final merge of two D^2FAs to create the final D^2FA.

4.4.3 Deferment to a Lower Level

Becchi and Crowley [8] propose a D^2FA algorithm where each state defers to a state at a lower level than itself (see Section 3.3.4.) More formally, they ensure that for all states u, $level(u) > level(F(u))$ if $F(u) \neq u$. We call this property the *back-pointer* property. If the back-pointer property holds, then every deferred transition taken decreases the level of the

current state by at least 1. Since a regular transition on an input character can only increase the level of the current state by at most 1, there have to be fewer deferred transitions taken on the entire input string than regular transitions. This gives an amortized cost of at most 2 transitions taken per input character.

Unfortunately, if D^{2}FAs D_1 and D_2 have the back-pointer property, the merged D^{2}FA $D_3 = \text{D2FAMerge}(D_1, D_2)$ is not guaranteed to have the back-pointer property. A simple counter example is when trying to merge the D^{2}FAs corresponding to the REs `/(aaa)+/` and `/(aaaa)+/`. Typically, for practical cases, if the initial D^{2}FAs have the back-pointer property, in the final merged D^{2}FA, almost all of the states have the back-pointer property.

In order to guarantee the D^{2}FA D_3 has the back-pointer property, we perform a similar modification to the `FindDefState` subroutine in Algorithm 4.4 as we performed when we wanted to limit the maximum deferment depth. When selecting candidate pairs for the deferred state, we only consider states with a lower level. Specifically, we replace line 24 with the following:

$$S := \{\langle p_i, q_{z-i}\rangle \mid (\max(0, z-m) \leq i \leq \min(l, z)) \wedge$$
$$(\langle p_i, q_{z-i}\rangle \in Q_3) \wedge (level(\langle v_1, v_2\rangle) > level(\langle p_i, q_{z-i}\rangle))\}$$

For states for which no candidate pairs are found, we just search through all states in Q_3 that are at a lower level for the deferred state. In practice, this search through all the states needs to be done for very few states because if D^{2}FAs D_1 and D_2 have the back-pointer

property, then almost all the states in D^2FAs D_3 have the back-pointer property. As with limiting maximum deferment depth, we only apply this restriction when performing the final merge of two D^2FAs to create the final D^2FA.

4.4.4 Algorithmic Complexity

The time complexity of the original D^2FA algorithm proposed in [26] is $O(n^2(\log(n)+|\Sigma|))$. The SRG has $O(n^2)$ edges, and $O(|\Sigma|)$ time is required to add each edge to the SRG and $O(\log(n))$ time is required to process each edge in the SRG during the maximum spanning tree routine. The time complexity of the D^2FA algorithm proposed in [8] is $O(n^2|\Sigma|)$. Each state is compared with $O(n)$ other states, and each comparison requires $O(|\Sigma|)$ time.

The time complexity of our new D2FAMerge algorithm to merge two D^2FAs is $O(n\Psi_1\Psi_2|\Sigma|)$, where n is the number of states in the merged D^2FA, and Ψ_1 and Ψ_2 are the maximum deferment depths of the two input D^2FAs. When setting the deferment for any state $u = \langle u_1, u_2\rangle$, in the worst case the algorithm compares $\langle u_1, u_2\rangle$ with all the pairs along the deferment chains of u_1 and u_2, which are at most Ψ_1 and Ψ_2 in length, respectively. Each comparison requires $O(|\Sigma|)$ time. In practice, the time complexity is $O(n|\Sigma|)$ as each state needs to be compared with very few states for the following three reasons. First, the maximum deferment depth Ψ is usually very small. The largest value of Ψ among our 8 primary RE sets in Section 4.5 is 7. Second, the length of the deferment chains for most states is much smaller than Ψ. The largest value of average deferment depth $\overline{\psi}$ among our 8 RE sets is 2.54. Finally, many of the state pairs along the deferment chains are

not reachable in the merged D^2FA. Among our 8 RE sets, the largest value of the average number of comparisons needed is 1.47.

When merging all the D^2FAs together for an RE set $\mathcal{R}$, the total time required in the worst case would be $O(n\Psi_1\Psi_2|\Sigma|\log(|\mathcal{R}|))$. The worst case would happen when the RE set contains strings and there is no state explosion. In this case, each merged D^2FA would have a number of states roughly equal to the sum of the sizes of the D^2FAs being merged. When there is state explosion, the last D^2FA merge would be the dominating factor, and the total time would just be $O(n\Psi_1\Psi_2|\Sigma|)$.

When modifying the D2FAMerge algorithm to maintain back-pointers, the worst case time would be $O(n^2|\Sigma|)$ because we would have to compare each state with $O(n)$ other states if none of the candidate pairs are found at a lower level than the state. In practice, this search needs to be done for very few states, typically less than 1%.

The worst case time complexity of the final compression step is the same as that of Kumar *et al.*'s D^2FA algorithm, which is $O(n^2(\log(n) + |\Sigma|))$, since both involve computing a maximum weight spanning tree on the SRG. However, because we only consider edges which improve upon the existing deferment forest, the actual size of the SRG in practice is typically linear in the number of nodes. In particular, for the real-world RE sets that we consider in the experiments section, the size of the SRG generated by our final compression step is on average 100 times smaller than the SRG generated by Kumar *et al.*'s algorithm. As a result the optimization step requires much less memory and time compared to the original algorithm.

4.5 Experimental Results

In this section, we evaluate the effectiveness of our algorithms on real-world and synthetic RE sets. We consider two variants of our D^{2}FA merge algorithm: the main variant $D^2FAMERGE$ which just merged the D^{2}FAs, and $D^2FAMERGEOPT$, which applies our final compression algorithm after running D^{2}FAMERGE. We compare our algorithms with the original D^{2}FA construction algorithm proposed in [26] $ORIGINAL$ that optimizes transition compression and the D^{2}FA construction algorithm proposed in [8] $BACKPTR$ that enforces the back-pointer property described in Section 4.4.3.

4.5.1 Methodology

4.5.1.1 Data Sets

Our main results are based on eight real RE sets, four proprietary RE sets C7, C8, C10, and C613 from a large networking vendor and four public RE sets Bro217, Snort 24, Snort31, and Snort 34, that we partition into three groups, STRING, WILDCARD, and SNORT, based upon their RE composition. For each RE set, the number indicates the number of REs in the RE set. The STRING RE sets, C613 and Bro217, contain mostly string matching REs. The WILDCARD RE sets, C7, C8 and C10, contain mostly REs with multiple wildcard closures '$.\star$'. The SNORT RE sets, Snort24, Snort31, and Snort34, contain a more diverse set of REs, roughly 40% of which have wildcard closures. To test scalability, we use Scale, a synthetic RE set consisting of 26 REs of the form

/.⋆c_u0123456.⋆c_l789!#%&/, where c_u and c_l are the 26 uppercase and lowercase alphabet letters. Even though all the REs are nearly identical differing only in the character after the two .⋆'s, we still get the full multiplicative effect where the number of states in the corresponding minimum state DFA roughly doubles for every RE added.

4.5.1.2 Metrics

We use the following metrics to evaluate the algorithms. First, we measure the resulting D^{2}FA size (# transitions) to assess transition compression performance. Our D^{2}FAMERGE algorithm typically performs almost as well as the other algorithms even though it builds up the D^{2}FA incrementally rather than compressing the final minimum state DFA. Second, we measure the the maximum deferment depth (Ψ) and average deferment depth ($\overline{\psi}$) in the D^{2}FA to assess how quickly the resulting D^{2}FA can be used to perform regular expression matching. Smaller Ψ and $\overline{\psi}$ mean that fewer deferment transitions that process no input characters need to be traversed when processing an input string. Our D^{2}FAMERGE significantly outperforms the other algorithms. Finally, we measure the space and time required by the algorithm to build the final automaton. Again, our D^{2}FAMERGE significantly outperforms the other algorithms. When comparing the performance of D^{2}FAMERGE with another algorithm A on a given RE or RE set, we define the following quantities to compare them: transition increase is (D^{2}FAMERGE D^{2}FA size - A D^{2}FA size) divided by A D^{2}FA size, transition decrease is (A D^{2}FA size - D^{2}FAMERGE D^{2}FA size) divided by A D^{2}FA size, average (maximum) deferment depth

ratio is A average (maximum) deferment depth divided by D^{2}FAMERGE average (maximum) deferment depth, space ratio is A space divided by D^{2}FAMERGE space, and time ratio is A build time divided by D^{2}FAMERGE build time.

4.5.1.3 Measuring Space

When measuring the required space for an algorithm, we measure the maximum amount of memory required at any point in time during the construction and then final storage of the automaton. This is a difficult quantity to measure exactly; we approximate this required space for each of the algorithms as follows. For D^{2}FAMERGE, the dominant data structure is the D^{2}FA. For a D^{2}FA, the transitions for each state can be stored as pairs of input character and next state id, so the memory required to store a D^{2}FA is calculated as $= (\#transitions) \times 5$ bytes. However, the maximum amount of memory required while running D^{2}FAMERGE may be higher than the final D^{2}FA size because of the following two reasons. First, when merging two D^{2}FAs, we need to maintain the two input D^{2}FAs as well as the output D^{2}FA. Second, we may create an intermediate output D^{2}FA that has more transitions than needed; these extra transitions will be eliminated once all D^{2}FA states are added. We keep track of the worst case required space for our algorithm during D^{2}FA construction. This typically occurs when merging the final two intermediate D^{2}FA to form the final D^{2}FA.

For ORIGINAL, we measure the space required by the minimized DFA and the SRG. For the DFA, the transitions for each state can be stored as an array of size Σ with each array

entry requiring four bytes to hold the next state id. For the SRG, each edge requires 17 bytes as observed in [8]. This leads to a required memory for building the D^{2}FA of $= |Q| \times |\Sigma| \times 4 + (\#edges\ in\ SRG) \times 17$ bytes.

For D^{2}FAMERGEOPT, the space required is the size of the final D^{2}FA resulting from the merge step, plus the size of the SRG used by the final compression algorithm. The sizes are computed as in the case of D^{2}FAMERGE and ORIGINAL.

For BACKPTR, we consider two variants. The first variant builds the minimized DFA directly from the NFA and then sets the deferment for each state. For this variant, no SRG is needed, so the space required is the space needed for the minimized DFA which is $|Q| \times |\Sigma| \times 4$ bytes. The second variant goes directly from the NFA to the final D^{2}FA; this variant uses less space but is much slower as it stores incomplete transition tables for most states. Thus, when computing the deferment state for a new state, the algorithm must recreate the complete transition tables for each state to determine which has the most common transitions with the new state. For this variant, we assume the only space required is the space to store the final D^{2}FA which is $= (\#transitions) \times 5$ bytes even though more memory is definitely needed at various points during the computation. We also note that both implementations must perform the NFA to DFA subset construction on a large NFA which means even the faster variant runs much more slowly than D^{2}FAMERGE.

4.5.1.4 Correctness

We tested correctness of our algorithms by verifying the final D^{2}FA is equivalent to the corresponding DFA. Note, we can only do this check for our RE sets where we were able to compute the corresponding DFA. Thus, we only verified correctness of the final D^{2}FA for our eight real RE sets and the smaller Scale RE sets.

4.5.2 D^{2}FAMERGE versus ORIGINAL

We first compare D^{2}FAMERGE with ORIGINAL that optimizes transition compression when both algorithms have unlimited maximum deferment depth. These results are shown in Table 4.1 for our 8 primary RE sets.

RE set	# States	ORIGINAL					D^{2}FAMERGE				
		# Trans	Def. depth		RAM (MB)	Time (s)	# Trans	Def. depth		RAM (MB)	Time (s)
			Avg.	Max.				Avg.	Max.		
Bro217	6533	9816	3.42	8	179.3	119.4	11737	2.15	5	0.13	3.2
C613	11308	21633	8.43	16	1039.5	326.0	26709	2.69	7	0.23	9.7
C7	24750	205633	19.18	30	47.4	397.7	207540	1.14	3	1.07	0.9
C8	3108	23209	8.95	13	4.9	14.5	23334	1.14	2	0.14	0.2
C10	14868	96793	13.68	27	25.5	141.0	97296	1.18	3	0.52	0.6
Snort24	13886	38485	9.53	20	861.2	299.2	39409	1.56	4	0.32	0.2
Snort31	20068	70701	11.41	23	298.5	244.3	92284	2.00	6	1.29	2.6
Snort34	13825	40199	9.99	17	795.4	309.9	43141	1.38	5	0.27	1.8

Table 4.1: The D^{2}FA size, D^{2}FA average $\overline{\psi}$ and maximum Ψ deferment depths, space estimate and time required to build the D^{2}FA for ORIGINAL and D^{2}FAMERGE.

Table 4.2 summarizes these results by RE group. We make the following observations.

(1) D^{2}FAMERGE uses much less space than ORIGINAL. On average, D^{2}FAMERGE uses 1500 times less memory than ORIGINAL to build the resulting D^{2}FA. This difference

RE set group	D^{2}FAMERGE					D^{2}FAMERGEOPT				
	Trans increase	Def. depth ratio		Space ratio	Time ratio	Trans increase	Def. depth ratio		Space ratio	Time ratio
		Avg.	Max.				Avg.	Max.		
All	10.8%	7.5	5.2	1499.8	154.5	0.4%	7.4	5.4	113.1	9.4
STRING	21.5%	2.4	1.9	2994.8	35.4	0.0%	2.1	1.6	103.5	0.8
WILDCARD	1.0%	12.1	8.5	42.8	246.6	1.0%	12.1	10.0	16.8	10.8
SNORT	13.3%	6.3	4.1	1960.3	141.8	0.0%	6.1	3.3	215.8	13.7

Table 4.2: Average values of transition increase, deferment depth ratios, space ratios, and time ratios for D^{2}FAMERGE and D^{2}FAMERGEOPT compared with ORIGINAL.

is most extreme when the SRG is large, which is true for the two STRING RE sets and Snort24 and Snort34. For these RE sets, D^{2}FAMERGE uses between 1422 and 4568 times less memory than ORIGINAL. For the RE sets with relatively small SRGs such as those in the WILDCARD and Snort31, D^{2}FAMERGE uses between 35 and 231 times less space than ORIGINAL.

(2) D^{2}FAMERGE is much faster than ORIGINAL. On average, D^{2}FAMERGE builds the D^{2}FA 155 times faster than ORIGINAL. This time difference is maximized when the deferment chains are shortest. For example, D^{2}FAMERGE only requires an average of 0.05 msec and 0.09 msec per state for the WILDCARD and SNORT RE sets, respectively, so D^{2}FAMERGE is, on average, 247 and 142 times faster than ORIGINAL for these RE sets, respectively. For the STRING RE sets, the deferment chains are longer, so D^{2}FAMERGE requires an average of 0.67 msec per state, and is, on average, 35 times faster than ORIGINAL.

(3) D^{2}FAMERGE produces D^{2}FA with much smaller average and maximum deferment depths than ORIGINAL. On average, D^{2}FAMERGE produces D^{2}FA that have average deferment depths that are 7.5 times smaller than ORIGINAL and maximum de-

ferment depths that are 5.2 times smaller than ORIGINAL. In particular, the average deferment depth for D^{2}FAMERGE is less than 2 for all but the two STRING RE sets, where the average deferment depths are 2.15 and 2.69. Thus, the expected number of deferment transitions to be traversed when processing a length n string is less than n. One reason D^{2}FAMERGE works so well is that it eliminates low weight edges from the SRG so that the deferment forest has many shallow deferment trees instead of one deep tree. This is particularly effective for the WILDCARD RE sets and, to a lesser extent, the SNORT RE sets. For the STRING RE sets, the SRG is fairly dense, so D^{2}FAMERGE has a smaller advantage relative to ORIGINAL.

(4) D^{2}FAMERGE produces D^{2}FA with only slightly more transitions than ORIGINAL, particularly on the RE sets that need transition compression the most. On average, D^{2}FAMERGE produces D^{2}FA with roughly 11% more transitions than ORIGINAL does. D^{2}FAMERGE works best when state explosion from wildcard closures creates DFA composed of many similar repeating substructures. This is precisely when transition compression is most needed. For example, for the WILDCARD RE sets that experience the greatest state explosion, D^{2}FAMERGE only has 1% more transitions than ORIGINAL. On the other hand, for the STRING RE sets, D^{2}FAMERGE has, on average, 22% more transitions. For this group, ORIGINAL needed to build a very large SRG and thus used much more space and time to achieve the improved transition compression. Furthermore, transition compression is typically not needed for such RE sets as all string matching REs can be placed into a single group and the resulting DFA can be built.

In summary, D^{2}FAMERGE achieves its best performance relative to ORIGINAL on the WILDCARD RE sets (except for space used for construction of the D^{2}FA) and its worst performance relative to ORIGINAL on the STRING RE sets (except for space used to construct the D^{2}FA). This is desirable as the space and time efficient D^{2}FAMERGE is most needed on RE sets like those in the WILDCARD because those RE sets experience the greatest state explosion.

4.5.3 Assessment of Final Compression Algorithm

We now assess the effectiveness of our final compression algorithm by comparing D^{2}FAMERGEOPT to ORIGINAL and D^{2}FAMERGE. The results are shown in Table 4.3 for our 8 primary RE sets.

RE set	# States	# Trans	Def. depth		RAM (MB)	Time (s)
			Avg.	Max.		
Bro217	6533	9816	2.44	7	2.64	99.2
C613	11308	21633	3.04	8	7.48	940.4
C7	24750	207540	1.14	3	2.49	45.7
C8	3108	23334	1.14	2	0.32	1.0
C10	14868	97296	1.17	2	1.61	14.8
Snort24	13886	38601	1.57	4	2.67	19.9
Snort31	20068	70780	2.17	8	15.61	59.1
Snort34	13825	40387	1.42	8	2.60	14.2

Table 4.3: The D^{2}FA size, D^{2}FA average $\overline{\psi}$ and maximum Ψ deferment depths, space estimate and time required to build the D^{2}FA for D^{2}FAMERGEOPT.

Table 4.2 summarizes these results by RE group. As expected D^{2}FAMERGEOPT produces a D^{2}FA that is almost as small as that produced by ORIGINAL; on average, the number of transitions increases by only 0.4%. There is a very small increase for WILDCARD and

SNORT because ORIGINAL also considers all edges with weight > 1 in the SRG, whereas D^{2}FAMERGEOPT does not use edges with weight < 10. There is a significant benefit to not using these low weight SRG edges; the deferment depths are much higher for the D^{2}FA produced by ORIGINAL when compared to the D^{2}FA produced by D^{2}FAMERGEOPT.

The final compression algorithm of D^{2}FAMERGEOPT does require more resources than are required by D^{2}FAMERGE. In some cases, this may limit the size of the RE set D^{2}FAMERGEOPT can be used for. However, as explained earlier, D^{2}FAMERGE performs best on the WILDCARD (which has the most state explosion) and performs the worst on the STRING (which has the no or limited state explosion). So the final compression algorithm is only needed for and is most beneficial for RE sets with limited state explosion. Finally, we observe that D^{2}FAMERGEOPT requires on average 113 times less RAM than ORIGINAL, and, on average, runs 9 times faster than ORIGINAL.

4.5.4 D^{2}FAMERGE versus ORIGINAL with Bounded Maximum Deferment Depth

We now compare D^{2}FAMERGE and ORIGINAL when they impose a maximum deferment depth bound Ω of 1, 2, and 4. Because time and space do not change significantly, we focus only on number of transitions and average deferment depth. These results are shown in Table 4.4. Note that for these data sets, the resulting maximum depth Ψ typically is identical to the maximum depth bound Ω; the only exception is for D^{2}FAMERGE and $\Omega = 4$; thus we omit the maximum deferment depth from Table 4.4.

RE set	ORIGINAL						D^2FAMERGE					
	# Trans			Avg. def. depth			# Trans			Avg. def. depth		
	$\Omega = 1$	$\Omega = 2$	$\Omega = 4$	Ω=1	Ω=2	Ω=4	$\Omega = 1$	$\Omega = 2$	$\Omega = 4$	Ω=1	Ω=2	Ω=4
Bro217	698229	296433	52628	0.62	1.18	2.09	50026	15087	11757	1.00	1.83	2.15
C613	1204831	507613	102183	0.62	1.17	2.16	154548	51858	27735	1.00	1.94	2.64
C7	2044171	597544	206814	0.71	1.24	2.07	215940	208044	207540	0.97	1.13	1.14
C8	206897	40411	23261	0.77	1.32	2.51	24090	23334	23334	0.98	1.14	1.14
C10	1105160	325536	97137	0.75	1.31	2.39	101556	97326	97296	0.98	1.18	1.18
Snort24	1376779	543378	106211	0.66	1.25	2.39	68906	42176	39409	0.99	1.47	1.56
Snort31	2193679	1102693	405785	0.62	1.11	2.08	208136	119810	95496	1.00	1.52	1.97
Snort34	1357697	559255	85800	0.66	1.19	2.17	57187	44607	43231	1.00	1.34	1.38

Table 4.4: The D^2FA size and D^2FA average $\overline{\psi}$ deferment depth for ORIGINAL and D^2FAMERGE on our eight primary RE sets given maximum deferment depth bounds of 1, 2 and 4.

Table 4.5 summarizes the results by RE group highlighting how much better or worse D^2FAMERGE does than ORIGINAL on the two metrics of number of transitions and average deferment depth $\overline{\psi}$.

RE set group	$\Omega = 1$		$\Omega = 2$		$\Omega = 4$	
	Trans decr.	Avg. def. depth ratio	Trans decr.	Avg. def. depth ratio	Trans decr.	Avg. dptr len ratio
All	91.3%	0.7	79.4%	0.9	42.5%	1.5
STRING	90.0%	0.6	92.5%	0.6	75.5%	0.9
WILDCARD	89.3%	0.8	59.0%	1.1	0.0%	2.0
SNORT	94.0%	0.7	91.0%	0.8	63.0%	1.4

Table 4.5: Average values of transition decrease and average deferment depth ratios for D^2FAMERGE compared with ORIGINAL for our RE set groups given maximum deferment depth bounds of 1, 2 and 4.

Overall, D^2FAMERGE performs very well when presented a bound Ω. In particular, the average increase in the number of transitions for D^2FAMERGE with Ω equal to 1, 2 and 4, is only 131%, 20% and 1% respectively, compared to D^2FAMERGE with unbounded maximum deferment depth. Stated another way, when D^2FAMERGE is required to have a maximum deferment depth of 1, this only results in slightly more than twice the number

of transitions in the resulting D^{2}FA. The corresponding values for ORIGINAL are 3121%, 1216% and 197%.

These results can be partially explained by examining the average deferment depth data. Unlike in the unbounded maximum deferment depth scenario, here we see that D^{2}FAMERGE has a larger average deferment depth $\overline{\psi}$ than ORIGINAL except for the WILDCARD when Ω is 1 or 2. What this means is that D^{2}FAMERGE has more states that defer to at least one other state than ORIGINAL does. This leads to the lower number of transitions in the final D^{2}FA. Overall, for $\Omega = 1$, D^{2}FAMERGE produces D^{2}FA with roughly 91% fewer transitions than ORIGINAL for all RE set groups. For $\Omega = 2$, D^{2}FAMERGE produces D^{2}FA with roughly 59% fewer transitions than ORIGINAL for the WILDCARD RE sets and roughly 92% fewer transitions than ORIGINAL for the other RE sets.

4.5.5 D^{2}FAMERGE versus BACKPTR

RE set	BACKPTR							D^{2}FAMERGE with back-pointer				
	# Trans	Def. depth		RAM (MB)	Time (s)	RAM2 (MB)	Time2 (s)	# Trans	Def. depth		RAM (MB)	Time (s)
		Avg.	Max.						Avg.	Max.		
Bro217	11247	2.61	6	6.38	88.08	0.05	273.95	13567	2.33	6	0.13	6.24
C613	26222	2.50	5	11.04	55.91	0.13	971.45	33777	2.30	5	0.25	10.78
C7	217812	5.94	13	24.17	277.80	1.04	1950.00	219684	1.15	4	1.12	4.51
C8	34636	2.44	8	3.04	12.61	0.17	27.76	35476	1.20	4	0.19	0.69
C10	157139	2.13	7	14.52	96.86	0.75	476.54	158232	1.21	4	0.80	11.94
Snort24	46005	8.74	17	13.56	70.95	0.22	1130.00	58273	1.62	8	0.41	47.77
Snort31	82809	2.87	8	19.60	109.56	0.39	1110.00	124584	1.74	6	1.29	3.61
Snort34	46046	7.05	14	13.50	94.19	0.22	983.98	51557	1.42	5	0.30	6.06

Table 4.6: The D^{2}FA size, D^{2}FA average $\overline{\psi}$ and maximum Ψ deferment depths, space estimate and time required to build the D^{2}FA for both variants of BACKPTR and D^{2}FAMERGE with the back-pointer property.

We now compare D^{2}FAMERGE with BACKPTR which enforces the back-pointer property described in Section 4.4.3. We adapt D^{2}FAMERGE to also enforce this back-pointer property. The results for all our metrics are shown in Table 4.6 for our 8 primary RE sets. We consider the two variants of BACKPTR described in Section 4.5.1.3, one which constructs the minimum state DFA corresponding to the given NFA and one which bypasses the minimum state DFA and goes directly to the D^{2}FA from the given NFA. We note the second variant appears to use less space than D^{2}FAMERGE. This is partially true since BACKPTR creates a smaller D^{2}FA than D^{2}FAMERGE. However, we underestimate the actual space used by this BACKPTR variant by simply assuming its required space is the final D^{2}FA size. We ignore, for instance, the space required to store intermediate complete tables or to perform the NFA to DFA subset construction. Table 4.7 summarizes these results by RE group displaying ratios for many of our metrics that highlight how much better or worse D^{2}FAMERGE does than BACKPTR.

RE set group	Trans increase	Def. depth ratio		Space ratio	Time ratio	Space2 ratio	Time2 ratio
		Avg.	Max.				
All	17.9%	2.9	1.9	30.4	19.3	0.7	142.5
STRING	25.0%	1.1	1.0	47.3	9.7	0.5	67.0
WILDCARD	1.3%	3.0	2.3	18.5	29.3	0.9	170.8
SNORT	29.7%	4.0	2.1	31.1	15.8	0.5	164.5

Table 4.7: Average values of transition increase, deferment depth ratios, space ratios, and time ratios for D^{2}FAMERGE compared with both variants of BACKPTR for RE set groups.

Similar to D^{2}FAMERGE versus ORIGINAL, we find that D^{2}FAMERGE with the back-pointer property performs well when compared with both variants of BACKPTR. Specifically, with an average increase in the number of transitions of roughly 18%, D^{2}FAMERGE

runs on average 19 times faster than the fast variant of BACKPTR and 143 times faster than the slow variant of BACKPTR. For space, D^{2}FAMERGE uses on average almost 30 times less space than the first variant of BACKPTR and on average roughly 42% more space than the second variant of BACKPTR. Furthermore, D^{2}FAMERGE creates D^{2}FA with average deferment depth 2.9 times smaller than BACKPTR and maximum deferment depth 1.9 times smaller than BACKPTR. As was the case with ORIGINAL, D^{2}FAMERGE achieves its best performance relative to BACKPTR on the WILDCARD RE sets and its worst performance relative to BACKPTR on the STRING RE sets. This is desirable as the space and time efficient D^{2}FAMERGE is most needed on RE sets like those in the WILDCARD because those RE sets experience the greatest state explosion.

4.5.6 Scalability results

Finally, we assess the improved scalability of D^{2}FAMERGE relative to ORIGINAL using the Scale RE set assuming that we have a maximum memory size of 1GB. For both ORIGINAL and D^{2}FAMERGE, we add one RE at a time from Scale until the space estimate to build the D^{2}FA goes over the 1GB limit. For ORIGINAL, we are able only able to add 12 REs; the final D^{2}FA has $397,312$ states and requires over 71 hours to compute. As explained earlier, we include the SRG edges in the RAM size estimate. If we exclude the SRG edges and only include the DFA size in the RAM size estimate, we would only be able to add one more RE before we reach the 1GB limit. For D^{2}FAMERGE, we are able to add 19 REs; the final D^{2}FA has $80,216,064$ states and requires only 77 minutes to

compute. This data set highlights the quadratic versus linear running time of ORIGINAL and D^{2}FAMERGE, respectively. Figure 4.5 shows how the space and time requirements grow for ORIGINAL and D^{2}FAMERGE as REs from Scale are added one by one until 19 have been added.

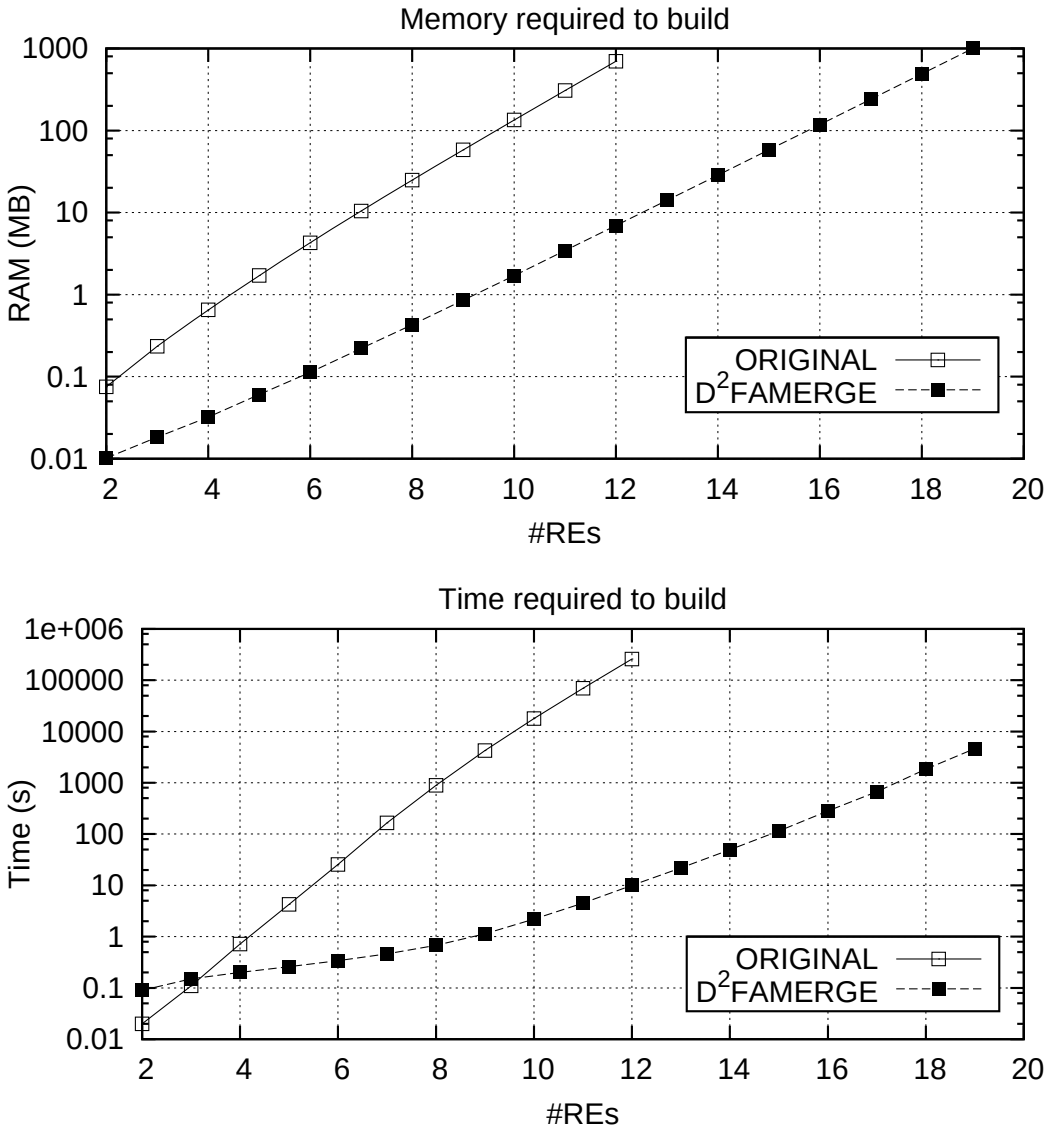


Figure 4.5: Memory and time required to build D^{2}FA versus number of Scale REs used for ORIGINAL's D^{2}FA and D^{2}FAMERGE's D^{2}FA.

Chapter 5

TCAM Implementation

In this section we present our work on the hardware implementation of RE matching using TCAM, which we call *RegCAM*.

5.1 Introduction/Motivation

Previous hardware solutions of RE matching have be based on FPGA. Although FPGA-based solutions can be modified, resynthesizing and updating FPGA circuitry in a deployed system to handle RE updates is slow and difficult. This makes FPGA-based solutions difficult to be deployed in many networking devices (such as NIDS/NIPS and firewalls) where the RE need to be updated frequently.

We propose the first TCAM based RE matching solution. TCAMs are prevalent in networking devices because TCAM-based packet classification is the de facto industry stan-

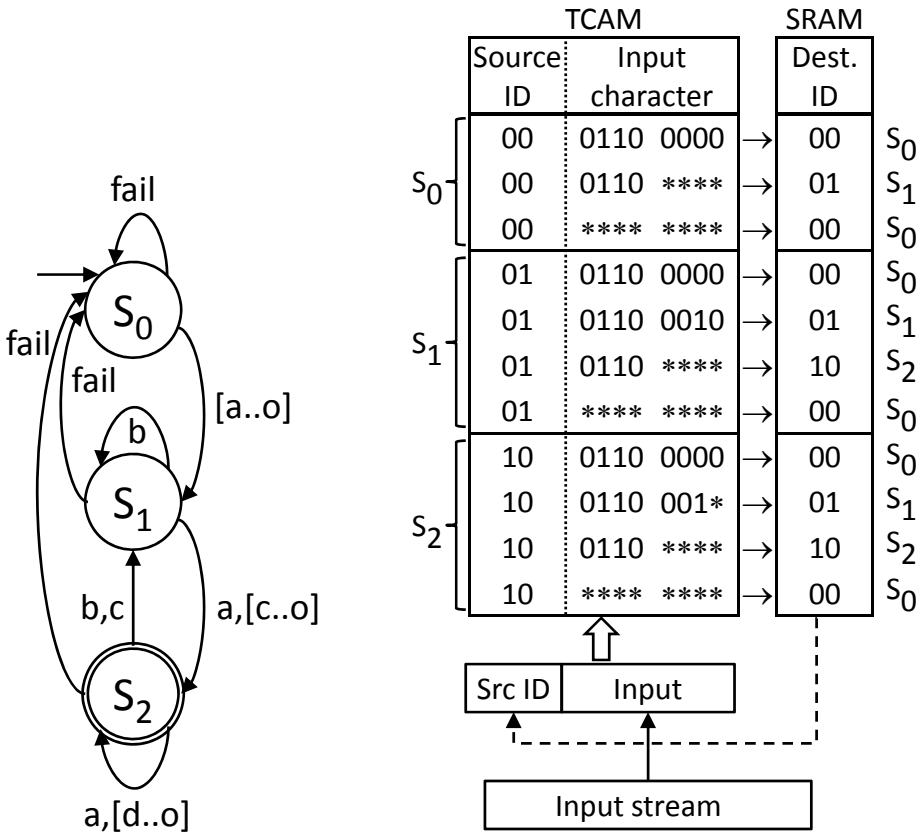
dard for high-speed packet classification, *i.e.*, header-based filtering. We show that TCAMs are also very effective for high-speed DPI, *i.e.*, payload-based filtering.

5.1.1 TCAM Architecture for RE matching

We first explain the straightforward implementation of RE matching using TCAM without any compression.

Given a RE set, we first construct an equivalent minimum state DFA. Second, we build a two column TCAM lookup table where each column encodes one of the two inputs to δ: the *source state ID* and the *input character*. Third, for each TCAM entry, we store the *destination* state ID in the same entry of the associated SRAM. Figure 5.1 shows an example DFA, its TCAM lookup table, and its SRAM decision table. We illustrate how this DFA processes the input stream "01101111, 01100011". We form a TCAM lookup key by appending the current input character to the current source state ID; in this example, we append the first input character "01101111" to "00", the ID of the initial state s_0, to form "0001101111". The first matching entry is the second TCAM entry, so "01", the destination state ID stored in the second SRAM entry is returned. We form the next TCAM lookup key "0101100011" by appending the second input character "011000011" to this returned state ID "01", and the process repeats.

Directly encoding a DFA in a TCAM using one TCAM entry per transition is infeasible. For example, consider a DFA with $25,000$ states that consumes one 8 bit character per transition. Each state has 2^8 transitions, and each transition needs 8 bits for the character



(a) Example DFA. (b) Corresponding TCAM table.

Figure 5.1: A DFA with its TCAM table.

and $\lceil \log 25000 \rceil$ bits for the source state ID. Thus, we would need a total of 140.38 Mb ($= 25000 \times 2^8 \times (8 + \lceil \log 25000 \rceil)$). This is infeasible given the largest available TCAM chip has a capacity of only 72 Mb. To address this challenge, we use two techniques that minimize the TCAM space for storing a DFA: *transition sharing* and *table consolidation*.

