I dedicate this to everyone who motivated and helped me complete the thesis.
ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor Prof. Sartaj Sahni for his guidance and support throughout the course of my studies for PhD. I would also like to thank the Professors in my PhD committee, Prof. Shigang Chen, Prof. Sanjay Ranka, Prof. Ye Xia and Prof. Ravindra Ahuja.

I am grateful to the U.S. Air Force for funding my research and to Dr. Gunasekaran Sitharaman, in particular, for reviewing and providing feedback on my papers.

I would also like to express my gratitude to Ernest, Paul and Dan for supporting all my system requirements over this time and to graduate advisors John and Joan for clarifying the numerous queries.

Thanks to my friends - Shibdas, Xinyan, Junjie and Radhika for their support. I am eternally grateful to my parents, my husband, and our little son Ahan for the love and support which made this dissertation possible.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>4</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>9</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>11</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>17</td>
</tr>
<tr>
<td>CHAPTER</td>
<td>19</td>
</tr>
<tr>
<td>1 INTRODUCTION</td>
<td>19</td>
</tr>
<tr>
<td>1.1 Routers and Packet Classifiers</td>
<td>19</td>
</tr>
<tr>
<td>1.2 Packet Forwarding</td>
<td>20</td>
</tr>
<tr>
<td>1.3 Control and Data Planes in Routers</td>
<td>20</td>
</tr>
<tr>
<td>1.4 Power Consumption in Routers</td>
<td>22</td>
</tr>
<tr>
<td>1.5 Ternary Content Addressable Memories</td>
<td>23</td>
</tr>
<tr>
<td>1.6 Contributions</td>
<td>23</td>
</tr>
<tr>
<td>1.7 Thesis Overview</td>
<td>25</td>
</tr>
<tr>
<td>2 PETCAM: A POWER EFFICIENT TCAM FOR FORWARDING TABLES</td>
<td>26</td>
</tr>
<tr>
<td>2.1 Background and Related Work</td>
<td>26</td>
</tr>
<tr>
<td>2.2 Deficiencies in Prefix Preprocessing</td>
<td>29</td>
</tr>
<tr>
<td>2.2.1 Issues Related to Methods of Liu</td>
<td>29</td>
</tr>
<tr>
<td>2.2.2 Issues Related to EaseCAM</td>
<td>30</td>
</tr>
<tr>
<td>2.2.2.1 Prefix aggregation</td>
<td>30</td>
</tr>
<tr>
<td>2.2.2.2 Prefix expansion</td>
<td>31</td>
</tr>
<tr>
<td>2.3 PETCAM</td>
<td>31</td>
</tr>
<tr>
<td>2.3.1 Compaction of Prefix Set</td>
<td>32</td>
</tr>
<tr>
<td>2.3.2 Representation using Ternary Tries</td>
<td>36</td>
</tr>
<tr>
<td>2.3.3 Carving Trie to Create Suffix Nodes</td>
<td>39</td>
</tr>
<tr>
<td>2.3.3.1 Suffix-node structure</td>
<td>40</td>
</tr>
<tr>
<td>2.3.3.2 Our carving heuristic</td>
<td>44</td>
</tr>
<tr>
<td>2.3.4 Storing and Updating Prefixes in TCAM-SRAM</td>
<td>46</td>
</tr>
<tr>
<td>2.3.4.1 PETCAM structure</td>
<td>46</td>
</tr>
<tr>
<td>2.3.4.2 PETCAM updates</td>
<td>48</td>
</tr>
<tr>
<td>2.4 Experimental Results</td>
<td>48</td>
</tr>
<tr>
<td>2.4.1 Compaction Efficiency</td>
<td>49</td>
</tr>
<tr>
<td>2.4.2 Power Efficiency</td>
<td>50</td>
</tr>
<tr>
<td>2.4.3 Area Efficiency</td>
<td>55</td>
</tr>
<tr>
<td>2.4.4 PETCAMLite</td>
<td>57</td>
</tr>
<tr>
<td>2.4.5 Implementation of Nexthop Computation</td>
<td>58</td>
</tr>
<tr>
<td>2.4.6 Performance Analysis</td>
<td>60</td>
</tr>
</tbody>
</table>
3 DUO: DUAL TCAM ARCHITECTURE FOR FORWARDING TABLES WITH INCREMENTAL UPDATE ............................................. 62

3.1 Background and Related Work ........................................... 62
3.2 Simple Dual TCAM – DUOS .................................................. 65
  3.2.1 DUOS Incremental Update Algorithms .............................. 69
    3.2.1.1 Insert ................................................................. 69
    3.2.1.2 Delete ............................................................... 72
    3.2.1.3 Change ............................................................. 74
  3.2.2 ITCAM Algorithms ...................................................... 74
  3.2.3 LTCAM Algorithms ..................................................... 75
  3.2.4 ITCAM Memory Management ......................................... 75
    3.2.4.1 Memory management scheme 1 ................................ 76
    3.2.4.2 Memory management scheme 2 ............................... 80
    3.2.4.3 Memory management scheme 3 ............................... 83
    3.2.4.4 Scheme 4 ......................................................... 88
  3.3 Wide Dual TCAM–DUOW .................................................. 92
  3.4 Indexed DUOW–IDUOW ................................................... 99
    3.4.1 Memory Management for DLTCAM and ILTCAM .................. 99
    3.4.2 1-12Wc ................................................................. 106
    3.4.3 M-12Wb ............................................................... 108
  3.5 Experimental Results ..................................................... 112
    3.5.1 Evaluation of Memory Management Schemes ..................... 113
    3.5.2 Evaluation of DUOS ............................................... 118
    3.5.3 Evaluation of DUOW ............................................... 122
    3.5.4 Evaluation of IDUOW ............................................... 124
    3.5.5 Comparison with MIPS and CAO_OPT ............................. 126