5.1.2 Reducing TCAM size

Recall that the two causes of DFA space explosion are transitions sharing and state replication (Section 3.2). We propose two techniques to reduce the size of TCAM required

to implement a DFA: *Transitions Sharing* that exploits transitions sharing and *Table Consolidation* that exploits state replication. The basic idea is to combine multiple transitions into one such that we use the ternary nature and first-match semantics of TCAMs to encode multiple DFA transitions using one TCAM entry.

5.1.2.1 Transitions Sharing

The two reasons for transition sharing are character redundancy and state redundancy.

Character redundancy: Prior work exploits character redundancy mainly by alphabet encoding, where the alphabet Σ is mapped to a smaller alphabet Σ'. Alphabet encoding cannot fully leverage all the compression opportunities presented by character redundancy, as it can only exploit global character redundancy that is common to all states in the DFA. Specifically, alphabet encoding can map two characters σ_1 and σ_2 in Σ to the same character σ' in Σ' if and only if $\forall q \in Q$, $\delta(q, \sigma_1) = \delta(q, \sigma_2)$.

To exploit character redundancy at each state, we propose the technique of *character bundling*. In character bundling, we leverage the ternary nature and first-match semantics of TCAMs on the input character field to represent multiple characters and thus multiple transitions that share the same source and destination states.

State redundancy: Prior work exploits state redundancy mainly by *deferred transitions*, where one state p might defer most of the transitions for another state q. Existing

deferred transition based solutions cannot fully exploit state redundancy because of the speed penalty, *i.e.*, traversal of a deferred transition leads to no input being processed. Thus, to alleviate this speed penalty, such solutions often choose deferred transitions that do not fully compress the transition table.

To exploit state redundancy, we propose the technique of *shadow encoding*. In shadow encoding, we leverage the ternary nature and first-match semantics of TCAMs on the current state ID field to encode many incoming transitions of a state from different states using only one TCAM entry.

5.1.2.2 Table Consolidation

We get state explosion in a DFA because each NFA state is replicated multiple times in the DFA. Table Consolidation exploits state replication in a DFA based on the following observation: two DFA states that are replications of the same NFA state, will usually have transitions remaining in the D^2FA (*i.e.* non-deferred transitions) on the same set of input characters (although the corresponding transitions in the two states might go to different states.) In this case, the TCAM tables for the two states will be exactly the same except for the state IDs. If the corresponding transition go to different next state then the SRAM tables for the two states will be different.

The idea is that we can merge the TCAM tables for the two states into one TCAM table, and store both the SRAM tables side by side. This results in reduction in the TCAM size, at the cost of possibly increasing SRAM size, which is fine since TCAM size is much more

critical than SRAM size.

5.1.3 Increasing Matching Throughput

Another challenge that we address is improving RE matching speed and thus throughput. One way to improve the throughput by up to a factor of k is to use k-stride DFAs that consume k input characters per transition. However, this leads to an exponential increase in both state and transition spaces. For example, a k-stride DFA requires 2^{8*k} transitions per state, so the transition space grows exponentially in k. Previous multi-stride DFAs suffer from a significant increase in the number of states and the number of transitions such that only 2-stride DFAs are achieved in practice [9, 13].

To avoid this space explosion, we use the novel idea of *variable striding*. The basic idea is to use transitions with variable strides, *i.e.* different transitions can consume different numbers of input characters. This allows us to increase the average number of characters consumed per transition while ensuring all the transitions fit within the allocated TCAM space. This idea is based on two key observations. First, for many states, we can capture many but not all k-stride transitions using relatively few TCAM entries whereas capturing all k-stride transitions requires prohibitively many TCAM entries. Second, with TCAMs, we can easily store transitions with different strides in the same TCAM lookup table. Variable striding would be very difficult to implement without TCAMs and thus it is not surprising variable striding has not been considered before.

5.1.4 Comparison of Transition Sharing with D^{2}FA

The observation behind the transition sharing technique, namely many states share a large number of outgoing transactions, is similar to that of deferred transition in a D^{2}FA. We use a D^{2}FA as the starting point for transition sharing, and it can be viewed as a way of implementing a D^{2}FA in TCAM.

But there are several differences between transition sharing and D^{2}FA:

(1) The transitions stored at each state is given by the D^{2}FA. But our character bundling technique achieves further compression, and so the total number of TCAM rules is significantly less than the number of transitions in the D^{2}FA.

(2) D^{2}FA suffers from speed penalty, as no input is consumed when a deferred transition is taken. The number of lookups needed in the worst case is given the the deferment depth of the current state. Because of or shadow encoding technique, there is no speed penalty in transition sharing. Only one TCAM lookup is needed for each character, irrespective of the deferment depth of the current state.

(3) Because of the speed penalty in the D^{2}FA, for practical implementation, the deferment depth of the D^{2}FA is bounded, which significantly increases the number of transitions in the D^{2}FA. For transition sharing, we build D^{2}FA without any limit on the deferment depth, achieving maximum transition compression.

We now explain each of our techniques in detail.

5.2 Transition Sharing

The basic idea of transition sharing is to combine multiple transitions into a single TCAM entry. We propose two transition sharing ideas: character bundling and shadow encoding. Character bundling exploits intra-state optimization opportunities and minimizes TCAM tables along the input character dimension. Shadow encoding exploits inter-state optimization opportunities and minimizes TCAM tables along the source state dimension.

5.2.1 Character Bundling

Character bundling exploits character redundancy by combining multiple transitions from the same source state to the same destination into one TCAM entry. Character bundling consists of four steps. (1) Assign each state a unique ID of $\lceil \log |Q| \rceil$ bits. (2) For each state, enumerate all 256 transition rules where for each rule, the predicate is a transition's label and the decision is the destination state ID. (3) For each state, treating the 256 rules as a 1-dimensional packet classifier and leveraging the ternary nature and first-match semantics of TCAMs, we minimize the number of transitions using the optimal 1-dimensional TCAM minimization algorithm (Section 3.4.2). (4) Concatenate the $|Q|$ 1-dimensional minimal prefix classifiers together by prepending each rule with its source state ID. The resulting list can be viewed as a 2-dimensional classifier where the two fields are source state ID and transition label and the decision is the destination state ID. Figure 5.1 shows an example DFA and its TCAM lookup table built using character bundling. The three chunks of

TCAM entries encode the 256 transitions for s_0, s_1, and s_2, respectively. Because each TCAM entry matches one or more input characters, we need only 11 total TCAM entries instead of the naïve implementation that requires $256 \times 3 = 768$ entries.

5.2.2 Shadow Encoding

Whereas character bundling encodes multiple transitions with the same source and destination states using one TCAM entry, shadow encoding encodes multiple transitions with the same character label and destination state ID using one TCAM entry. This technique is based upon the observation of state redundancy. More specifically, character bundling uses ternary codes in the input character field to encode multiple input characters, and shadow encoding uses ternary codes in the source state ID field to encode multiple source states.

5.2.2.1 Observations

We use our running example in Figure 5.1 to illustrate shadow encoding. We observe that all transitions with source states s_1 and s_2 have the same destination state except for the transitions on character c. Likewise, source state s_0 differs from source states s_1 and s_2 only in the character range [a..o]. This implies there is a lot of state redundancy. The table in Figure 5.2 shows how we can exploit state redundancy to further reduce required TCAM space. First, since states s_1 and s_2 are more similar, we give them the state IDs 00 and 01, respectively. State s_2 uses the ternary code of $0*$ in the state ID field of its TCAM

entries to share transitions with state s_1. We give state s_0 the state ID of 10, and it uses the ternary code of $**$ in the state ID field of its TCAM entries to share transitions with both states s_1 and s_2. Second, we order the state tables in the TCAM so that state s_1 is first, state s_2 is second, and state s_0 is last. This facilitates the sharing of transitions among different states where earlier states have incomplete tables deferring some transitions to later tables. Specifically, s_1 has an incomplete table with only a single TCAM entry to specify the transitions it does not share with s_2, and s_2 has an incomplete table with only 3 TCAM entries to specify the transitions it (and s_1) does not share with s_0.

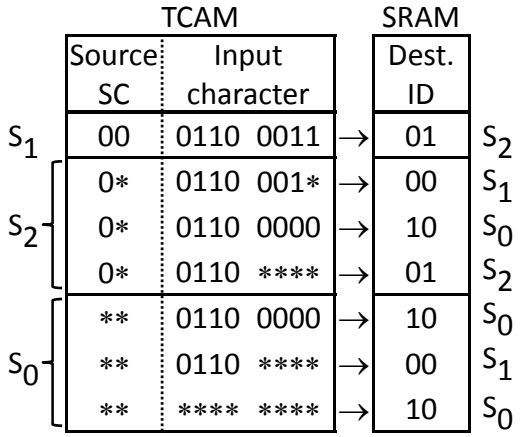
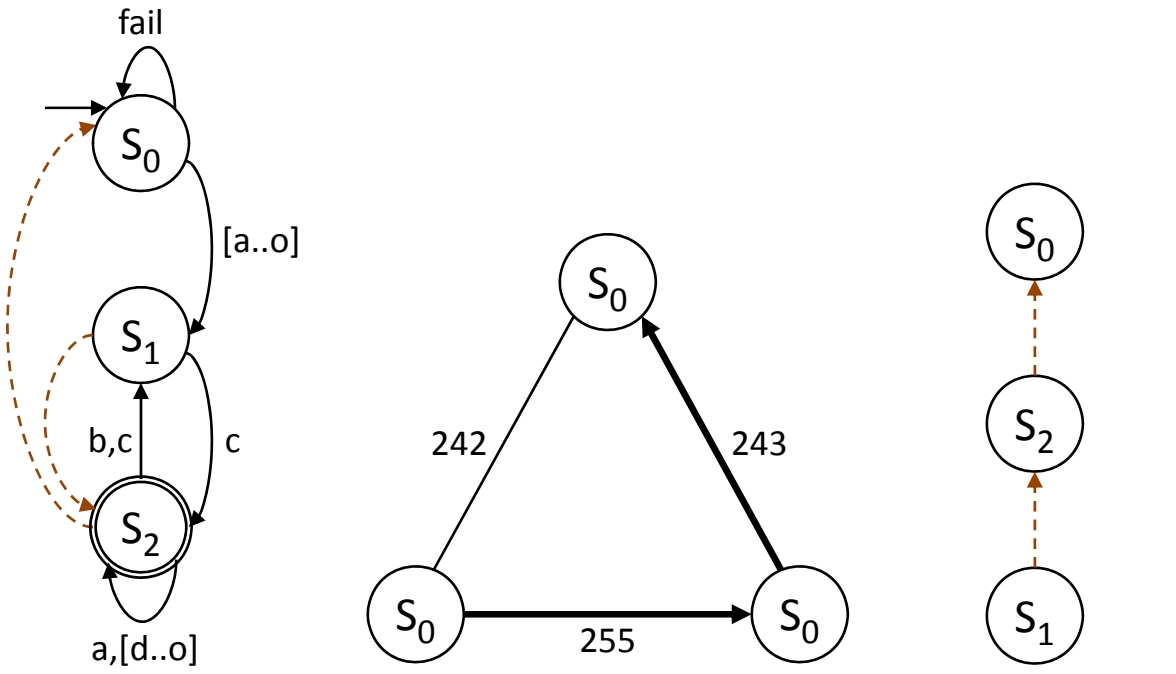


Figure 5.2: TCAM table with shadow encoding.

Implementing shadow encoding requires solving three key problems: (1) Find the best order of the state tables in the TCAM (any order is allowed). (2) Choose binary IDs and ternary codes for each state given the state table order. (3) Identify entries to remove from each state table.

5.2.2.2 Determining Table Order

We first describe how we compute the order of tables within the TCAM. In order to exploit inter-state transition sharing, we first build a D^2FA for the given RE set. If $p \twoheadrightarrow q$ (*i.e.* state p is a descendant of state q), we say that state p is in state q's *shadow*. We use the partial order of the deferment forest of the D^2FA to determine the order of state transition tables in the TCAM. Specifically, state q's transition table must be placed after the transition tables of all states in state q's shadow. That is, the state order in given by a depth first traversal of the deferment forest.



(a) D^2FA (b) SRG (c) Deferment tree

Figure 5.3: D^2FA, SRG, and deferment tree of the DFA in Figure 5.1.

Figure 5.3 shows the D^2FA, SRG, and the deferment tree, respectively, for the DFA in Figure 5.1.

5.2.2.3 Shadow Encoding Algorithm

We now describe our shadow encoding algorithm which takes as input a deferment forest F, and outputs the state IDs. We also use the term nodes to refer to states in the description of the algorithm. To ensure that proper sharing of transitions occurs, we need to compute a *shadow encoding* for the given deferment forest. In a valid shadow encoding, each state q is assigned a binary *state ID* (ID(q)) and a ternary *shadow code* (SC(q)). Binary state IDs are used in the destination state ID field (in the SRAM) of transition rules. Ternary shadow codes are used in the source state ID field (in the TCAM) of transition rules. The *shadow length* of a shadow encoding is the common length of every state ID and shadow code.

A valid shadow encoding for a given deferment forest F must satisfy the following four *Shadow Encoding Properties (SEP)*:

1. *Uniqueness Property*: For any two distinct states p and q, $ID(p) \neq ID(q)$ and $SC(p) \neq SC(q)$.

2. *Self-Matching Property*: For any state p, $ID(p) \in SC(p)$ (*i.e.*, ID(p) matches SC(p)).

3. *Deferment Property*: For any two states p and q, $p \twoheadrightarrow q$ (*i.e.*, q is an ancestor of p in the given deferment forest) if and only if $SC(p) \subset SC(q)$.

4. *Non-interception Property*: For any two distinct states p and q, $p \twoheadrightarrow q$ if and only if $ID(p) \in SC(q)$.

Lemma 3. *Given a valid shadow encoding for deferment forest* F*, for any state* q *and all states* p *in* q*'s shadow,* $ID(p) \in SC(q)$.

Proof. The deferment property implies that $SC(p) \subset SC(q)$. The self-matching property implies that $ID(p) \in SC(p)$. Thus, the result follows. □

Lemma 4. *Given a valid shadow encoding for deferment forest* F*, for any state* q *and all states* p *not in* q*'s shadow,* $ID(p) \notin SC(q)$.

Proof. This follows immediately from the non-interception property. □

Intuitively, q's shadow code must match the state ID of all states in q's shadow and cannot match the state ID of any states not in q's shadow.

Theorem 4. *Given a valid shadow encoding for a DFA* M *and deferment forest* F *and a TCAM classifier* $\mathbb{C}$ *that uses only binary state IDs for both source and destination state IDs in transition rules and that orders the state tables according to* F*, the TCAM classifier formed by replacing each source state ID in* $\mathbb{C}$ *with the corresponding shadow code and each destination state ID in* $\mathbb{C}$ *with the corresponding state ID will be equivalent to* $\mathbb{C}$.

Proof. This follows from the first match nature of TCAMs, the state tables are ordered according to F, and Lemmas 3 and 4. □

We give a shadow encoding algorithm where the deferment forest is a single deferment tree DT. We handle deferment forests by simply creating a virtual root node whose children

are the roots of the deferment trees in the forest and then running the algorithm on this tree.

Our algorithm uses the following internal variables for each node v: a local binary ID denoted $L(v)$, a global binary ID denoted $G(v)$, and an integer weight denoted $W(v)$ that is the shadow length we would use for the subtree of DT rooted at v. Intuitively, the state ID of v will be $G(v)|L(v)$ where $|$ denotes concatenation, and the shadow code of v will be the prefix string $G(v)$ followed by the required number of $*$'s; some extra padding characters may be needed. We use $\#L(v)$ and $\#G(v)$to denote the number of bits in $L(v)$ and $G(v)$, respectively.

Our algorithm works as follows. For all v, we initially set $L(v) = G(v) = \emptyset$ and $W(v) = 0$. Our algorithm works recursively in a bottom-up fashion. We mark nodes red when they have been processed. We begin by marking each leaf node of DT as processed. We process an internal node v when all its children $v_1, \cdots, v_n$ are marked red. Once a node v is processed, its weight $W(v)$ and its local ID $L(v)$ are fixed, but we will prepend additional bits to its global ID $G(v)$ when we process its ancestors in DT.

While precessing v, we assign v and each of its n children a variable-length binary code *HCode* that is prefix free (*i.e.* no HCode is a prefix of another HCode.) One option is to assign each of the $(n+1)$ nodes a binary number from 0 to n using $\log_2(n+1)$ bits. To minimize the shadow length $W(v)$, we use a Huffman coding style algorithm to compute the HCodes and $W(v)$. This algorithm uses two data structures: a binary encoding tree T with $n+1$ leaf nodes, one for v and each of its children, and a min-priority queue PQ,

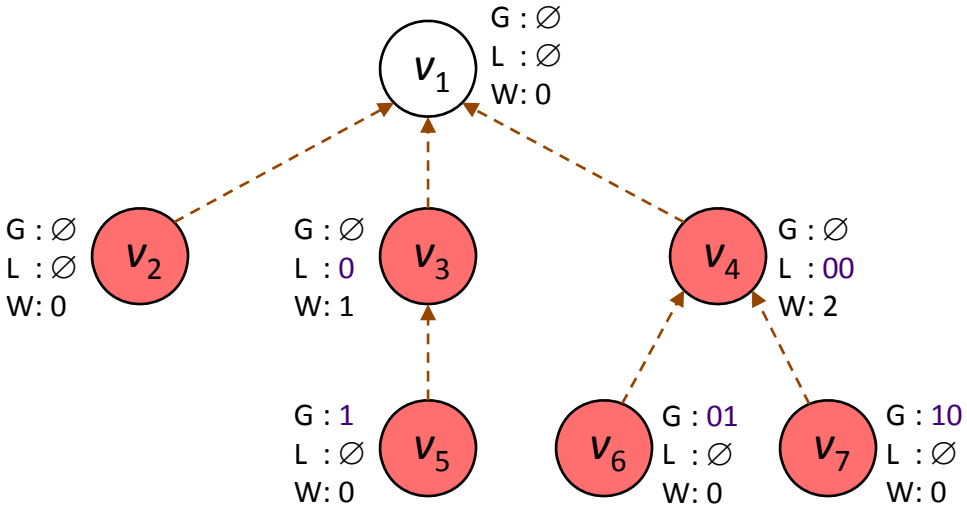
initialized with $n+1$ elements (one for v and each of its children) that is ordered by node weight. While PQ has more than one element, we remove the two elements x and y with lowest weight from PQ, create a new internal node z in T with two children x and y, and set weight(z)=maximum(weight(x), weight(y))+1, and then put element z into PQ. When PQ has only one element, T is complete. The HCode assigned to each leaf node v' is the path in T from the root node to v' where left edges have value 0 and right edges have value 1.

We update the internal variables of v and its descendants in DT as follows. We set $L(v)$ to be its HCode, and $W(v)$ to be the weight of the root node of T; $G(v)$ is left empty. For each child v_i, we prepend v_i's HCode to the global ID of every node in the subtree rooted at v_i including v_i itself. We then mark v as red. This continues until all nodes in DT are red.

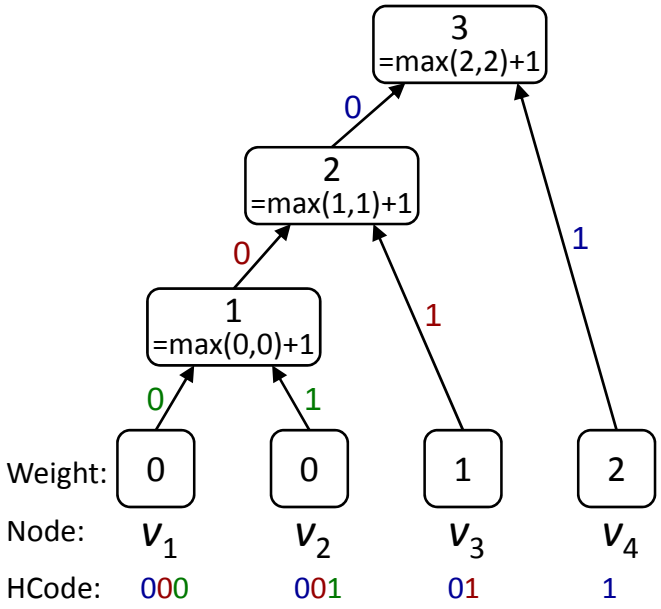
We now set state IDs and a shadow codes. The shadow length is k, the weight of the root node of DT. We use $\{*\}^m$ to denote a ternary string with m $*$'s and $\{0\}^m$ to denote a binary string with m 0's. For each node v, we compute v's state ID and shadow code as follows:

$$ID(v) = G(v)|L(v)|\{0\}^{k-\#G(v)-\#L(v)}, \quad SC(v) = G(v)|\{*\}^{k-\#G(v)}.$$

We illustrate our shadow encoding algorithm in Figure 5.4. Figure 5.4(a) shows all the internal variables just before v_1 is processed. Figure 5.4(b) shows the Huffman style binary encoding tree T built for node v_1 and its children v_2, v_3, and v_4 and the resulting HCodes.

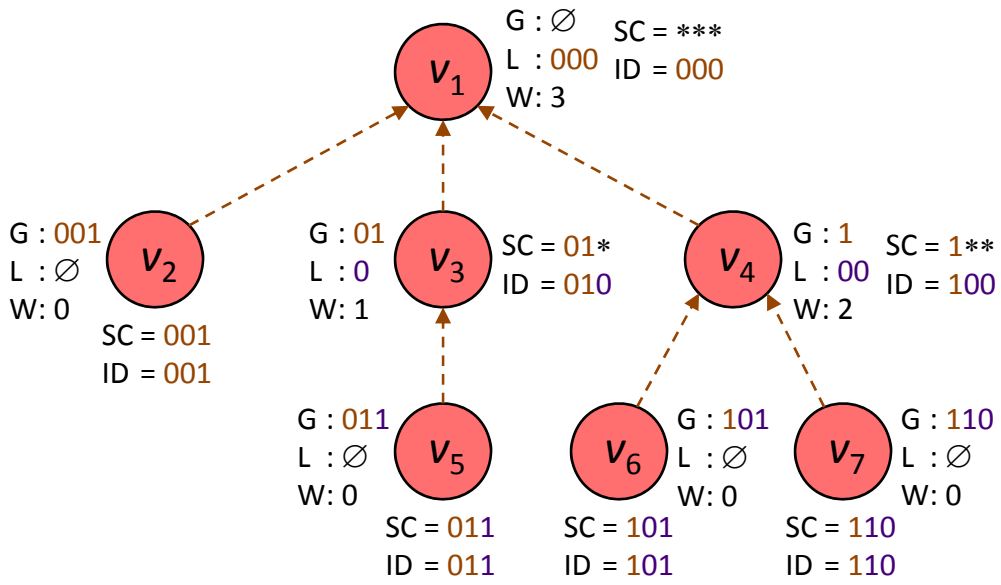


(a) Deferment tree with internal variables before processing v_1.



(b) Build Hufman tree and assigned HCodes while processing v_1.

Figure 5.4: Shadow encoding example.



(c) Internal variables before processing v_1 and assigned state IDs and shadow codes.

Figure 5.4: Shadow encoding example (cont'd).

Figure 5.4(c) shows each node's final weight, global ID, local ID, state ID and shadow code. The pseudo-code for the Shadow Encoding algorithm is given in figure Algorithm 5.5.

We now prove two properties of our shadow encoding algorithm using induction on the height n of the deferment tree T. In both proofs, in the inductive case, we let s denote the root node of T, s_1 through s_c denote the c children of s, and T_i for $1 \leq i \leq c$ denote the subtree rooted at s_i.

Theorem 5. *The state IDs and shadow codes generated by our Shadow Encoding algorithm satisfy the SEP.*

Proof. We prove by induction on the height n of T. The base case where $n = 0$ is trivial since there is only a single node. For the inductive case, our inductive hypothesis is that the shadow codes and state IDs generated for T_i for $1 \leq i \leq c$ satisfy the SEP. Note, we do

Input: Deferment forest, DF, with n states, $s_1, \ldots, s_n$.
Output: ID[1..n] and SC[1..n] for each state.

Add state s_0 to DF with all the tree roots as its children;
Set all ID[1..n] and SC[1..n] to the empty string;
ShadowEncode (s_0);
return ID[1..n] and SC[1..n];

Function ShadowEncode(s)
    // Base case
    if s *has no children* **then return** 0;
    // Recursive case
    $r \leftarrow$ Number of children of s;
    CHILD[1..r] $\leftarrow$ List of children of s;
    for $i = 1$ **to** r **do**
        W[i] $\leftarrow$ ShadowEncode(CHILD[i]);
    W[0] $\leftarrow 0$;
    G $\leftarrow$ HCode(W);
    $l \leftarrow \max_{0 \le i \le r}(|G[i]| + W[i])$;
    for $i = 1$ **to** r **do**
        Append 0's at end of G[i] to make $|G(i)| + W(i) = l$;
        Attach G[i] in front of ID and SC for each state in the subtree of CHILD[i];
    ID(s) $\leftarrow (0)^l$;
    SC(s) $\leftarrow (*)^l$;
    return l;

Function HCode(W[0..r])
    Initialize Q as a min priority queue of binary tree nodes;
    for $i = 0$ **to** r **do**
        Insert leaf node n_i in Q with value $V[n_i] \leftarrow W[i]$;
    while $|Q| > 1$ **do**
        $n_l \leftarrow$ pop(Q);
        $n_r \leftarrow$ pop(Q);
        Insert node n in Q with n_l and n_r as left and right children, and value $V[n] \leftarrow \max(V[n_l], V[n_r]) + 1$;
    $n \leftarrow$ pop(Q);
    Generate the codes based on the Huffman Tree rooted at n;
    return the codes assigned to the leaf nodes;

Figure 5.5: Shadow Encoding Algorithm.

not process the root node s in this assumption. We now consider what happens when we process s. For each node $v \in T_i$ for $1 \leq i \leq c$, $\text{HCode}(s_i)$ is prepended to the $\text{SC}(v)$ and $\text{ID}(v)$. Thus, the SEP still holds for all the nodes within T_i for $1 \leq i \leq c$. For any nodes p and q from different subtrees T_i and T_j, it follows that $\text{ID}(p) \notin \text{SC}(q)$ and $\text{ID}(q) \notin \text{SC}(p)$ because $\text{HCode}(s_i)$ and $\text{HCode}(s_j)$ are not prefixes of each other. Finally, for all nodes $v \in T$, $\text{ID}(v) \in \text{SC}(s)$ because $\text{SC}(s)$ contains only $*$'s. □

We define a prefix shadow encoding as a shadow encoding where all shadow codes are prefix strings; that is, all $*$'s are after any 0's or 1's. For any prefix shadow encoding $\mathcal{E}$ of T, $\mathcal{E}_{T_i}$ denotes the subset of state ids and shadow codes for all $v \in T_i$. For any state id or shadow code X, ${}_p\lfloor X$ denotes the first p characters of X, and $X\rfloor_p$ denotes the last p characters of X. We define $\mathcal{E}_{T_i}\rfloor_p = \{X\rfloor_p \mid X \in \mathcal{E}_{T_i}\}$.

Lemma 5. *Consider a deferment tree T with a valid length x prefix shadow encoding $\mathcal{E}$ that satisfies the SEP. For every child $s_i, 1 \leq i \leq c$, of the root of T, there exist two values p_i and q_i such that:*

1. $\forall i,\ 0 < p_i \leq x,\ 0 \leq q_i < x$ *and* $p_i + q_i = x$.
2. $\forall i,\ \forall v \in T_i,\ {}_{p_i}\lfloor \text{ID}(v) = {}_{p_i}\lfloor \text{SC}(v) = {}_{p_i}\lfloor \text{SC}(s_i)$.
3. $\forall i,\ \mathcal{E}_{T_i}\rfloor_{q_i}$ *is a valid prefix shadow encoding of* T_i.
4. *The set* $\mathcal{E}_{\text{ID}} = \{{}_{p_i}\lfloor \text{SC}(s_i) \mid 1 \leq i \leq c\}$ *is prefix free.*

Proof. Since $\mathcal{E}$ is a prefix shadow encoding, for any child s_i, $\text{SC}(s_i)$ must be of the form

$\{0,1\}^a\{*\}^{x-a}$. Let $p_i = a$ and $q_i = x - a$. Now, $p_i > 0$, otherwise we would have $SC(s_i) = \{*\}^x$, which is not possible as it would violate the deferment and non-interception properties. This proves (1). Also, since $\mathcal{E}$ satisfies the deferment and self-matching properties, we must have (2) and (3). And we must have (4) because of the non-interception property. □

Our shadow encoding algorithm produces minimum length encodings.

Theorem 6. *For any deferment tree* T*, our shadow encoding algorithm generates the shortest possible prefix shadow encoding that satisfies the SEP.*

Proof. First, our shadow encoding algorithm generates a prefix shadow encoding. We prove by induction on the height n of T that it is the shortest possible prefix shadow encoding. The base case where $n = 0$ is trivial since the encoding for a single node is empty and thus optimal. For the inductive case, our inductive hypothesis is that the prefix shadow encoding for T_i for $1 \leq i \leq c$ is the shortest possible.

Let $\mathcal{E}$ be the prefix shadow encoding generated by our shadow encoding algorithm and $\mathcal{F}$ be the optimal prefix shadow encoding. Let l and m be the lengths of $\mathcal{E}$ and $\mathcal{F}$ respectively. Let g_i and w_i be the values defined by Lemma 5 for $\mathcal{E}$. And let p_i and q_i be the corresponding values for $\mathcal{F}$. By the inductive hypothesis, we have $w_i \leq q_i$ for $1 \leq i \leq c$.

If $m < l$, this implies that the optimal shortest prefix shadow encoding for T must compute a better set of HCode equivalents for each child node s_i. In particular, we have that

$\max_i(p_i + q_i) < \max_i(g_i + w_i)$. That is, given equal or larger initial lengths, $\{q_i\}$, optimal prefix shadow encoding computes prefix-free codes $\mathcal{F}_{ID}$ for the children that are shorter than the prefix-free codes $\mathcal{E}_{ID}$ computed by the HCode subroutine. However, this is a contradiction, since the Huffman style encoding used to compute the HCodes minimizes the term $\max_i(g_i + w_i)$ [21]. Therefore, we must have $l \leq m$. □

Experimentally, we found that our shadow encoding algorithm is effective at minimizing shadow length. No DFA had a shadow length larger than $\lceil \log_2 |Q| \rceil + 3$ where $\lceil \log_2 |Q| \rceil$ is the shortest possible shadow length.

5.2.2.4 Choosing Transitions

Section 5.2.1 describes how the TCAM tables are generated for states with all 256 transitions (*i.e.* for root states) using 1-dimensional complete classifier minimization. But non-root states do not have complete tables. We now describe how we apply the character bundling technique to generate the TCAM tables for non-root states.

For a given DFA and a corresponding deferment forest, we construct a D^2FA by choosing which transitions to encode in each transition table as follows. If state p has a default transition to state q, we identify p's deferrable transitions which are the transitions that are common to both p's transition table and q's transition table. These deferrable transitions are optional for p's transition table; that is, they can be removed to create an incomplete transition table or included if that results in fewer TCAM entries. Figure 5.2 is an example where including a deferrable transition produces a smaller classifier. The second entry in

s_2's table in Figure 5.2 can be deferred to state s_0's transition table. However, this results in a classifier with at least 4 TCAM entries whereas specifying the transition allows a classifier with just 3 TCAM entries. This leads us to the following problem for which we give an optimal solution.

Definition 4 (Partially Deferred Incomplete One-dimensional TCAM Minimization Problem). *Given a one-dimensional packet classifier* f *on* $\{*\}^b$ *and a subset* $\mathcal{D} \subseteq \{*\}^b$*, find the minimum cost prefix classifier* f' *such that* $\text{Cover}(f') \supseteq \{*\}^b \setminus \mathcal{D}$ *and is equivalent to* f *over* $\text{Cover}(f')$.

Here b is the field width (in bits), $\mathcal{D}$ is the set of packets that can be deferred, and $\text{Cover}(c)$ is the union of the predicates of all the rules in c (*i.e.* all the packets matched by c). For simplicity of description, we assume that f has flattened rule set (*i.e.* one rule for each packet with the packet as the rule predicate). Assuming the packet is a one byte character, this implies f has 256 rules.

We provide a dynamic programming formulation for solving this problem that is similar to the dynamic programming formulation used in [31] and [47] to solve the related problem when all transitions must be specified. In these previous solutions for complete classifiers, for each prefix, the dynamic program maintains an optimal solution for each possible final decision. It then specifies how to combine these optimal solutions for matching prefixes into an optimal solution for the prefix that is the union of the two matching prefixes; in this step, two final rules for each prefix that have the same decision can be replaced by a single final rule for the combined prefix resulting in a savings of one TCAM entry.

The main change is to maintain an optimal solution for each prefix where we defer some transitions within the prefix.

We now formally specify this dynamic program introducing the following notation. Let $d_i, i \geq 1$ denote the actual decisions in a classifier. For a prefix $\mathcal{P} = \{0,1\}^k\{*\}^{b-k}$, we use $\underline{\mathcal{P}}$ to denote the prefix $\{0,1\}^k0\{*\}^{b-k-1}$, and $\overline{\mathcal{P}}$ to denote the matching prefix $\{0,1\}^k1\{*\}^{w-k-1}$. For a classifier f on $\{*\}^b$ and a prefix $\mathcal{P} \subseteq \{*\}^b$, $f_{\mathcal{P}}$ denotes a classifier on $\mathcal{P}$ that is equivalent to f (*i.e.* the subset of rules in f with predicates that are in $\mathcal{P}$). So $f = f_{\{*\}^b}$. For $i \geq 1$, $f_{\mathcal{P}}^{d_i}$ denotes a classifier on $\mathcal{P}$ that is equivalent to f and the decision of the last rule is d_i. Note that all packets in $\mathcal{P}$ are specified by such classifiers. Classifier $f_{\mathcal{P}}^{d_0}$ denotes the optimal classifier that is equivalent to f except that it possibly defers some packets within $\mathcal{D}$. We use $C(f_{\mathcal{P}}^{d_i})$ to denote the cost of the minimum classifier equivalent to $f_{\mathcal{P}}^{d_i}$ for $i \geq 0$. $[P(x)]$ evaluates to 1 when the statement inside is true; otherwise it evaluates to 0. We use x to represent some packet in the prefix $\mathcal{P}$ currently being considered.

Theorem 7. *Given a one-dimensional classifier f on $\{*\}^b$ and a subset $\mathcal{D} \subseteq \{*\}^b$ with a set of possible decisions $\{d_1, d_2, \ldots, d_z\}$ and a prefix $\mathcal{P} \subseteq \{*\}^b$, we have that $C(f_{\mathcal{P}}^{d_i})$ is calculated as follows:*

(1) For $i > 0$

$$C(f_{\mathcal{P}}^{d_i}) = \begin{cases} 1 + [f(x) \neq d_i] & \text{if } f \text{ is consistent on } \mathcal{P} \\ \min_{j=1}^{z}(C(f_{\underline{\mathcal{P}}}^{d_j}) + C(f_{\overline{\mathcal{P}}}^{d_j}) - 1 + [j \neq i]) & \text{else} \end{cases}$$

(2) For $i = 0$*:*

$$C(f_{\mathcal{P}}^{d_0}) = \begin{cases} 0 & \text{if } \mathcal{P} \subseteq \mathcal{D} \\ \min(\min_{i=1}^{z}(C(f_{\mathcal{P}}^{d_i})), C(f_{\underline{\mathcal{P}}}^{d_0}) + C(f_{\overline{\mathcal{P}}}^{d_0})) & \text{else} \end{cases}$$

Proof. (1) When $i > 0$, we just build a minimum cost complete classifier. The recursion and the proof is exactly the same as given in [31] Theorem 4.1 (with decision weights $= 1$).

(2) We consider two possibilities. Either the optimal classifier is a complete classifier or the optimal classifier is an incomplete classifier. If the optimal classifier is incomplete, we consider two cases. If the entire prefix $\mathcal{P}$ is contained with $\mathcal{D}$ and can be deferred, the minimum cost classifier is to defer all transitions and has cost 0. Otherwise, the minimum cost classifier for $\mathcal{P}$ would just be the minimum cost classifier for $\underline{\mathcal{P}}$ concatenated with the minimum cost classifier for $\overline{\mathcal{P}}$. This is represented by the last term in the minimization for case (2). The possibility that the optimal classifier is a complete classifier is handled by the first term in the first minimization for case (2). □

5.3 Table Consolidation

We now present *table consolidation* where we combine multiple transition tables for different states into a single transition table such that the combined table takes less TCAM space than the total TCAM space used by the original tables. To define table consolidation, we need two new concepts: k-decision rule and k-decision table. A k-*decision rule* is a rule

whose decision is an array of k decisions. A k-*decision table* is a sequence of k-decision rules following the first-match semantics. Given a k-decision table $\mathbb{T}$ and i ($0 \leq i < k$), if for any rule r in $\mathbb{T}$ we delete all the decisions except the i-th decision, we get a 1-decision table, which we denote as $\mathbb{T}[i]$. In table consolidation, we take a set of k 1-decision tables $\mathbb{T}_0, \cdots, \mathbb{T}_{k-1}$ and construct a k-decision table $\mathbb{T}$ such that for any i ($0 \leq i < k$), the condition $\mathbb{T}_i \equiv \mathbb{T}[i]$ holds where $\mathbb{T}_i \equiv \mathbb{T}[i]$ means that $\mathbb{T}_i$ and $\mathbb{T}[i]$ are equivalent (*i.e.*, they have the same decision for every search key). We call the process of computing k-decision table $\mathbb{T}$ *table consolidation*, and we call $\mathbb{T}$ the *consolidated table*.