4 CONSIST: CONSISTENT UPDATES FOR PACKET CLASSIFIERS ........ 137

4.1 Background and Related Work ........................................... 137
4.2 Consistent Updates .......................................................... 142
  4.2.1 Definitions and Properties .......................................... 142
  4.2.2 Batch Consistent Sequences ........................................ 151
    4.2.2.1 Graph representation of updates ............................. 151
    4.2.2.2 Constructing a near-optimal batch consistent sequence .... 152
  4.2.3 Incremental Consistent Sequences .................................. 158
  4.3 Experiments .............................................................. 160
    4.3.1 Benchmarks .......................................................... 160
    4.3.2 Results .............................................................. 163
      4.3.2.1 Forwarding tables ........................................... 163
      4.3.2.2 Packet classifiers .......................................... 164
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-1 An example 6-prefix forwarding table</td>
<td>20</td>
</tr>
<tr>
<td>2-1 Routing table with generalized prefixes</td>
<td>36</td>
</tr>
<tr>
<td>2-2 An example 6-prefix forwarding table</td>
<td>40</td>
</tr>
<tr>
<td>2-3 Number of routing table prefixes in PETCAM</td>
<td>50</td>
</tr>
<tr>
<td>2-4 TCAM power (in mW) for the different schemes</td>
<td>51</td>
</tr>
<tr>
<td>2-5 SRAM power (in mW) for the different schemes</td>
<td>53</td>
</tr>
<tr>
<td>2-6 TCAM area (in mm2) for the different schemes</td>
<td>54</td>
</tr>
<tr>
<td>2-7 SRAM area (in mm2) for the different schemes</td>
<td>55</td>
</tr>
<tr>
<td>2-8 TCAM size (in KB) for the different schemes</td>
<td>56</td>
</tr>
<tr>
<td>2-9 SRAM size (in KB) for the different schemes</td>
<td>57</td>
</tr>
<tr>
<td>2-10 Execution time in seconds for Steps 1 and 2 of Compact</td>
<td>58</td>
</tr>
<tr>
<td>2-11 Timing and power results for additional hardware</td>
<td>59</td>
</tr>
<tr>
<td>3-1 Datasets used in the experiments</td>
<td>112</td>
</tr>
<tr>
<td>3-2 Number of moves for the simple TCAM</td>
<td>115</td>
</tr>
<tr>
<td>3-3 Average number of moves for the simple TCAM</td>
<td>116</td>
</tr>
<tr>
<td>3-4 Maximum number of moves for the simple TCAM</td>
<td>117</td>
</tr>
<tr>
<td>3-5 Standard deviation in number of moves for the simple TCAM</td>
<td>118</td>
</tr>
<tr>
<td>3-6 Number of waitWrites for the simple TCAM</td>
<td>119</td>
</tr>
<tr>
<td>3-7 Number of Scheme 3 moves for inserts</td>
<td>120</td>
</tr>
<tr>
<td>3-8 Distribution of prefixes, inserts, and deletes for DUOS</td>
<td>121</td>
</tr>
<tr>
<td>3-9 Number of moves for inserts and deletes in the ITCAM of DUOS</td>
<td>122</td>
</tr>
<tr>
<td>3-10 Number of waitWrites in the ITCAM of DUOS</td>
<td>123</td>
</tr>
<tr>
<td>3-11 Number of LTCAM moves and waitWrites for DUOS</td>
<td>124</td>
</tr>
<tr>
<td>3-12 Number of prefixes to be stored in the LTCAM and associated wide SRAM</td>
<td>125</td>
</tr>
<tr>
<td>3-13 Number of waitWrites in the LTCAM of DUOW</td>
<td>126</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>1-1</td>
<td>Control and data planes in routers</td>
</tr>
<tr>
<td>1-2</td>
<td>Router architecture with batch updating policy</td>
</tr>
<tr>
<td>1-3</td>
<td>Router architecture with incremental updating policy</td>
</tr>
<tr>
<td>2-1</td>
<td>Trie for example prefix set $C = {000^<em>, 001^</em>, 010^<em>, 011^</em>, 00^<em>}$. Packet with address 000</em> is forwarded to next hop $H_1$.</td>
</tr>
<tr>
<td>2-2</td>
<td>Trie for prefix set in Figure 2-1, after logic minimization. Packet with address 000* is forwarded to next hop $H_2$.</td>
</tr>
<tr>
<td>2-3</td>
<td>Flow diagram for PETCAM construction</td>
</tr>
<tr>
<td>2-4</td>
<td>Algorithm for compaction of a prefix set</td>
</tr>
<tr>
<td>2-5</td>
<td>Ternary trie for the routing-table of Table 2-1</td>
</tr>
<tr>
<td>2-6</td>
<td>Ternary trie after merging of the $x$-child of root</td>
</tr>
<tr>
<td>2-7</td>
<td>Normalized ternary trie</td>
</tr>
<tr>
<td>2-8</td>
<td>Algorithm to normalize a ternary trie</td>
</tr>
<tr>
<td>2-9</td>
<td>Algorithm to merge an $x$-subtree</td>
</tr>
<tr>
<td>2-10</td>
<td>Suffix node of [21] with a 5-bit match start position field and representation of the first suffix</td>
</tr>
<tr>
<td>2-11</td>
<td>Six rules of Table 2-2 stored in TCAM + SRAM</td>
</tr>
<tr>
<td>2-12</td>
<td>Suffix node example</td>
</tr>
<tr>
<td>2-13</td>
<td>Type I suffix node</td>
</tr>
<tr>
<td>2-14</td>
<td>Type II suffix node</td>
</tr>
<tr>
<td>2-15</td>
<td>($numP$, $xNumP$, $size$) for nodes of trie in Figure 2-7.</td>
</tr>
<tr>
<td>2-16</td>
<td>Visit function for subtree carving heuristic[21]</td>
</tr>
<tr>
<td>2-17</td>
<td>PETCAM structure</td>
</tr>
<tr>
<td>2-18</td>
<td>Relative efficiency for table compaction</td>
</tr>
<tr>
<td>2-19</td>
<td>Comparison of total power (TCAM + SRAM) among different schemes</td>
</tr>
<tr>
<td>2-20</td>
<td>Pipeline processing suffix node with variable suffix lengths</td>
</tr>
</tbody>
</table>
3-1 Insertion of the root prefix into (a) requires the insertion of 4 new independent prefixes into the TCAM. Similarly, the deletion of the root prefix from (b) requires the withdrawal of these 4 prefixes from the TCAM.

3-2 Dual TCAM with simple SRAM

3-3 DUOS for an example 4-prefix forwarding table. Note that prefixes in ITCAM are stored in length order, whereas those in LTCAM are stored arbitrarily since the prefixes are disjoint.

3-4 Table of control-plane trie functions

3-5 Table of functions used for incremental update

3-6 Algorithm to insert into DUOS

3-7 Insert rule P5 - \{1^*, H5\} to the initial table in Figure 3-3. P5 is a leaf and hence is added to the LTCAM.

3-8 Insert rule P6 - \{011^*, H6\} to the prefixes in Figure 3-7. P6 is added to the LTCAM, while P3, which is no longer a leaf, is deleted from LTCAM and added to ITCAM.

3-9 Insert rule P7 - \{0^*, H7\} to the prefixes in Figure 3-8. P7 is added to the ITCAM since it involves an intermediate prefix.

3-10 Algorithm to delete from DUOS

3-11 Delete rule P7 - \{0^*, H7\} from the prefixes in Figure 3-9. P7 is deleted from ITCAM.

3-12 Delete rule P4 - \{000^*, H4\} from the prefixes in Figure 3-11. P4 is deleted from LTCAM. P2 is inserted to LTCAM and deleted from ITCAM as P2 is now a leaf.

3-13 Delete rule P5 - \{1^*, H5\} from the prefixes in Figure 3-12. P5 is deleted from LTCAM.

3-14 Algorithm to change a next hop in DUOS

3-15 ITCAM algorithms

3-16 LTCAM algorithms

3-17 Move from ITCAM[src] to ITCAM[dest]

3-18 Prefix arrangement in ITCAM for Scheme 1 for IPv4. The free space pool is indicated by hatched lines. Numbers 1, 2 by the curved arrow correspond to the first and second move, respectively.
5-1 Flow diagram for storing packet classifiers in TCAMs ............................................. 172
5-2 Example of a two-dimensional trie storing three rules: R1, R2 and R3 .................. 173
5-3 Selecting rules for insertion into LTCAM ................................................................. 175
5-4 Find overlapping rules by trie traversal ................................................................. 177
5-5 Table of control-plane trie functions ................................................................. 178
6-1 Flow diagram for storing packet classifiers in TCAMs ............................................. 187
6-2 Table of control-plane trie functions ..................................................................... 189
6-3 Setting $hpri$ on a new vertex in a priority graph .................................................. 190
6-4 Insert a rule in the ITCAM ..................................................................................... 191
6-5 Moving descendants downward in the ITCAM ....................................................... 192
6-6 Decision diagrams for priority adjustment of descendants vs. ancestors ............... 194
6-7 Initial ITCAM layout ............................................................................................... 196
6-8 Percentage of rules stored in the LTCAM and percentage improvement in lookup
time compared to STCAM architecture ....................................................................... 198
6-9 Number of TCAM writes with respect to PC-DUOS+ ............................................ 200
6-10 Run time normalized with respect to PC-DUOS+ .................................................. 200
6-11 Ratio of edges to vertices of graph ....................................................................... 201
6-12 Maximum chain length in graph before processing updates ............................... 202
6-13 A small graph representing test ipc3 ..................................................................... 202
6-14 Percentage of updates that require 1 write, $\leq 3$ and $\leq 10$ writes ................... 202
7-1 An example classifier ............................................................................................... 208
7-2 Classifier rules stored in a indexed TCAM ............................................................. 208
7-3 PC-TRIO Architecture ........................................................................................... 210
7-4 Selecting protocol nodes for leaves of leaves set ...................................................... 212
7-5 Data encoding in a wide SRAM word ...................................................................... 212
7-6 Nodes in a source trie is being carved. .................................................................... 214
7-7 Prefixes in forwarding table before and after applying updates ............................. 216
7-8 Differences among the architectures ............................................. 221
7-9 Comparison of compaction ratio, total power, lookup time and area .................................. 225
7-10 TCAM writes .................................................................................. 227
Ternary Content Addressable Memory (TCAM) is a hardware device which can support high-speed table lookups and is an attractive solution for applications such as packet forwarding and classification. A major drawback of TCAMs is that they are power-hungry. We investigate various TCAM architectures and propose PETCAM for TCAM power and memory reduction in packet forwarding and show that far better power and memory performance is possible when we use an optimal prefix set for the given forwarding table.

To keep up with the changes on the Internet, it is necessary to adopt efficient update algorithms for the TCAMs. Our dual TCAM architecture, DUO, and new memory management schemes for the TCAMs supports efficient control-plane incremental updates without delaying data-plane lookups. Compared to other TCAM architectures such as CAO_OPT that support incremental updates without delaying lookups, DUO offers reduction in power consumption.

We present a methodology CONSIST for constructing a consistent sequence of updates to be applied incrementally to packet classifiers when the updates arrive in a cluster, where consistency is with respect to the next hop/action returned from a packet forwarding table/classifier during lookup. The sequence of updates, built using our strategy, is free from redundancies in update operations and produces a near minimal increase in table size. We prove the existence of a consistent update sequence for any
given rule table and a cluster of updates. Our experiments validate our methodology and demonstrate a minimal increase in intermediate table size as a cluster of updates is applied.

For TCAMs storing packet classifier rules, we propose PC-DUOS, and its enhanced version PC-DUOS+, for distributing the classifier rules to two TCAMs and for incrementally updating the TCAMs. The update and lookup performance is compared against the prevalent scheme of storing classifier rules in a single TCAM in priority order. While PC-DUOS reduces the number of TCAM writes by up to 2.82 times, PC-DUOS+ reduces the number by up to 3.72 times, compared to a single TCAM architecture. Lookup speed in PC-DUOS and PC-DUOS+ improved by up to 48%.

Finally, we propose an indexed TCAM architecture, PC-TRIO, for packet classifiers. PC-TRIO uses wide SRAMs and index TCAMs and supports low power lookups and incremental updates.
1.1 Routers and Packet Classifiers

Internet routers are devices that connect several packet switched networks to allow communication and resource sharing among a large group of users. A router implements the packet forwarding and routing functions of network layer which is the third layer of the seven-layer OSI model of computer networking. A router has several input and output ports through which it receives and sends packets. A packet arriving at an input port of a router is transferred to an appropriate output port and from here to its next hop on the path to its destination. A router maintains a list of rules in a forwarding table that is used to determine the next hops for packets during packet forwarding. A router updates its forwarding table continuously in response to changes in the Internet. All the routers in a network communicate with each other using routing protocols to remain updated of the changes and to select the best paths to reachable destination addresses.

The data packets received by a router are often categorized into different “flows” using a table of rules in which each rule is of the form $(F, A)$, where $F$ is a filter and $A$ is an action. This is packet classification. When an incoming packet matches a rule in the classifier, its action determines how the packet is handled. For example, the packet could be forwarded to an appropriate output link, or it may be dropped. A $d$-dimensional filter $F$ is a $d$-tuple $(F[1], F[2], \ldots, F[d])$, where $F[i]$ is a range specified for an attribute in the packet header, such as destination address, source address, port number, protocol type, TCP flag, etc. A packet matches filter $F$, if its attribute values fall in the ranges of $F[1], \ldots, F[d]$. Since it is possible for a packet to match more than one of the filters in a classifier thereby resulting in a tie, each rule has an associated cost or priority. When a packet matches two or more filters, the action of the matching rule with the lowest cost (highest priority) is applied on the packet. It is assumed that filters
that match the same packet have different priorities. Packet classification is a key step in routers for various functions such as routing, creating firewalls, load balancing and differentiated services.

1.2 Packet Forwarding

Packet forwarding is a special case of packet classification in which a one-dimensional filter is used. In particular, the destination address on a packet is used to determine the next hop for the packet. A forwarding table often contains hundreds of thousands of rules. A packet forwarding rule \( (P,H) \) comprises a prefix \( P \) which acts as the filter and a next hop \( H \), which corresponds to the action of forwarding the packet to \( H \). Thus, a packet with destination address \( d \) is forwarded to \( H \) where \( H \) is the next hop associated with the rule that has the longest prefix matching \( d \). Table 1-1 shows a small forwarding table with 6 prefixes. The prefix associated with rule R5 is 100 (the * at the end indicates a sequence of don’t care bits) and the associated next hop is H5. Rule R5 matches all destination addresses that begin with 100. The length of the prefix 100 associated with R5 is 3. A destination address that begins with 100 is matched by rules R1, R3, and R5. Among these rules, R5 is the one with the longest prefix. So, H5 is the next hop for packets with a destination address that begins with 100.

Table 1-1. An example 6-prefix forwarding table

<table>
<thead>
<tr>
<th>Prefixes</th>
<th>Next Hop</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1 *</td>
<td>H1</td>
</tr>
<tr>
<td>R2 00*</td>
<td>H2</td>
</tr>
<tr>
<td>R3 10*</td>
<td>H3</td>
</tr>
<tr>
<td>R4 11*</td>
<td>H4</td>
</tr>
<tr>
<td>R5 100*</td>
<td>H5</td>
</tr>
<tr>
<td>R6 111*</td>
<td>H6</td>
</tr>
</tbody>
</table>

1.3 Control and Data Planes in Routers

Figure 1-1 shows a block diagram of the internals of a router. The part of a router that deals with packet forwarding is called the data plane and the part that communicates with other routers and selects the best routes to different destinations
and keeps the forwarding table updated is called the control plane. The control plane updates the forwarding table either in a batch or incrementally. When the updates are done in a batch, two copies of the forwarding table are maintained as shown in the Figure 1-2. While one copy is used for data plane lookups, the other copy is made up-to-date by the control plane. When the updating is complete, the data plane switches to the freshly updated copy for lookups, and the outdated copy is updated by the control plane. Thus batch updates require twice the memory for storing two copies of the forwarding table and introduce a latency between arrival of an update request and the
time the update is incorporated. In contrast, when the updates are applied incrementally, lookups and updates are done on the same table as shown in Figure 1-3.