5.3.1 Observations

Table consolidation is based on three observations. First, semantically different TCAM tables may share common entries with possibly different decisions. For example, the three tables for s_0, s_1 and s_2 in Figure 5.1 have three entries in common: 01100000, 0110****, and ********. Table consolidation provides a novel way to remove such information redundancy. Second, given any set of k 1-decision tables $\mathbb{T}_0, \cdots, \mathbb{T}_{k-1}$, we can always find a k-decision table $\mathbb{T}$ such that for any i ($0 \leq i < k$), the condition $\mathbb{T}_i \equiv \mathbb{T}[i]$ holds. This is easy to prove as we can use one entry per each possible binary search key in $\mathbb{T}$. Third, a TCAM chip typically has a build-in SRAM module that is commonly used to store lookup decisions. For a TCAM with n entries, the SRAM module is arranged as an array of n entries where SRAM[i] stores the decision of TCAM [i] for every i. A TCAM lookup returns the index of the first matching entry in the TCAM, which is then used as the

index to directly find the corresponding decision in the SRAM. In table consolidation, we essentially trade SRAM space for TCAM space because each SRAM entry needs to store multiple decisions. As SRAM is cheaper and more efficient than TCAM, moderately increasing SRAM usage to decrease TCAM usage is worthwhile.

Figure 5.6 shows the TCAM lookup table and the SRAM decision table for a 3-decision consolidated table for states s_0, s_1, and s_2 in Figure 5.1. In this example, by table consolidation, we reduce the number of TCAM entries from 11 to 5 for storing the transition tables for states s_0, s_1, and s_2. This consolidated table has an ID of 0. As both the table ID and column ID are needed to encode a state, we use the notation $< \text{Table ID} > @ < \text{Column ID} >$ to represent a state.

TCAM			SRAM		
Consolidated Src Table ID	Input Character		Column ID 00	01	10
0	0110 0000	$\rightarrow$	s_0	s_0	s_0
0	0110 0010	$\rightarrow$	s_1	s_1	s_1
0	0110 0011	$\rightarrow$	s_1	s_2	s_1
0	0110 ****	$\rightarrow$	s_1	s_2	s_2
0	**** ****	$\rightarrow$	s_0	s_0	s_0

Figure 5.6: 3-decision table for 3 states in Figure 5.1

We illustrate input character stream processing with table consolidation using this example 3-decision table. Suppose the input character string is "01101111, 01100011". The initial state is state s_0 which is represented as 0@00. We append s_0's table ID of 0 to the first character 01101111 to form the lookup key 001101111. This matches the fourth TCAM entry in the 3-decision table. We now need to find the decision. We use s_0's column ID 00 to determine that the first decision is the correct decision. This gives us the state s_1

which is represented as 0@01. We then prepend s_1's table ID of 0 to the second character 01100011 to form the lookup key 001100011. This matches the third TCAM entry. We use s_1's column ID of 01 to determine that the second decision is the correct decision. This gives us the next state s_2 which has code 0@10. Because s_2 is an accepting state, we would accept the input string. Note that because this DFA has only 3 states which have all been consolidated together, all three states have the same table ID of 0. In general, with more states than just those consolidated together, we would have more table IDs.

There are two key technical challenges in table consolidation. The first challenge is how to consolidate k 1-decision transition tables into a k-decision transition table. The second challenge is which 1-decision transition tables should be consolidated together. Intuitively, the more similar two 1-decision transition tables are, the more TCAM space saving we can get from consolidating them together. However, we have to consider the deferment relationship among states. We present our solutions to these two challenges.

5.3.2 Computing a k-decision table

In this section, we assume we know which states need to be consolidated together and present a local state consolidation algorithm that takes a k_1-decision table for state set S_i and a k_2-decision table for another state set S_j as its input and outputs a consolidated $(k_1 + k_2)$-decision table for state set $S_i \cup S_j$. For ease of presentation, we first assume that $k_1 = k_2 = 1$.

Let s_1 and s_2 be the two input states which have default transitions to states s_3 and s_4.

The consolidated table will be assigned a common table ID X. We assign state s_1 column ID 0 and state s_2 column ID 1. Thus, we encode s_1 as X@0 and s_2 as X@1. We enforce a constraint that if we do not consolidate s_3 and s_4 together, then s_1 and s_2 cannot defer any transitions at all. If we do consolidate s_3 and s_4 together, then s_1 and s_2 may have incomplete transition tables due to default transitions to s_3 and s_4, respectively.

The key concepts underlying this algorithm are breakpoints and critical ranges. To define breakpoints, it is helpful to view Σ as numbers ranging from 0 to $|\Sigma| - 1$; given 8 bit characters, $|\Sigma| = 256$. For any state s, we define a character $i \in \Sigma$ to be a *breakpoint* for s if $\delta(s, i) \neq \delta(s, i-1)$. For the end cases, we define 0 and $|\Sigma|$ to be breakpoints for every state s. Let $b(s)$ be the set of breakpoints for state s. We then define $b(S) = \bigcup_{s \in S} b(s)$ to be the set of breakpoints for a set of states $S \subset Q$. Finally, for any set of states S, we define $r(S)$ to be the set of ranges defined by $b(S)$: $r(S) = \{[0, b_2 - 1], [b_2, b_3 - 1], \ldots, [b_{|b(S)|-1}, |\Sigma| - 1]\}$ where b_i is ith smallest breakpoint in $b(S)$. Note that $0 = b_1$ is the smallest breakpoint and $|\Sigma|$ is the largest breakpoint in $b(S)$. Within $r(S)$, we label the range beginning at breakpoint b_i as r_i for $1 \leq i \leq |b(S)| - 1$. If $\delta(s, b_i)$ is deferred, then r_i is a deferred range.

When we consolidate s_1 and s_2 together, we compute $b(\{s_1, s_2\})$ and $r(\{s_1, s_2\})$. For each $r' \in r(\{s_1, s_2\})$ where r' is not a deferred range for both s_1 and s_2, we create a consolidated transition rule where the decision of the entry is the ordered pair of decisions for state s_1 and s_2 on r'. For each $r' \in r(\{s_1, s_2\})$ where r' is a deferred range for one of s_1 but not the other, we fill in r' in the incomplete transition table where it is deferred, and we create a consolidated entry where the decision of the entry is the ordered pair of decisions for

state s_1 and s_2 on r'. Finally, for each $r' \in r(\{s_1, s_2\})$ where r' is a deferred range for both s_1 and s_2, we do not create a consolidated entry. This produces a non-overlapping set of transition rules that may be incomplete if some ranges do not have a consolidated entry. If the final consolidated transition table is complete, we minimize it using the optimal 1-dimensional TCAM minimization algorithm in [30, 47]. If the table is incomplete, we minimize it using the 1-dimensional incomplete classifier minimization algorithm in [31]. We generalize this algorithm to cases where $k_1 > 1$ and $k_2 > 1$ by simply considering $k_1 + k_2$ states when computing breakpoints and ranges.

5.3.3 Choosing States to Consolidate

We now describe our global consolidation algorithm for determining which states to consolidate together. As we observed earlier, if we want to consolidate two states s_1 and s_2 together, we need to consolidate their parent nodes in the deferment forest as well or else lose all the benefits of shadow encoding. Thus, we propose to consolidate two deferment trees together.

A consolidated deferment tree must satisfy the following properties. First, each node is to be consolidated with at most one node in the second tree; some nodes may not be consolidated with any node in the second tree. Second, a level i node in one tree must be consolidated with a level i node in the second tree. The level of a node is its distance from the root. We define the root to be a level 0 node. Third, if two level i nodes are consolidated together, their level $i-1$ parent nodes must also be consolidated together.

An example legal matching of nodes between two deferment trees is depicted in Figure 5.7.

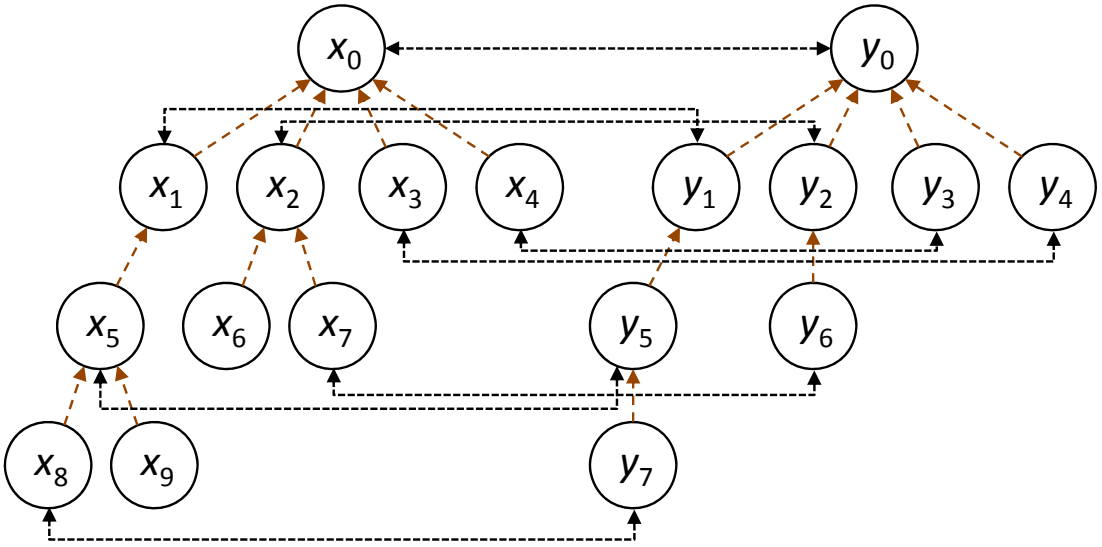


Figure 5.7: Consolidating two trees.

Given two deferment trees, we start the consolidation process from the roots. After we consolidate the two roots, we need to decide how to pair their children together. For each pair of nodes that are consolidated together, we again must choose how to pair their children together, and so on. We make an optimal choice using a combination of dynamic programming and matching techniques. Suppose we wish to compute the minimum cost $C(x, y)$, measured in TCAM entries, of consolidating two subtrees rooted at nodes x and y where x has u children $X = \{x_1, \ldots, x_u\}$ and y has v children $Y = \{y_1, \ldots, y_v\}$. We first recursively compute $C(x_i, y_j)$ for $1 \leq i \leq u$ and $1 \leq j \leq v$ using our local state consolidation algorithm as a subroutine. We then construct a complete bipartite graph $K_{X,Y}$ such that each edge (x_i, y_j) has the edge weight $C(x_i, y_j)$ for $1 \leq i \leq u$ and $1 \leq j \leq v$. Here $C(x, y)$ is the cost of a minimum weight matching [24, 35] of $K(X, Y)$ plus the cost of consolidating x and y. When $|X| \neq |Y|$, to make the sets equal in size, we pad the smaller

set with null states that defer all transitions.

Finally, we must decide which trees to consolidate together. We assume that we produce k-decision tables where k is a power of 2. We describe how we solve the problem for $k = 2$ first. We create an edge-weighted complete graph with where each deferment tree is a node and where the weight of each edge is the cost of consolidating the two corresponding deferment trees together. We find a minimum weight matching [16, 18] of this complete graph to give us an optimal pairing for $k = 2$. For larger $k = 2^l$, we then repeat this process $l - 1$ times. Our matching is not necessarily optimal for $k > 2$.

In some cases, the deferment forest may have only one tree. In such cases, we consider consolidating the subtrees rooted at the children of the root of the single deferment tree. We also consider similar options if we have a few deferment trees but they are not structurally similar.

Figure Algorithm 5.8 shows the pseudo-code for the algorithm.

5.3.3.1 Greedy Matching

Our algorithm using the matching subroutines gives the optimal pairing of deferment trees but can be relatively slow on larger DFAs. When running time is a concern, we present a greedy matching routine. When we need to match children of two nodes, x and y, we consider one child at a time from the node with fewer children (say x). First all children of y are set *unmarked*. For each child, x_i, of x, we find the *best match* from the unmarked

Input: Deferment forest, DF, with r tree roots, $s_1, \ldots, s_r$.
Output: Optimal matching of the r roots.

1 For each pair of roots, s_i and s_j, compute $C(s_i, s_j)$;
2 Construct complete graph K_r, with the roots as vertices and $C(s_i, s_j)$ as edge weights;
3 **return** Minimum_Weight_Matching(K_r);

4 **Function** $C(s_1, s_2)$
    // Base case
5 **if** s_1 *and* s_2 *have no children* **then**
6 **return** Consolidated_Cost(s_1, s_2);
    // Recursive case
7 Attach NULL children so that both s_1 and s_2 have same number of children, q;
8 Construct complete bipartite graph $K_{q,q}$, with the children of s_1 and s_2 as the vertices, and set $C(s_x, s_y)$ as the edge weight between vertices s_x and s_y;
9 M = Minimum_Weight_Bipartite_Matching($K_{q,q}$) gives the matching of the children;
10 count $\leftarrow 0$;
11 **foreach** *matching* $(s_x, s_y) \in M$ **do**
12 count $\leftarrow$ count $+ C(s_x, s_y)$;
13 **return** (count + Consolidated_Cost(s_1, s_2));

Figure 5.8: Algorithm for Consolidating Trees.

children of y, match them up, and set the matched child in y as *marked*. The best match for x_i is given by

$$\text{argmin}_{y_j \in \{\text{unmarked children of } y\}} \frac{C(x_i, y_j)}{C(x_i) + C(y_j)}$$

where $C(x)$ is just the cost (in TCAM entries) of the subtree rooted at x. If $C(x_i)+C(y_j) = 0$, then we set the ratio to 0.5. All unmarked children of y at the end are matched with null states. We consider the children of x in decreasing order of $C(x_i)$ to prioritize the larger children of x. We use the same approach for matching roots. First all roots are set unmarked. Each time we consider the largest unmarked root, find the best match for it, and then mark the newly matched roots.

In our experiments, this greedy approach runs much faster than the optimal approach

and the resulting classifier size is not much larger. We also observe that another greedy approach that uses $C(x_i, y_j)$ instead of $\frac{C(x_i,y_j)}{C(x_i)+C(y_j)}$ produces classifiers with much larger TCAM sizes. This approach often matches a large child of x with a small child of y that it does not align well with.

5.3.4 Effectiveness of Table Consolidation

We now explain why table consolidation works well on real-world RE sets.

Most real-world RE sets contain REs with wildcard closures '.⋆' where the wildcard '.' matches any character and the closure '⋆' allows for unlimited repetitions of the preceding character. Wildcard closures create deferment trees with lots of structural similarity. For example, consider the D^{2}FA in Figure 5.9 for RE set {/abc/, /abd/, /e.⋆f/}

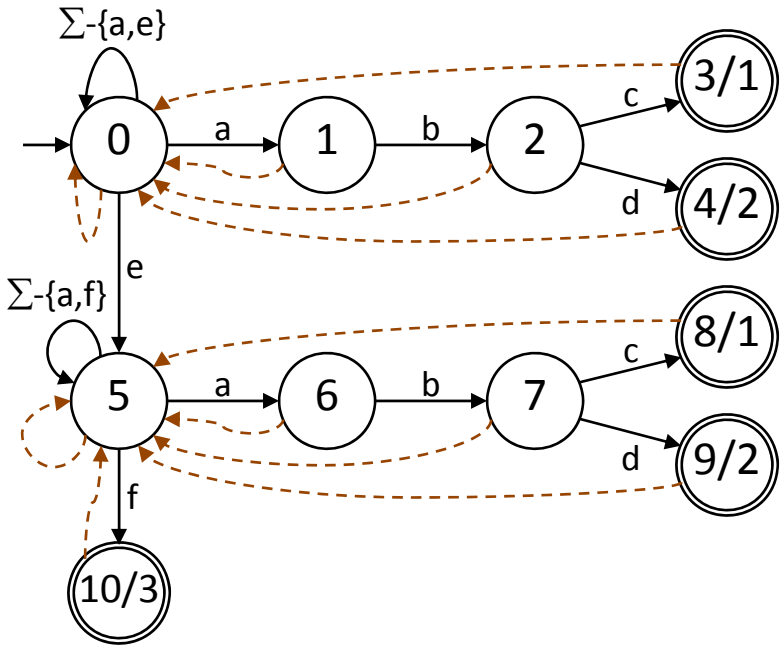


Figure 5.9: D^{2}FA for RE set {/abc/, /abd/, /e.⋆f/}.

where we use dashed arrows to represent the default transitions. The second wildcard closure '.$\star$' in the RE /e.$\star$f/ duplicates the entire DFA sub-structure for recognizing REs/abc/ and /abd/. Thus, table consolidation of the subtree $(0, 1, 2, 3, 4)$ with the subtree $(5, 6, 7, 8, 9, 10)$ will lead to significant space saving.

5.4 Variable Striding

We explore ways to improve RE matching throughput by consuming multiple characters per TCAM lookup. One possibility is a k-stride DFA which uses k-stride transitions that consume k characters per transition. Although k-stride DFAs can speed up RE matching by up to a factor of k, the number of states and transitions can grow exponentially in k. To limit the state and transition space explosion, we propose variable striding using *variable-stride DFAs*. A k-var-stride DFA consumes between 1 and k characters in each transition with at least one transition consuming k characters. Conceptually, each state in a k-var-stride DFA has 256^k transitions, and each transition is labeled with (1) a unique string of k characters and (2) a stride length j ($1 \leq j \leq k$) indicating the number of characters consumed.

In TCAM-based variable striding, each TCAM lookup uses the next k consecutive characters as the lookup key, but the number of characters consumed in the lookup varies from 1 to k; thus, the lookup decision contains both the destination state ID and the stride length.

There are many technical challenges in implementing variable striding. First, we need to control the exponential growth in the number of states. Second, we need to control the exponential growth in the number of transitions. Third, we need to carefully choose which transitions to expand from 1-stride to multi-stride given a specific amount of available TCAM space. Fourth, we need to carefully decide on the maximum stride length k. Increasing k can help by increasing average RE matching throughput; however, increasing k can hurt by requiring more TCAM space. Specifically, implementing a k-var-stride DFA in TCAM requires 8k bits for the k input characters in each lookup key. The width of a TCAM chip is configurable, but not arbitrary. Commercially available TCAM chips typically can be configured with length 36, 72, 144, 288, or 576 bits. We must choose k so that we optimize throughput while not wasting bits in each TCAM entry.

5.4.1 Observations

We use an example to show how variable striding can achieve a significant RE matching throughput increase with a small and controllable space increase. Figure 5.10 shows a 3-var-stride transition table that corresponds to state s_0 in Figure 5.1. This table only has 7 entries as opposed to 116 entries in a full 3-stride table for s_0. If we assume that each of the 256 characters is equally likely to occur, the average number of characters consumed per 3-var-stride transition of s_0 is $1 * 1/16 + 2 * 15/256 + 3 * 225/256 = 2.82$.

TCAM					SRAM	
Src state	Inp char1	Inp char2	Inp char3		Dest state	Stride
s_0	0110 0000	**** ****	**** ****	→	s_0	1
s_0	0110 ****	**** ****	**** ****	→	s_1	1
s_0	**** ****	0110 0000	**** ****	→	s_0	2
s_0	**** ****	0110 ****	**** ****	→	s_1	2
s_0	**** ****	**** ****	0110 0000	→	s_0	3
s_0	**** ****	**** ****	0110 ****	→	s_1	3
s_0	**** ****	**** ****	**** ****	→	s_0	3

Figure 5.10: 3-var-stride transition table for s_0

5.4.2 Eliminating State Explosion

We first explain how converting a 1-stride DFA to a k-stride DFA causes state explosion. For a source state and a destination state pair (s, d), a k-stride transition path from s to d may contain $k-1$ intermediate states (excluding d); for each unique combination of accepting states that appear on a k-stride transition path from s to d, we need to create a new destination state because a unique combination of accepting states implies that the input has matched a unique combination of REs. This can be a very large number of new states.

We eliminate state explosion by ending any k-var-stride transition path at the first accepting state it reaches. Thus, a k-var-stride DFA has the exact same state set as its corresponding 1-stride DFA. Ending k-var-stride transitions at accepting states does have subtle interactions with table consolidation and shadow encoding. We end any k-var-stride consolidated transition path at the first accepting state reached in any one of the paths being consolidated which can reduce the expected throughput increase of variable striding. There is a similar but even more subtle interaction with shadow encoding which we

describe in the next section.

5.4.3 Controlling Transition Explosion

In a k-stride DFA converted from a 1-stride DFA with alphabet Σ, a state has $|\Sigma|^k$ outgoing k-stride transitions. Although we can leverage our techniques of character bundling and shadow encoding to minimize the number of required TCAM entries, the rate of growth tends to be exponential with respect to stride length k. We have two key ideas to control transition explosion: self-loop unrolling and k-var-stride transition sharing.

5.4.3.1 Self-Loop Unrolling Algorithm

We now consider root states, all of which are self-looping states. We have two methods to compute the k-var-stride transition tables of root states. The first is direct expansion (stopping transitions at accepting states) since these states do not defer to other states which results in an exponential increase in table size with respect to k. The second method, which we call *self-loop unrolling*, scales linearly with k.

Self-loop unrolling increases the stride of all the self-loop transitions encoded by the last default TCAM entry. Self-loop unrolling starts with a root state j-var-stride transition table encoded as a compressed TCAM table of n entries with a final default entry representing most of the self-loops of the root state. Note that given any complete TCAM table where the last entry is not a default entry, we can always replace that last entry with a default entry without changing the semantics of the table. We generate the (j+1)-var-stride

transition table by expanding the last default entry into n new entries, which are obtained by prepending 8 $*$'s as an extra default field to the beginning of the original n entries. This produces a (j+1)-var-stride transition table with $2n - 1$ entries. Figure 5.10 shows the resulting table when we apply self-loop unrolling twice on the DFA in Figure 5.1.

We next illustrate the idea of self-loop unrolling using an example. Consider state s_0 of Figure 5.1. The default transition in s_0's table is a self-loop that is matched by 240 characters; one self-loop is matched by the first TCAM entry in s_0's table. We can "unroll" this self-loop and increase the stride of many but not all 2-stride and 3-stride transitions as follows. First, we leave in place the first two 1-stride transitions. We then make 2-stride copies of these transitions where we shift the characters over by one and put a default character in the first position. These 2-stride transitions capture the case where the first character in the transition self-loops but is not 01100000 and the second character leaves state s_0 or is 01100000. We then make 3-stride copies of these transitions where we shift the characters over by one again and put default characters for the first two positions. Finally, we include a stride-3 default transition that self-loops back to state 0. The resulting 7 transition variable-stride table is shown in Figure 5.10. In this example, we could continue using self-loop unrolling to create even larger stride transitions with an additional cost of only 2 TCAM entries per extra character consumed.

5.4.3.2 k-var-stride Transition Sharing Algorithm

Similar to 1-stride DFAs, there are many transition sharing opportunities in a k-var-stride DFA. Consider two states s_0 and s_1 in a 1-stride DFA where s_0 defers to s_1. The deferment relationship implies that s_0 shares many common 1-stride transitions with s_1. In the k-var-stride DFA constructed from the 1-stride DFA, all k-var-stride transitions that begin with these common 1-stride transitions are also shared between s_0 and s_1. Furthermore, two transitions that do not begin with these common 1-stride transitions may still be shared between s_0 and s_1. For example, in the 1-stride DFA fragment in Figure 5.11, although s_1 and s_2 do not share a common transition for character a, when we construct the 2-var-stride DFA, s_1 and s_2 share the same 2-stride transition on string *aa* that ends at state s_5.

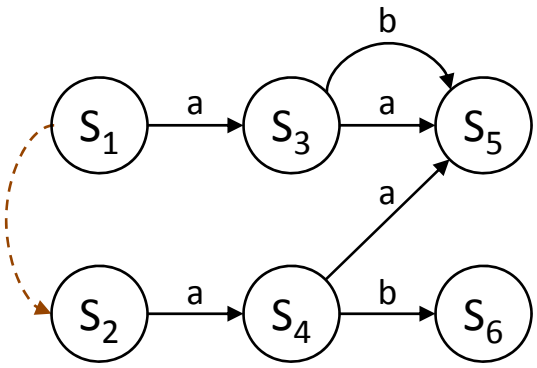


Figure 5.11: States s_1 and s_2 share transition `aa`

To promote transition sharing among states in a k-var-stride DFA, we first need to decide on the deferment relationship among states. The ideal deferment relationship should be calculated based on the SRG of the final k-var-stride DFA. However, the k-var-stride DFA cannot be finalized before we need to compute the deferment relationship among states

because the final k-var-stride DFA is subject to many factors such as available TCAM space. There are two approximation options for the final k-var-stride DFA for calculating the deferment relationship: the 1-stride DFA and the full k-stride DFA. We have tried both options in our experiments, and the difference in the resulting TCAM space is negligible. Thus, we simply use the deferment forest of the 1-stride DFA in computing the transition tables for the k-var-stride DFA.

Second, for any two states s_1 and s_2 where s_1 defers to s_2, we need to compute s_1's k-var-stride transitions that are not shared with s_2 because those transitions will constitute s_1's k-var-stride transition table. Although this computation is trivial for 1-stride DFAs, this is a significant challenge for k-var-stride DFAs because each state has too many (256^k) k-var-stride transitions. The straightforward algorithm that enumerates all transitions has a time complexity of $O(|Q|^2|\Sigma|^k)$, which grows exponentially with k. We propose a dynamic programming algorithm with a time complexity of $O(|Q|^2|\Sigma|k)$, which grows linearly with k. Our key idea is that the non-shared transitions for a k-stride DFA can be quickly computed from the non-shared transitions of a (k-1)-var-stride DFA. For example, consider the two states s_1 and s_2 in Figure 5.11 where s_1 defers to s_2. For character a, s_1 transits to s_3 while s_2 transits to s_4. Assuming that we have computed all (k-1)-var-stride transitions of s_3 that are not shared with the (k-1)-var-stride transitions of s_4, if we prepend all these (k-1)-var-stride transitions with character a, the resulting k-var-stride transitions of s_1 are all not shared with the k-var-stride transitions of s_2, and therefore should all be included in s_1's k-var-stride transition table. Formally, using $n(s_i, s_j, k)$ to

denote the number of k-stride transitions of s_i that are not shared with s_j, our dynamic programming algorithm uses the following recursive relationship between $n(s_i, s_j, k)$ and $n(s_i, s_j, k-1)$:

$$n(s_i, s_j, 0) = \begin{cases} 0 & \text{if } s_i = s_j \\ 1 & \text{if } s_i \neq s_j \end{cases} \tag{5.1}$$

$$n(s_i, s_j, k) = \sum_{c \in \Sigma} n(\delta(s_i, c), \delta(s_j, c), k-1) \tag{5.2}$$

The above formulae assume that the intermediate states on the k-stride paths starting from s_i or s_j are all non-accepting. For state s_i, we stop increasing the stride length along a path whenever we encounter an accepting state on that path or on the corresponding path starting from s_j. The reason is similar to why we stop a consolidated path at an accepting state, but the reasoning is more subtle. Let p be the string that leads s_j to an accepting state. The key observation is that we know that any k-var-stride path that starts from s_j and begins with p ends at that accepting state. This means that s_i cannot exploit transition sharing on any strings that begin with p.

Figure 5.12 shows the resultant 2-var-stride transition tables for all three states s_0, s_1, and s_2 of the D^2FA in Figure 5.3(a). Note that the one transition out of state s_1 and two self-loop transitions for state s_2 have stride-1 because they end at s_2, an accepting state.

The above dynamic programming algorithm produces non-overlapping and incomplete

TCAM				SRAM	
Src state	Inp char1	Inp char2		Dest state	Stride
s_1	[c]	*	→	s_2	1
s_2	[b..c]	[c]	→	s_2	2
s_2	[a]	*	→	s_2	1
s_2	[d..o]	*	→	s_2	1
s_0	[a..o]	[0..96]	→	s_0	2
s_0	[a..o]	[a]	→	s_2	2
s_0	[a..o]	[b]	→	s_1	2
s_0	[a..o]	[c..o]	→	s_2	2
s_0	[a..o]	[112..255]	→	s_0	2
s_0	[0..96]	[0..96]	→	s_0	2
s_0	[0..96]	[a..o]	→	s_1	2
s_0	[0..96]	[112..255]	→	s_0	2
s_0	[112..255]	[0..96]	→	s_0	2
s_0	[112..255]	[a..o]	→	s_1	2
s_0	[112..255]	[112..255]	→	s_0	2

Figure 5.12: Uncompressed 2-var-stride transition tables for D^2FA in Figure 5.3(a) ($a = 97$, $o = 111$)

transition tables that we compress using the 1-dimensional incomplete classifier minimization algorithm in [31].

5.4.4 Variable Striding Selection Algorithm

We now propose solutions for the third key challenge - which states should have their stride lengths increased and by how much, *i.e.*, how should we compute the transition function δ. Note that each state can independently choose its variable striding length as long as the final transition tables are composed together according to the deferment forest. This can be easily proven based on the way that we generate k-var-stride transition tables. For any two states s_1 and s_2 where s_1 defers to s_2, the way that we generate s_1's k-var-stride transition table is seemingly based on the assumption that s_2's transition table is also

k-var-stride; actually, we do not have this assumption. For example, if we choose k-var-stride ($2 \leq k$) for s_1 and 1-stride for s_2, all strings from s_1 will be processed correctly; the only issue is that strings deferred to s_2 will process only one character.

We view this as a packing problem: given a TCAM capacity C, for each state s, we select a variable stride length value K_s, such that $\sum_{s \in Q} |\mathbb{T}(s, K_s)| \leq C$, where $\mathbb{T}(s, K_s)$ denotes the K_s-var-stride transition table of state s. This packing problem has a flavor of the knapsack problem, but an exact formulation of an optimization function is impossible without making assumptions about the input character distribution. We propose the following algorithm for finding a feasible δ that strives to maximize the minimum stride of any state. First, we use all the 1-stride tables as our initial selection. Second, for each j-var-stride ($j \geq 2$) table t of state s, we create a tuple $(l, d, |t|)$ where l denotes variable stride length, d denotes the distance from state s to the root of the deferment tree that s belongs to, and $|t|$ denotes the number of entries in t. As stride length l increases, the individual table size $|t|$ may increase significantly, particularly for the complete tables of root states. To balance table sizes, we set limits on the maximum allowed table size for root states and non-root states. If a root state table exceeds the root state threshold when we create its j-var-stride table, we apply self-loop unrolling once to its $(j-1)$-var-stride table to produce a j-var-stride table. If a non-root state table exceeds the non-root state threshold when we create its j-var-stride table, we simply use its $(j-1)$-var-stride table as its j-var-stride table. Third, we sort the tables by these tuple values in increasing order first using l, then using d, then using $|t|$, and finally a pseudorandom coin flip to break

ties. Fourth, we consider each table t in order. Let t' be the table for the same state s in the current selection. If replacing t' by t does not exceed our TCAM capacity C, we do the replacement.

5.5 Implementation and Modeling

We now describe some implementation issues associated with our TCAM based RE matching solution. First, the only hardware required to deploy our solution is the off-the-shelf TCAM (and its associated SRAM). Many deployed networking devices already have TCAMs, but these TCAMs are likely being used for other purposes. Thus, to deploy our solution on existing network devices, we would need to share an existing TCAM with another application. Alternatively, new networking devices can be designed with an additional dedicated TCAM chip.

Second, we describe how we update the TCAM when an RE set changes. First, we must compute a new DFA and its corresponding TCAM representation. For the moment, we recompute the TCAM representation from scratch, but we believe a better solution can be found and is something we plan to work on in the future. We report some timing results in our experimental section. Fortunately, this is an offline process during which time the DFA for the original RE set can still be used. The second step is loading the new TCAM entries into TCAM. If we have a second TCAM to support updates, this rewrite can occur while the first TCAM chip is still processing packet flows. If not, RE matching must halt

while the new entries are loaded. This step can be performed very quickly, so the delay will be very short. In contrast, updating FPGA circuitry takes significantly longer.

We have not developed a full implementation of our system. Instead, we have only developed the algorithms that would take an RE set and construct the associated TCAM entries. Thus, we can only estimate the throughput of our system using TCAM models. We use Agrawal and Sherwood's TCAM model [3] assuming that each TCAM chip is manufactured with a $0.18\mu m$ process to compute the estimated latency of a single TCAM lookup based on the number of TCAM entries searched. These model latencies are shown in Table 5.1. We recognize that some processing must be done besides the TCAM lookup such as composing the next state ID with the next input character; however, because the TCAM lookup latency is much larger than any other operation, we focus only on this parameter when evaluating the potential throughput of our system.

Entries	TCAM Chip size (36-bit wide)	TCAM Chip size (72-bit wide)	Latency ns
1024	0.037 Mb	0.074 Mb	0.94
2048	0.074 Mb	0.147 Mb	1.10
4096	0.147 Mb	0.295 Mb	1.47
8192	0.295 Mb	0.590 Mb	1.84
16384	0.590 Mb	1.18 Mb	2.20
32768	1.18 Mb	2.36 Mb	2.57
65536	2.36 Mb	4.72 Mb	2.94
131072	4.72 Mb	9.44 Mb	3.37

Table 5.1: TCAM size and Latency

5.6 Experimental Results

In this section, we evaluate our TCAM-based RE matching solution on real-world RE sets focusing on two metrics: TCAM space and RE matching throughput.

5.6.1 Methodology

We use the same 8 RE sets used in Section 4.5 for the main results.

To test the scalability of our algorithms, we use one family of 34 REs from a recent public release of the Snort rules with headers ($EXTERNAL_NET, $HTTP_PORTS, $HOME_NET, any), most of which contain wildcard closures '.*'. We added REs one at a time until the number of DFA states reached 305,339. We name this family Scale.

We calculate TCAM space by multiplying the number of entries by the TCAM width: 36, 72, 144, 288, or 576 bits. For a given DFA, we compute a minimum width by summing the number of state ID bits required with the number of input bits required. In all cases, we needed at most 16 state ID bits. For 1-stride DFAs, we need exactly 8 input character bits, and for 7-var-stride DFAs, we need exactly 56 input character bits. We then calculate the TCAM width by rounding the minimum width up to the smallest larger legal TCAM width. For all our 1-stride DFAs, we use TCAM width 36. For all our 7-var-stride DFAs, we use TCAM width 72.

We estimate the potential throughput of our TCAM-based RE matching solution by using the model TCAM lookup speeds we computed in Section 5.5 to determine how many

TCAM lookups can be performed in a second for a given number of TCAM entries and then multiplying this number by the number of characters processed per TCAM lookup. With 1-stride TCAMs, the number of characters processed per lookup is 1. For 7-var-stride DFAs, we measure the average number of characters processed per lookup in a variety of input streams.

We use Becchi *et al.*'s network traffic generator [11] to generate a variety of synthetic input streams. This traffic generator includes a parameter that models the probability of malicious traffic p_M. With probability p_M, the next character is chosen so that it leads away from the start state. With probability $(1 - p_M)$, the next character is chosen uniformly at random.

5.6.2 Results on 1-stride DFAs

RE set	#states	TS			TS + TC2			TS + TC4		
		tcam Mbits	#rows per state	thru Gbps	tcam Mbits	#rows per state	thru Gbps	tcam Mbits	#rows per state	thru Gbps
Bro217	6533	0.31	1.40	3.64	0.21	0.94	4.35	0.17	0.78	4.35
C613	11308	0.63	1.61	3.11	0.52	1.35	3.64	0.45	1.17	3.64
C10	14868	0.61	1.20	3.11	0.31	0.61	3.64	0.16	0.32	4.35
C7	24750	1.00	1.18	3.11	0.53	0.62	3.64	0.29	0.34	3.64
C8	3108	0.13	1.20	5.44	0.07	0.62	5.44	0.03	0.33	8.51
Snort24	13886	0.55	1.16	3.64	0.30	0.64	3.64	0.18	0.38	4.35
Snort31	20068	1.43	2.07	2.72	0.81	1.17	2.72	0.50	0.72	3.64
Snort34	13825	0.56	1.18	3.11	0.30	0.62	3.64	0.17	0.36	4.35

Table 5.2: TCAM size and throughput for 1-stride DFAs

Table 5.2 shows our experimental results on the 8 RE sets using 1-stride DFAs. We use TS to denote our transition sharing algorithm including both character bundling and

shadow encoding. We use TC2 and TC4 to denote our table consolidation algorithm where we consolidate at most 2 and 4 transition tables together, respectively. For each RE set, we measure the number states in its 1-stride DFA, the resulting TCAM space in megabits, the average number of TCAM table entries per state, and the projected RE matching throughput; the number of TCAM entries is the number of states times the average number of entries per state. The TS column shows our results when we apply TS alone to each RE set. The TS+TC2 and TS+TC4 columns show our results when we apply both TS and TC under the consolidation limit of 2 and 4, respectively, to each RE set.

We draw the following conclusions from Table 5.2. (1) *Our RE matching solution is extremely effective in saving TCAM space.* Using TS+TC4, the maximum TCAM size for the 8 RE sets is only 0.50 Mb, which is two orders of magnitude smaller than the current largest commercially available TCAM chip size of 72 Mb. More specifically, the number of TCAM entries per DFA state ranges between .32 and 1.17 when we use TC4. We require 16, 32, or 64 SRAM bits per TCAM entry for TS, TS+TC2, and TS+TC4, respectively as we need to record 1, 2, or 4 state 16 bit state IDs in each decision, respectively. (2) *Transition sharing alone is very effective.* With the transition sharing algorithm alone, the maximum TCAM size is only 1.43Mb for the 8 RE sets. Furthermore, we see a relatively tight range of TCAM entries per state of 1.16 to 2.07. Transition sharing works extremely well with all 8 RE sets including those with wildcard closures and those with primarily strings. (3) *Table consolidation is very effective.* On the 8 RE sets, adding TC2 to

TS improves compression by an average of 41% (ranging from 16% to 49%) where the maximum possible is 50%. We measure improvement by computing $(TS-(TS+TC2))/TS)$. Replacing TC2 with TC4 improves compression by an average of 36% (ranging from 13% to 47%) where we measure improvement by computing $((TS+TC2)-(TS+TC4))/(TS+TC2)$. Here we do observe a difference in performance, though. For the two RE sets Bro217 and C613 that are primarily strings without table consolidation, the average improvements of using TC2 and TC4 are only 24% and 15%, respectively. For the remaining six RE sets that have many wildcard closures, the average improvements are 47% and 43%, respectively. The reason, as we touched on in Section 5.3.4, is how wildcard closure creates multiple deferment trees with almost identical structure. Thus wildcard closures, the prime source of state explosion, is particularly amenable to compression by table consolidation. In such cases, doubling our table consolidation limit does not greatly increase SRAM cost. Specifically, while the number of SRAM bits per TCAM entry doubles as we double the consolidation limit, the number of TCAM entries required almost halves! (4) *Our RE matching solution achieves high throughput with even* 1*-stride DFAs.* For the TS+TC4 algorithm, on the 8 RE sets, the average throughput is 4.60Gbps (ranging from 3.64Gbps to 8.51Gbps).

We use our Scale dataset to assess the scalability of our algorithms' performance focusing on the number of TCAM entries per DFA state. Figure 5.13(a) shows the number of TCAM entries per state for TS, TS+TC2, and TS+TC4 for the Scale REs containing 26 REs (with DFA size 1275) to 34 REs (with DFA size 305,339). The DFA size roughly doubled for

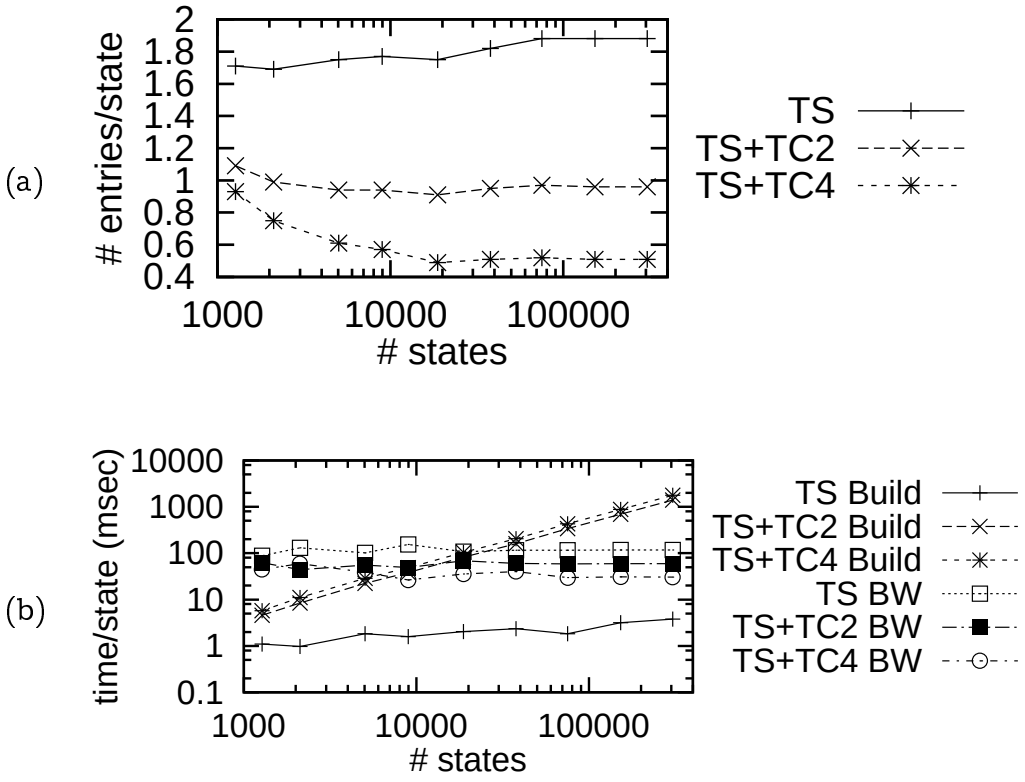


Figure 5.13: TCAM entries per DFA state (a) and compute time per DFA state (b) for Scale 26 through Scale 34.

every RE added. In general, the number of TCAM entries per state is roughly constant and actually decreases with table consolidation. This is because table consolidation performs better as more REs with wildcard closures are added as there are more trees with similar structure in the deferment forest.

We now analyze running time. We ran our experiments on the Michigan State University High Performance Computing Center (HPCC). The HPCC has several clusters; most of our experiments were executed on the fastest cluster which has nodes that each have 2 quad-core Xeons running at 2.3GHz. The total RAM for each node is 8GB. Figure 5.13(b) shows the compute time per state in milliseconds. The build times are the time per DFA

state required to build the non-overlapping set of transitions (applying TS and TC); these increase linearly because these algorithms are quadratic in the number of DFA states. For our largest DFA Scale 34 with 305,339 states, the total time required for TS, TS+TC2, and TS+TC4 is 19.25 mins, 118.6 hrs, and 150.2 hrs, respectively. These times are cumulative; that is going from TS+TC2 to TS+TC4 requires an additional 31.6 hours. This table consolidation time is roughly one fourth of the first table consolidation time because the number of DFA states has been cut in half by the first table consolidation and table consolidation has a quadratic running time in the number of DFA states. The BW times are the time per DFA state required to minimize these transition tables using the Bitweaving algorithm in [31]; these times are roughly constant as Bitweaving depends on the size of the transition tables for each state and is not dependent on the size of the DFA. For our largest DFA Scale 34 with $305,339$ states, the total Bitweaving optimization time on TS, TS+TC2, and TS+TC4 is 10 hrs, 5 hrs, and 2.5 hrs. These times are not cumulative and fall by a factor of 2 as each table consolidation step cuts the number of DFA states by a factor of 2.

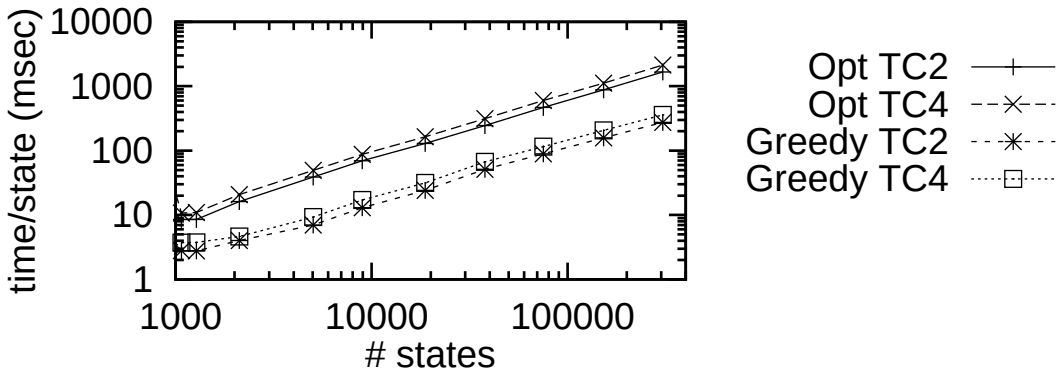


Figure 5.14: Consolidation times for Scale 26 through Scale 34 for Optimal and Greedy consolidation algorithms.

Figure 5.14 shows the time required per state for the greedy and optimal consolidation algorithms on the Scale dataset. The greedy algorithm runs roughly 6 times faster than the optimal algorithm. The average increase in the number of resulting TCAM rules is around 4% for TC2 and around 9% for TC4.

The partially deferred algorithm given in Section 5.2.2.4 always performs at least as well as the completely deferred minimization algorithm given in [31]. For the three Snort RE sets and C613, the partially deferred algorithm results in a reduction of 1, 2, 152, and 194 TCAM entries over the completely deferred algorithm. For the other RE sets, both algorithms perform equally well. The partially deferred algorithm is slower than the completely deferred algorithm because there are more unique decisions during minimization, so we use the completely deferred minimization algorithm for computing classifier sizes during consolidation, and we use the partially deferred minimization algorithm for generating the final TCAM classifiers for each state.

5.6.3 Results on 7-var-stride DFAs

We consider two implementations of variable striding assuming we have a 2.36 megabit TCAM with TCAM width 72 bits (32,768 entries). Using Table 5.1, the latency of a lookup is 2.57 ns. Thus, the potential RE matching throughput of by a 7-var-stride DFA with average stride S is $8 \times S/.00000000257 = 3.11 \times S$ Gbps.

In our first implementation, we only use self-loop unrolling of root states in the deferment forest. Specifically, for each RE set, we first construct the 1-stride DFA using transition

sharing. We then apply self-loop unrolling to each root state of the deferment forest to create a 7-var-stride transition table. Because of the linear increase in transition table size, we know that the resulting TCAM table will increase in size by at most a factor of 7. In all our experiments, the size never increased by more than a factor of 2.25, and the largest DFA (for C7) required only 2.25 megabits. We can decrease the TCAM space by using table consolidation; this was very effective for all RE sets except the string matching RE sets Bro217 and C613. This was unnecessary since all self-loop unrolled tables fit within our available TCAM space.

Second, we apply full variable striding. Specifically, we first create 1-stride DFAs using transition sharing and then apply variable striding with no table consolidation, table consolidation with 2-decision tables, and table consolidation with 4-decision tables. We use the best result that fits within the 2.36 megabit TCAM space. For the RE sets Bro217, C8, C613, Snort24 and Snort34, no table consolidation is used. For C10 and Snort31, we use table consolidation with 2-decision tables. For C7, we use table consolidation with 4-decision tables.

We now run both implementations of our 7-var-stride DFAs on traces of length 287484 to compute the average stride. For each RE set, we generate 4 traces using Becchi *et al.*'s trace generator tool using default values 35%, 55%, 75%, and 95% for the parameter p_M. These generate increasingly malicious traffic that is more likely to move away from the start state towards distant accept states of that DFA. We also generate a completely random string to model completely uniform traffic such as binary traffic patterns which

we treat as $p_M = 0$.

We group the 8 RE sets into 3 groups: group (a) represents the two string matching RE sets Bro217 and C613; group (b) represents the three RE sets C7, C8, and C10 that contain all wildcard closures; group (c) represents the three RE sets Snort24, Snort31, and Snort34 that contain roughly 40% wildcard closures. Figure 5.15 shows the average stride length and throughput for the three groups of RE sets according to the parameter p_M (the random string trace is $p_M = 0$).

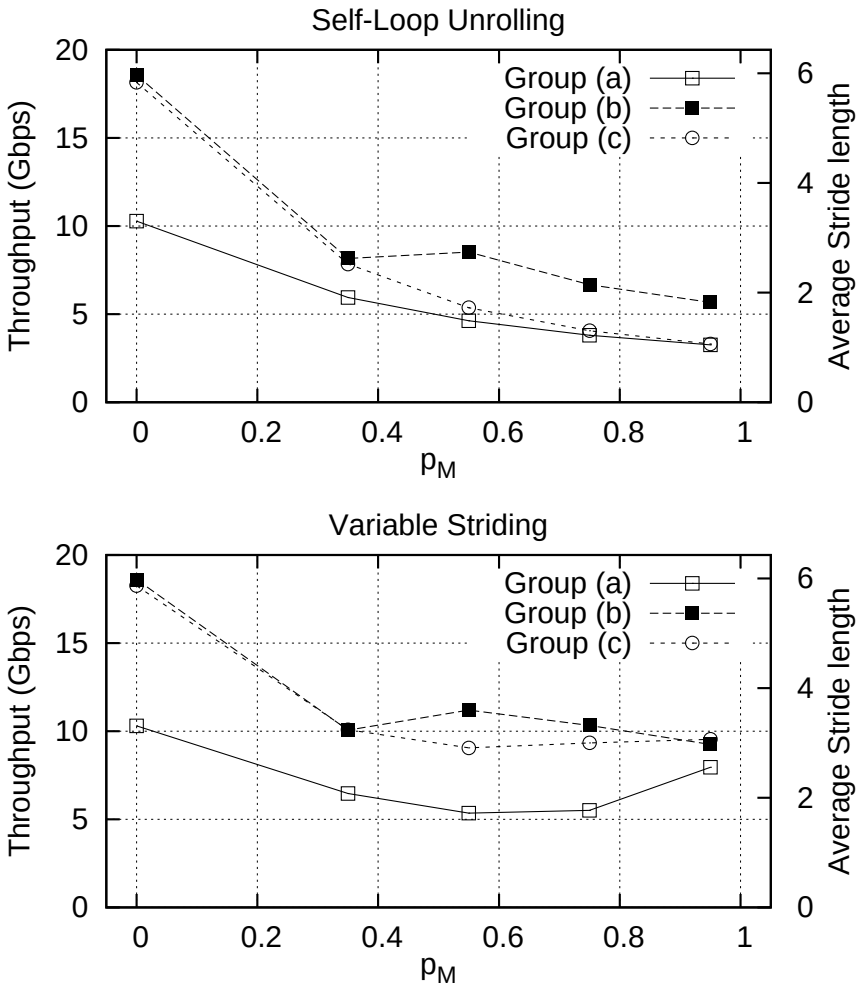


Figure 5.15: The throughput and average stride length of RE sets.

We make the following observations. *(1) Self-loop unrolling is extremely effective on the uniform trace.* For the non string matching sets, it achieves an average stride length of 5.97 and 5.84 and RE matching throughput of 18.58 and 18.15 Gbps for groups (b) and (c), respectively. For the string matching sets in group (a), it achieves an average stride length of 3.30 and a resulting throughput of 10.29 Gbps. Even though only the root states are unrolled, self-loop unrolling works very well because the non-root states that defer most transitions to a root state will still benefit from that root state's unrolled self-loops. In particular, it is likely that there will be long stretches of the input stream that repeatedly return to a root state and take full advantage of the unrolled self-loops. *(2) The performance of self-loop unrolling does degrade steadily as* p_M *increases for all RE sets except those in group (b).* This occurs because as p_M increases, we are more likely to move away from any default root state. Thus, fewer transitions will be able to leverage the unrolled self-loops at root states. *(3) For the uniform trace, full variable striding does little to increase RE matching throughput.* Of course, for the non-string matching RE sets, there was little room for improvement. *(4) As* p_M *increases, full variable striding does significantly increase throughput, particularly for groups (b) and (c).* For example, for groups (b) and (c), the minimum average stride length is 2.91 for all values of p_M which leads to a minimum throughput of 9.06Gbps. Also, for all groups of RE sets, the average stride length for full variable striding is much higher than that for self-loop unrolling for large p_M. For example, when $p_M = 95\%$, full variable striding achieves average stride lengths of 2.55, 2.97, and 3.07 for groups (a), (b), and (c), respectively, whereas self-loop unrolling achieves average stride lengths of only 1.04, 1.83,

and 1.06 for groups (a), (b), and (c), respectively.

These results indicate the following. First, self-loop unrolling is extremely effective at increasing throughput for random traffic traces. Second, other variable striding techniques can mitigate many of the effects of malicious traffic that lead away from the start state.

Chapter 6

Overlay Automata

In this section we present our overlay automata model for handling DFA state replication, and the implementation of the overlay automata in both software and hardware.

6.1 Introduction

As discussed in Section 3.2, the main reason for redundancy in a DFA is state replication, which causes the exponential increase in the size of the DFA as multiple REs are combines. Ideally we would like to build an automata whose size is proportional to a NFA and matching speed close to that of a DFA. We achieve this goal using our new overlay automata model.

6.1.1 Limitations of Prior Automata Models

DFA-based automata models have been developed to address DFA space explosion. Two representative models are D^{2}FA proposed by Kumar *et al.* [26] and XFA proposed by Smith *et al.* [41]. D^{2}FAs reduce the number of transitions stored per state by using deferred transitions to compactly represent common transitions, *i.e.*, the transitions with the same input character and destination state. This elegant solution can be automated; however, it only handles transition sharing, and does not address state replication, and resulting replicated transitions. So although there is a huge reduction in space required, it is still proportional to the number of DFA states, which grows exponentially with the number of REs in the RE set. XFAs deal with state replication using scratch memory and auxiliary code stored at each state that must be executed before or after each transition. This interesting solution models state replication; however, it cannot be fully automated [50]. Furthermore, the code that needs to be executed for each transition limits the throughput that can be achieved.

Our technique of table consolidation presented in Section 5.3 actually exploits state replication to reduce the size of TCAM required, but it does so accidentally. That is, table consolidation works well because of state replication, but the the technique is oblivious to state replication. The algorithm does not explicitly search for replicated states, it only looks for state pairs that are good matches for consolidation. But replicate states are usually good matches for consolidation, and so states that are consolidated together are usually replications of the same NFA state. There are several limitations of table consoli-

dation because of which state replication is not fully exploited. First, there is a practical limit on the number of TCAM tables that can be consolidated. For instance we only consider consolidating 4 tables together. Thus, table consolidation can only lead to a constant factor reduction in TCAM storage no matter how much state replication exists in the DFA. So the final TCAM size can still be exponential in the size of the RE set. Ideally we would like to combine together all the replications of a NFA state. Second, table consolidation does not reduce the associated SRAM required to store decisions because although the TCAM entries are merged, the decisions are not. Furthermore, the SRAM required by table consolidation might increase due to imperfect merging of tables.

6.1.2 Summary of Overlay Automata Approach

We developed a new overlay automata model which exploit state replication to compress the size of the DFA. The idea is to group together the replicated DFA structures instead of repeating them multiple times. We briefly describe here the overlay automata model and how the automata is implmented in software and hardware.

6.1.2.1 Overlay DFA

We propose *Overlay Deterministic Finite state Automata (ODFA)* that models state replication in DFAs. The basic idea is to overlay all the DFA states that are replications of the same NFA state vertically together into what we call a super-state. If we view a DFA as a 2-D object, then an ODFA can be viewed as a 3-D object. Figure 6.2 depicts

the DFA and ODFA for the RE set {/abc/, /abd/, /e.⋆f/}. The ODFA model gives us the following key benefits. *First, it allows us to easily identify replications of the same NFA state as they are all in the same super-state.* For example, in Figure 6.2, we merge states 0 and 5 and states 1 and 6 into super-states S_0 and S_1, respectively. *Second, it allows us to represent replications of the same NFA transition by one super-state transition between two super-states.* For example, for any NFA transition from s_1 to s_2 on character σ, in the corresponding ODFA, all replications of state s_1 are in the same super-state say S_1, all replications of state s_2 are in the same super-state say S_2, and all replicates of state s_1 have a transition on σ to their corresponding replicates on state s_2. We merge these replicate transitions into one combined super-state transition from super-state S_1 to super-state S_2 on character σ. For example, in Figure 6.2, we merge the two transitions from states 0 and 5 on character 'a' into one super-state transition on character 'a'.

6.1.2.2 Overlay D^{2}FA

Combining our overlay idea, which models state replication and replicated transitions, and the delayed input idea in D^{2}FA, which models sharing non-replicated transitions among non-replicated DFA states (*i.e.* transition sharing) through a state deferment relationship, we propose *Overlay Delayed Input DFA (OD^{2}FA)* to model state replication, replicated transitions, and transition sharing. The relationship among these automata models, DFA, D^{2}FA, ODFA, and OD^{2}FA, is illustrated in Figure 6.1. A key benefit of OD^{2}FA

is that we can represent the deferment relationship among D^2FA states more compactly using deferment among OD^2FA super-states. From the perspective of transitions, OD^2FA optimizes both deferred transitions (*i.e.*, common transitions among states) and replicated transitions.

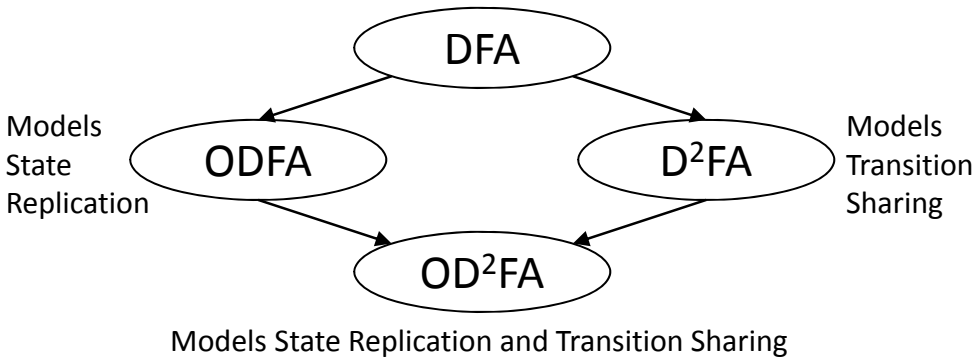


Figure 6.1: Relationship of Automata Models.

6.1.2.3 Building OD^2FA

To build an OD^2FA, we propose algorithms for constructing it from a given set of REs incrementally. We first construct the equivalent OD^2FA for each RE. We then efficiently merge OD^2FAs until only a single OD^2FA for the entire set of REs is left. We propose an incremental construction algorithm that builds the OD^2FA $\mathcal{D}$ for RE set $R_1 \cup R_2$ by merging the OD^2FA $\mathcal{D}_1$ for R_1 with the OD^2FA $\mathcal{D}_2$ for R_2. This algorithm automatically identifies and groups together replicate states in $\mathcal{D}$ into super-states and replicate transitions into super-state transitions without having to perform an expensive analysis of the final DFA structure.

6.1.2.4 Implementing OD^{2}FA

We develop techniques for implementing the OD^{2}FA is software and hardware.

We extend the software implementation of a D^{2}FA to OD^{2}FA. The main problem we need to solve is that, since an OD^{2}FA only stores super-state transitions, how do we efficiently lookup state transitions from the super-state transitions. Our efficient encoding of super-state transition facilitates in performing this lookup very quickly.

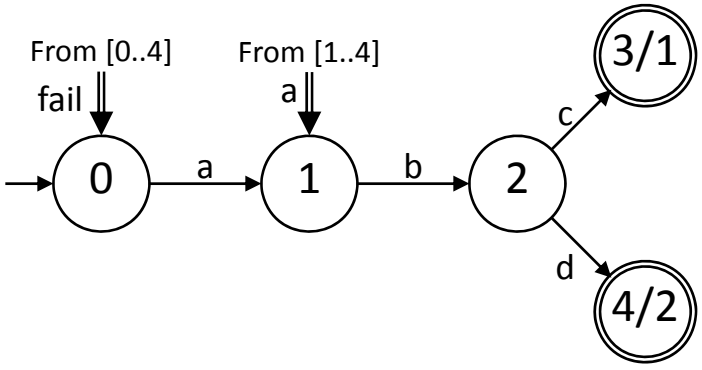
For the hardware implmentation, we develop a solution which we call OverlayCAM, by extending RegCAM to implement the OD^{2}FA in TCAM. Again, our efficient encoding of super-state transition allows us to implement each super-state transition using only one TCAM entry. So OverlayCAM not only encodes multiple deferred state transitions using one TCAM entry but also encodes multiple non-deferred state transitions that are replications of the same NFA transition using only one TCAM entry. We also extend the variable striding technique in RegCAM for use with OverlayCAM to increase the matching throughput.

6.2 Overlay DFA

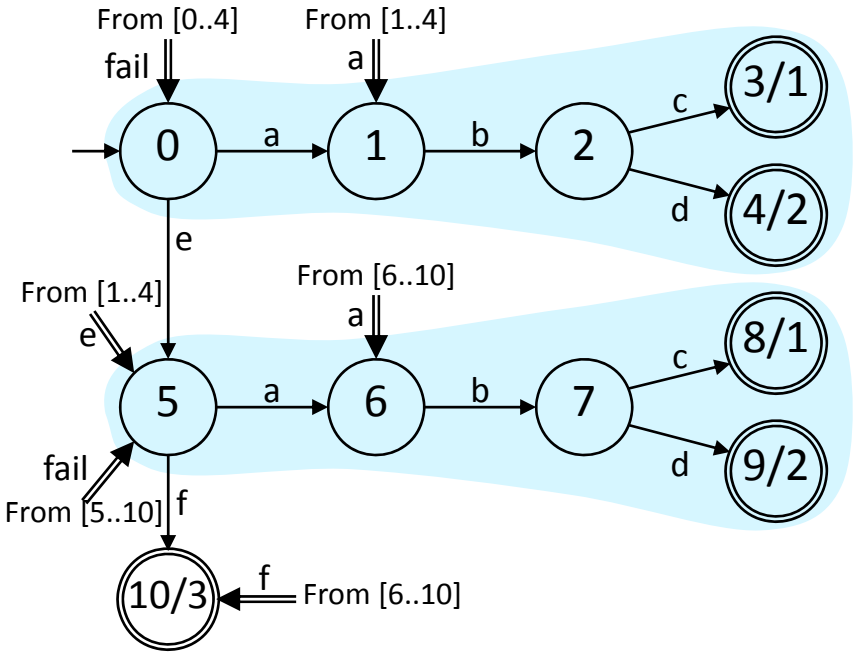
In this section, we formally define a new automata, Overlay Deterministic Finite state Automata (ODFA), which we propose to deal with state explosion in DFA.

There are two ideas behind an ODFA. The first is to group all DFA states that are repli-

cations of the same NFA state into a single super-state. The second is to merge as many transitions from the replicate states within a super-state as possible. To define ODFA, we will use the concepts of super-states, overlays, super-state transitions, and overlay offsets. We begin by informally defining ODFA and these concepts using the ODFA in Figure 6.2 as a running example.

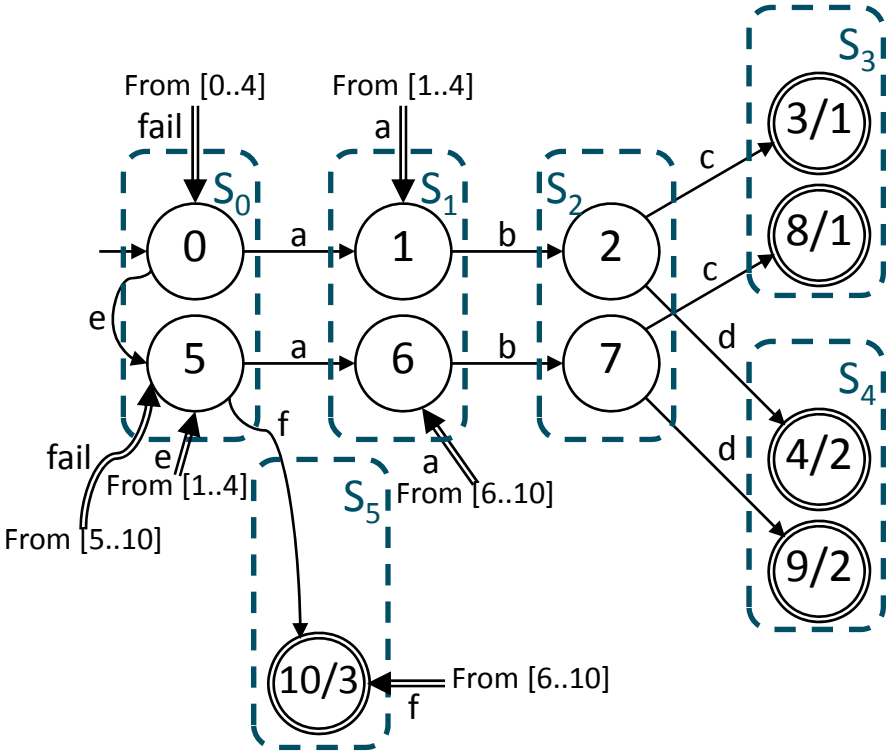


(a) DFA for RE set {/abc/, /abd/}

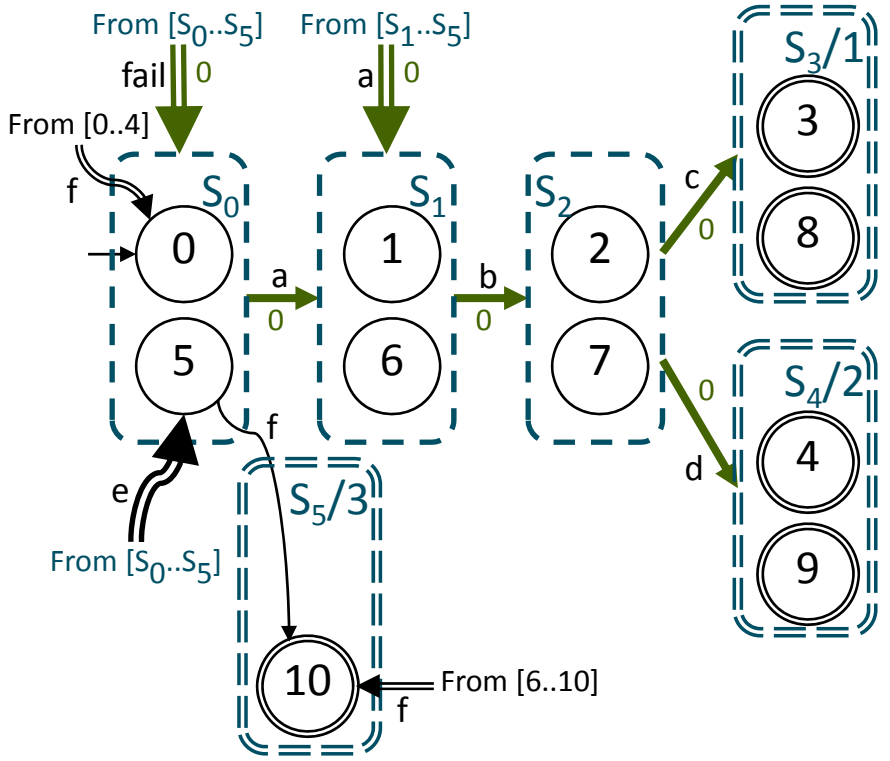


(b) DFA for RE set {/abc/, /abd/, /e.⋆f/}

Figure 6.2: Example of DFA, state replication and Overlay DFA.



(c) Corresponding ODFA



(d) ODFA with super-state transitions

Figure 6.2: Example of DFA, state replication and Overlay DFA (cont'd).

Figure 6.2(a) shows the DFA for the RE set {/abc/, /abd/} from Figure 3.1(a). The notations used in the figure are explained in Section 3.2. Figure 6.2(b) shows the DFA after the RE /e.⋆f/ is added to the RE set (same as Figure 3.1(b).) This DFA illustrates the potential for ODFA as the entire DFA for the RE set {/abc/, /abd/} is replicated twice. The corresponding ODFA is shown in Figure 6.2(c).

In Figure 6.2(c), we overlay the two copies of the DFA for the RE set {/abc/, /abd/}) on top of each other. Each pair of replicated DFA states is a *super-state* in the ODFA. Each layer of states is called an *overlay*. The ODFA in Figure 6.2(c) has six super-states $S_0, \ldots, S_5$ and two overlays. Each overlay contains a subset of the states in the entire DFA; in Figure 6.2(c), the first overlay does not contain a state from super-state S_5.

We now introduce the concept of *super-state transitions*. One super-state transition represents multiple DFA transitions much as one super-state represents a group of DFA states. In a standard DFA transition, the source state is a DFA state. In a super-state transition, the source state is an ODFA super-state and represents transitions from all the replicated DFA states within the super-state. The destination state is usually an ODFA super-state but can sometimes be a DFA state. The two super-state transition forms are $S_1 \xrightarrow{\sigma} S_2, o, 1$ and $S_1 \xrightarrow{\sigma} S_2, O, 0$ (distinguished by the last bit value 1/0). In the first form, the semantics are that each DFA state q in super-state S_1 transitions on character σ to a DFA state q' in super-state S_2, with $o = (\text{overlay of } q' - \text{overlay of } q) \bmod \#\text{overlays}$. We call this difference in the overlay value the *overlay offset* (or just offset for short.) The value of the overlay offset o is usually 0. In the second form, the semantics are that

each DFA state q in super-state S_1 transitions on character σ to the DFA state located in super-state S_2 at overlay O. For example, consider the two DFA state transitions $1 \xrightarrow{b} 2$ and $6 \xrightarrow{b} 7$ in Figure 6.2(c). These two transitions can be represented by one super-state transition $S_1 \xrightarrow{b} S_2, 0$; the 0 denotes no change in overlay. As a second example, consider the two DFA state transitions $3 \xrightarrow{e} 5$ and $8 \xrightarrow{e} 5$ in Figure 6.2(c). These two transitions can be represented by one super-state transition $S_3 \xrightarrow{e} S_2, 1, 0$.

In the ideal case, all DFA transitions can be replaced by super-state transitions which reduces the total number of transitions by the number of overlays in the ODFA. In some cases, not all states in a super-state have transitions that can be merged. We generalize super-state transitions to allow super-state transitions to be defined for a specific subset of overlays X within a given super-state. Technically, traditional transitions from a single state s are super-state transitions where X contains only s's overlay. We refer to these as *singleton super-state transitions.*

Figure 6.2(d) shows the ODFA for our running example with non-singleton super-state transitions denoted with thick edges. For example, the two transitions $0 \xrightarrow{a} 1$ and $5 \xrightarrow{a} 6$ from Figure 6.2(c) are represented with one super-state transition $S_0 \xrightarrow{a} S_1, 0, 1$. For super-state transitions of the form $S_1 \xrightarrow{\sigma} S_2, o, 1$ (*i.e.* destination is also a super-state), the number besides the thick edge gives the overlay offset o. As we use double arrows to represent multiple transitions, we use thick double arrows to represent multiple non-singleton super-state transitions. For example, the two transitions $0 \xrightarrow{e} 5$ and $5 \xrightarrow{e} 5$ from Figure 6.2(c) are included in one super-state transition $S_0 \xrightarrow{e} S_0, 1, 0$ which is part of the

thick double arrow labeled with 'e' ending at state 5. The DFA in Figure 6.2(b) has $11 \times 256 = 2816$ total transitions; the ODFA in Figure 6.2(d) has 1542 total super-state transitions which is close to the best possible result of $2816/2 = 1408$ total super-state transitions; only a few of these transitions are singleton super-state transitions.

Recall the DFA is defined as a 5-tuple $(Q, \Sigma, q_0, M, \delta)$ (Section 3.1). We now formally define the ODFA.

Definition 5 (Overlay Deterministic Finite state Automata (ODFA)). *An ODFA for a set of REs $\mathcal{R}$ is defined as a 7-tuple $\mathcal{D} = (Q, \Sigma, q_0, \mathcal{S}, \mathcal{O}, \mathcal{M}, \Delta)$. The first three terms are the same as those in the above DFA definition.*

The next two terms define the overlay structure on top of a DFA: $\mathcal{S} = \{S_0, \ldots, S_{|\mathcal{S}|-1}\}$ is a set of super-states that partitions Q, while $\mathcal{O} = \{O_0, \ldots, O_{|\mathcal{O}|-1}\}$ is a set of overlays that also partitions Q. We shall treat each overlay as a unique number in the range $[0..|\mathcal{O}|)$. We overload notation and define $\mathcal{S}: Q \to \mathcal{S}$ and $\mathcal{O}: Q \to \mathcal{O}$ as functions mapping states to super-states and overlays, respectively. For any two states $s_i \neq s_j$, it must be the case that $(\mathcal{S}(s_i), \mathcal{O}(s_i)) \neq (\mathcal{S}(s_j), \mathcal{O}(s_j))$. For any super-state S and overlay O, $S \cap O$ is either empty or contains one state $s \in Q$.