![Router architecture with incremental updating policy.](image)

**Figure 1-3.** Router architecture with incremental updating policy.

### 1.4 Power Consumption in Routers

With the rapid growth of the Internet, the number of routers being used is increasing dramatically. While routers used at homes consume about 5-10W, edge routers consume about 4kW of power and the core routers about 10kW of power per rack [51]. It was measured that in Japan, ICT equipment in 2006 consumed about 45TWh or 4% of the electricity generated and 1% of the total energy consumption of the country [23]. Approximately, 25% of the ICT equipment energy is consumed by routers. The packet forwarding engine of a router needs a lot of energy to perform high-speed lookups and packet switching. For example, to keep up to a line rate of 40 Gbps, a packet forwarding engine must perform 125 million searches per second, assuming the minimum IPv4 packet size of 40 bytes. It was found that about 62% of the router power consumption happens in the packet forwarding engine. With the growing trend of network usage, the total energy consumption in routers is only going to get worse. For example, with the current growth rate of Internet traffic at 40% per year in Japan, it is projected that by the year 2022, the energy consumption will exceed 10,000 TWh which was the total energy produced in Japan in 2005 [40]. Thus, unless we can make routers far more energy efficient, routers may soon consume most of the produced energy. This makes the design of low power packet forwarding engines essential.
1.5 Ternary Content Addressable Memories

Our research is focussed on reduction of energy consumption in a packet forwarding engine that uses a particular type of memory called TCAM which is the acronym for Ternary Content Addressable Memory. A TCAM is different from a conventional memory in that, each bit may be set to one of the three states namely, 0, 1, and \( x \) (don’t care). This makes it particularly convenient to store the destination prefixes along with their trailing sequence of ‘\( x \)’s. A TCAM has an associated SRAM which is used to store the next hops. A TCAM often has a priority encoder to choose the best match among multiple matches. TCAMs are attractive for use in edge and core routers, because like an associative memory, TCAMs enable a parallel search across all the rules and complete a table lookup in one clock cycle. Even though TCAMs support high-speed lookups, they are power hungry. So, a lot of research has been done to reduce the power consumption both in the hardware and software domains.

1.6 Contributions

This thesis presents low power design strategies for packet forwarding and classification in TCAM based routers. Specifically, there are four main contributions, as described below.

1. We propose a methodology for packet forwarding tables that compacts the table and then stores the prefixes in TCAM, leading to significant reduction in TCAM power and space consumption. We propose the use of a minimum set of rules equivalent to those in the given forwarding table coupled with the wide SRAM strategy of \cite{21}. We introduce the notion of a ternary trie to represent the logically minimized prefixes and also present algorithms to normalize and carve the trie to fill up a two level TCAM architecture using wide SRAM words. Since in our scheme we get suffixes with ‘\( x \)’ bits in them we propose a new format for storing these ternary suffixes in the SRAM. We perform batch updates to the set of rules to accommodate the incoming
route advertisements. The low power lookup schemes are generally not suitable for incremental updates.

2. We develop a new dual TCAM architecture, generally referred to as DUO, for packet forwarding tables along with advanced memory management schemes for the TCAMs. The advantage of DUO is that it supports incremental updates while still supporting low power TCAM lookup. The memory management schemes result in fewer moves compared to existing TCAM memory management schemes.

3. We formalize the consistency properties of incrementally updated packet classifiers when updates arrive in clusters. The updates arriving in a cluster are arranged in a consistent sequence that leads to proper packet forwarding and classification, with the same results as if the updates were applied all at a time in a batch. We define and analyze requirements and properties for two types of consistency, namely, batch consistency and incremental consistency. We represent the cluster of updates using a precedence graph and obtain batch consistent sequences as topological orderings of vertices of the precedence graph. We show that it is not possible to construct a precedence graph representation for all incremental consistent sequences. We also present a heuristic to obtain a near optimal batch consistent sequence as a topological ordering of the precedence graph. Here, optimality is defined with respect to the increase in size of the intermediate rule table, where an optimal sequence guarantees minimum increase in the maximum table size.

4. We develop a dual TCAM architecture and algorithms for storing packet classifiers. The advantage of using two TCAMs instead of one is support for efficient incremental updates and low power TCAM-lookups.

5. We develop an indexed, triple TCAM architecture augmented with wide SRAMs for packet classifiers that lead to low power lookups, in addition to, efficient incremental updates. The usage of wide SRAMs alleviates the range expansion problem for classifiers to a large extent.
1.7 Thesis Overview

The remainder of this thesis is organized as follows.

Chapter 2 describes how various existing techniques for power reduction in TCAMs such as reduction in forwarding table size, using an index TCAM and wide SRAMs, may be combined effectively to result in PETCAM, with significant reduction in TCAM power during prefix lookup in packet forwarding. Proofs for equivalence of original prefix set and reduced prefix set are given.

Chapter 3 presents DUO, which is a dual TCAM architecture for storing incrementally updatable packet forwarding tables. Detailed memory management algorithms for incremental updates are presented in this chapter. The update performance and power consumption are compared with existing techniques.

Chapter 4 presents a scheme, CONSIST, for applying updates consistently on packet classifiers and forwarding tables. This results in fewer packets being sub-optimally classified and fewer packets being forwarded to sub-optimal next hops, as lookups and incremental updates happen simultaneously.

Chapter 5 presents PC-DUOS, a TCAM architectures for packet classifiers and the associated update algorithms. A notable feature of PC-DUOS is its efficient incremental update algorithms.

Chapter 6 presents an enhanced version of PC-DUOS, PC-DUOS+, which employs more efficient incremental update algorithms. A detailed comparison between PC-DUOS and PC-DUOS+ is made in this chapter.

Chapter 7 presents PC-TRIO, which is the triple TCAM architecture augmented with wide SRAMs, for packet classifiers.

Finally, we conclude in Chapter 8.
CHAPTER 2
PETCAM: A POWER EFFICIENT TCAM FOR FORWARDING TABLES

The main challenge of using TCAMs in routers is the power consumption in these memory chips during a lookup. This chapter presents a methodology to process and store forwarding tables in a router in a manner that leads to a very low power consumption per lookup.

The chapter is organized as follows. In Section 2.1, we will look at the existing techniques used for cooler TCAMs and present definitions for terms used here. Later, in Section 2.2 we will present deficiencies in existing schemes and in Section 2.3 we show how our PETCAM methodology solves these issues and helps to bring down the power consumption in TCAMs, and finally conclude in Section ??.

2.1 Background and Related Work

A lot of research has been done to improve the power efficiency of TCAM-based routing tables [2, 20, 21, 32, 35, 54, 55]. Shah et al. [41] propose two algorithms for incremental updates to the TCAM table while keeping the number of prefix moves small in the worst case. Wang et al. [54] present an algorithm for consistent and incremental updates to TCAMs. Draves et al. [14] and Suri et al. [49] have proposed use of dynamic programming to remove redundancies in a forwarding table by computing equivalent forwarding table with minimal number of prefixes. We describe the results reported in [20, 21, 35, 55] in this section as these are most relevant to the work we report in this chapter. We define four terms that will be used for our analysis.

Definition 1. \( P_1 \subset P_2 \) iff \( \text{addr}(P_1) \subset \text{addr}(P_2) \), where \( \text{addr}(P) \) is the set of addresses matched by prefix \( P \). Note that \( P_1 \subset P_2 \) iff \( P_2 \) is a proper prefix of \( P_1 \).