The term $\mathcal{M}: \mathcal{S} \to 2^{\mathcal{R}}$ gives the subset of REs matched by any super-state. The set of REs matched by any state $s \in Q$ is then given by $\mathcal{M}(\mathcal{S}(s))$. The final term $\Delta: \mathcal{S} \times 2^{\mathcal{O}} \times \Sigma \to \mathcal{S} \times [0..|\mathcal{O}|) \times \{0, 1\}$ is a partial function and defines the super-state transition function. For any $s \in Q$ and any $\sigma \in \Sigma$, all the transition $(\mathcal{S}(s), X, \sigma) \in \mathrm{dom}(\Delta)$ with $\mathcal{O}(s) \in X$ must have the same value; i.e. *if we have two transitions*

$(\mathcal{S}(s), X, \sigma) \in \mathrm{dom}(\Delta)$ *and* $(\mathcal{S}(s), Y, \sigma) \in \mathrm{dom}(\Delta)$, *with* $\mathcal{O}(s) \in X \cap Y$, *then we must have* $\Delta(\mathcal{S}(s), X, \sigma) = \Delta(\mathcal{S}(s), Y, \sigma)$. *We define the derived total state transition function* $\delta''(s, \sigma)$ *based on this unique transition value, say* (S', o, b), *as follows. First, if* $b = 0$, *we call the transition a* non-offset transition, *and* $\delta''(s, \sigma) = S' \cap o$. *Otherwise (*$b = 1$*), we call the transition an* offset transition, *and* $\delta''(s, \sigma) = S' \cap ((\mathcal{O}(s) + o) \bmod |\mathcal{O}|)$. *The value* b *is called the* offset bit. *It must be the case that overlay* $(\mathcal{O}(s) + o) \bmod |\mathcal{O}|$ *does intersect* S'. *Normally for offset transitions* $o = 0$, *so the resulting overlay is just* $\mathcal{O}(s)$.

We use the notation $(S_1, O) \xrightarrow{\sigma} (S_2, o, b)$ to denote the super-state transition $\Delta(S_1, O, \sigma) = (S_2, o, b)$. Even though an ODFA has super-states and overlays, an ODFA processes an input string much like a DFA does. That is, the ODFA is always in a unique state and each character processed moves the ODFA to a potentially new state. The main difference is that the ODFA hopefully compresses multiple DFA transitions into a single ODFA super-state transition, and the RE matching information is stored at the super-state level rather than at the state level. For example, given the ODFA in Figure 6.2(d) and the input string `abea`, the ODFA begins in state 0. After processing character `a`, the ODFA moves to state 1. After processing character `b`, the ODFA moves to state 2. After processing character `e`, the ODFA moves to state 5. Finally, after processing character `a`, the ODFA moves to state 6. The first and fourth transitions are actually the same super-state transition. The third transition corresponds to the first form of super-state transition with specified destination state 5. In all cases, $\mathcal{M}(\mathcal{S}(s')) = \emptyset$, so no RE is matched at any point in time.

Overlays and super-states are two orthogonal partitionings of states in Q; intuitively, super-states partition Q vertically and overlays partition Q horizontally. There exist many possible ways to partition the states of a DFA into super-states and overlays. The benefits of an ODFA are only realized by a careful partitioning; for example, grouping replicate states of the same NFA state together in a super-state. Note that some super-states may not have DFA states in each overlay. If overlay O in super-state S is empty, we denote it by $S \cap O = \perp$ (*i.e.* $\perp$ denotes an empty location). In Figure 6.2(d), super-state S_5 contains only one DFA state 10 which belongs to the second overlay. The compressive power of a super-state transition increases with the number of overlays that it includes. In the best case, all overlays are included in a super-state transition. In Figure 6.2(d), most super-state transitions include all overlays; there are only a few singleton super-state transitions. In more complex ODFA, there may be cases where a given super-state transition includes more than one overlay but not all overlays.

In an ODFA the RE matching is stored at the super-state level (*i.e.* $\mathcal{M}$) and state matching is defined by $\mathcal{M}$. So when constructing an ODFA $\mathcal{D}$ for a given DFA D, we must create the super-states such that the following condition is satisfied

$$\forall S \in \mathcal{S}_{\mathcal{D}}, \forall s_1, s_2 \in S,\ M_D(s_1) = M_D(s_2), \tag{C1}$$

6.3 Overlay D^{2}FA

In this section we present another new automata, Overlay Delayed Input DFA (OD^{2}FA), which we propose to deal with both state and transition explosion in DFA.

Recall that, given a DFA $D=(Q, \Sigma, q_0, M, \delta)$, its corresponding D^{2}FA D' is defined as a 6-tuple $(Q, \Sigma, q_0, M, \rho, F)$ (Section 3.3).

ODFAs address state explosion and D^{2}FAs address transition explosion. We propose OD^{2}FA to address both state and transition explosion in DFAs.

Definition 6 (Overlay D^{2}FA (OD^{2}FA)). *We define an OD^{2}FA as an 8-tuple* (Q, Σ, q_0, $\mathcal{F}$, $\mathcal{S}$, $\mathcal{O}$, $\mathcal{M}$, Δ), *where the first three terms are the same as in defining D^{2}FA. The last four terms are the same as in defining ODFA. The only difference is that, we derive a partial state transition function* $\rho' : Q \times \Sigma \rightarrow Q$ *from* Δ. *Since* ρ' *is a partial function, we do not require the existence of a covering transition in* Δ *for each* $s \in Q$ *and* $\sigma \in \Sigma$. $\mathcal{F} : \mathcal{S} \rightarrow \mathcal{S}$ *is the super-state deferment function, and gives the deferred super-state for each super-state. We define the D^{2}FA state deferment function* F *from* $\mathcal{F}$ *as* $F(s) = \mathcal{F}(\mathcal{S}(s)) \cap \mathcal{O}(s))$. *To ensure this is a valid deferment function,* $\mathcal{F}$ *must satisfy the following two conditions. First,*

$$\forall s \in Q, \mathcal{F}(\mathcal{S}(s)) \cap \mathcal{O}(s)) \neq \perp, \tag{C2}$$

Second, the deferment forest of super-states defined by $\mathcal{F}$ *has no cycles other than self-loops. Finally,* ρ' *and* F *define the derived total state transition function* δ'' *as*

follows.

$$\delta''(s,\sigma) = \begin{cases} \rho'(s,\sigma) & \textit{if } \langle s,\sigma\rangle \in \mathrm{dom}(\rho') \\ \delta''(F(s),\sigma) & \textit{else} \end{cases}$$

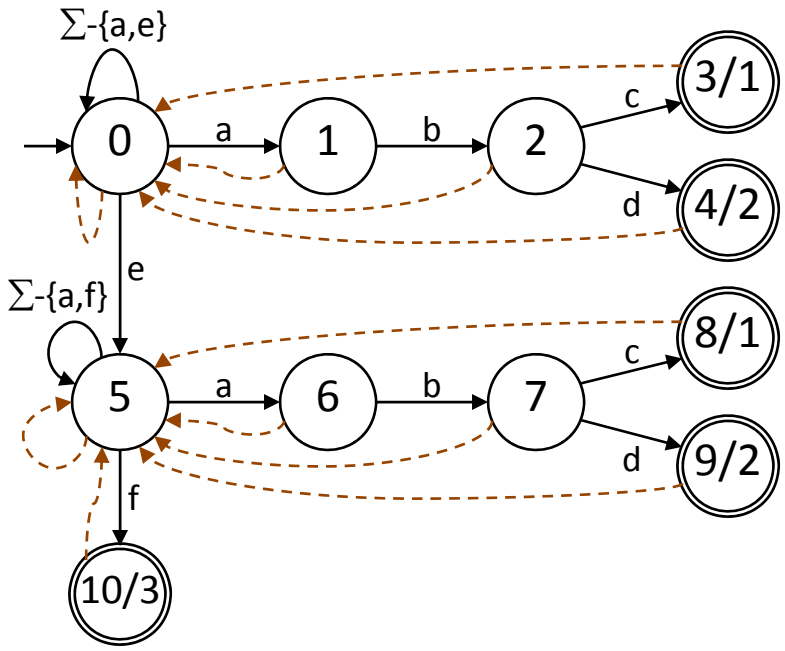
We say that $\langle s,\sigma\rangle \in \mathrm{dom}(\rho')$ *if there exists a transition* $(\mathcal{S}(s),X,\sigma) \in \Delta$ *with* $\mathcal{O}(s) \in X$. *If* $\langle s,\sigma\rangle \in \mathrm{dom}(\rho')$, *then* $\rho'(s,\sigma)$ *is defined as* δ'' *is defined for ODFA.*

We say that super-state S *overlay covers* super-state S' if $\forall O \in \mathcal{O}, (S \cap O = \perp) \rightarrow (S' \cap O = \perp)$. That is, every overlay that is empty in S is also empty in S'. Then, Condition (C2) says that for every super-state S, super-state $\mathcal{F}(S)$ overlay covers S.

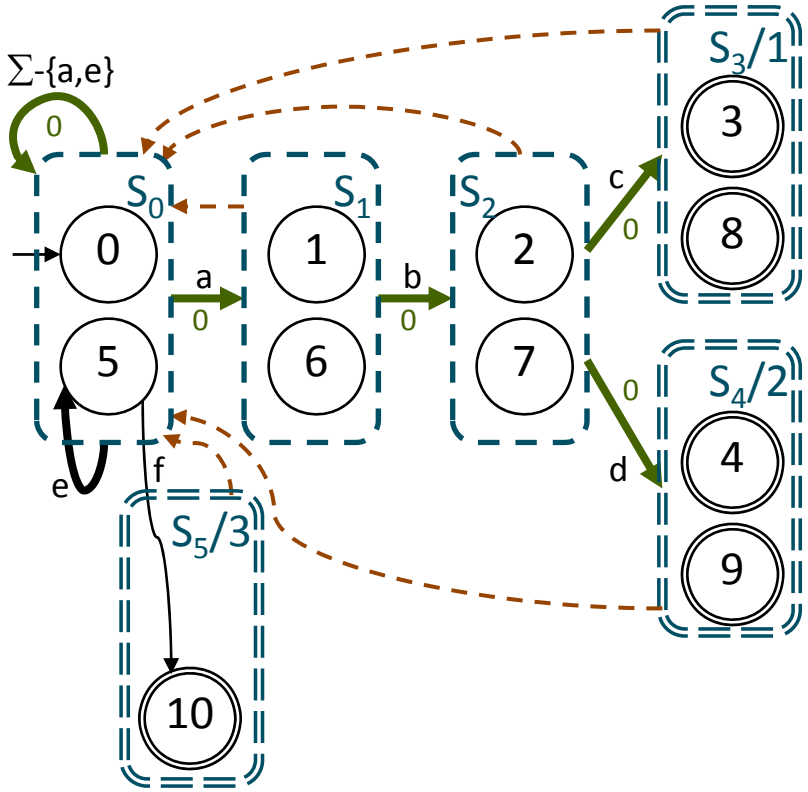
The transition function δ'' is computed by finding the transition $(\mathcal{S}(s),X,\sigma) \in \Delta$ with $\mathcal{O}(s) \in X$ if such a transition exists. If not, the OD^{2}FA follows the super-state deferment function.

As defined, we store $\mathcal{F}$ rather than F; thus deferment information is stored only at the super-state level. Likewise, we store just RE matching information $\mathcal{M}$ at the super-state level. Finally, with Δ, many super-state transitions represent multiple singleton transitions. Combined, we can achieve significant savings.

Figure 6.3(a) shows the D^{2}FA for the RE set {`/abc/`, `/abd/`, `/e.*f/`}. The dashed edges are deferment transitions. Figure 6.3(b) shows the corresponding OD^{2}FA. The D^{2}FA needs to store 518 actual transitions and 10 deferment transitions while the OD^{2}FA only needs to store 260 actual transitions, most of which are non-singleton super-state transitions, and 5 super-state deferred transitions. For this example, we achieve near optimal compression



(a) D^{2}FA for RE set {/abc/, /abd/, /e.⋆f/}



(b) Corresponding OD^{2}FA

Figure 6.3: OD^{2}FA Example.

given only two overlays in the OD^{2}FA when compared to the D^{2}FA.

6.3.1 OD^{2}FA Multiplicative Compression

OD^{2}FA multiplies the compressive effect of D^{2}FA and ODFA to significantly reduce the space required to store transitions. ODFA reduces the storage space for transitions *among DFA replicates* by storing one super-state transition for each replicated transition. The compression limit for ODFA is the number of DFA replicates. D^{2}FA reduces the storage space for transitions *within each DFA replicate* using deferment transitions. The compression limit for D^{2}FA is the number of states within each DFA replicate. OD^{2}FA is able to do both simultaneously. The compression limit is the number of DFA replicates multiplied by the number of states within each replicate which is essentially the total number of DFA states.

To illustrate this multiplicative compression, consider again the OD^{2}FA in Figure 6.3(b). The original DFA for this RE set requires $11 \times 256 = 2816$ transitions. The corresponding ODFA in Figure 6.2(d) is able to reduce the number of transitions by almost a factor of 2 by storing one super-state transition for each pair of replicated transitions. The corresponding D^{2}FA in Figure 6.3(a) is able to reduce the number of transitions by more than a factor of 5 using deferment transitions. In particular, in both replicates, almost all of the transitions for all states except the self-looping start states are eliminated. Finally, the OD^{2}FA in Figure 6.3(b) multiplies both effects and ends up with 260 super-state transitions and 5 super-state deferment transitions. This is almost a factor of 11 times smaller than the

original DFA where 11 is the compression limit since the DFA has 11 states. Starting from the D^2FA, the OD^2FA is able to replicate all the self-looping transitions out of the two self-looping states in the D^2FA (adding one singleton transition on '`f`' for state 5). This is critical since the vast majority of transitions remaining in many D^2FA are self-looping transitions.

6.3.2 Effectiveness of OD^2FA on Ideal RE set

We can further demonstrate the effectiveness of OD^2FA using an example set of n REs where each RE is of the form $/A_{i,1}A_{i,2}\cdots A_{i,p}.\star B_{i,1}B_{i,2}\cdots B_{i,p}/$, $1 \leq i \leq n$; that is, each RE has p characters followed by '$.\star$' and another p characters and all the $2np$ characters are unique. This is a simple RE set, in the sense that there is no interaction between the REs in the set, and we get a simple exponential increase in the size of the DFA relative to the number of REs in the set n because of state replication.

In this case, the NFA has $(2p+1)n+2$ $(O(pn))$ states and the DFA has $((2p-1)n+2)2^{n-1}$ $(O(pn2^n))$ states. The D^2FA has $((p-1)n+256)2^n$ $(O(pn2^n))$ transitions, and our RegCAM presented in Section 5.2 will generate $(pn+1)2^n$ $(O(pn2^n))$ TCAM entries. The OD^2FA only has $pn+1$ $(O(pn))$ super-states, $2pn+256$ $(O(pn))$ super-state transitions, and a straightforward TCAM implementation of these transitions needs only $2pn+1$ $(O(pn))$ TCAM entries. The number of rules with the OD^2FA is the same as the NFA size, which is a lower bound on the compression any method can achieve.

6.4 OD^{2}FA Construction

In this section we present our algorithms for constructing an OD^{2}FA for a set of REs. Given a set of REs, we construct its equivalent OD^{2}FA incrementally in two phases. In the first phase, we construct an equivalent individual OD^{2}FA for each RE. In the second phase, we merge all the individual OD^{2}FAs in a binary tree fashion; *i.e.* we merge two OD^{2}FAs into one OD^{2}FA at a time until there is only one OD^{2}FA for the entire given RE set.

Constructing an OD^{2}FA involves three main steps: (1) creating the super-states (*i.e.* assigning a super-state, overlay pair for each DFA state), (2) setting the deferment for each super-state and (3) for each super-state creating the (combined) super-state transitions from the (singleton) state transitions. The algorithms for the first two steps (creating super-states and setting deferment) are different for the two phases mentioned above. However the algorithms for the third step (creating super-state transitions) are almost identical for the two phases. So we describe the OD^{2}FA construction algorithms in two parts. In this section we demonstrate how the super-states are created and how super-state deferment is set (*i.e.* steps 1 and 2) during both the phases. In the next section we show how super-state transitions are built from state transitions (*i.e.* step 3).

6.4.1 OD^{2}FA Construction from One RE

Given one RE, we first build its equivalent D^{2}FA using the technique described in Section 4.3.1. The deferment relationship among states in this D^{2}FA defines a deferment forest. The root states in this forest are all self-looping states which means they transit to themselves for more than $|\Sigma|/2 = 128$ characters. Most failure transitions end in self-looping states. For example, in the D^{2}FA in Figure 6.4, states 0 and 2 are self-looping states. An important property of the D^{2}FA constructed using the technique described in Section 4.3.1 is that each self-looping state in the DFA is the root of a tree in the deferment forest of the D^{2}FA, and vice versa. Furthermore, all the states whose failure transitions go to a self-looping state s are in the deferment tree rooted at s.

Now we describe our algorithm for constructing the OD^{2}FA from a D^{2}FA using the example in Figure 6.4 for the RE `/ab[^n]*pq/`. A key observation is that any D^{2}FA is also a valid OD^{2}FA with only a single overlay, singleton super-states, and singleton super-state transitions. We gradually convert the D^{2}FA into a more compact OD^{2}FA first creating valid overlays and super-states and then updating the super-state transition function to combine multiple transitions into one super-state transition.

We begin by specifying the number of deferment trees in the super-state deferment forest and the number of overlays in a super-state. We accomplish these tasks by partitioning the self-looping root states of the D^{2}FA into two groups, accepting root states and rejecting root states. If either partition is empty, we create one deferment tree in the OD^{2}FA; otherwise there are two deferment trees. The number of overlays in the OD^{2}FA is the

larger of the number of accepting root states and the number of rejecting root states. For any non-empty partition, we merge the root states in that partition into a single root super-state in the OD^{2}FA. Typically, self-looping states are failure states, so the accepting root state partition is empty and the resulting root super-state is not formed. This observation holds for all of our experimental RE sets. Thus, the deferment forest of the OD^{2}FA typically has one deferment tree rooted at the rejecting root super-state. For example, the OD^{2}FA in Figure 6.4 has one deferment tree with two overlays, 0 and 1, and the rejecting root super-state is $\boxed{0}\boxed{2}$.

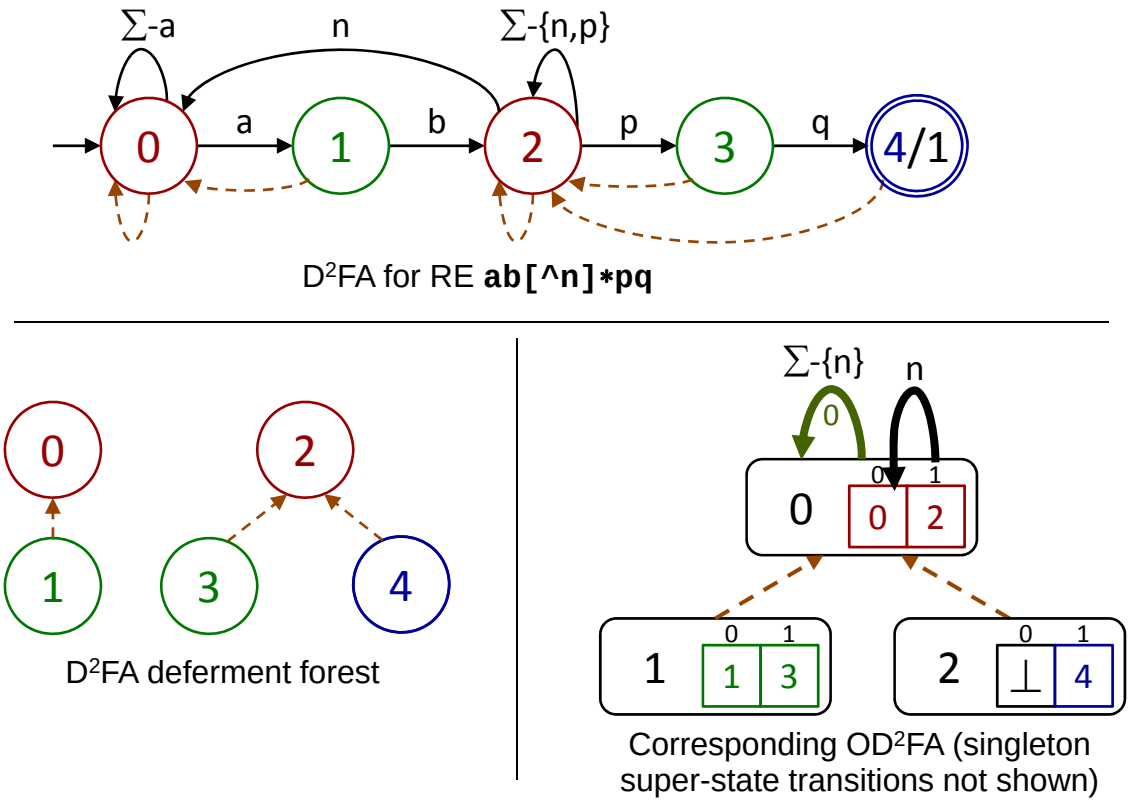


Figure 6.4: OD^{2}FA construction from one RE.

There are two reasons we group root states into super-states even though the self-looping states in the D^{2}FA are usually not replications of the same NFA state. First, all the com-

mon self-loops can be merged into super-state transitions. We specify this more precisely in Section 6.5. Second, as self-looping states are typically the "replication points" when combining REs, grouping self-looping states into a common super-state helps us automatically identify the state replications and replicated transitions when we merge two OD^{2}FAs. We elaborate this more in Section 6.4.2. Condition (C2) is satisfied as the root super-state defers to itself.

We now describe how we assign the remaining states to super-states and overlays ensuring Condition (C2) is maintained. Given a super-state S that is in the OD^{2}FA deferment forest, our algorithm groups the children of the states in S into new super-states that defer to S. This grouping is recursively applied to the new super-states formed until all states are assigned to super-states. We now specify how the children of the states of S are grouped into super-states. Let n be the number of non-empty overlays in S, and let $s_1, \ldots, s_n$ be the states in these overlays. Let $C_i = F^{-1}(s_i)$ be the set of children for each state s_i in S, and let $U = \bigcup_{i=1}^{n} C_i$ be the total set of states to be grouped into super-states. To ensure all states in a super-state match the same REs, we partition U into accepting states and rejecting states and work with each partition independently. Without loss of generality, we assume U has one partition. We create super-states with the following two goals in mind: grouping together states $u \in U$ from different C_i to (1) maximize the number of super-state transitions that can be formed and (2) minimize the total number of super-states formed.

We propose the following greedy strategy. We start with an arbitrary state u from the first

non-empty C_i removing u from C_i and creating super-state S' with just u in $\mathcal{O}(s_i)$. From each of the remaining non-empty C_k, we pick the state u_k that has the most common non-deferred transitions with u, delete u_k from C_k, and add u_k to super-state S' in $\mathcal{O}(s_k)$. State u_k must have at least one common non-deferred transition with u to be selected. We repeat this process until all the C_i are empty. Condition (C2) is maintained because a state s' in a super-state S' is added to overlay O if and only if the corresponding state s in $\mathcal{F}(S)$ is in overlay O. For the D^{2}FA in Figure 6.4 with root super-state $\boxed{0}\boxed{2}$ as S, we have $C_0 = \{1\}$ and $C_1 = \{3, 4\}$, and we create three super-states, $\boxed{1}\boxed{\perp}$, $\boxed{\perp}\boxed{3}$ and $\boxed{\perp}\boxed{4}$, each of which defers to $\boxed{0}\boxed{2}$. No super-states with more than one overlay occupied are formed because states 1 and 3 as well as 1 and 4 do not have any common non-deferred transitions.

After the super-states have been created, we greedily merge together compatible pairs of super-states. Two super-states are *compatible* if there is no overlay that is non-empty in both super-states. For our example in Figure 6.4, the super-states $\boxed{1}\boxed{\perp}$ and $\boxed{\perp}\boxed{3}$ will be merged together, giving us two final super-states $\boxed{1}\boxed{3}$ and $\boxed{\perp}\boxed{4}$.

The last step is to create the super-state transitions which is discussed in Section 6.5.

We use greedy algorithms in several of our steps. This does not have much effect on overall compression because most compression opportunities are accidental; they are not the result of replications of the same NFA state. The key compression that is attained results from grouping the root states together and combining the resulting self-loops into super-state transitions; everything else is a bonus.

6.4.2 OD^{2}FA Construction from 2 OD^{2}FAs

We present our OD^{2}FA merge algorithm, which we call *OD2FAMerge*, that constructs OD^{2}FA $\mathcal{D}_3$ with underlying D^{2}FA D_3 for the RE set $R_3 = R_1 \cup R_2$ given two OD^{2}FAs, $\mathcal{D}_1$ with underlying D^{2}FA D_1 for RE set R_1 and $\mathcal{D}_2$ with underlying D^{2}FA D_2 for RE set R_2 where $R_1 \cap R_2 = \emptyset$.

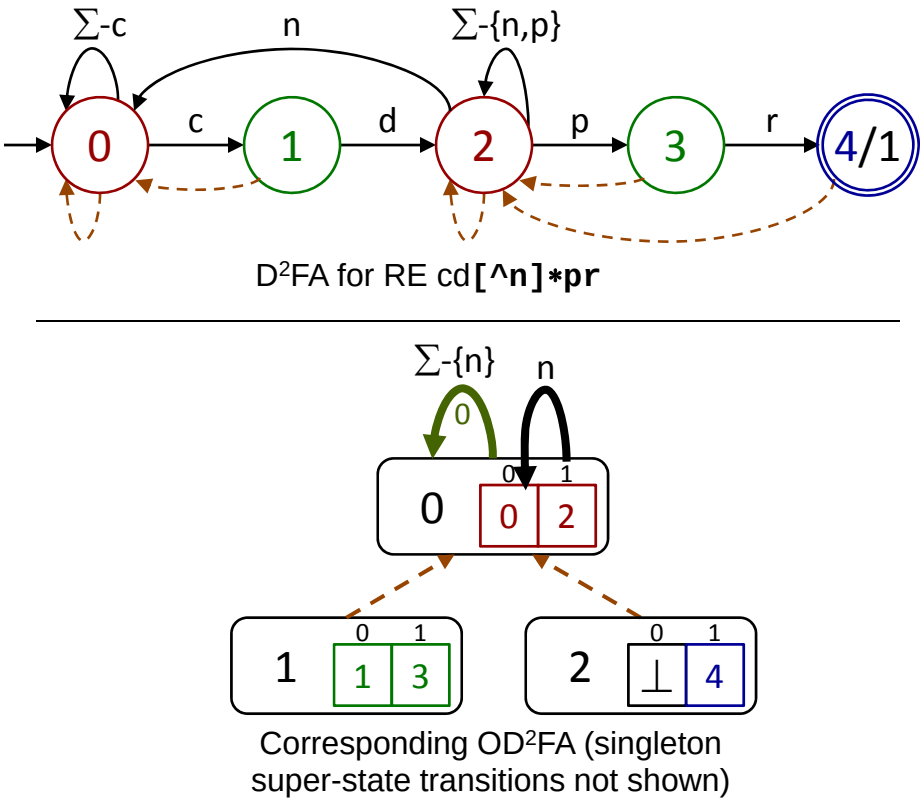
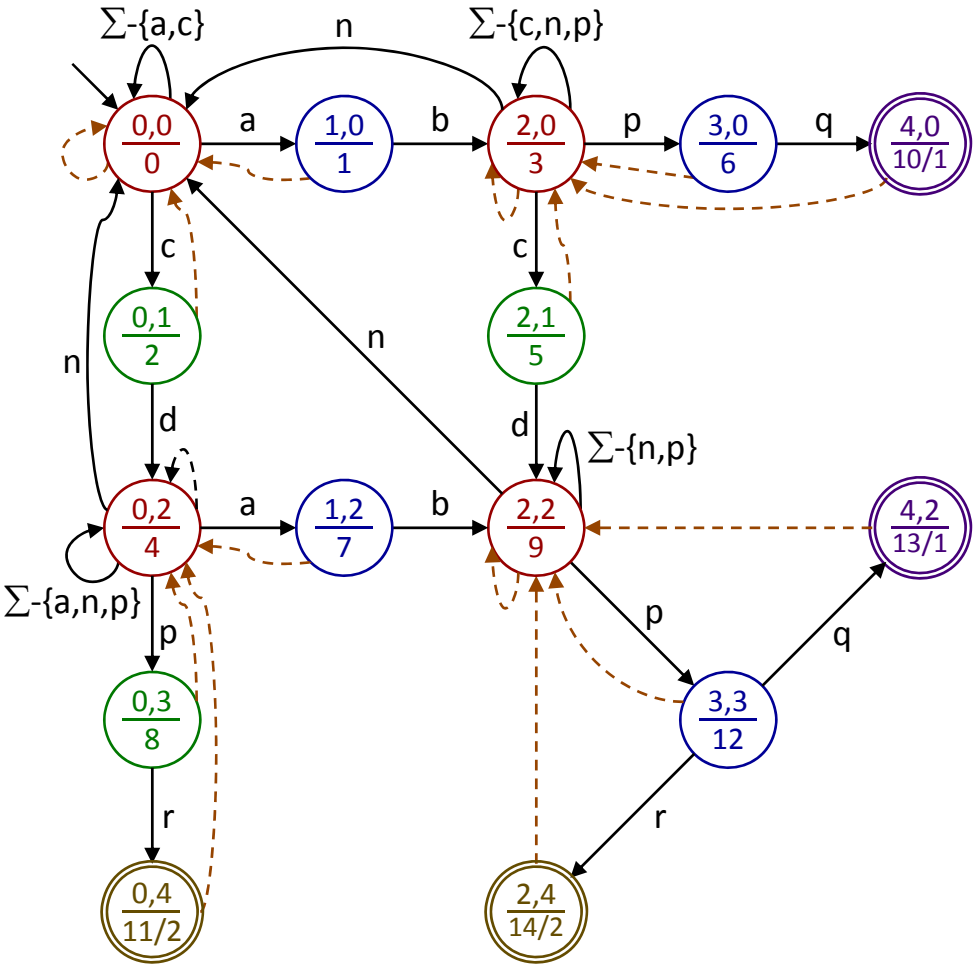


Figure 6.5: D^{2}FA and OD^{2}FA for RE `/cd[^n]*pr/`.

The first step is to create the merged D^{2}FA D_3 using the the D^{2}FA merge algorithm described in Section 4.3.2. For example, Figure 6.6(a) shows the D^{2}FA constructed from the D^{2}FAs in Figure 6.4 and Figure 6.5. For each state, the number below the line is the state id in D_3 and the two numbers above the line are the state ids of the states in D_1

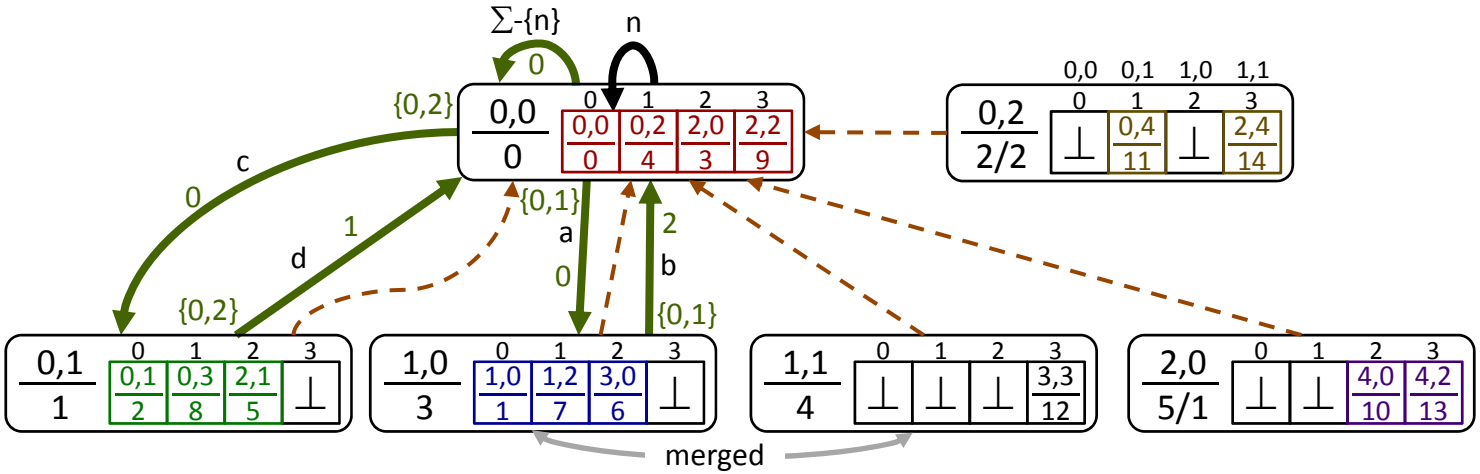
and D_2 that this state corresponds to.

We now construct OD^{2}FA $\mathcal{D}_3 = (Q_3, \Sigma, q_{03}, \mathcal{F}_3, \mathcal{S}_3, \mathcal{O}_3, \mathcal{M}_3, \Delta_3)$ from the input OD^{2}FAs $\mathcal{D}_1 = (Q_1, \Sigma, q_{01}, \mathcal{F}_1, \mathcal{S}_1, \mathcal{O}_1, \mathcal{M}_1, \Delta_1)$ and $\mathcal{D}_2 = (Q_2, \Sigma, q_{02}, \mathcal{F}_2, \mathcal{S}_2, \mathcal{O}_2, \mathcal{M}_2, \Delta_2)$ as well as the merged D^{2}FA D_3. The first three terms in $\mathcal{D}_3$ are derived from D_3. We then set $\mathcal{S}_3 = \mathcal{S}_1 \times \mathcal{S}_2$ and $\mathcal{O}_3 = \mathcal{O}_1 \times \mathcal{O}_2$. We reduce $\mathcal{S}_3$ to only include reachable super-states (a super-state is reachable if it contains at least one reachable state). We discuss how we handle empty overlays in Section 6.5.4.

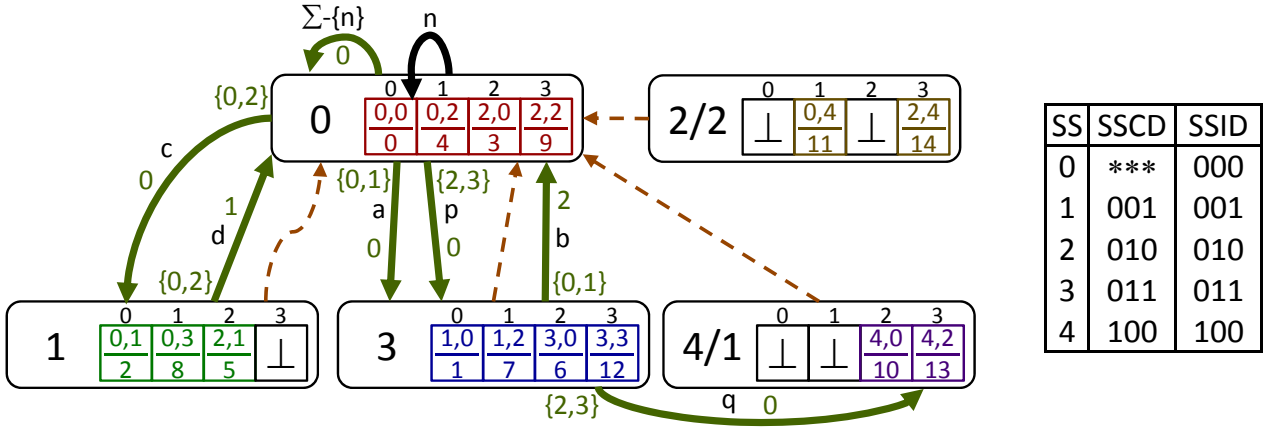


(a) D^{2}FA merged from D^{2}FAs in Figures 6.4 and 6.5.

Figure 6.6: Merged OD^{2}FA construction example.



(b) OD^{2}FA merged from OD^{2}FAs in Figures 6.4 and 6.5.



SS	SSCD	SSID
0	***	000
1	001	001
2	010	010
3	011	011
4	100	100

(c) Corresponding optimized OD^{2}FA.

Figure 6.6: Merged OD^{2}FA construction example (cont'd).

Recall that the notation $S_3{=}\langle S_1, S_2\rangle$ means super-state S_3 in $\mathcal{D}_3$ corresponds the pair of super-states super-state S_1 from $\mathcal{D}_1$ and S_2 from $\mathcal{D}_2$. Both S_3 and $\langle S_1, S_2\rangle$ refer to the same super-state in $\mathcal{D}_3$. Then for any super-state $S_3{=}\langle S_1, S_2\rangle \in \mathcal{S}_3$, we set $\mathcal{M}_3(S_3) = \mathcal{M}_1(S_1) \cup \mathcal{M}_2(S_2)$. Condition (C1) holds because all the states in super-state S_1 match the REs in $\mathcal{M}_1(S_1)$ and all the states in super-state S_2 match the REs in $\mathcal{M}_2(S_2)$.

Just as each state in D_3 ($\mathcal{D}_3$) corresponds to a pair of states from D_1 ($\mathcal{D}_1$) and D_2 ($\mathcal{D}_2$), each super-state in $\mathcal{D}_3$ will correspond to a pair of super-states from $\mathcal{D}_1$ and $\mathcal{D}_2$, and similarly each overlay in $\mathcal{D}_3$ will correspond to a pair of overlays from $\mathcal{D}_1$ and $\mathcal{D}_2$. Any state in $\mathcal{D}_3$ is assigned to a super-state and an overlay as follows. Let $u{=}\langle v, w\rangle$ be a state in $\mathcal{D}_3$. Then $\mathcal{S}_3(u) \leftarrow \langle \mathcal{S}_1(v), \mathcal{S}_2(w)\rangle$ and $\mathcal{O}_3(u) \leftarrow \langle \mathcal{O}_1(v), \mathcal{O}_2(w)\rangle$. That is, we assign u to the super-state (overlay) that corresponds to the pair of super-states (overlay) that v and w belong to in $\mathcal{D}_1$ and $\mathcal{D}_2$ respectively.

Figure 6.6(b) shows the OD^{2}FA $\mathcal{D}_3$ constructed from OD^{2}FA $\mathcal{D}_1$ in Figure 6.4 and OD^{2}FA $\mathcal{D}_2$ in Figure 6.5. In this figure, for each super-state, the number below the line is the super-state ID in $\mathcal{D}_3$ and the pair numbers above the line are the super-state IDs of the super-states in $\mathcal{D}_1$ and $\mathcal{D}_2$ that this super-state corresponds to. For instance, consider state 7 in $\mathcal{D}_3$, which corresponds to state 1 in $\mathcal{D}_1$ and state 2 in $\mathcal{D}_2$. As we can see from Figures 6.4 and 6.5, state $1 \in \mathcal{D}_1$ belongs to super-state 1 and overlay 0, and state $2 \in \mathcal{D}_2$ belongs super-state 0 and overlay 1. Therefore, in OD^{2}FA $\mathcal{D}_3$, we assign state 7 to super-state 3, which corresponds to super-state 1 from $\mathcal{D}_1$ and super-state 0 from $\mathcal{D}_2$; similarly, we assign state 7 to overlay 1, which corresponds to overlay 0 from $\mathcal{D}_1$ and

overlay 1 from $\mathcal{D}_2$. In Figure 6.6(b), the input character and overlay offset are shown along each super-state transition. For super-state transitions that do not include all the overlays in the super-state, the set of numbers at the base of the transition gives the included overlays.

We define the super-state deferment relationship $\mathcal{F}_3$ as follows: for any super-state S, which contains one or more states in Q_3, we defer it to the super-state that contains most of the states that the states in S defer to; *i.e.*, $\forall S \in \mathcal{S}$, $\mathcal{F}_3(S) \leftarrow \text{mode}(\{\mathcal{S}_3(F_3(u)) \mid u \in S\})$. After defining $\mathcal{F}_3$, we need to adjust the deferment relationship F for D^{2}FA D_3. Specifically, for each state s in a super-state S where S defers to super-state S', we let s defer to state s' in S' where s and s' are in the same overlay if $s' \neq \perp$. If $s' = \perp$, we split S into two super-states $S_1 = S \setminus \{s\}$ and $S_2 = \{s\}$, where S_2 defers to the super-state that contains the state that s defers to (*i.e.*, $\mathcal{F}_3(S_2) := \mathcal{S}_3(F_3(s))$). Note that the case that $s' = \perp$ rarely happens in our experimental RE sets. This super-state splitting ensures that Condition (C2) holds for $\mathcal{D}_3$.

We show how the super-state transitions are created for the merged OD^{2}FA Section 6.5.

Pseudo-code for our OD2FAMerge algorithm is given in Algorithm 6.7.

We now consider the following optimization for $\mathcal{D}_3$. Among the super-states that defer to the same super-state, we merge two compatible super-states into one super-state if merging them results in more super-state transitions. This will commonly be the case when we lose a D^{2}FA state we expect to generate from a self-looping state. For example, in D^{2}FA Figure 6.6(a), we lost the expected states $\langle 2, 3\rangle$ and $\langle 3, 2\rangle$ getting instead state $12 = \langle 3, 3\rangle$.

Input: OD^{2}FAs, $\mathcal{D}_1$ and $\mathcal{D}_2$, with underlying D^{2}FAs D_1 and D_2, corresponding to RE sets R_1 and R_2.

Output: An OD^{2}FA and its underlying D^{2}FA corresponding to the RE set $R_1 \cup R_2$.

Let $D_3 \leftarrow$ `D2FAMerge`(D_1, D_2) `// algorithm from Section 4.3.2`
Set #overlays in $\mathcal{D}_3$, $|\mathcal{O}_3| = n \leftarrow |\mathcal{O}_1| \times |\mathcal{O}_2|$;
foreach $S_i \in \mathcal{S}_1 \times S_j \in \mathcal{S}_2$ **do** `// Create the super-states`
    Initialize super-state $S{=}\langle S_i, S_j\rangle$ with n NULL states;
    foreach $O_k{\in}\mathcal{O}_1, 0{\leq}\, k{<}\, |\mathcal{O}_1| \times O_l \in \mathcal{O}_2, 0{\leq}\, l{<}\, |\mathcal{O}_2|$ **do**
        if *state* $s{=}\langle S_i \cap O_k, S_j \cap O_l\rangle \in Q_3$ **then**
            Assign s to overlay $O_{(k\times|\mathcal{O}_2|+l)}$ in super-state S;
    if *at least one non-NULL state in* S **then**
        Add S to $\mathcal{S}_3$;
        $\mathcal{M}_3(S) \leftarrow \mathcal{M}_1(S_i) \cup \mathcal{M}_2(S_j)$;
foreach $S \in \mathcal{S}_3$ **do** `// set super-state deferment`
    Set $\mathcal{F}_3(S) \leftarrow$ `mode`$(\{\mathcal{S}_3(F_3(s)) \mid s \in S\})$;
    Let $P = \{s \mid (s \in S) \wedge (\mathcal{F}_3(S) \cap \mathcal{O}_3(s) = \perp)\}$;
    foreach *state* $u \in P$ **do**
        Remove u from super-state S;
        Create new super-state S' with just state u in overlay $\mathcal{O}_3(u)$ and add S' to $\mathcal{S}_3$;
        Set $\mathcal{M}_3(S') \leftarrow M_{D_3}(u)$;
        Set $\mathcal{F}(S') \leftarrow \mathcal{S}_3(F_3(u))$;
    foreach *state* $s \in S$ *with* $F_3(s) \neq \mathcal{F}_3(S) \cap \mathcal{O}_3(s)$ **do**
        Set $F_3(s) \leftarrow \mathcal{F}_3(S) \cap \mathcal{O}_3(s)$, and regenerate non-deferred transitions for ρ_3 in D_3 for state s;
foreach $S \in \mathcal{S}_3 \times c \in \Sigma$ **do** `// create super-state trans.`
    `CreateSupreStateTrans`(S, c);

Function `CreateSupreStateTrans`(S, c)
    $C \leftarrow$ `CreateSupreStateTransClassifier`$(S, \mathcal{F}_3(S), c)$;
    For each rule, $r_i \in C$ add super-state transition $\Delta_3(S, \mathbb{P}(r_i), c) = \mathbb{D}(r_i)$;

Function `CreateSupreStateTransClassifier`(S, DS, c)
`/* Generate transitions for character c and super-state S when it defers to DS */`
    Let ODec[n] be the offset decision vector initialized to $\circledast$;
    Let NODec[n] be the non-offset decision vector initialized to $\circledast$;
    Let Reqd[n] be the required vector initialized to False;

(cont'd)

Figure 6.7: Algorithm OD2FA*Merge*$(\mathcal{D}_1, \mathcal{D}_2)$ for merging two OD^{2}FAs.

```
(cont'd)
30  foreach O ∈ 𝒪₃ do
31      if S ∩ O ≠ ⊥ then
32          u=⟨u₁, u₂⟩ ← S ∩ O; // current state
33          nu ← δ'₃(u, c); // next state
34          if ρ₃(u, c) is defined then // not deferred
35              if S ≠ DS ∨ u ≠ nu then Reqd[O] ← True
36          ODec[O] ← (𝒮₃(nu), (𝒪₃(nu)−O) mod n, 1);
37          NODec[O] ← (𝒮₃(nu), 𝒪₃(nu), 0);
38  if #Unique values in ODec ≤ #Unique values in NODec then
39      return CreateOverlayClassifier(ODec, Reqd);
40  else
41      return CreateOverlayClassifier(NODec, Reqd);
```

Figure 6.7: Algorithm $\mathrm{OD2FAMerge}(\mathcal{D}_1, \mathcal{D}_2)$ for merging two OD^{2}FAs (cont'd).

As a result, in Figure 6.6(b), the super-states $1_3 = \boxed{2}\boxed{8}\boxed{5}\boxed{\perp}$ and $3_3 = \boxed{1}\boxed{7}\boxed{6}\boxed{\perp}$ have $\perp$ in overlay 3, and there is the super-state $4_3 = \boxed{\perp}\boxed{\perp}\boxed{\perp}\boxed{12}$ with just state 12 in overlay 3, and super-state 4_3 is compatible with both super-states 1_3 and 3_3. We can create new super-state transitions by merging super-state 4_3 with either 1_3 or 3_3. In Figure 6.6(c), we show the resulting OD^{2}FA when we merge 4_3 from Figure 6.6(b) with 3_3 adding the super-state transitions out of super-state 0_3 on 'p' to super-state 3_3 for overlays 2 and 3 with offset $o = 0$ and the super-state transitions out of super-state 3_3 to super-state 5_3 (renamed 4_3 in Figure 6.6(c)) on 'q' for overlays 2 and 3 with offset $o = 0$. Alternatively, we could have merged super-state 4_3 from Figure 6.6(b) with super-state 1_3 and added a super-state transition out of super-state 0_3 on 'p' to super-state 1_3 for overlays 1 and 3 with offset $o = 0$ and a super-state transition out of super-state 1_3 on r to super-state 2_3 for overlays 1 and 3 with offset $o = 0$. After merging super-states, we regenerate the super-state transitions for all the super-states and not just the super-states that were

merged, as merging super-states could lead additional transition merging opportunities in other super-states too.

Theorem 8. *Given as input OD^{2}FAs $\mathcal{D}_1$ and $\mathcal{D}_2$ and corresponding equivalent D^{2}FAs D_1 and D_2 for RE sets R_1 and R_2, the OD2FAMerge algorithm outputs an OD^{2}FA $\mathcal{D}_3$ that is equivalent to D^{2}FA D_3 for RE set $R_1 \cup R_2$.*

Proof. The D^{2}FA D_3 constructed by merging D^{2}FAs D_1 and D_2 using D2FAMerge algorithm is equivalent to RE set $R_1 \cup R_2$ ([36]). Line 20 only changes the deferred state for some states and so D_3 is equivalent to RE set $R_1 \cup R_2$.

We now show that the generated OD^{2}FA $\mathcal{D}_3$ is equivalent to D^{2}FA D_3. To show equivalence, we need to show that for each state $s \in Q_3$, the deferred state for s, the non-deferred transitions for s, and the matched REs for s, derived from $\mathcal{D}_3$ are same as in D_3. Let $s = \langle s_1, s_2 \rangle \in Q_3$ be any state in D_3. First, $\mathcal{S}_3(s)$ and $\mathcal{O}_3(s)$ are defined as we take a complete cross product of $\mathcal{S}_1 \times \mathcal{S}_2$ and $\mathcal{O}_1 \times \mathcal{O}_2$. The super-state transitions are directly generated from the D^{2}FA state transitions. It is easy to see that $\forall \sigma \in \Sigma$, $\rho'_3(s, \sigma)$ is defined in $\mathcal{D}_3 \iff \rho_3(s, \sigma)$ is defined in D_3; and when defined $\rho'_3(s, \sigma) = \rho_3(s, \sigma)$.

Then we have the following two cases.

Case 1: $\mathcal{S}_3(s)$ added to $\mathcal{S}_3$ on line 16. Then REs matched in $\mathcal{D}_3$ by $s = M_{\mathcal{D}_3}(s) \cup \mathcal{M}_3(\mathcal{S}(s)) = M_{D_3}(s)$ ($\because M_{\mathcal{D}_3}(s) = \emptyset$).

Deferred state of s in $\mathcal{D}_3 = \mathcal{F}_3(\mathcal{S}_3(s)) \cap \mathcal{O}_3(s) = \mathcal{S}_3(F_3(s)) \cap \mathcal{O}_3(F_3(s)) = F_3(s)$.

Case 2: $\mathcal{S}_3(s)$ added on line 9. Then let $\mathcal{S}_3(s) = S = \langle S_1, S_2 \rangle$. REs matched in $\mathcal{D}_3$ by $s =$

$M_{\mathcal{D}_3}(s) \cup \mathcal{M}_3(S) = \mathcal{M}_1(S_1) \cup \mathcal{M}_2(S_2) = M_{D_1}(s_1) \cup M_{D_2}(s_2) = M_{D_3}(s)$.

Deferred state of s in $\mathcal{D}_3 = \mathcal{F}_3(S) \cap \mathcal{O}_3(s) = F_3(s)$. □

6.4.3 Direct OD^{2}FA Construction from 2 OD^{2}FAs

Our OD^{2}FA merge algorithm presented in Section 6.4.2 requires the underlying D^{2}FA to be stored along with the OD^{2}FA. This underlying D^{2}FA requirement for merging OD^{2}FAs is problematic for two main reasons. First, in most practical cases, we would need to update the RE set over time. If the underlying D^{2}FA is discarded, then when a new RE is added to the RE set, we cannot use the merge algorithm to merge the OD^{2}FA for the new RE into the existing OD^{2}FA. Instead, we will have to build the entire OD^{2}FA again. This defeats one of the main advantages of the merge approach to building the OD^{2}FA which is automatic support for updating the RE set. The second problem is that because the underlying D^{2}FA is generally orders of magnitude larger than the OD^{2}FA, the size of the D^{2}FA limits the scalability of the algorithm.

We now present our algorithm, called *DirectOD2FAMerge*, to merge two OD^{2}FAs which does not require storing the underlying D^{2}FA. After the initial OD^{2}FAs have been built for each individual RE, we only store the OD^{2}FA at each merge step.

The input is two OD^{2}FAs, $\mathcal{D}_1 = (Q_1, \Sigma, q_{01}, \mathcal{F}_1, \mathcal{S}_1, \mathcal{O}_1, \mathcal{M}_1, \Delta_1)$ for RE set R_1 and $\mathcal{D}_2 = (Q_2, \Sigma, q_{02}, \mathcal{F}_2, \mathcal{S}_2, \mathcal{O}_2, \mathcal{M}_2, \Delta_2)$ for RE set R_2 where $R_1 \cap R_2 = \emptyset$, and we construct construct OD^{2}FA $\mathcal{D}_3 = (Q_3, \Sigma, q_{03}, \mathcal{F}_3, \mathcal{S}_3, \mathcal{O}_3, \mathcal{M}_3, \Delta_3)$ for the RE set $R_3 = R_1 \cup R_2$.

Just as in our OD2FAMerge algorithm in Section 6.4.2, each state (super-state) in $\mathcal{D}_3$ corresponds to a pair of states (super-states) from $\mathcal{D}_1$ and $\mathcal{D}_2$. The first step is to compute Q_3, *i.e.* find which states in the underlying DFA for $\mathcal{D}_3$ that are reachable. The set Q_3 is not stored explicitly but is implicit from the set of non-empty overlays for each super-state. If we store the set of non-empty overlays for each super-state as a list, the total size will be proportional to Q_3, which can be very large. So the set of non-empty overlays for each super-state is stored as a ternary classifier (similar to how we store super-state transitions which is discussed in Section 6.5.)

One option to find the reachable states is to simulate a UCP construction of the underlying DFAs of $\mathcal{D}_1$ and $\mathcal{D}_2$. That is, we do the UCP construction, but after computing the transitions of each merged state, we do not store them. The UCP construction also gives the state to super-states and overlay assignment. The problem with this method is that the queue of unexplored states while doing the UCP construction can be proportional to $|Q_3|$.

To avoid this, we simulate the UCP construction focusing on super-states instead of states. The construction works as follows. For each discovered super-state in $\mathcal{D}_3$, we maintain two sets of overlays: (1) the $\mathtt{Explored}$ set containing the overlays which have a reachable DFA state that have already been explored, and (2) the $\mathtt{Unexplored}$ set containing the overlays which have a reachable DFA state that have not already been explored. We maintain a queue, $\mathtt{Queue}$, of super-states in $\mathcal{D}_3$ that currently need to be explored, and explore one super-state from the queue at a time. For the super-state, say S, currently being explored,

we explore all the states corresponding to the overlays in S's $\mathit{Unexplored}$ set, and them move all the overlays from the $\mathit{Unexplored}$ to the $\mathit{Explored}$ set.

When a new state, say $(S' \cap O')$, is discovered, it is processed as follows. If S' is a newly discovered super-state, we add it to Queue and set $\mathit{Explored}(S') = \emptyset$ and $\mathit{Unexplored}(S') = O'$. Otherwise S' is already discovered and so is in $\mathcal{S}_3$. In this case, if $O' \in \mathit{Explored}(S')$ or $O' \in \mathit{Unexplored}(S')$, then we do not have to do anything as the state has already been discovered. Otherwise, this is a newly discovered state, so we add O' to $\mathit{Unexplored}(S')$, and add S' to Queue if S' is not already there.

A super-state may be added to Queue and explored multiple times because all non-empty overlays within a super-state are not discovered at the same time. As mentioned earlier, the $\mathit{Explored}$ and $\mathit{Unexplored}$ overlay sets are maintained as ternary classifiers. As new overlays are added to the sets, the classifiers are minimized using the bit merging algorithm that is explained in Section 6.5.3.

After computing the reachable states, we have all the terms in $\mathcal{D}_3$ constructed except for $\mathcal{F}_3$ and Δ_3.

For the OD^{2}FAs in Figure 6.4 and Figure 6.5, this new merge algorithm results in the same OD^{2}FA as earlier shown in Figure 6.6(b).

To set the super-state deferment, we use a method similar to that used in Section 4.3.2 to set state deferment when merging D^{2}FAs. Let $S{=}\langle S_0, T_0\rangle$ be the current super-state in $\mathcal{D}_3$ for which we need to compute the deferment. Let $S_0 \rightarrow S_1 \rightarrow \cdots \rightarrow S_l$ be the maximal

deferment chain DC_1 (*i.e.* S_l is the root super-state) in $\mathcal{D}_1$ starting at S_0, and $T_0 \rightarrow T_1 \rightarrow \cdots \rightarrow T_m$ be the maximal deferment chain DC_2 in $\mathcal{D}_2$ starting at T_0. We will choose some super-state $\langle S_i, T_j \rangle$ where $0 \leq i \leq l$ and $0 \leq j \leq m$ to be $\mathcal{F}_3(S)$. We only consider a candidate super-state pair if it is reachable in $\mathcal{D}_3$ and it overlay covers super-state S (so Condition (C2) holds). Ideally, we want i and j to be as small as possible though not both 0. For example, our best choices are typically $\langle S_0, T_1 \rangle$ or $\langle S_1, T_0 \rangle$. However, it is possible that both super-states are not eligible (either not reachable or do not overlay cover S). This leads us to consider other possible $\langle S_i, T_j \rangle$.

For any candidate super-state pair $\langle S_i, T_j \rangle$, we build the super-state transitions for super-state S as if it were to defer to super-state $\langle S_i, T_j \rangle$ in $\mathcal{D}_3$ (we show how to build the super-state transitions in Section 6.5). The number of super-state transitions built gives us the measure of the effectiveness of the deferment; the fewer transitions built, the better it is. One strategy (the *best match method*) is to consider all candidate super-state pairs, and pick the one that results in the fewest super-state transitions built for super-state S. A faster strategy (the *first match method*) is to consider the 'distance sum' $z = i + j$ in increasing order, from 1 to $l + m$. For the current distance sum z, we consider all super-state pairs at that distance; *i.e.* the set of super-states $Z = \{\langle S_i, T_{z-i} \rangle \mid (\max(0, z - m) \leq i \leq \min(l, z)) \wedge (\langle S_i, T_{z-i} \rangle \in Q_3) \wedge (\langle S_i, T_{z-i} \rangle \text{ overlay covers} S)\}$. From the set of super-states Z, we choose the super-state that results in the fewest super-state transitions built for super-state S. We can always find an eligible super-state to set as $\mathcal{F}_3(S)$, since the root super-state pair $\langle S_l, T_m \rangle$ is always reachable in $\mathcal{D}_3$ and it overlay covers all other

super-states.

For example in Figure 6.6(b), for super-state $4{=}\langle 1,1\rangle$, there are three reachable super-state pairs along the deferment chains: $1{=}\langle 0,1\rangle$, $3{=}\langle 1,0\rangle$ and $0{=}\langle 0,0\rangle$. However super-states $1{=}\langle 0,1\rangle$ and $3{=}\langle 1,0\rangle$ do not overlay cover super-state $4{=}\langle 1,1\rangle$, leaving the super-state $0{=}\langle 0,0\rangle$ as the only candidate pair, which is chosen as the deferred super-state.

How the super-state transitions are created for the merged OD^{2}FA is shown in Section 6.5.

Pseudo-code for our DirectOD2FAMerge algorithm is given in Algorithm 6.8.

At the end, we apply the same optimization of merging sibling super-states together as in the case of our OD2FAMerge algorithm.

6.5 Building Super-state Transitions

In this section we describe how we combine state transitions to create super-state transitions after the super-states have been created. The OD^{2}FA captures similarity among states in different overlays within a super-state. So we would expect that state transitions (which are just singleton super-state transitions) would be combined over the overlay field; *i.e.* multiple singleton super-state transitions with the same current super-state, current input character and decision values but different overlay values will be combined.

The super-state transitions are created for each super-state and input character at a time. In the rest of the section, S refers to the current super-state and σ refers to the current

Input: OD^{2}FAs, $\mathcal{D}_1 = (Q_1, \Sigma, q_{01}, \mathcal{F}_1, \mathcal{S}_1, \mathcal{O}_1, \mathcal{M}_1, \Delta_1)$ and $\mathcal{D}_2 = (Q_2, \Sigma, q_{02}, \mathcal{F}_2, \mathcal{S}_2, \mathcal{O}_2, \mathcal{M}_2, \Delta_2)$, corresponding to RE sets R_1 and R_2.
Output: An OD^{2}FA and its underlying D^{2}FA corresponding to the RE set $R_1 \cup R_2$.

```
1  Initialize D3 to an empty OD²FA;
2  Set #overlays in D3, |O3| = n ← |O1| × |O2|;
   // Create the super-states
3  Initialize queue as an empty queue;
4  queue.push (⟨q01, q02⟩);
5  while queue not empty do
6      u=⟨u1, u2⟩ ← queue.pop();
7      Q3 ← Q3 ∪ {u};
8      S1 ← S1(u1); O1 ← O1(u1);
9      S2 ← S2(u2); O2 ← O2(u2);
10     if super-state S=⟨S1, S2⟩ ∉ S3 then
11         Initialize super-state S=⟨S1, S2⟩ with n NULL states;
12         Add S to S3;
13         M3(S) ← M1(S1) ∪ M2(S2);
14     Assign u to overlay (O1 × |O2| + O2) in super-state S;
15     foreach c ∈ Σ do
16         nxt ← ⟨δ1''(u1, c), δ2''(u2, c)⟩;
17         if nxt ∉ Q3 ∧ nxt ∉ queue then queue.push (nxt);
18 foreach S ∈ S3 do F3(S) ← FindDefState(S); // set super-state deferment
19 foreach S ∈ S3 × c ∈ Σ do // create super-state trans.
20     CreateSupreStateTrans(S, c);

21 Function FindDefState(⟨S1, S2⟩)
22     Let ⟨p0 = S1, p1, ..., pl⟩ be the list of super-states on the deferment chain from S1 to the
       root super-state in D1;
23     Let ⟨q0 = S2, q1, ..., qm⟩ be the list of super-states on the deferment chain from S2 to the
       root super-state in D2;
24     for z = 1 to (l + m) do
25         S' ← {⟨pi, q_{z−i}⟩ | (max(0, z − m) ≤ i ≤ min(l, z)) ∧ (⟨pi, q_{z−i}⟩ ∈ S3)};
26         if S' ≠ ∅ then return argmin_{DS∈S'}(Σ_{c∈Σ}
           Cost(CreateSupreStateTransClassifier(⟨S1, S2⟩, DS, c)));
27     return ⟨S1, S2⟩;

28 Function CreateSupreStateTrans(S, c)
29     C ← CreateSupreStateTransClassifier(S, F3(S), c);
30     For each rule, ri ∈ C add super-state transition Δ3(S, P(ri), c) = D(ri);
```

(cont'd)

Figure 6.8: Algorithm DirectOD2FAMerge($\mathcal{D}_1, \mathcal{D}_2$) for merging two OD^{2}FAs.

```
(cont'd)
31 Function CreateSupreStateTransClassifier(S, DS, c)
   /* Generate transitions for character c and super-state S when it defers to
      DS  */
32   Let ODec[n] be the offset decision vector initialized to ⊛;
33   Let NODec[n] be the non-offset decision vector initialized to ⊛;
34   Let Reqd[n] be the required vector initialized to False;
35   foreach O ∈ 𝒪3 do
36       if S ∩ O ≠⊥ then
37           u=⟨u1, u2⟩ ← S ∩ O; // current state
38           nu1 ← δ1''(u1, c); nu2 ← δ2''(u2, c); // next state
39           if S = DS then // for the root super-state
40               if (u1 ≠ nu1) ∨ (u2 ≠ nu2) then Reqd[O] ← True; // not a self-loop
41           else
42               du=⟨du1, du2⟩ ← DS ∩ O;
43               if (δ1''(du1, c) ≠ nu1) ∨ (δ2''(du2, c) ≠ nu2) then Reqd[O] ← True; // not
                 deferred
44           ODec[O] ← (𝒮3(⟨nu1, nu2⟩), (𝒪3(⟨nu1, nu2⟩)−O) mod n, 1);
45           NODec[O] ← (𝒮3(⟨nu1, nu2⟩), 𝒪3(⟨nu1, nu2⟩), 0);
46   if #Unique values in ODec ≤ #Unique values in NODec then
47       return CreateOverlayClassifier(ODec, Reqd);
48   else
49       return CreateOverlayClassifier(NODec, Reqd);
```

Figure 6.8: Algorithm DirectOD2FAMerge($\mathcal{D}_1, \mathcal{D}_2$) for merging two OD^{2}FAs (cont'd).

input character for which we want to build the super-state transitions. T refers to the current (or potential) deferred super-state of S.

6.5.1 Combining State Transitions

To combine the state (singleton super-state) transitions, we first need to identify the (subset of) overlays that have the same decision; that is, the same next super-state, overlay value and the offset bit.

A trivial way to combine state transitions is to create one super-state transition for each unique decision value among all the overlay decisions. All the overlays (*i.e.* state transitions) having the same decision will be combined into the super-state transition for that decision. In this case, we will have the smallest possible number of super-state transitions, which is equal to the number of unique decisions.

The problem with this approach is that we may have any arbitrary subset of overlays in a super-state transition. Thus, we will need to represent arbitrary subsets of overlays. This is problematic because any such representation will have a size that will be linear in the size of the overlay set, $\mathcal{O}$. The combined memory requirement of such a representation over all the super-state transitions for all super-states will essentially be linear in terms of the number of state transitions. This defeats the purpose of combining the state transitions.

To address this issue, we only create overlay subsets (*i.e.* only combine state transitions) for states whose overlay sets that can be concisely represented. Specifically, we only create overlay subsets that can be represented as a ternary value; *i.e.* the set of overlays in each combined super-state transition is equal to the ternary expansion of a ternary value. Recall that we treat the overlays as integers in the range $[0..|\mathcal{O}|)$ and $|\mathcal{O}|$ is always a power of 2. In most cases this restriction does not result in lost state combining opportunities. In almost all cases, we are able to combine all state transitions with the same decision into a single super-state transition.

6.5.1.1 Computing State Transitions

For each overlay $O \in \mathcal{O}$, we can have one of the following three cases: (a) $S \cap O = \perp$, *i.e.* the overlay is empty, (b) $S \cap O = s$ and $\delta''(s, \sigma) \neq \delta''(T \cap O, \sigma)$, *i.e.* state transition is not deferred and (c) $S \cap O = s$ and $\delta''(s, \sigma) = \delta''(T \cap O, \sigma)$, *i.e.* state transition is deferred. $\mathcal{O}_f \subseteq \mathcal{O}$ denotes the set of filled overlays, and $\mathcal{O}_r \subseteq \mathcal{O}_f$ denotes the set of overlays for which the state transition is not deferred. Note that $\mathcal{O}_f$ depends on S, and $\mathcal{O}_r$ depends on S, T and σ. The super-state transitions generated for super-state S need to cover all the overlays in $\mathcal{O}_r$. We make the following two observations which help us combine the state transitions into fewer super-state transitions.

- *We never do a lookup on the OD^2FA for any overlay $O \in \mathcal{O} \setminus \mathcal{O}_f$ for super-state S.* Because of this, empty overlays can have any decision, and so can be 'merged' with any overlay. For example, suppose we have $|\mathcal{O}| = 4$, where overlay $2 = (10)_2$ is empty, and overlays $0 = (00)_2$, $1 = (01)_2$ and $3 = (11)_2$ all have the same decision. If we try to combine just the filled overlays, we get two super-state transitions with overlay sets $0*$ and 11. But since we would never do a lookup on the empty overlay, we can include it in the super-state transition, which results in only one transition with overlay set $**$. For every empty overlay we designate a special *wildcard decision*, denoted by $\circledast$, that matches any actual decision. Also note that if we include empty overlays in super-state transitions, Condition (C2) is necessary and sufficient to ensure that transition deferment works correctly.

- *It is not necessary to defer transitions that match the deferred state.* When

combining state transitions, including transitions that can be deferred can result in fewer super-state transitions. For example, suppose we have $|\mathcal{O}| = 4$, where all four overlays are filled and all have the same decision, but the transition for overlay $2 = (10)_2$ is deferred, whereas transitions for overlays $0 = (00)_2$, $1 = (01)_2$ and $3 = (11)_2$ are not deferred. If we require that the transition for overlay 2 must be deferred, then we need need two super-state transitions with overlay sets $0*$ and 11 to cover the remianing overlays. Including the state transition for overlay 2 in the combined super-state transition results in only one super-state transition with overlay set $**$.

Before we can combine state transitions, we first need to compute the state transition and deferment for each overlay. We create a *Decision* array which records the decision for each overlay, and a corresponding Boolean *Required* array which records whether the decision is necessary or not (*i.e.* whether it must be specified or it can be deferred). For empty overlays, the *Decision* value is set to $\circledast$ and *Required* is set to false. For filled overlays, how the state transitions are computed depends on the stage of the OD^{2}FA construction.

During initial OD^{2}FA construction for one RE: The underlying D^{2}FA is available during the initial OD^{2}FA construction, so the state transitions and deferments are determined by the D^{2}FA.

During OD2FAMerge: Since the underlying D^{2}FA is stored in OD2FAMerge, again the state transitions and deferments are determined by the stored D^{2}FA. The D^{2}FA lookup

from the underlying D^2FA corresponds to lines 33 and 35 in Algorithm 6.7.

During DirectOD2FAMerge: During DirectOD2FAMerge, we do a lookup from the input OD^2FAs to compute the state transitions and deferments. The lookup from the two input OD^2FAs corresponds to lines 38 and 43 in Algorithm 6.8.

For the root super-state, for self-loop state transitions we set the Required value to false, even though these transitions are not deferred. As a result, the root super-state will not store the self-looping super-state transitions. If lookup fails for a non-root super-state, then we would follow the deferment pointer and do a lookup on its deferred super-state. If lookup fails for the root super-state, there no deferment pointer to follow along; however we would know that the missing transition is a self-loop (on the root super-state), so the destination super-state is the root super-state and the destination overlay is the current overlay. Since most transitions for the root super-state are self-loops, this greatly reduces the resulting number of super-state transitions.

We need to determine which of the two forms of super-state transitions (offset transitions or non-offset transitions) to create. Clearly we would use the form which results in fewer super-state transitions. So we create a *Decision* array for both offset and non-offset decision, and use the one which has fewer unique values in it to create the super-state transitions. In most of the cases using the offset decisions results in fewer super-state transitions.

We only compute and store transitions for all states in one super-state at at time. Once the

super-state transitions have been constructed, the state transitions are discarded. Hence we never store state transitions for all states in the OD^{2}FA at the same time.

For example consider super-state 1 and input character d in the OD^{2}FA in Figure 6.6(c). The OD^{2}FA has four overlays so $\mathcal{O} = \{0, 1, 2, 3\}$. In this case we have $\mathcal{O}_f = \{0, 1, 2\}$ and $\mathcal{O}_r = \{0, 2\}$. The *Decision* array will be $[(0, 1, 1), (0, 0, 1), (0, 1, 1), \circledast]$ and the *Required* array will be $[true, false, true, false]$.

6.5.2 Creating Overlay Classifier

The set of state transitions for each overlay for super-state S and input character σ essentially forms a 1-dimensional classifier over the overlay field. The problem of creating a minimum set of covering super-state transitions then boils down to finding an equivalent ternary minimized classifier.

We introduce some standard terminology first. A 1-dimensional classifier is defined over a field F and consists of a list of rules. Each rule r has a predicate $\mathbb{P}(r) \subseteq F$ and a decision $\mathbb{D}(r)$. A packet $p \in F$ matches rule r if $p \in \mathbb{P}(r)$. The decision of the classifier C for a packet p is given by the first rule in C that matches p. For our purpose of using a classifier to build super-state transitions, we define a generalized version of a classifier that we call an *overlay classifier*.

Definition 7 (Overlay classifier). *An overlay classifier, C, is 1-dimensional classifier over the field $\mathcal{O}$. Each rule r has a Boolean flag, denoted by $\mathbb{R}(r)$, that indicates*

whether the rule is required or not. Rules with decision $\circledast$ *have their flag* $\mathbb{R}(r)$ *set to false. The rules in* C *satisfy the following properties:*

- Ternary classifier: *For each rule* $r \in C$*, its predicate* $\mathbb{P}(r)$ *is a ternary value.*
- Non-conflicting property: *For every packet* $p \in \mathcal{O}_f$*, all the rules that match* p *(if any) also have matching decisions (note that* $\circledast$ *matches any actual decision.)*
- Covering property: *For every packet* $p \in \mathcal{O}_r$*, there is at least one rule* $r \in C$ *that matches* p *and* $\mathbb{R}(r)$ *is true (which also implies* $\mathbb{D}(r) \neq \circledast$*.)*
- Restricted equivalence: *Two overlay classifiers are equivalent if, for every packet in* $\mathcal{O}_f$ *for which both overlay classifiers have a match, they both have the same decision.*

Given the *Decision* and *Required* values for each overlay, we first construct an overlay classifier with one rule for each overlay. Specifically, we create an empty overlay classifier C over $\mathcal{O}$. Then for each overlay O, we add the rule $\text{Rule}(O, \text{Decision}[0], \text{Required}[O])$ to C. Here $\text{Rule}(x, y, z)$ refers to creating a rule r with $\mathbb{P}(r) = x$, $\mathbb{D}(r) = y$ and $\mathbb{R}(r) = z$. Next we minimize the rules in C to get an equivalent overlay classifier C' (which is discussed in the next section). After minimizing, each rule $r \in C'$ with $\mathbb{R}(r) = \text{true}$ gives us a combined super-state transition $\Delta(S, \mathbb{P}(r), \sigma) = \mathbb{D}(r)$ in the OD^{2}FA.

The covering property of overlay classifiers ensures that super-state S will have a super-state transition covering every overlay in $\mathcal{O}_r$. The non-conflicting property of overlay classifier ensures that each overlay in $\mathcal{O}_f$ has at most one decision. Note that we can have more

than one super-state transition covering an overlay, but in that case the non-conflicting property ensures that they all have the same decision.

For example with super-state 1 and input character d in the OD^{2}FA in Figure 6.6(c), the overlay classifier created will have just one required rule $*0 \rightarrow (0,1,1)$, which gives us the super-state transition $(1,*0) \xrightarrow{d} (0,1,1)$. Figure 6.9 shows the overlay classifiers and corresponding super-state transitions generated for all the super-states in the OD^{2}FA in Figure 6.6(c).

Super-state	Char.	Overlay classifier	Super-state transition
0	a	$0* \rightarrow (3,0,1)$	$(0,0*) \xrightarrow{a} (3,0,1)$
	c	$*0 \rightarrow (1,0,1)$	$(0,*0) \xrightarrow{c} (1,0,1)$
	n	$** \rightarrow (0,0,0)$	$(0,**) \xrightarrow{n} (0,0,0)$
	p	$01 \rightarrow (1,0,1)$ $1* \rightarrow (3,0,1)$	$(0,01) \xrightarrow{p} (1,0,1)$ $(0,1*) \xrightarrow{p} (3,0,1)$
1	d	$*0 \rightarrow (0,1,1)$	$(1,*0) \xrightarrow{d} (0,1,1)$
	r	$01 \rightarrow (2,0,1)$	$(1,01) \xrightarrow{r} (2,0,1)$
3	b	$0* \rightarrow (0,2,1)$	$(3,0*) \xrightarrow{b} (0,2,1)$
	q	$1* \rightarrow (4,0,1)$	$(3,1*) \xrightarrow{q} (4,0,1)$
	r	$11 \rightarrow (2,0,1)$	$(3,11) \xrightarrow{r} (2,0,1)$

Figure 6.9: Overlay classifier and corresponding super-state transitions for the super-states in OD^{2}FA in Figure 6.6(c).