Definition 2. A rule \((P_1, H_1)\) is Type I redundant iff (a) there exists a rule \((P_2, H_2)\) such that \( P_1 \subset P_2 \) and \( H_1 = H_2 \) and (b) there is no rule \((P_3, H_3)\) such that \( P_1 \subset P_3 \subset P_2 \) and \( H_3 \neq H_1 \).
Definition 3. A generalized prefix is a sequence comprised of the symbols 0, 1, and ‘x’ (don't care) and possibly terminated by the symbol * (1 or more don't cares). A simple prefix (or simply, prefix) is a generalized prefix that has no occurrence of the symbol ‘x’. (Alternatively, we may limit the occurrence of the symbol ‘x’ to the right end of the sequence. Note that ‘x’s at the right end of a sequence may be replaced by a * so that the sequence 10xxx may be regarded as a simple prefix by rewriting it as 10*.)

For example, “0xx1xx0*” and “xx100x11*” are generalized prefixes. In routing table applications, a generalized prefix may be stored in a word of TCAM by replacing * with a suitable number of ‘x’s.

Definition 4. Two sets of generalized prefixes are equivalent iff they match the same addresses.

Liu [20] proposes two schemes – pruning and mask extension – to compact the rules of a routing table. In pruning, rules with type I redundant prefixes are eliminated from the rule table. It is easy to see that the elimination of type I redundant prefixes does not change the next-hop decision for any destination address. Following the pruning process, each set, $S$, of prefixes that have the same length and the same next hop, is subjected to mask extension in which $S$ is replaced by a set of generalized prefixes $T$ such that $|T| \leq |S|$. This compaction process may lead to inaccurate behavior as described in Section 2.2.1.

Ravikumar et al. [35] extend the work of [20] and propose the 2-level EaseCAM architecture for routing tables. For an IPv4 routing table, the first level stores 8-bit sub-prefixes. Prefixes that have the same first 8 bits define a prefix cluster. Pruning, prefix aggregation, and prefix expansion are used to replace the simple prefixes in each cluster with a smaller set of generalized prefixes with the property that a search of the TCAM segment that contains this smaller set of generalized prefixes results in the same next hop as does a search in the TCAM segment for the original cluster of simple prefixes. Since the generalized prefixes in a cluster have the same first 8 bits,
it is necessary to store only the remaining 24 bits of each generalized prefix in the second-level TCAM. Consequently, second-level TCAM words are 25% smaller than the TCAM words in the design of [20]. Prefixes shorter than 8 bits are stored in a separate bucket. Section 2.2.2 describes how this method can lead to incorrect packet forwarding.

Zane et al. [55] propose two schemes to achieve power reduction. In the first, a few bits of each prefix are used to partition the prefix set so that each partition agrees on these selected bits. The bits are called the partition selector bits. Prefixes in the same partition are stored together in decreasing order of length. To search for the longest matching prefix for a given destination address d, the partition selector bits are extracted from d and used to determine which partition is to be searched. Although all prefixes of an uncompacted routing table are stored in the TCAM, power reduction results from having to search only one partition\(^1\). The second strategy proposed by Zane et al. [55] is a 2-level TCAM architecture in which the first level TCAM is an index to the partitions in the second level TCAM. The partitions and index are constructed by decomposing the binary trie representation of the routing-table prefixes.

The most recent work on TCAM power reduction in the context of routing tables appears to be that of Lu and Sahni [21]. They augment the traditional 1-level TCAM lookup structure as well as the 2-level TCAM structure of Zane et al. [55] with wide SRAMs and store the suffixes of several prefixes in a single wide SRAM word. This enables a reduction in both power consumption and total TCAM memory requirement.

Our proposed approach can significantly improve both area and power requirements compared to existing techniques.

---

1 The power required by a TCAM lookup is proportional to the total number of bits in the TCAM partition that is searched.
2.2 Deficiencies in Prefix Preprocessing

This section illustrates the deficiencies in two of the closely related approaches that may lead to incorrect packet forwarding.

2.2.1 Issues Related to Methods of Liu

As noted in Section 2.1, when logic minimization is applied to a set of same-hop same-length prefixes, we get a set of generalized prefixes. The placement of the generalized prefixes in the TCAM is not directly addressed in [20], and an incorrect placement will result in return of incorrect next-hop. For example, \( A = \{000*, 001*, 010*, 011*\} \) optimizes to \( B = \{0*\} \). While it may be natural to assign \( 0* \) a length of 1, such a length assignment can result in an incorrect next hop computation. To see this, suppose that the next hop associated with the prefixes of \( A \) is \( H1 \) and that the routing table has another prefix \( 00* \) whose next hop is \( H2 \) and \( H1 \neq H2 \). When using the original prefix set \( C = \{000*, 001*, 010*, 011*, 00*\} \) as represented by the trie in Figure 2-1, packets with destination address beginning with 000 are sent to \( H1 \). Consider what happens when we apply the compaction scheme of Liu [20]. Since \( C \) has no type I redundancy, pruning does not weed out any member of \( C \). Mask extension compacts \( A \) to \( B \). So, the compacted prefix set is \( D = \{0*, 00*\} \) with \( 0* \) having \( H1 \) as its next hop and \( 00* \) having \( H2 \) as shown in Figure 2-2. Using the prefix set \( D \) and entering prefixes in TCAM in order of length, packets with destination addresses that begin with 000 are sent to \( H2 \).

![Figure 2-1. Trie for example prefix set \( C = \{000*, 001*, 010*, 011*, 00*\} \). Packet with address 000* is forwarded to next hop \( H1 \).](image)
While this issue can be fixed by placing the generalized prefixes in the TCAM in order of length of the corresponding input prefixes to mask extension (so 0* is assigned length 3 and placed in the TCAM before 00*), we discuss in Section 2.3.1 a strategy such that mask extension will output a generalized prefix set that is equivalent to the input prefix set with an intuitive definition of length, and hence the generalized prefix set will result in correct packet forwarding when the prefixes are arranged in length order.

2.2.2 Issues Related to EaseCAM

EaseCAM[35] has two important steps: prefix aggregation and prefix expansion. The remainder of this section describes how each of these steps can cause functional inaccuracies.

2.2.2.1 Prefix aggregation

In prefix aggregation, prefixes that have the same hop are aggregated into clusters with each cluster containing prefixes that have the same common sub-prefix. The common sub-prefix length is constrained to be a multiple of 8. So, for example if two prefixes that have the same next hop agree on their first 18 bits only, then they will be in a cluster of same-hop prefixes that agree on their first 16 bits. Logic minimization is then applied to each cluster. Since the prefixes in a cluster have different length, there appears to be no reasonable way to determine where to place the generalized prefixes that result from logic minimization into the TCAM so as to correctly route packets.
Neither of the length resolution methods proposed for mask extension in Section 2.2.1 work when aggregation is employed. For example, consider the rule set \{(1*,A), (10*,B), (101*,A)\}, where the first 8 bits of each prefix are omitted and are the same. There is no redundant rule, so no rule is eliminated in the initial pruning step. In the aggregation step, 1* and 101* form a cluster and 10* is in a different cluster as it has a different next hop. Logic minimization reduces the first cluster to 1* and has no effect on the second cluster. The new rule set is \{(1*,A), (10*,B)\}. 1* was derived from a prefix of length 1 and one of length 3. Neither length assignment 1 or 3 for 1* allows the new rule set to work like the original rule set. For example, with the natural length assignment of 1 to 1*, packets destined to 101* addresses get routed to B rather than to A; with a length assignment of 3, packets to 100* get sent to A rather than to B!