The pseudo-code for creating the overlay classifier is given in Algorithm 6.10.

6.5.3 Minimizing Overlay Classifier

We now explain how we minimize the initial overlay classifier created from the *Decision* and *Required* arrays. We generalize the bit merging algorithm proposed in [31] to handle

Input: The decision, Dec[], and required value, Reqd[], for each overlay.
Output: An equivalent ternary minimized overlay classifier.

```
1 n ← len(Dec); // number of overlays, will be a power or 2
2 w ← log2(n); // number of bits
3 Create empty overlay classifier C with field width w;
4 foreach overlay o ∈ [0..n) do
5     Insert Rule(o, Dec[o], Reqd[o]) in C;
6 return MinimizeOverlayClassifier(C); // minimize the rules and return
```

Figure 6.10: Algorithm $CreateOverlayClassifier(Dec, Reqd)$.

wildcard decision $\circledast$ and optional deferment.

We introduce some standard terminology first. For a ternary value T, the *ternary position mask* of T, denoted by $\tau(T)$, is the binary value obtained by replacing all binary bits in T by 0 and all ternary bits ($*$) in T by 1. The ternary position mask of T basically indicates the positions in T which have a ternary bit. The *binary bit mask* of T, denoted by $\beta(T)$, is the binary value obtained by replacing all ternary bits in T by 1. The ternary position mask and binary bit mask together represent a ternary value using two binary values. If bit location b is a 1 bit in $\tau(T)$ then T has a $*$ in location b; otherwise T has the same binary bit in location b as in $\beta(T)$. So we can represent at ternary value T as the pair of binary values $(\tau(T), \beta(T))$.

Two ternary values, T_1 and T_2, are said to be *ternary adjacent* if $\tau(T_1) = \tau(T_2)$ and $\beta(T_1)$ and $\beta(T_2)$ differ in exactly one bit. In other words, T_1 and T_2 are ternary adjacent if they differ in exactly one location which has a binary bit in both T_1 and T_2. The *ternary cover* of T_1 and T_2 is the ternary value $(\tau(T_1) \mid (\beta(T_1) \wedge \beta(T_2)), \beta(T_1) \mid (\beta(T_1) \wedge \beta(T_2)))$ (here $\mid$ is the bitwise OR, and $\wedge$ is the bitwise XOR). That is, the ternary cover is the ternary value

obtained by replacing the differing binary bit location in T_1 (or in T_2) by the ternary bit $*$. Two rules are said to be *ternary adjacent* if their predicates are ternary adjacent and their decision match.

We first minimize the rules in the overlay classifier and then remove rules that are not required (*i.e.* have the $\mathbb{R}(r)$ flag set to false). Minimizing the overlay classifier is done in two steps, pre-merging bits and bit merging. We explain these two steps using the example in Figure 6.11.

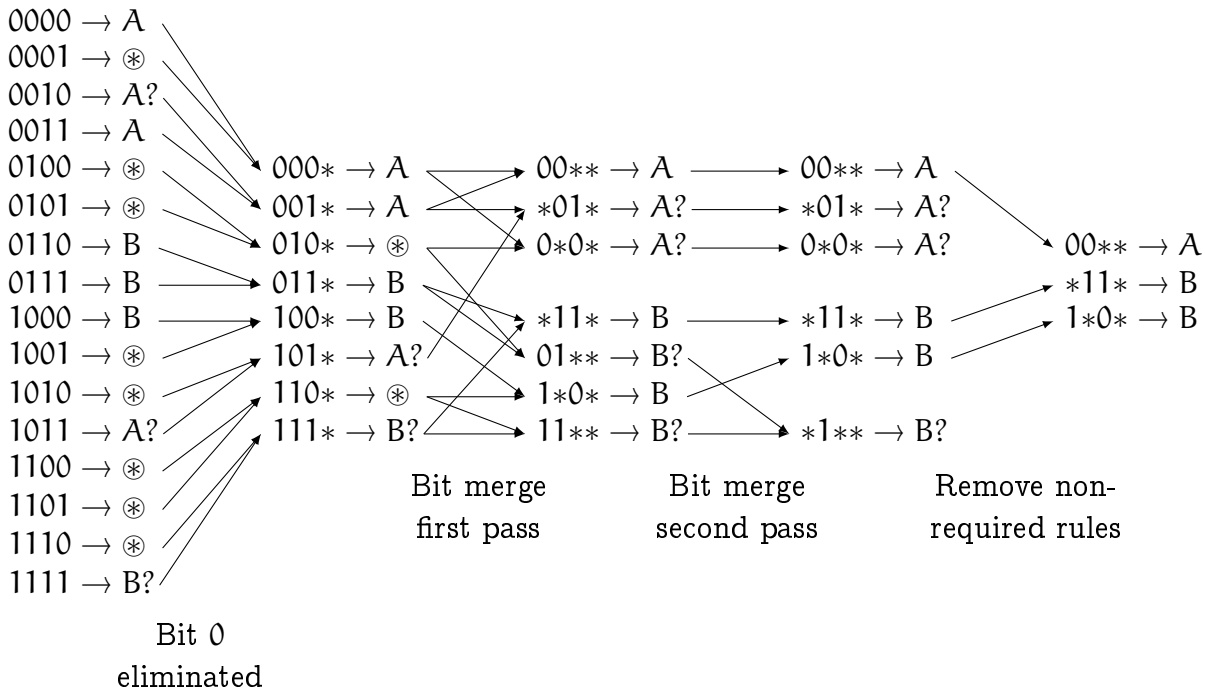


Figure 6.11: Minimizing overlay classifier example.

The pseudo-code for minimizing the overlay classifier is given in Algorithm 6.12

```
Input: A initial overlay classifier C with n = 𝒪 rules.
Output: Equivalent overlay classifier with rules minimized.
w ← log2(n); // number of bits
foreach bit k ∈ [0..w) do // first try pre-merging bits
    premerge ← True;
    foreach pair of rules, r_i, r_j, such that ℙ(r_i) and ℙ(r_i) differ only in bit k do
        if r_i and r_j are not ternary adjacent then // i.e. decisions of r_i and r_j do not match
            premerge ← False;
            break;
    if premerge then // bit k is pre-merged
        foreach pair of rules, r_i, r_j, such that ℙ(r_i) and ℙ(r_i) differ only in bit k do
            Remove rules r_i and r_j from C;
            Insert rule MergedRule(r_i, r_j) in C;
C ← BitMerge(C); // then do bit merging
foreach rule r_i ∈ C do if ℝ(r_i) = False then Remove r_i from C; // remove non-required rules
return C;

Function BitMerge(C)
    Create empty overlay classifier C′;
    foreach rule r_i ∈ C do  Initialize covered[i] ← False;
    PM ← Partition of rules in C based on rule predicate ternary position masks;
    foreach Partition pm ∈ PM do
        PD ← Partition of rules in pm based on rule decision;
        foreach Partition pd ∈ PD with corresponding decision d do
            foreach pair or rules r_i, r_j ∈ pd do
                if r_i and r_j are ternary adjacent then
                    Insert MergedRule(r_i, r_j) in C′;
                    covered[i] ← covered[j] ← True;
                    ℝ(r_i) ← ℝ(r_j) ← False;
            if d ≠ ⊛ then
                psd ← Partition in PD corresponding to ⊛;
                foreach pair or rules r_i ∈ pd × r_j ∈ psd do
                    if r_i and r_j are ternary adjacent then
                        Insert MergedRule(r_i, r_j) in C′;
                        covered[i] ← covered[j] ← True;
                        ℝ(r_i) ← ℝ(r_j) ← False;
(cont'd)
```

Figure 6.12: Algorithm MinimizeOverlayClassifier(C).

(cont'd)

```
34  if C′ is empty then // no rules merged
35      return C;
36  foreach rule r_i ∈ C do if covered[i] = False then Insert r_i in C′;
37  Remove duplicate rules from C′;
38  return BitMerge(C′); // recursively call BitMerge and return the result

39 Function MergedRule(r_1, r_2)
40  T ← ternary cover of ℙ(r_1) and ℙ(r_2);
41  if 𝔻(r_1) ≠ ⊛ then D ← 𝔻(r_1) else D ← 𝔻(r_2) ;
42  reqd ← ℝ(r_i) ∨ ℝ(r_j);
43  return Rule(T, D, reqd);
```

Figure 6.12: Algorithm $MinimizeOverlayClassifier(C)$ (cont'd).

6.5.3.1 Pre-merging Bits

The initial overlay classifier created from the *Decision* and *Required* arrays will have $|\mathcal{O}|$ rules, one rule for each overlay, and the predicate of any rule r_i is i (the corresponding overlay (binary) value). For our example, the first column in Figure 6.11 shows the initial overlay classifier. We have $|\mathcal{O}| = 16$. There are two unique actual decisions denoted by A and B. A '?' next to an actual decision indicates that the rule is not required (rules with a $\circledast$ decision are always not required).

At this point we can directly apply the bit merging algorithm, which will result in a minimized set of rules. But in most cases, all except for a few overlays have the same decision. So only a few bits that distinguish the overlays having different decisions will vary in the minimized rules. All the other bits will be merged to $*$'s in all the minimized rules. We can accelerate the bit merging step by identifying these bits and pre-merging them so that the bit-merging algorithm only needs to work on the few remaining bits that

are not pre-merged.

The pre-merging works as follows. For a binary value p, $\hat{0}_b(p)$ denotes the value obtained by inserting a 0 bit at location b, and $\hat{1}_b(p)$ denotes the value obtained by inserting a 1 bit at location b. Bit location b is pre-merged if the following condition is true: $\forall p \in [0..|\mathcal{O}|/2)$, $\mathbb{D}(r_{\hat{0}_b(p)})$ matches $\mathbb{D}(r_{\hat{1}_b(p)})$. That is, for every pair of rules whose predicates differ only in bit location b, their decisions match. Since the decisions for every such pair of rules match, we merge these pair of rules. A pair of such rules, lets say r_i and r_j are merged as follows. We create a new merged rule, say r_k. $\mathbb{P}(r_k)$ is set to the ternary cover of $\mathbb{P}(r_i)$ and $\mathbb{P}(r_j)$. If $\mathbb{D}(r_i) \neq \circledast$ then we set $\mathbb{D}(r_k) \leftarrow \mathbb{D}(r_i)$ otherwise we set $\mathbb{D}(r_k) \leftarrow \mathbb{D}(r_j)$, and we set $\mathbb{R}(r_k) \leftarrow \mathbb{R}(r_i) \vee \mathbb{R}(r_j)$. Rules r_i and r_j are replaced with the merged rule r_k.

We test and pre-merge one bit location at a time. Every time a bit is pre-merged, the number of rules is reduced by half.

In our example in Figure 6.11, bit location 0 gets pre-merged, and the resulting rules are shown in the second column.

6.5.3.2 Bit Merging Algorithm

The bit merging algorithm runs in several iterations. The input to each iteration is an overlay classifier C, and the output is an equivalent overlay classifier C'. Each iteration works as follows.

We first initialize a $Covered$ flag to false for each rule in C. For rule r_i, $Covered[r_i]$

indicates if rule r_i is covered by some rule in C'. Then for every pair of rules, say r_i and r_j, in C that are ternary adjacent, we insert the merged rule r_k in C'. The merged rule r_k is created in the same way as during the pre-merging step. After inserting merged rule r_k to C', we set $Covered[r_i]$ and $Covered[r_j]$ to true and set $\mathbb{R}(r_i)$ and $\mathbb{R}(r_j)$ to false. The idea behind setting the required flags for r_i (and r_j) to false is that since a rule has already been added to C' that covers r_i, any further rules we add to C' should not be set as required because of r_i.

To speed up bit merging, we partition the rules based on the ternary position mask of the each rule's predicate and each rule's decision. This reduces the number of pairs of rules we need to check for merging. After all pairs have been checked for merging, any rules left in C with their $Covered$ flag false are added to C'. The bit merging iterations continue as long as there is at least one merged rule added to C' When no pair of rules is merged, we stop and return the current overlay classifier.

For our example in Figure 6.11, we have two iterations of bit merging. After the first iteration, we get the rules in column 3. The first rule in column 3 is obtained by merging the first two rules in column 2. After merging the first two rules in column 2, both rules will be marked as non-required. Therefore when the third rule in column 3 is created by merging the first and third rule in column 2, it is marked as non-required. We get the rules in column 4 after the second iteration of bit merging. No more rules can be merged after that, so bit merging stops. Finally, we remove the non-required rules to get the final overlay classifier shown in column 5.

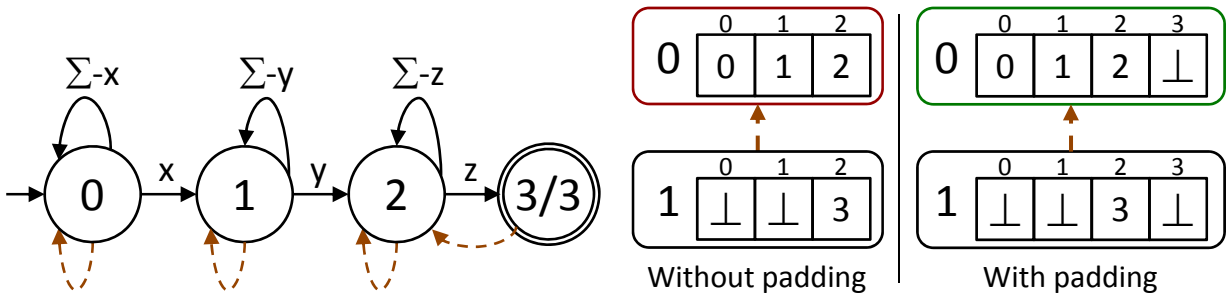
6.5.4 Overlay Discussion

6.5.4.1 Restricting Overlay Count to Power of 2

We keep the number of overlays in all intermediate OD^{2}FAs and the final OD^{2}FA to be a power of 2 and number the overlays starting with 0 and ending with $|\mathcal{O}| - 1$. We achieve this by modifying the algorithm that constructs an OD^{2}FA from one RE to pad empty overlays at the end if necessary. The OD^{2}FA merge algorithm requires no modification since the number of overlays in the merged OD^{2}FA is equal to the product of the number of overlays in the two input OD^{2}FAs.

We explain by example the benefit of requiring the number of overlays to be a power of 2.

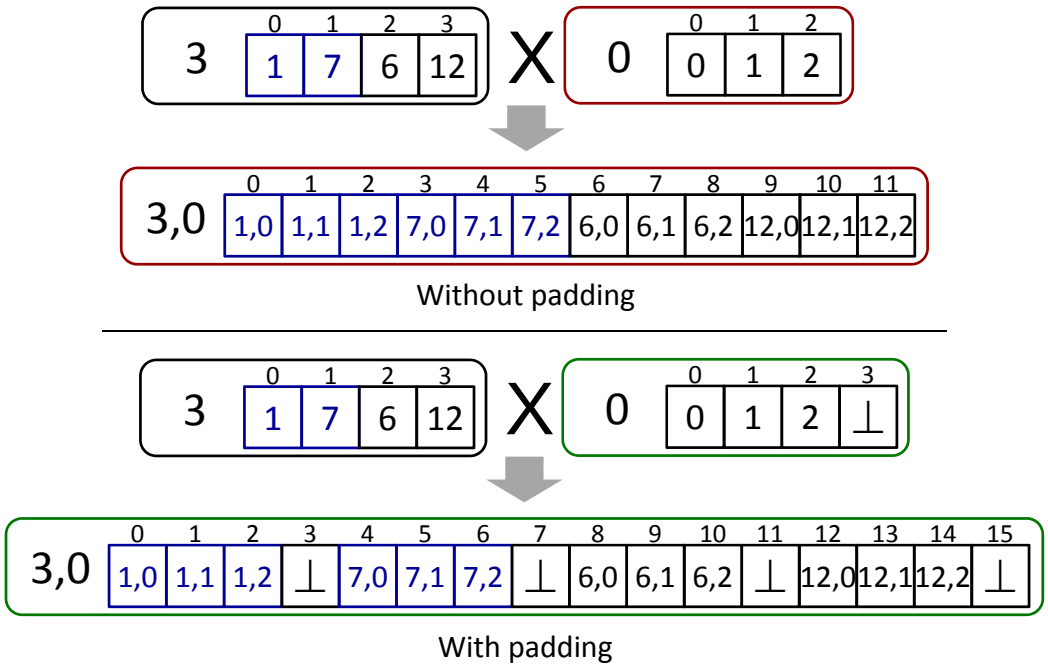
Figure 6.13(a) shows the D^{2}FA for the RE `/x.⋆y.⋆z/` and Figure 6.13(b) shows two possible overlay structures for the OD^{2}FA. Since there are three self-looping states in the D^{2}FA, 0, 1 and 2, our algorithm places them in the root super-state. The overlay structure on the left has three overlays, with the three self-looping states in them, with no padding.



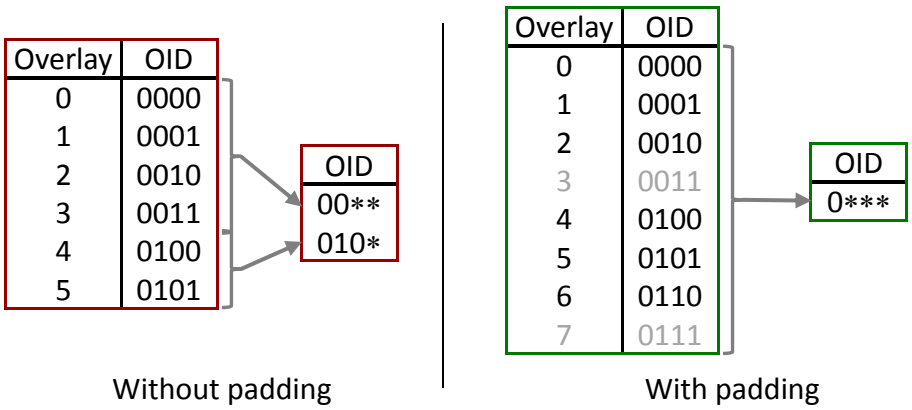
(a) D^{2}FA for RE `/x.⋆y.⋆x/`.

(b) Possible overlay structures for the corresponding OD^{2}FA.

Figure 6.13: Overlay Padding Example.



(c) Merged super-state.



(d) TCAM rules.

Figure 6.13: Overlay Padding Example (cont'd).

In the right overlay structure, we pad one empty overlay, so that the number of overlays is a power of 2. Now consider what happens when this new OD^{2}FA in Figure 6.13(b) with and without padding, is merged with the OD^{2}FA in Figure 6.6(c). As an example, we consider the merging of super-state 3 in Figure 6.6(c), which we call S_3 and super-state 0 for the new OD^{2}FA, which we call S_0. For both cases, Figure 6.13(c) shows the resulting

super-state in the merged OD^{2}FA, which we call S_m. In both cases, there will be 12 states in the merged super-state. The first three of these states are replications of state 1 in S_3, the next three states are replications of state 7 in S_3, and so on. Furthermore, states 1 and 7 in S_3 were itself replications of the state 1 of the D^{2}FA in Figure 6.4. Hence, the first six states in S_m are replications of the same state (*i.e.* state 1) of the D^{2}FA in Figure 6.4. For the case without padding, S_m has 12 overlays, with one state in each overlay. For the case with padding, S_m has 16 overlays, with the overlays 3, 7, 11 and 15 being empty. Now, since the first six states in S_m are replications of state 1 of the D^{2}FA in Figure 6.4, in the merged OD^{2}FA, they all will have one non-deferred transitions on input character `a`. In both cases, the overlay offsets will also be the same for all six state transitions. So all six overlays will have the same decision, and will bit-merge in the overlay classifier. Figure 6.13(d) shows the (predicates of the) rules in the minimized overlay classifier for both cases. For the case without padding, we can only get down to two rules from six rules. In the case with padding, the overlays $3 = 0011$ and $7 = 0111$ are empty overlays, and hence will have $\circledast$decision during bit-merging. As a result, we can merge all six rules into a single rule.

6.5.4.2 Eliminating Overlay Bits

We modify the OD^{2}FA merge algorithm to eliminate unnecessary overlay ID bits and thus reduce the required TCAM entry width. The idea behind doing a cross product of overlays while merging is to capture the replication of states. Replicated states get assigned

to different overlays in the same super-state. However, sometimes there is no replication and we do not need to create extra overlays. For example, consider the merging of the OD^{2}FAs for REs/ab.⋆cd/ and /ab.⋆ef/. The two input OD^{2}FAs will both have two overlays 0 and 1, so in the merged OD^{2}FA we will create four overlays 0, 1, 2, and 3. In this case, since both REs have a common prefix, there is no state replication and overlays 1 and 2 will be empty in the merged OD^{2}FA. The two filled overlays, 0 and 3, have overlay IDs 00 and 11. Since the two overlays differ in both the bits, either bit is redundant and can be removed from the overlay ID producing only two overlays 0 and 1. In general, after merging two OD^{2}FAs, we eliminate as many overlay ID bits as possible by searching for overlay ID bits i where in every pair of overlays whose overlay ID differs only in bit i, at least one of the two overlays is empty. If bit i is eliminated, one empty overlay from each pair that differ in bit i is removed. We note that the overlay count stays a power of 2.

6.6 OD^{2}FA Software Implementation

In this section we discus the implementation of OD^{2}FA in software on a general purpose processor. We first review the implementation of DFA and D^{2}FA in software, then present our proposed implementation of OD^{2}FA.

Implementation of any finite automta mainly involves choosing a data structure to store the transition function and then implementing the lookup function using the given data structure. In a DFA $(Q, \Sigma, q_0, M, \delta)$, each state in Q has $|\Sigma|$ transitions. The transition

function δ can be stored in memory as a 2-dimensional array of next state values, indexed over Q and Σ. Looking up the next state requires just one memory lookup in the array using the current state and input character as indices. If we assume a 4 byte state ID value, then the amount of memory required to implement the transition function is $|Q| \times |\Sigma| \times 4$ bytes.

For a D^{2}FA $(Q, \Sigma, q_0, M, \rho, F)$, each state in Q has 0 to $|\Sigma|$ transition plus the deferment pointer. Most states have only a couple of transitions. So the transitions for each state can be stored as a list of (current character, next state) pairs in memory. To do a lookup, we go through the list of transitions for the current state to check if there is a transition on the current input character or not. If there is one, we get the next state, otherwise we go to the deferred state of the current state and check its transition table. The amount of memory required to implement the transition function is $\#\ transitions\ in\ \rho \times 5$ bytes for the transitions and $|Q| \times 4$ bytes for the deferment pointers.

6.6.1 Implementing OD^{2}FA

We now discuss the implementation for an OD^{2}FA $(Q, \Sigma, q_0, \mathcal{F}, \mathcal{S}, \mathcal{O}, \mathcal{M}, \Delta)$. All of the fields of an OD^{2}FA are simple to implement except for Δ. To implement Δ, we use a structure similar to that of a D^{2}FA except that instead of storing next state values, we store pointers to overlay classifiers. Specifically, for each super-state, we store a list of (current character, pointer to overlay classifier) pairs in memory for each character that is not deferred. Note that a character may be deferred for some overlays, but we say it is

not deferred if there is at least one overlay where it is not deferred.

Given the current super-state S, current overlay O and current character σ, the lookup is done as follows. We go through the transition list for the super-state S to check if there is an entry for character σ. If there is no entry for σ, we perform the lookup using the deferred super-state for S $\mathcal{F}(S)$. If there is an entry for σ, that gives us the location of the overlay classifier to use. We do a lookup in this overlay classifier for overlay O (we discuss next how to do this). If we find a match, the decision gives us the next super-state and overlay values. If we do not find a match, then overlay O is deferred for character σ, so we again perform the lookup using the deferred super-state for S $\mathcal{F}(S)$.

6.6.2 Overlay Classifier Storage and Lookup

An overlay classifier is just a list of rules. Each rule has a rule predicate, which is a ternary value, and a rule decision, which is a triple of next super-state, overlay value and the offset bit. If we use 4 byte overlay id values, then the rule predicate can be stored using two 4 byte values. One value will be the ternary position mask of the rule predicate and the other value will be the binary bit mask of the rule predicate. The rule decision can also be stored as two 4 byte values, one for the next super-state and the other for the overlay value. The single offset bit can be encoded in either of these two values. We would just store the list of rules in memory requiring 16 bytes per rule.

The lookup for an overlay O is done as follows. We just go through the list of rules and check if any rule matched the overlay O. To check if a rule r matches overlay O, we need

to check if the rule predicate $\mathbb{P}(r)$ covers O. $\mathbb{P}(r)$ will cover O if all the bit locations that contain a binary bit in $\mathbb{P}(r)$ have the same bit in both $\mathbb{P}(r)$ and O. This check may be done very efficiently using just one bitwise OR by testing $(O \mid \tau(\mathbb{P}(r))) = \beta(\mathbb{P}(r))$.

6.6.3 Space Requirement

For the OD^{2}FA, we need $|\mathcal{S}| \times 4$ bytes to store the super-state deferment pointers and roughly $|\mathcal{S}|$ bytes to store the super-state match function $\mathcal{M}$. If $m = \Sigma_{S \in \mathcal{S}}(\#$ of non-deferred characters for $S)$, then we need $m \times 5$ bytes to store the overlay classifier pointers. We optimize the size required to store the overlay classifiers using the following observation. The same overlay classifier may be used by multiple super-states for multiple characters. Rather than storing the same overlay classifier multiple times, we store one copy of each unique overlay classifier. In each super-state transition list, the same pointer is used by each entry that points to the same overlay classifier. The memory required to store the overlay classifiers will be 16 times the total number of rules among all the unique overlay classifier stores.

6.7 OD^{2}FA Implementation in TCAM

In this section, we describe how OD^{2}FA can be implemented in TCAM and present our OverlayCAM algorithm for doing so. We extend our solution of the RegCAM algorithm described in Chapter 5 to implement the OD^{2}FA in TCAM. The RegCAM implementation

uses two tables to represent an automata: a TCAM lookup table with a source state ID column and an input character column, and a corresponding SRAM decision table which contains the next state ID. To implement OD^{2}FA in TCAM, we use the unique pair of super-state ID and overlay ID as source state ID in the TCAM lookup table and next state ID (which is pair of next super-state ID and next overlay ID) in the SRAM decision table. The super-state ID and overlay ID columns in TCAM will be filled with ternary values that together match multiple states rather than a single state whereas the super-state ID and overlay ID columns in SRAM will be binary values that together give a single state. We add an extra bit in the SRAM decision table to specify the overlay bit in the super-state transition decision. Just as in RegCAM, we leverage the first match feature of TCAMs to ensure that the correct transition will be found in the TCAM lookup table. Specifically, if super-state S defers to super-state S', then we list all the super-state transitions for super-state S before those of super-state S'. We describe the specific challenges of implementing OD^{2}FA in TCAM including dealing with super-states, overlays, and super-state transitions in the remainder of this section.

6.7.1 Generating Super-state IDs and Codes

For the super-states, we apply the shadow encoding algorithm described in Section 5.2.2.3 on the super-state deferment forest of the given OD^{2}FA. This generates a binary super-state ID SSID(S) and a ternary super-state shadow code SSCD(S) for each super-state S that satisfies the Shadow Encoding Properties (SEP). Figure 6.6(c) shows the SSIDs and

SSCDs generated for that OD^{2}FA.

6.7.2 Implementing Super-state Transitions

We now address the implementation of super-state transitions in TCAM. Let $(S_1, X) \xrightarrow{\sigma} (S_2, o, b)$ be the super-state transition we want to implement in TCAM. In the TCAM table, we use $SSCD(S_1)$ in the super-state ID column. Since we restrict the set of overlays in any super-state transition to ternary values, we can just use X in the overlay ID column of the TCAM. For the SRAM, in the super-state ID column, we use $SSID(S_2)$, In the overlay ID column, we use the binary representation of the overlay value o. And the offset bit b is stored in the offset bit location in the SRAM.

The RE matching process works as follows. Let S be the current super-state, O be the current overlay, and σ the current input character. So $s = SSID(S) \cdot O$ denotes the current state; s concatenated with σ is used as a TCAM lookup key. Let uid be the SSID stored in super-state ID column in SRAM and o be the value stored in the overlay ID column in SRAM and b be the value of the offset bit stored in SRAM. We compute the next super-state ID and overlay ID as follows. The next super-state ID will be uid. The next overlay ID will be $(b \times \mathcal{O}(s) + o) \bmod |\mathcal{O}|$. If $b = 0$, the next overlay ID is simply o. If $b = 1$, the next overlay ID is $(\mathcal{O}(s) + o) \bmod |\mathcal{O}|$. In most cases where $o = 0$, the next overlay ID is $(\mathcal{O}(s) + 0) \bmod |\mathcal{O}| = \mathcal{O}(s)$. For example, consider the OD^{2}FA in Figure 6.6(c). We represent the super-state transition $\Delta(0_3, \{0, 1\}, a) = (3_3, 0, 1)$ as follows. The TCAM super-state ID column is filled with $SSCD(0_3) = ***$, the TCAM overlay ID column is $0*$,

the SRAM super-state ID column is filled with $SSID(3_3) = 011$, the overlay ID column is filled with 0, and the offset bit is set to 1.

6.7.3 TCAM Table Generation

We now explain how we generate the TCAM entries for OD^{2}FA. We generate the TCAM entries for one super-state at a time. Say S is the current super-state. We use the overlay classifiers of super-state S to generate its TCAM rules. For each character for which S has an overlay classifier, we add a TCAM entry for each rule in the overlay classifier as described in the previous section. After building this initial TCAM table for S, we reduce the TCAM entries as follows. We apply the bit merging algorithm explained in Section 6.5.3.2 on the TCAM entries generated for the super-state. The predicate of each rule corresponding to the TCAM entries has three parts: the current super-state code $SSCD(S)$, the overlay set X, and the current input character. The $SSCD(S)$ part will be the same in all TCAM rules for S, and the bit merging algorithm was already applied on the overlay field while building the overlay classifiers, so we cannot merge TCAM rules using any bits from these two fields. However, we can merge rules based on the current input character field. This mostly helps with case insensitive searches where transitions on the alphabet characters will mostly occur in pairs and such pairs can be merged because they differ on only one bit in ASCII encoding.

We order the TCAM tables of the super-states according to the super-state deferment relationship (every super-state table occurs before its deferred super-state table).

The overlay classifiers for the root super-state exclude all the self-looping transitions. All of these transitions are handled by the last rule added in the TCAM which is all $*$s.

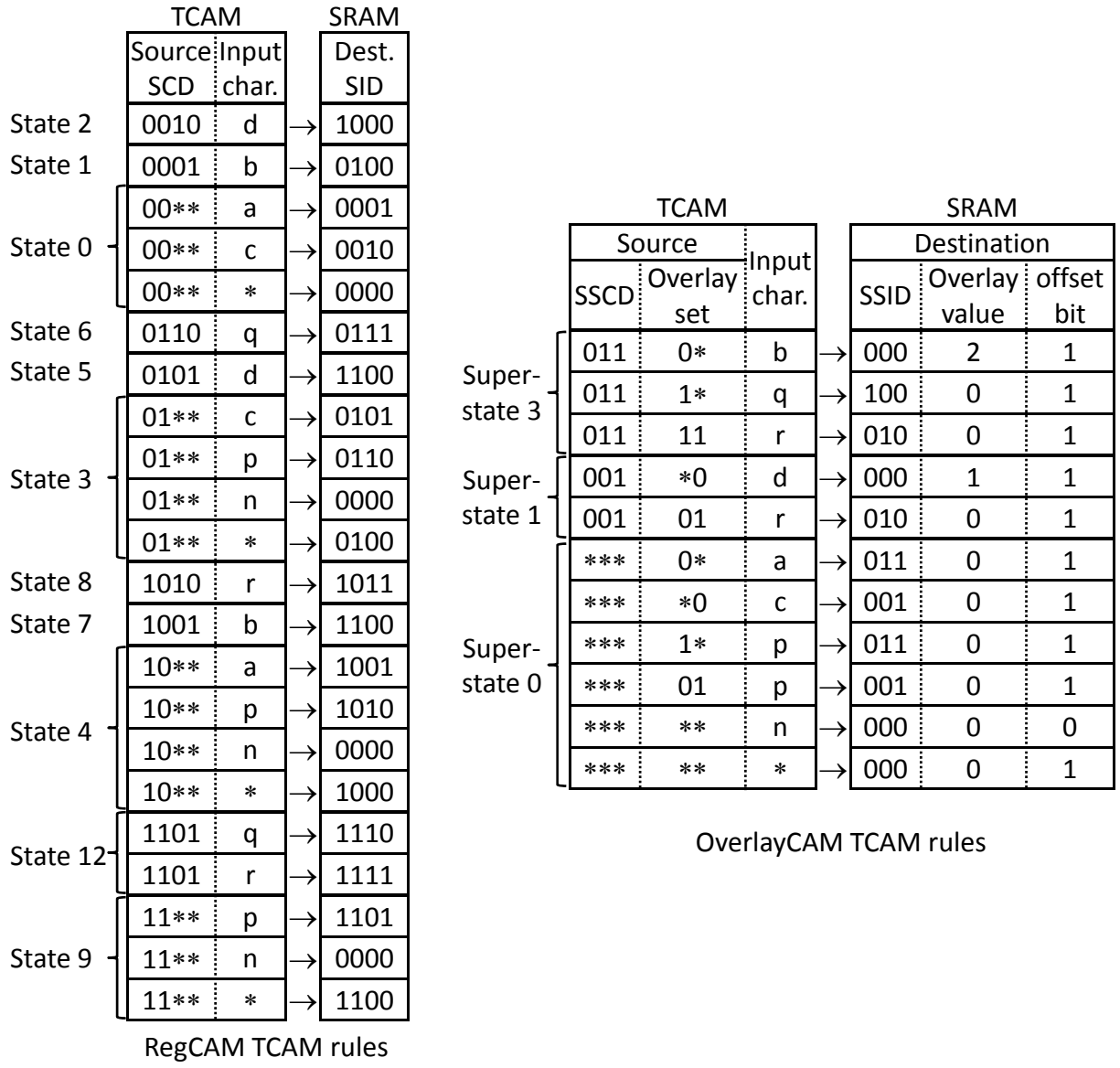


Figure 6.14: TCAM rules for RegCAM and OD^{2}FA.

Figure 6.14 shows the final TCAM and SRAM tables for the OD^{2}FA in Figure 6.6, and, for comparison purposes, the TCAM and SRAM tables generated by the RegCAM algorithm for the same RE set {`/ab[^n]*pq/`, `/cd[^n]*pr/`}.

6.7.4 Variable Striding

In this section, we describe how we adapt the technique of variable striding introduced in Section 5.4 to use with OD^{2}FA. We first explain the basic idea of a variable striding in a DFA. Creating a full k-stride DFA leads to space explosion because of two reasons. First each state in a k-stride DFA has $|\Sigma|^k$ transitions. This leads to transition explosion. Second, anytime a k-stride transition passes through an accepting state, we might need to create multiple copies of the destination state in order to record the matching. This leads to state explosion.

A k-var-stride DFA handles both these problems by generating variable (between 1 and k) stride transitions. The transition decision stores the stride length of the transition along with the destination state. The problem of transition explosion is managed by selectively extending the stride of a limited number of transitions. The problem of state explosion is eliminated by never extending a transition past an accepting state.

There are two implementations of variable striding that we considered in Section 5.4, self-loop unrolling and full variable striding.

6.7.4.1 Self-loop Unrolling

The self-loop unrolling technique for the OD^{2}FA works in the same way as for the D^{2}FA as presented in Section 5.4. The basic idea behind self-loop unrolling is as follows. The last rule in the TCAM table for the root super-state is always the self-loop rule which handles

all the self-looping transitions for all the states in the root super-state. For example consider the TCAM table for the root super-state (0) in Figure 6.14, which is also shown in Figure 6.15(a).

Consider the lookup when the next two input characters are `xa` and 0 is the current super-state. On the first input character `x`, we will match the last self-loop rule. This indicates that after processing the current character, we return to the same state. We can replace the last self-loop rule with another copy of super-state 0s TCAM table with the input character over the second stride and $*$s in the first stride. This is shown in Figure 6.15(b) with this second copy of the rules marked as Stride-2. If we do a lookup for `xa`, we will match the first Stride-2 rule. Thus, instead performing two lookups in the 1-stride table, we get the same decision by performing one lookup in the unrolled 2-stride table.

If we unroll the self-loop rule at the end of the second copy of the TCAM rules one more time, we get the table shown in Figure 6.15(b). We can further unroll the self-loop rule to extend to a k-stride table. If the 1-stride TCAM table has n rules, then the self-loop unrolled k-stride table will have only $(n-1)k+1$ rules.

6.7.4.2 Full Variable Striding

Adapting the full variable striding technique for the OD^{2}FA is more challenging. The k-var-stride transition sharing algorithm presented in Section 5.4 generates k-var-stride tables which correctly handle state deferment in the D^{2}FA. What we mean by this is the

TCAM				SRAM		
Source		Input char.		Destination		
SSCD	Overlay set			SSID	Overlay value	offset bit
***	0*	a	→	011	0	1
***	*0	c	→	001	0	1
***	1*	p	→	011	0	1
***	01	p	→	001	0	1
***	**	n	→	000	0	0
***	**	*	→	000	0	1

(a) 1-stride table for super-state 0.

	TCAM						SRAM			
	Source		Input				Destination			
	SSCD	Overlay set	char1	char2	char3		SSID	Overlay value	offset bit	Stride
Stride 1	***	0*	a	*	*	→	011	0	1	1
	***	*0	c	*	*	→	001	0	1	1
	***	1*	p	*	*	→	011	0	1	1
	***	01	p	*	*	→	001	0	1	1
	***	**	n	*	*	→	000	0	0	1
Stride 2	***	0*	*	a	*	→	011	0	1	2
	***	*0	*	c	*	→	001	0	1	2
	***	1*	*	p	*	→	011	0	1	2
	***	01	*	p	*	→	001	0	1	2
	***	**	*	n	*	→	000	0	0	2
Stride 3	***	0*	*	*	a	→	011	0	1	3
	***	*0	*	*	c	→	001	0	1	3
	***	1*	*	*	p	→	011	0	1	3
	***	01	*	*	p	→	001	0	1	3
	***	**	*	*	n	→	000	0	0	3
	***	**	*	*	*	→	000	0	1	3

(b) Super-state 0 table unrolled to 3-var-stride.

Figure 6.15: Root super-state self loop unrolling example for TCAM rules in Figure 6.14.

following. Suppose s_1 is the current state and it defers to state s_2. If we lookup a character and match a rule from state s_2's TCAM table giving the next state s_3, then state s_1 also transitions to state s_3 on the same input. In general, a match found in the TCAM table of an ancestor of s_1 when doing a lookup for s_1 will always be correct.

We cannot extend the k-var-stride transition sharing algorithm to OD^{2}FA to generate tables that correctly handle deferment. The difficulty arises from the following. In an OD^{2}FA, each super-state has multiple states. On the same input, different states in the same super-state might transition to states in different super-states. Thus, we propose an alternate technique to generate variable stride tables.

For each super-state S, we generate a k-var-stride table in addition to its 1-stride table. When the k-var-stride table is implemented in TCAM, in the current super-state column of the TCAM, we use SSID(S) instead of the SSCD(S). That way, the k-var-stride rules of super-state S will only match when doing a lookup for itself, and will not match when doing a lookup for any other super-state. So the k-var-stride rules only have to be correct for S. The k-var-stride table for S is placed just before its 1-stride table in TCAM, so higher priority is given to k-var-stride rules over the 1-stride rules.

We now explain our algorithm to generate the k-var-stride table for a super-state. We define the variable stride transition function as Γ: $\mathcal{S} \times 2^{\mathcal{O}} \times (\bigcup_{1 \leq i \leq k} \Sigma^i) \rightarrow \mathcal{S} \times [0..|\mathcal{O}|) \times \{0, 1\}$, which is same as Δ except that Γ transitions over a string of characters of length between 1 and k. Let S be the super-state for which we are generating the k-var-stride transitions. For each 1-stride transition for super-state S, we build k-var-stride transitions

by extending the transitions of super-state S_2 with that transition in two ways: first by composing with S_2's k-var-stride table, then by composing with S_2's 1-stride table. More specifically, let $(S, X) \xrightarrow{\sigma} (S_1, o_1, 1) \in \Delta$ be any 1-stride transition for S such that $S < S_1$ and $\mathcal{M}(S_1) = \emptyset$. We add the condition $S < S_1$ because we only want to extend forward transitions and this condition is true for most forward transitions. We add the condition $\mathcal{M}(S_1) = \emptyset$ because we stop a variable stride transition at matching super-states.

If we have not already built the k-var-stride transition table for super-state S_1, we recursively build it first. Then we first extend the transitions in the k-var-stride table of S_1: for each transition $(S_1, Y) \xrightarrow{w} (S_2, o_2, 1)$ in the k-var-stride transition table of S_1, if $|X \cap Y|$ is large enough and $len(w) < k$, we add the extended transition $(S, X \cap Y) \xrightarrow{\sigma.w} (S_2, (o_1 + o_2) \bmod |\mathcal{O}|, 1)$ to the k-var-stride transition table for S. Next we extend the transitions in the 1-stride table of S_1: for each transition $(S_1, Y) \xrightarrow{\sigma_2} (S_2, o_2, 1)$ in the 1-stride transition table of S_1, if $|X \cap Y|$ is large enough, we add the extended transition $(S, X \cap Y) \xrightarrow{\sigma.\sigma_2} (S_2, (o_1 + o_2) \bmod |\mathcal{O}|, 1)$ to the k-var-stride transition table for S. We use the condition $|X \cap Y| \geq \min(|X|, |Y|)/4$ as the measure for large enough in our experiments. When we extend one transition to the next, the extended transition can only cover overlays that are common in both initial transitions. Ideally we would like both transitions to cover the exact same set of overlays (in most cases this is true). But even when we do not have the same overlay set, if the size of the intersection is significant compared to the number of overlays covered by the two initial transitions, it is worthwhile to add the extended transition. We do not extend 1-stride transitions that are on the whitespace characters.

We have found experimentally that extending 1-stride transitions on these characters significantly increases the number of TCAM rules while only marginally (if at all) increasing the average stride. Figure 6.16 shows the k-var-stride transition table built for super-state O from the 1-stride transition tables in Figure 6.9.

Super-state 0 rule	Next super-state rule	Extended var-stride rule
$(0, 0*) \xrightarrow{a} (3, 0, 1)$	$(3, 0*) \xrightarrow{b} (0, 2, 1)$	$(0, 0*) \xrightarrow{ab} (0, 2, 1)$
$(0, *0) \xrightarrow{c} (1, 0, 1)$	$(1, *0) \xrightarrow{d} (0, 1, 1)$	$(0, *0) \xrightarrow{cd} (0, 1, 1)$
$(0, 1*) \xrightarrow{p} (3, 0, 1)$	$(3, 1*) \xrightarrow{q} (4, 0, 1)$	$(0, 1*) \xrightarrow{pq} (4, 0, 1)$
$(0, 01) \xrightarrow{p} (1, 0, 1)$	$(1, 01) \xrightarrow{r} (2, 0, 1)$	$(0, 01) \xrightarrow{pr} (2, 0, 1)$

Figure 6.16: variable stride transitions generated for super-state 0 from 1-stride transition in Figure 6.9.

The pseudo-code of our algorithm for building the k-var-stride transition tables is shown in Algorithm 6.17.

6.8 Experimental Results

We implemented OverlayCAM using C++ and conducted experiments to evaluate its effectiveness and scalability. We verify our results by confirming that the TCAM table generated by OverlayCAM is equivalent to the original DFA. That is, for every pair of current state and input character, the next state returned by the TCAM lookup matches the next state returned by the DFA.

Input: OD^{2}FAs, $\mathcal{D} = (Q, \Sigma, q_0, \mathcal{F}, \mathcal{S}, \mathcal{O}, \mathcal{M}, \Delta)$.
Output: Builds multi-stride transitions for $\mathcal{D}$.

foreach $S_i \in \mathcal{S}$ **do** Initialize Built[S_i] $\leftarrow$ False;
foreach $S_i \in \mathcal{S}$ **do**
  if Built[S_i] = False **then** `BuildVarStrideTrans` (S_i);

Function `BuildVarStrideTrans`(S)
  foreach *offset transition* $(S, X) \xrightarrow{c} (S_i, o, 1) \in \Delta$ *for super-state* S **do**
    if $S_i \leq S$ **then** **Continue**; `// skip backward transition`
    if $\mathcal{M}(S_i) \neq \emptyset$ **then** **Continue**; `// stop at accepting super-states`
    if Built[S_i] = False **then**
      `BuildVarStrideTrans` (S_i);
    `// extend var-stride transitions of destination super-state`
    foreach *transition* $(S_i, Y) \xrightarrow{w} (S_j, o_2, 1) \in \Gamma$ *for super-state* S_i **do**
      if $|X \cap Y| \geq \min(|X|, |Y|)/4$ **then**
        if $len(w) < k$ **then** `// max stride limit not reached`
          Add transition $(S, X \cap Y) \xrightarrow{c.w} (S_j, (o + o_2) \mod |\mathcal{O}|, 1)$ to Γ;
    `// extend 1-stride transitions of destination super-state`
    foreach *offset transition* $(S_i, Y) \xrightarrow{c_2} (S_j, o_2, 1) \in \Delta$ *for super-state* S_i **do**
      if $|X \cap Y| \geq \min(|X|, |Y|)/4$ **then**
        Add transition $(S, X \cap Y) \xrightarrow{c.c_2} (S_j, (o + o_2) \mod |\mathcal{O}|, 1)$ to Γ;
  Built[S] $\leftarrow$ True;

Figure 6.17: Algorithm BuildVarStrideOD2FA($\mathcal{D}$) to build k-var-stride rules.

6.8.1 Effectiveness of OverlayCAM

We use the same 8 RE sets used in Section 4.5 for the main results. We define the following metric for measuring the amount of state replication in the DFA that corresponds to an RE set. *For any RE set* R, *we define* SR(R) *to be the ratio of the number of states in the minimum state DFA corresponding to* R *divided by the number of states in the standard NFA without* ϵ *transitions corresponding to* R. Based on the characteristics of the REs, these eight sets are partitioned into three groups,

STRING ={C613, Bro217}, which contains mostly strings, causing little state replication (SR(Bro271) = 3.0, SR(C613) = 2.1); WILDCARD ={C7, C8 and C10}, which contains multiple wildcard closures '.⋆', causing lots of state replication (SR(C7) = 231, SR(C8) = 43, and SR(C10) = 162); and SNORT ={Snort24, Snort31, and Snort34}, which contain a diverse set of REs, roughly 40% of the REs have wildcard closures, causing moderate state replication (SR(Snort24) = 24, SR(Snort31) = 22, and SR(Snort34) = 16).

We conducted side-by-side comparison with RegCAM-TC (RegCAM without Table Consolidation) and RegCAM+TC (RegCAM with Table Consolidation) on all 8 real-world RE sets. For RegCAM+TC, we consolidated 4 tables together. The results are shown in Table 6.1. For TCAM space, we only report the number of TCAM entries because the TCAM widths for all TCAM tables generated by RegCAM-TC, RegCAM+TC, and OverlayCAM on all 8 RE sets. Since TCAM width typically is only allowed to be configured as 36, 72, or 144 bits, we use a TCAM width of 36 in all cases.

RE set	# NFA States	SR	# NFA Trans.	# Over-lays	# Super states	# TCAM entries			SRAM size (Kb)			Throughput (Gbps)		
						RegCAM -TC	RegCAM +TC	Overlay CAM	RegCAM -TC	RegCAM +TC	Overlay CAM	RegCAM -TC	RegCAM +TC	Overlay CAM
C8	72	43.17	2177	72	85	3722	1012	125	47.25	51.39	1.83	5.44	8.51	12.50
C10	92	161.61	2982	288	133	17824	4739	263	261.09	277.68	4.62	3.11	4.35	12.12
C7	107	231.31	3261	648	127	29196	8315	234	456.19	519.69	4.57	3.11	3.64	12.31
Snort24	575	24.15	4054	30	897	16130	5310	1426	236.28	331.88	26.46	3.64	4.35	7.27
Snort34	891	15.52	4731	48	1151	16297	5026	2293	238.73	294.49	42.55	3.64	4.35	5.44
Snort31	917	21.88	5738	32	2395	41539	14464	9478	689.61	960.50	185.12	2.72	3.64	3.64
Bro217	2132	3.06	5424	2	3401	9143	5087	6028	133.93	317.94	88.30	3.64	4.35	4.35
C613	5343	2.12	14563	1	11308	18256	13182	18256	320.91	978.35	338.73	3.11	3.64	3.11

Table 6.1: Experimental results of OverlayCAM on 8 RE sets in comparison with RegCAM-TC and RegCAM+TC

TCAM lookup speed is typically higher for smaller TCAM chips. We use the TCAM model discussed in Section 5.5 to calculate RE matching throughput. For the two string-based RE sets Bro217 and C613, we observe that OverlayCAM does not significantly outperform the two RegCAM algorithms. This is expected as OverlayCAM is designed to handle state replication and string-based RE sets have little state replication.

For the other RE sets, OverlayCAM significantly outperforms RegCAM and often outperforms NFAs. (1) *OverlayCAM uses orders of magnitude less TCAM and SRAM than RegCAM.* On average, OverlayCAM uses 41 times less TCAM and 33 times less SRAM than `RegCAM-TC` and 12 times less TCAM and 38 times less SRAM than `RegCAM+TC`. (2) *OverlayCAM has significantly higher throughput than RegCAM.* On average, OverlayCAM has 2.5 and 1.93 times higher throughput than `RegCAM-TC` and `RegCAM+TC`, respectively. (3) *The total number of TCAM entries used by OverlayCAM is often (far) smaller than the total number of NFA transitions.* For C7, OverlayCAM's number of TCAM entries is 14 times less than the number of NFA transitions.

We now describe why OverlayCAM performs so well. (4) *OverlayCAM is very effective in conquering state replication.* OverlayCAM effectively and automatically identifies all NFA state replicates and groups them together into super-states. The number of super-states is, on average, 1.55 times the number of NFA states and is never more than 2.61 times the number of NFA states. Because of this, the larger SR(R) is, the more that OverlayCAM outperforms RegCAM. For C7, OverlayCAM uses 125 times less TCAM and 100 times less SRAM than `RegCAM-TC` and 36 times less TCAM and 114 times less SRAM

than RegCAM+TC. (5) *OverlayCAM effectively multiplies the compression benefits of conquering state replication and transition sharing.* That is, OverlayCAM effectively multiplies the benefits of ODFA and D^2FA. The average number of TCAM entries per super-state is only 2.14, even when super-states have hundreds of constituent states.

We wanted to conduct side-by-side comparison with Peng *et al.*'s scheme [38]; however, we do not have access to their code. Fortunately, Peng *et al.* have reported their results on the two public RE sets Snort24 and Snort34. For these two sets, OverlayCAM requires 2.15 and 1.44 times less TCAM and SRAM space.

6.8.2 Results on 7-var-stride

We now compare the results of applying the variable striding technique with $k = 7$ on OverlayCAM with the results for RegCAM–TC. We compare the average stride values achieved using the same traces that were used for the experiments in Section 5.6.3 as well as the number of TCAM rules.

We only compare using the RE sets in the WILDCARD and SNORT groups since the RE sets in the STRING group have no (or limited) state replication.

6.8.2.1 Self-loop Unrolling

The root state in both RegCAM–TC and OverlayCAM are exactly the same since the self-looping states are selected as the root states. As a result, the resulting TCAM rules

after unrolling the roots states are semantically equivalent. Hence we get the exact same average stride values for both algorithms (which are shown in Table 6.3). Table 6.2 shows the number of TCAM rules required without self-loop unrolling (*i.e.* for 1-stride) and with self-loop unrolling for both algorithms.

RE set	RegCAM–TC			OverlayCAM		
	1-stride	Unroll	7-var-stride	1-stride	Unroll	7-var-stride
C8	3722	7794	8192	125	310	814
C10	17824	36336	65536	263	590	1113
C7	29196	64356	65536	234	442	1381
Snort24	16130	18627	32768	1426	1482	6942
Snort34	16297	19825	32768	2293	2577	9654
Snort31	41539	43920	65536	9478	9819	32243

Table 6.2: Number of TCAM rules for RegCAM–TC and OverlayCAM for 1-stride, with self-loop unrolling and with 7-var-stride

Compared to RegCAM–TC, OverlayCAM requires on average 77 times fewer TCAM rules for the WILDCARD group and 8 times fewer TCAM rules for the SNORT group. The average percentage increase in the number of TCAM rules resulting from unrolling the roots for the SNORT group is 14.3% for RegCAM–TC and only 6.6% for OverlayCAM. This is because in RegCAM–TC, there are many root states that are unrolled. On the other hand, in OverlayCAM, there is only one root super-state that is unrolled.

6.8.2.2 Full Variable Striding

Table 6.2 shows the number of TCAM rules required for full variable striding, and Table 6.3 shows the average stride values for RegCAM–TC and OverlayCAM. As we can see, OverlayCAM requires many fewer TCAM rules than RegCAM–TC. On average OverlayCAM

requires 38.8 times fewer rules for the WILDCARD group and 3.4 times fewer TCAM rules for the SNORT.

RE set	Self-loop unroll			7-var-stride					
				RegCAM-TC			OverlayCAM		
	0	50	95	0	50	95	0	50	95
C8	6.1	2.9	1.8	6.1	4.1	2.9	6.1	3.8	3.7
C10	5.9	3.4	1.9	6.0	4.5	3.2	5.9	4.1	3.6
C7	6.1	1.9	1.8	6.1	3.7	3.8	6.1	2.7	3.8
Snort24	5.6	1.7	1.1	5.7	2.9	3.6	5.6	2.4	4.0
Snort34	5.9	1.7	1.1	5.9	3.4	3.7	5.9	2.5	4.1
Snort31	6.1	1.7	1.1	6.2	2.8	2.3	6.1	2.3	2.9

Table 6.3: Average stride values for self-loop unrolling and 7-var-stride for RegCAM-TC and OverlayCAM for $p_M = 0$, 50 and 95.

In general OverlayCAM is able to achieve nearly the same average stride values as RegCAM-TC. For random traffic ($p_M = 0$), OverlayCAM has nearly identical average stride value as RegCAM-TC. This is because with random traffic, most of the transitions taken are self-loops around the root state, whichh are unrolled to 7-stride in both algorithms. For $p_M = 95$, OverlayCAM is able to achieve equal or higher average stride value than RegCAM-TC for all the RE sets. This is because with $p_M = 95$, most of the transitions taken are forward transitions, and OverlayCAM is able to selectively combine longer chains of forward transitions into higher stride transitions than RegCAM-TC. The average of the ratio of the stride values across all RE sets and p_M values is only 1.09.

6.8.3 Scalability of OverlayCAM

We evaluated the scalability of OverlayCAM on synthetic RE sets constructed by adding new REs from 13 REs from a recent release of the Snort rules one at a time. Each RE

contains closure on the wildcard or a range; these cause the DFA size to double as each RE is added. The final DFA has 225,040 states.

We first define the *TCAM Expansion Factor (TEF)* of an RE set to be the number of TCAM entries divided by the number of NFA transitions. In Figure 6.18(a), we plot the TEF for `RegCAM-TC`, `RegCAM+TC` and OverlayCAM. We omit the first 5 data points because the corresponding 5 DFAs are too small. As expected, the TEF of the RegCAM algorithms grows exponentially with the number of NFA states due to state replication. In contrast, the TEF of OverlayCAM grows linearly at a very slow growth rate with the number of NFA states. We next define the *super-state expansion factor (SEF)* of an RE set to be the number of super-states divided by the number of NFA states. Figure 6.18(b) shows that the SEF of OverlayCAM also grows linearly and slowly with the number of NFA states. Note that for any RE set, the number of NFA states is the minimum compared to any other automaton.

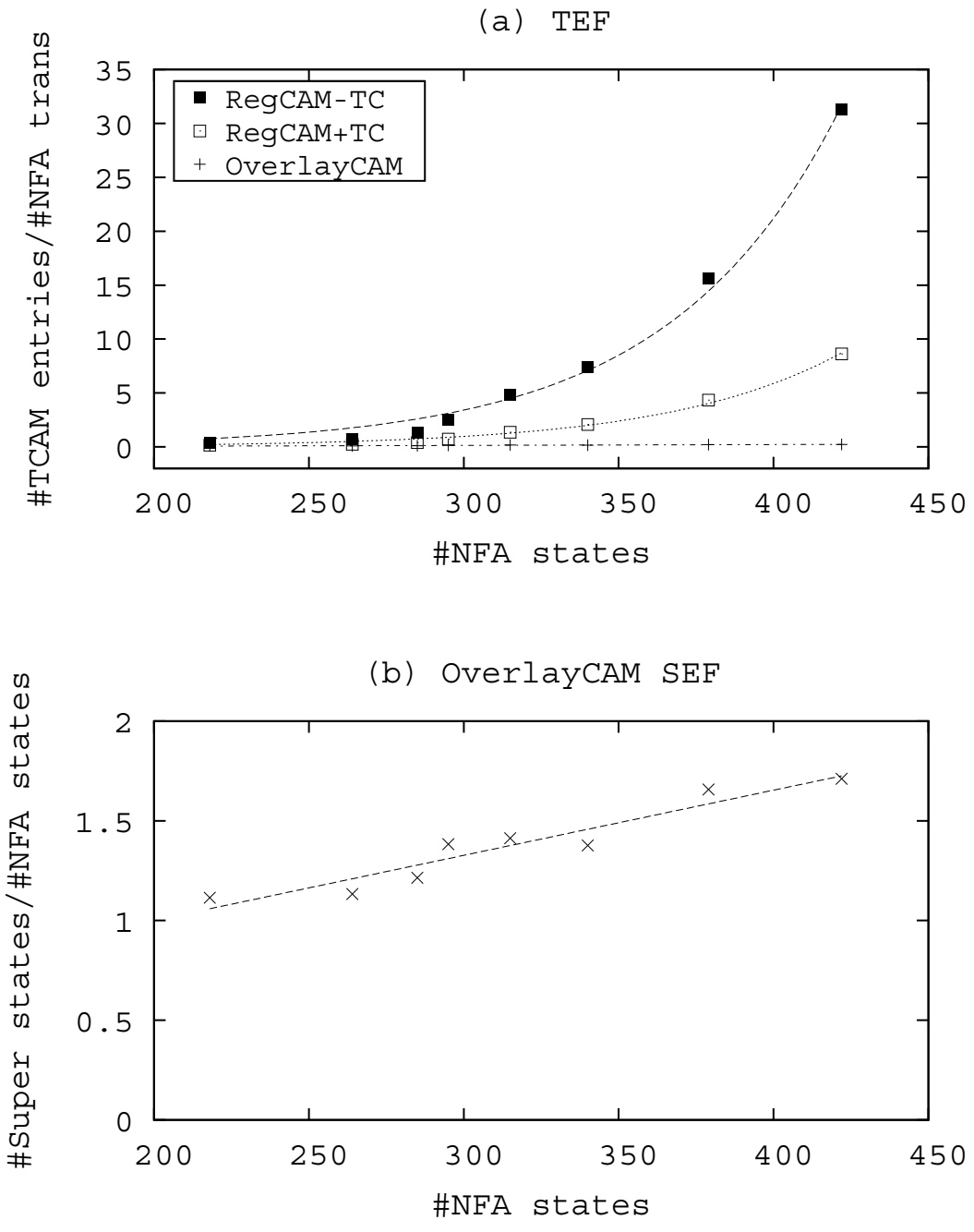


Figure 6.18: (a) TEF vs. # NFA states for OverlayCAM and RegCAM, (b) SEF vs. # NFA states for OverlayCAM

Chapter 7

Conclusion

In this dissertation, we consider the problem of RE matching in DPI for networking applications. We survey current solutions for RE matching for DPI and identify their limitations. We then develop several techniques and algorithms for fast and efficient RE matching.

For a software solution of RE matching, we use an existing automata model D^2FA. We propose a novel Minimize then Union framework and develop efficient algorithms for building D^2FA based on the framework. Our approach requires a fraction of the memory and time required by current algorithms. This allows us to build much larger D^2FAs than was possible with previous techniques. Our algorithm naturally supports frequent RE set updates. We conducted experiments on real-world and synthetic RE sets that verify our claims. For example, our algorithm requires an average of 1400 times less memory and 300 times less time than the original D^2FA construction algorithm of Kumar *et al.*. We believe our Minimize then Union framework can be incorporated with other alternative

automata for RE matching.

We propose the first TCAM-based RE matching solution. We prove that this unexplored direction works very well for RE matching. We implemented our techniques and conducted experiments on real-world RE sets. We show that small TCAMs are capable of storing large DFAs. For example, in our experiments, we were able to store a DFA with 25K states in a 0.5Mb TCAM chip. We also develop multi-striding techniques to increase matching throughput wihtout significantly increasing the memory requirement. We are able to achieve a matching throughput of nearly 20Gbps.

The D^2FA and our TCAM-based solution only partially handles the problem of state replication in a DFA. We propose a new overlay automata model called the OD^2FA, which fully exploits state replication in a DFA. We develop algorithms for efficiently constructing the OD^2FA. We also develop techniques to implement the OD^2FA in software and in hardware using TCAMs. Our experiments indicate that OD^2FA is able to effectively manage state replication. This results in a memory requirement proportional to that of a NFA while maintaining fast and deterministic matching throughput like that of a DFA.

APPENDICES

Glossary

character redundancy Redundant/shared transitions within a state. **17**, 78

deferment forest The directed graph with states as vertices and edges given by the deferment relation F. **22**, 211

deferment pointer The deferred state, $F(s)$, of a state s. **23**

deferment tree A tree (connected component) in the deferment forest. **23**

self-looping state State with more than $\Sigma/2$ of its transitions looping back to itself. **16**

state redundancy Redundant/shared transitions between two states. **17**, 78

state replication Multiple replications of same NFA state in a DFA when DFAs for two REs are combined. **14**, 77, 78, 131

transitions sharing Multiple transitions within a state or between different states going to the same next state. **14**, 77, 78

Acronyms

D^{2}FA Delayed Input DFA. 4, 19, 208, 209

DFA Deterministic Finite state Automata. ii, iii, **3**, 12, 209

DPI Deep Packet Inspection. ii, **1**, 208

NFA Nondeterministic Finite state Automata. iii, **3**, 209

OD^{2}FA Overlay Delayed Input DFA. 5, 134, 144, 209

ODFA Overlay Deterministic Finite state Automata. 5, 133, 136, 141

RE Regular Expression. ii, iii, **2**, 208, 209

SEP Shadow Encoding Properties. **86**, 91, 93, 94, 189

SRG Space Reduction Graph. **24**, 42

TCAM Ternary Content Addressable Memory. iii, **5**, 30, 209

Notation

D A DFA/D^{2}FA. **12**

$\mathcal{D}$ An ODFA/OD^{2}FA. **141**

Q Set of states in the DFA/D^{2}FA/ODFA/OD^{2}FA. **12**

Σ The input alphabet. **12**

$\mathcal{S}$ The set of super-states in an ODFA/OD^{2}FA. **141**

$\mathcal{O}$ The set of overlays an ODFA/OD^{2}FA. **141**

s, q, u A DFA/D^{2}FA/ODFA/OD^{2}FA state. **13**

S An ODFA/OD^{2}FA super-state. **141**

O An ODFA/OD^{2}FA overlay. **141**

X A set of overlays in an ODFA/OD^{2}FA. **140**

$M(s)$ Set of REs accepted by state s. **14**

$\mathcal{M}(S)$ Set of REs accepted by all states in super-state S. **141**

$F(s)$ Deferred state of state s. **19**, 20, 211

$\mathcal{F}(S)$ Deferred super-state of super-state S. **144**

$u \rightarrow v$ State u defers to state v. **23**

$u \twoheadrightarrow v$ State u descendant of state v. **23**

$\perp$ NULL state/empty location. 143

$\delta(s, \sigma)$ The state transition function for a DFA. 13

$\rho(s, \sigma)$ Partial state transition function for a D^2FA. 22

$\Delta(S, X, \sigma)$ Super-state transition function for a $ODFA/OD^2FA$. 141

$\rho'(s, \sigma)$ Partial state transition function derived from Δ for OD^2FA. 144

$\delta'(s, \sigma)$ Total transaction function derived from ρ for D^2FA. 22

$\delta''(s, \sigma)$ Total transaction function derived from Δ (ρ') for ODFA (OD^2FA). 142

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Application layer packet classifier for linux. http://l7-filter.clearfoundation.com/.

[2] Snort. http://www.snort.org/.

[3] B. Agrawal and T. Sherwood. Modeling TCAM power for next generation network devices. In *Proc. IEEE Int. Symposium on Performance Analysis of Systems and Software*, pages 120– 129, 2006.

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

[5] M. Alicherry, M. Muthuprasanna, and V. Kumar. High speed pattern matching for network ids/ips. In *Proc. 2006 IEEE International Conference on Network Protocols*, pages 187–196. Ieee, 2006.

[6] M. Becchi and S. Cadambi. Memory-efficient regular expression search using state merging. In *Proc. INFOCOM*. IEEE, 2007.

[7] M. Becchi and P. Crowley. A hybrid finite automaton for practical deep packet inspection. In *Proc. ACM Int. Conf. on emerging Networking EXperiments and Technologies (CoNEXT)*. ACM Press, 2007.

[8] M. Becchi and P. Crowley. An improved algorithm to accelerate regular expression evaluation. In *Proc. ACM/IEEE ANCS*, 2007.

[9] M. Becchi and P. Crowley. Efficient regular expression evaluation: Theory to practice. In *Proc. ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 50–59, 2008.

[10] M. Becchi and P. Crowley. Extending finite automata to efficiently match perl-compatible regular expressions. In *Proc. ACM Int. Conf. on emerging Networking EXperiments and Technologies (CoNEXT)*. ACM Press, 2008. Article Number 25.

[11] M. Becchi, M. Franklin, and P. Crowley. A workload for evaluating deep packet inspection architectures. In *Proc. IEEE IISWC*, 2008.

[12] A. Bremler-Barr, D. Hay, and Y. Koral. Compactdfa: Generic state machine compression for scalable pattern matching. In *Proc. IEEE INFOCOM*, pages 1–9. Ieee, 2010.

[13] B. C. Brodie, D. E. Taylor, and R. K. Cytron. A scalable architecture for high-throughput regular-expression pattern matching. *SIGARCH Computer Architecture News*, 2006.

[14] C. R. Clark and D. E. Schimmel. Efficient reconfigurable logic circuits for matching complex network intrusion detection patterns. In *Proc. Field-Programmable Logic and Applications*, pages 956–959, 2003.

[15] C. R. Clark and D. E. Schimmel. Scalable pattern matching for high speed networks. In *Proc. 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Washington, DC, 2004.

[16] J. Edmonds. Paths, trees, and flowers. *Canad. J. Math.*, 17:449–467, 1965.

[17] D. Ficara, S. Giordano, G. Procissi, F. Vitucci, G. Antichi, and A. D. Pietro. An improved DFA for fast regular expression matching. *Computer Communication Review*, 38(5):29–40, 2008.

[18] H. N. Gabow. An efficient implementation of edmonds' algorithm for maximum matching on graphs. *J. ACM*, 23:221–234, April 1976.

[19] J. E. Hopcroft. *The Theory of Machines and Computations*, chapter An nlogn algorithm for minimizing the states in a finite automaton, pages 189–196. Academic Press, 1971.

[20] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 2000.

[21] D. E. Knuth. Huffman's algorithm via algebra. *Journal of Combinatorial Theory, Series A*, 32(2):216 – 224, 1982.

[22] S. Kong, R. Smith, and C. Estan. Efficient signature matching with multiple alphabet compression tables. In *Proc. 4th Int. Conf. on Security and privacy in communication netowrks (SecureComm)*, page 1. ACM Press, 2008.

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

[24] H. W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2:83–97, 1955.

[25] S. Kumar, B. Chandrasekaran, J. Turner, and G. Varghese. Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia. In *Proc. ACM/IEEE ANCS*, pages 155–164, 2007.

[26] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In *Proc. SIGCOMM*, pages 339–350, 2006.

[27] S. Kumar, J. Turner, and J. Williams. Advanced algorithms for fast and scalable deep packet inspection. In *Proc. IEEE/ACM ANCS*, pages 81–92, 2006.

[28] T. Liu, Y. Yang, Y. Liu, Y. Sun, and L. Guo. An efficient regular expressions compression algorithm from a new perspective. In *Proc. IEEE INFOCOM*, pages 2129–2137, 2011.

[29] Y. Liu, L. Guo, M. Guo, and P. Liu. Accelerating DFA construction by hierarchical merging. In *Proc. IEEE 9th Int. Symposium on Parallel and Distributed Processing with Applications*, 2011.

[30] C. R. Meiners, A. X. Liu, and E. Torng. TCAM Razor: A systematic approach towards minimizing packet classifiers in TCAMs. In *Proc. 15th IEEE Conf. on Network Protocols (ICNP)*, pages 266–275, October 2007.

[31] C. R. Meiners, A. X. Liu, and E. Torng. Bit weaving: A non-prefix approach to compressing packet classifiers in TCAMs. In *Proc. 17th IEEE Conf. on Network Protocols (ICNP)*, October 2009.

[32] C. R. Meiners, J. Patel, E. Norige, E. Torng, and A. X. Liu. Fast regular expression matching using small TCAMs for network intrusion detection and prevention systems. In *Proc. 19th USENIX Security Symposium (USENIX Security)*, pages 111–126, Washington, DC, August 2010.

[33] A. Mitra, W. Najjar, and L. Bhuyan. Compiling PCRE to FPGA for accelerating SNORT IDS. In *Proc. 3rd ACM/IEEE Symposium on Architecture for networking and communications systems ANCS*. ACM Press, 2007.

[34] J. Moscola, J. Lockwood, R. P. Loui, and M. Pachos. Implementation of a content-scanning module for an internet firewall. In *Proc. 11th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 31–38. IEEE Comput. Soc, 2003.

[35] J. Munkres. Algorithms for the assignment and transportation problems. *Journal of the Society of Industrial and Applied Mathematics*, 5(1):32–38, March 1957.

[36] J. Patel, A. X. Liu, and E. Torng. Bypassing space explosion in regular expression matching for network intrusion detection and prevention systems. In *Proc. Network and Distributed System Security Symposium (NDSS'12)*, February 2012.

[37] V. Paxson. Bro: a system for detecting network intruders in real-time. *Computer Networks*, 31(23-24):2435–2463, 1999.

[38] K. Peng, S. Tang, M. Chen, and Q. Dong. Chain-based DFA deflation for fast and scalable regular expression matching using TCAM. In *Proc. ACM ANCS*, pages 24–35, 2011.

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

[40] R. Sidhu and V. K. Prasanna. Fast regular expression matching using fpgas. In *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines FCCM*, pages 227–238, 2001.

[41] R. Smith, C. Estan, and S. Jha. Xfa: Faster signature matching with extended automata. In *Proc. IEEE Symposium on Security and Privacy*, pages 187–201, 2008.

[42] R. Smith, C. Estan, S. Jha, and S. Kong. Deflating the big bang: fast and scalable deep packet inspection with extended finite automata. In *Proc. SIGCOMM*, pages 207–218, 2008.

[43] R. Sommer and V. Paxson. Enhancing byte-level network intrusion detection signatures with context. In *Proc. 10th ACM Conf. on Computer and Communications Security (CCS)*, pages 262–271, 2003.

[44] I. Sourdis and D. Pnevmatikatos. Pnevmatikatos: Fast, large-scale string match for a 10gbps fpga-based network intrusion detection system. In *Proc. Int. on Field Programmable Logic and Applications*, pages 880–889, 2003.

[45] I. Sourdis and D. Pnevmatikatos. Pre-decoded cams for efficient and high-speed nids pattern matching. In *Proc. 12th IEEE Symposium on FieldProgrammable Custom Computing Machines*, volume C, pages 258–267. Ieee, 2004.

[46] J.-S. Sung, S.-M. Kang, Y. Lee, T.-G. Kwon, and B.-T. Kim. A multi-gigabit rate deep packet inspection algorithm using tcam. In *Proc. IEEE GLOBECOM*, pages 453–457, 2005.

[47] S. Suri, T. Sandholm, and P. Warkhede. Compressing two-dimensional routing tables. *Algorithmica*, 35:287–300, 2003.

[48] L. Tan and T. Sherwood. A high throughput string matching architecture for intrusion detection and prevention. In *Proc. 32nd Annual Int. Symposium on Computer*

Architecture (ISCA), pages 112–122, 2005.

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

[50] L. Yang, R. Karim, V. Ganapathy, and R. Smith. Fast, memory-efficient regular expression matching with NFA-OBDDs. *Computer Networks*, 55(55):3376–3393, 2011.

[51] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz. Fast and memory-efficient regular expression matching for deep packet inspection. In *Proc. ACM/IEEE Symposium on Architecture for Networking and Communications Systems (ANCS)*, pages 93–102, 2006.

[52] F. Yu, R. H. Katz, and T. V. Lakshman. Gigabit rate packet pattern-matching using TCAM. In *Proc. 12th IEEE Int. Conf. on Network Protocols (ICNP)*, pages 174–183, 2004.