**2.2.2.2 Prefix expansion**

EaseCAM\[^{35}\] proposes using prefix expansion within an aggregated cluster to improve compaction and runtime performance of logic minimization. In prefix expansion, short prefixes in a cluster are replaced by a set of prefixes whose length equals that of the longest prefix in the cluster. So, following prefix expansion, all prefixes in a cluster have the same length. Since logic minimization is faster when the input prefixes are of the same size, runtime efficiency is achieved \[^{35}\]. In the example cluster \{1*,101*\} of Section 2.2.2.1 of prefixes with same next hop A, prefix expansion yields the cluster \{100*, 101*, 110*, 111*\}, which is reduced to 1* by logic minimization. The new rule set is \{(1*,A),(10*,B)\}, which, as noted in Section 2.2.2.1 cannot be made to work the same as the original rule set.

**2.3 PETCAM**

Figure 2-3 shows the overall flow of our PETCAM construction algorithm. The first phase is the prefix compaction phase which results in a set of generalized prefixes and is explained in Section 2.3.1. In the second phase, the set of generalized prefixes is stored in router memory in the form of a ternary trie. This representation is explained
in Section 2.3.2. The third phase is the application of a carving heuristic to carve out 1-bit subtrees from the trie. The information in a 1-bit subtree is stored in a particular structure called the suffix node. The structure of a suffix node and our carving heuristic are presented in Section 2.3.3. Finally, the fourth phase deals with storing and updating the prefixes and suffix nodes in a TCAM and SRAM system which is described in Section 2.3.4.

Figure 2-3. Flow diagram for PETCAM construction

2.3.1 Compaction of Prefix Set

Our goal here is to compact the prefix set. Towards that end, we first apply a dynamic programming algorithm [14] which compacts the prefix set to a optimal prefix set removing all redundancies. To obtain further reduction in size, we apply logic minimization using mask extension [20] on the optimal prefix set. A combination of these two techniques reduces the prefix table size significantly, as is also demonstrated in our experiments. Algorithm Compact in Figure 2-4 shows the two important steps in compaction of prefix set. We will prove that the set of generalized prefixes obtained using Compact is equivalent to the input prefix set in the original routing table in this subsection.
Algorithm: Compact
Input: Original Prefix Set
Output: Compacted Prefix Set
Step 1: Transform the given routing table to an equivalent optimal routing table using the dynamic programming algorithm of [14].
Step 2: Use mask extension [20] to reduce the number of prefixes in the optimal routing table obtained in Step 1.
Return Compacted Prefix Set

Figure 2-4. Algorithm for compaction of a prefix set

Definition 5. A rule \((P_1, H_1)\) is Type II redundant iff the routing table contains a set of rules \(\{(P_2, H_2), \cdots, (P_k, H_k)\}\) such that \(|P_1| < |P_i|, 2 \leq i \leq k\) and every address matched by \(P_1\) is also matched by at least one \(P_i, 2 \leq i \leq k\).

In Figure 2-1 the rule \((00^*, H2)\) is type II redundant as it is completely covered by rules \((000^*, H1), (001^*, H1)\), that is, every address matched by \((00^*, H2)\) is also matched by either \((000^*, H1)\) or \((001^*, H1)\). Similarly, in the rule set \(\{(10^*, H1), (100^*, H2), (101^*, H3)\}\), no prefix is type I redundant. However, \((10^*, H1)\) is type II redundant. Neither Liu [20] nor Ravikumar et al. [35] remove type II redundant rules.

Since the dynamic programming algorithm of [14] transforms a set of prefix rules into a provably optimal equivalent set of prefix rules, the transformed set used in PETCAM is guaranteed to be free of type I and type II redundancies.

We note that every generalized prefix may be written as the sum of simple prefixes that have the same length as the generalized prefix and such that the addresses matched by the generalized prefix are the union of those matched by the simple prefixes. So, for example, \(1x00x1^* = "100001^* + "100011^* + "110001^* + "110011^*"\).

This decomposition of a generalized prefix into the sum of simple prefixes that have the same length as the generalized prefix is referred to as generalized prefix decomposition (GPD) and \(GPD(X)\) is the generalized prefix decomposition of the generalized prefix \(X\).

Definition 6. Let \(R = \{R_1, R_2, \cdots, R_r\}\) be a set of generalized prefixes that is equivalent to the set of simple equal-length same-hop prefixes \(S = \{S_1, \cdots, S_s\}\). \(R\) is a canonical equivalent set iff each \(R_i\) is the sum of some of the \(S_s\)s.
For example, \( R = \{0^*\} \) is a canonical equivalent set for \( S = \{01^*, 00^*\} \). However, even though \( R = \{0001^*, 00x0^*, 0011^*\} \) and \( S = \{00^*, 001^*\} \) are equivalent, \( R \) is not a canonical equivalent set for \( S \). For example, \( 0001^* \) is not the sum of any subset of the \( S_q \)s.

**Theorem 2.1.** Let \( R \) and \( S \) be as in Definition 6. There exists a canonical equivalent set for \( S \) that has the same number of generalized prefixes as does \( R \).

**Proof.** Consider an \( R_i \) in \( R \). Let \( R_i = R_{i1} + R_{i2} + \cdots + R_{iq(i)} \) be the GPD of \( R_i \). Since \( R \) and \( S \) are equivalent and prefixes of the same length are disjoint (i.e., have no common matching address), there is exactly one \( f(i, j) \), \( 1 \leq f(i, j) \leq s \), such that \( R_{ij} \) and \( S_{f(i,j)} \) are not disjoint, \( 1 \leq i \leq r, 1 \leq j \leq q(i) \). We consider 3 cases.

- **Case 1:** If \( |R_i| = |S_1| \), \( R_{ij} = S_{f(i,j)} \) for all \( j \) and so \( R_i \) is the sum of some of the \( S_q \)s.

- **Case 2:** If \( |R_i| > |S_1| \), let \( R_i^* \) be the first \( |S_1| \) bits of \( R_i \). So, the addresses matched by \( R_i \) are a subset of those matched by \( R_i^* = R_{i1}^* + R_{i2}^* + \cdots + R_{iq(i)}^* = S_{f(i,1)} + S_{f(i,2)} + \cdots + S_{f(i,q(i))} \), where \( R_{ij}^* \) is obtained from \( R_{ij} \) by truncating the last \( |R_i| - |S_1| \) bits. Since \( R_i^* \) matches no address not matched by \( S \), replacing \( R_i \) by \( R_i^* \) in \( R \) preserves the equivalence between \( R \) and \( S \) and doesn’t increase the number of \( R_i \)s in \( R \). We may use this replacement transformation as often as need to replace all \( R_i \)s in \( R \) whose length is more than \( |S_1| \) increase the number of \( R_i \)s in \( R \). We may use this replacement transformation as often as need to replace all \( R_i \)s in \( R \) whose length is more than \( |S_1| \) with \( R_i^* \)s whose length equals \( |S_1| \). From Case 1, it follows that each of the replacing \( R_i^* \)s is the sum of some of the \( S_q \)s.

- **Case 3:** When \( |R_i| < |S_1| \), we may use prefix expansion to represent each \( R_{ij} \) as the sum of \( 2^t \), \( t = |S_1| - |R_i| \) simple prefixes whose length is \( |S_1| \). From the equivalence of \( R \) and \( S \) and the fact that prefixes of the same length are disjoint, it follows that each expanded prefix is one of the \( S_q \)s. So, each \( R_{ij} \) and hence \( R_i \) is the sum of some of the \( S_q \)s.

\( \square \)

The prefixes of a canonical equivalent set are called **canonical prefixes** and \( CD(R_{ij}) \) is the set of prefixes of \( S \) that sum to \( R_{ij} \). From Theorem 2.1, it follows that for every set of equivalent generalized prefixes computed by a minimization algorithm, there is a
canonical equivalent set with the same number of generalized prefixes. So, henceforth, we assume that minimization algorithms return canonical prefixes.

**Theorem 2.2.** Let $U$ be a set of rules comprised of simple prefixes that is free of type II redundancies. Let $V$ be the set of rules comprised of (canonical) generalized prefixes obtained from $U$ by applying logic minimization to the equal-length same-hop prefixes of $U$ as is done in mask extension [20]. Longest prefix matching in $U$ and $V$ results in the same next hop for every destination address $A$.

**Proof.** Suppose there is an address $A$ for which the longest matching simple prefix in $U$ is $U_1$ with next hop $H_1$ and for which the longest matching generalized prefix in $V$ is $V_2$ with next hop $H_2$ and $H_1 \neq H_2$. Let $V_{21}$ be the prefix of $GPD(V_2)$ that matches $A$. Note that since all prefixes in $GPD(V_2)$ have the same length, they are disjoint and so exactly one of these matches $A$. Further, let $U_2$ be the prefix of $CD(V_{21})$ that matches $A$. Again, exactly one prefix of $CD(V_{21})$ matches $A$. Since $U_1$ is the longest prefix of $U$ that matches $A$, $|U_1| > |U_2|$. Let $V_1$ be the generalized prefix of $V$ such that $V_{11} \in GPD(V_1)$ matches $A$ and $U_1 \in CD(V_{11})$. Such a $V_1$ must exist in $V$ because of the way $V$ is constructed from $U$ using logic minimization. Since $V_2$ is the longest matching generalized prefix for $A$ in $V$ and $V_1$ also matches $A$, $|V_{21}| = |V_2| \geq |V_1| = |V_{11}|$. Now, since two prefixes are either disjoint or nest and since $U_1$, $U_2$, $V_{11}$, and $V_{21}$ match $A$,

$$addr(U_1) \subset addr(U_2) \subseteq addr(V_{21}) \subseteq addr(V_{11})$$

From this and the observation that all prefixes in $CD(V_{11})$ are of the same length and hence are disjoint, it follows that some subset of $CD(V_{11})$ that includes $U_1$ sums to $U_2$. Hence, $U_2$ is type II redundant. $\square$

From Theorem 2.2, it follows that if we start with a set of original prefixes that contains no type II redundancy, apply logic minimization to obtain generalized prefixes, then the set of generalized prefixes is equivalent to the set of original prefixes. Thus in the context of PETCAM, the set of generalized prefixes is equivalent to the set
of prefixes in the optimal routing table. The prefixes in the optimal routing table, on the other hand, are equivalent to the original set of prefixes \([14]\). Hence, the set of generalized prefixes in PETCAM is equivalent to the set of prefixes in the original routing table.

Thus, when these generalized prefixes are entered into a TCAM in decreasing order of length, then lookups yield the same next hops as when we load the TCAM with the original prefix set in length order, where the length of a generalized prefix is intuitively defined as the index of the rightmost symbol that is not a ‘x’ or a *. So, the length of “1xx01*” is 5 and the length of “xx00xx1*” is 7. This is consistent with the accepted definition of the length of a simple prefix where, for example, the length of “001*” is 3. We use the notation \(|G|\) to denote the length of a generalized prefix.

In our example prefix set \(C\) in Section 2.2.1 and Figure 2-1, the prefix 00* is Type II redundant and hence is removed. Then logic minimization is applied to the prefix set and finally a prefix set \(B = \{0*\}\) is obtained that is equivalent to \(C\).

2.3.2 Representation using Ternary Tries

To map the generalized prefixes that result from steps 1 and 2 of our PETCAM construction algorithm we first construct a ternary trie\(^2\). Table 2-1 shows an example routing table following steps 1 and 2 of our PETCAM construction algorithm. Figure 2-5 shows the corresponding ternary trie.

Table 2-1. Routing table with generalized prefixes

<table>
<thead>
<tr>
<th>Address</th>
<th>Next Hop</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x0</td>
</tr>
<tr>
<td>2</td>
<td>00x0</td>
</tr>
<tr>
<td>3</td>
<td>00x1</td>
</tr>
<tr>
<td>4</td>
<td>1100</td>
</tr>
<tr>
<td>5</td>
<td>11x1</td>
</tr>
</tbody>
</table>

\(^2\) A ternary trie differs from a binary trie in that each node of a ternary trie may have up to 3 children depending on whether the branching bit is a 0, 1, or ‘x’.
A normalized ternary trie is a ternary trie in which each node that is the $x$-child (i.e., the don’t care child) of its parent has no sibling. So, in a normalized ternary trie, the children of degree 2 nodes are 0- and 1-children, the child of a degree 1 node may be a 0-, 1-, or $x$-child, and there are no degree 3 nodes. A ternary trie may be normalized by eliminating the $x$-child of each degree 3 node by merging the subtree rooted at this $x$-child with the subtrees rooted at the two siblings of this $x$-child. For example, in the ternary trie of Figure 2-5, the root is a degree 3 node and the subtree rooted at its $x$-child may be merged with the subtree rooted at the root’s 0-child as well as with that rooted at the root’s 1-child to obtain the ternary tree of Figure 2-6. One may verify that the ternary tries of Figures 2-5 and 2-6 are equivalent in that both route all packets to the same next hop.

The trie of Figure 2-6 is not yet a normalized ternary trie as it contains an $x$-child that has a sibling (i.e., the $x$-child with $Q(R) = 11x$). This subtree rooted at this $x$-child may be merged with that rooted at its 0-sibling and its empty 1-sibling to obtain the normalized ternary trie of Figure 2-7.

Figure 2-8 shows our algorithm to normalize a ternary trie. This algorithm assumes that each node $y$ of a ternary trie has 3 children fields with $y.child[0]$ and $y.child[1]$ pointing to the 0- and 1-children of node $y$ and $y.child[2]$ pointing to the $x$-child of node.
Figure 2-6. Ternary trie after merging of the $x$-child of root

Figure 2-7. Normalized ternary trie

The algorithm employs two other algorithms—delete, which deletes a subtree given its root and merge, which merges two subtrees together. We do not further specify delete as this is a simple postorder traversal. Algorithm merge is specified in Figure 2-9.

In algorithm merge, $oChild$ and $xChild$ are children of parent, where, $xChild$ is the $x$-child while $oChild$ is the $oChildID$-child. Notice that when we start with a prefix set that has no type I and II redundancies and perform mask extension, at most one of $oChild$ and $xChild$ may have a non-null next hop. Further, note that an optimal prefix set is devoid of type I and type II redundancies.
Algorithm normalize(root)
{
    if (!root) return;
    if (root.child[2]) {
        if (root.child[0] || root.child[1]) {
            merge (root, root.child[0], 0, root.child[2]);
            merge (root, root.child[1], 1, root.child[2]);
            delete(root.child[2]);
        }
        normalize(root.child[0]);
        normalize(root.child[1]);
        normalize(root.child[2]);
    }
    normalize(root.child[0]);
    normalize(root.child[1]);
    normalize(root.child[2]);
}

Figure 2-8. Algorithm to normalize a ternary trie

Algorithm merge(parent, oChild, oChildID, xChild)
{
    if (!xChild) return;
    if (!oChild) {
        oChild = new node;
        oChild.nextHop = xChild.nextHop;
        parent.child[oChildID] = oChild;
    }
    else {
        if (xChild.nextHop) then
            oChild.nextHop = xChild.nextHop;
        }
        merge (oChild, oChild.child[0], 0, xChild.child[0]);
        merge (oChild, oChild.child[1], 1, xChild.child[1]);
        merge (oChild, oChild.child[2], 2, xChild.child[2]);
    }

Figure 2-9. Algorithm to merge an x-subtree

### 2.3.3 Carving Trie to Create Suffix Nodes

Our carving heuristic starts with the normalized ternary trie for the canonical prefixes that result when mask extension is done on an optimal prefix set. The carving of a node in the normalized ternary trie, generates a prefix (represented by bits in path from the root to the carved node) and a 1-bit subtree of suffixes. The 1-bit subtree of suffixes is encoded in a format, also called the suffix-node structure which is explained below.
Table 2-2. An example 6-prefix forwarding table

<table>
<thead>
<tr>
<th>Prefixes</th>
<th>Next Hop</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>*</td>
</tr>
<tr>
<td>R2</td>
<td>0*</td>
</tr>
<tr>
<td>R3</td>
<td>1*</td>
</tr>
<tr>
<td>R4</td>
<td>01*</td>
</tr>
<tr>
<td>R5</td>
<td>010*</td>
</tr>
<tr>
<td>R6</td>
<td>111*</td>
</tr>
</tbody>
</table>

2.3.3.1 Suffix-node structure

Our suffix-node structure is motivated by the structure in [21] and we develop necessary adaptations to accommodate generalized prefixes and parallel suffix matching. For this, we need to modify the structure of a suffix node as well as develop an algorithm to map suffixes into suffix nodes. Before developing these adaptations, we provide a brief overview of the the motivation behind creating suffix-nodes and the suffix-node structure itself.

The use of wide SRAMs in conjunction with TCAMs reduces power consumption and increases effective TCAM capacity. Whereas a next hop can often be encoded using 10 to 12 bits we can fetch, in a single memory fetch cycle, 72 bits from a QDRII SRAM in dual burst mode and 144 bits in quad burst mode. Further, given the orders of magnitude discrepancy between the time for an SRAM fetch cycle and the time to perform an arithmetic, it is possible to do significant processing of the data stored in a word of a wide SRAM in much less time than it takes to fetch that word of data from the SRAM. To capitalize on these observations, the suffixes of several prefixes that are in the same subtree of the binary trie for the forwarding table prefixes are packed into a suffix node, which is then stored in one or more SRAM words in such a way that the entire suffix node may be retrieved in a single memory cycle. Figure 2-10 shows the
Figure 2-11. Six rules of Table 2-2 stored in TCAM + SRAM

Figure 2-12. Suffix node example

structure of the suffix node of [21]. We have added a 5-bit match start position field which indicate the bit position in the destination prefix from where suffix matching can start for all suffixes encoded in the suffix node. The suffix count field gives the number of suffixes packed in the suffix node. For each suffix Si stored in a suffix node, we keep the suffix length, len(Si), the suffix, Si, and the next hop associated with the suffix. Using the suffix node creation scheme, each suffix node must have exactly one suffix of length 0. This suffix can come from either a prefix that is stored in the root of the subtree that is carved to form the suffix node, or a covering prefix which is inherited from the nearest

Figure 2-13. Type I suffix node
Figure 2-14. Type II suffix node

ancestor with a prefix in case the root of the subtree does not store a prefix. To optimize SRAM usage further, we store this suffix as the first suffix in the node, and since it has a length 0, we drop the suffix length field for the first suffix. Thus a suffix of length 0 appears as the first suffix in a suffix node, and is represented only by its next hop.

Figure 2-12 shows the binary trie for the prefixes of Table 2-2 together with a mapping of these prefixes into a simple TCAM with wide SRAM. For this example, we assume an SRAM word width of 32 bits with 2 bits allocated for the match start position field (allowing prefixes to be of length 5 bits), 2 bits allocated for the count field of a suffix node (permitting up to 4 suffixes to be stored in a node), 2 bits for the suffix length field (permitting suffixes of length between 0 and 3), and 12 bits for the next-hop field (permitting up to 4096 different next hops). In the worst-case, a suffix node stores a single suffix of length 0, for which a next hop field of 12 bits is used along with the match start position and suffix count fields, utilizing only 16 of the 32 bits. In the best case, we may store a suffix of length 0 and one of length 2 resulting in the utilization of all 32 bits in the node. The allocation of suffixes to suffix nodes is done by carving out subtries of the binary trie for the prefix set. For example, from the binary trie of Figure 2-12, we first carve out the binary trie rooted at node $A$. The path from the root to $A$ is $Q(A) = 01$. $Q(A)$ is stored in the TCAM and the suffixes * (length 0) and 0* (length 1) that result from eliminating $Q(A)$ from the front of each prefix in the carved subtrie are packed into a suffix node. This carving-packing process is repeated at nodes $B$, $C$, and $D$ resulting in the suffix nodes of Figure 2-12. When carving is done at a node, say $R$, whose subtree doesn’t contain a matching prefix for every destination address that begins with $Q(R)$ (that is, there is no next hop stored at the node $R$), we add a covering prefix into the suffix node for this carving. The covering prefix for node $R$, which is stored as a

<table>
<thead>
<tr>
<th>5 bits</th>
<th>4 bits</th>
<th>1 bit</th>
<th>3 bits</th>
<th>12 bits</th>
<th>4 bits</th>
<th>4 bits</th>
<th>12 bits</th>
<th>12 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Match start position</td>
<td>Suffix count</td>
<td>Type</td>
<td>Index</td>
<td>next hop of S1</td>
<td>len(S2)</td>
<td>...</td>
<td>len(Sk)</td>
<td>next hop of S2</td>
</tr>
</tbody>
</table>
suffix whose length is 0 is the prefix in the nearest binary trie ancestor of $R$. Assuming
that each routing table contains the default prefix $*$, each node of the binary trie has
a well defined covering prefix. An example of $R$ in Figure 2-12, is the node $B$, since it
doesn’t contain a next hop. The remaining three nodes: $A$, $C$ and $D$ which are carved,
each has a next hop specified. Thus, covering prefix for node $B$ of the binary trie of
Figure 2-12 is prefix $1^*$ that has a next hop of H3. So, for example, the next hop for an
address 1101* will be H3. In practice, we store a covering prefix whenever the root of
the carved subtree does not contain a prefix. Hence, every suffix node has a prefix, its
first one, whose length is 0.

The $Q(R)$s and associated suffix nodes are assigned, respectively, to TCAM and
SRAM words in descending order of length.

For PETCAM we need to define a new format for a suffix node as well as formulate
an algorithm to populate suffix nodes with the suffixes of generalized prefixes. Our new
suffix-node structure has a 1-bit type field that permits the use of two variants. A type I
suffix node is used to store simple suffixes exclusively (i.e., all suffixes in a type I suffix
node are comprised of 0s and 1s). Such a suffix node is structured the same as the
suffix node of [21] except for the addition of a type field (Figure 2-13).

A type II suffix node (Figure 2-14) stores a mix of simple and non-simple suffixes
(i.e., suffixes that have at least one don’t care bit). Simple suffixes are stored first using
triples (length, suffix, next hop) as used in Figure 2-13. These triples are followed by
4-tuples (length, suffix, mask, next hop) that represent non-simple suffixes. The suffix
and mask entries are of the same length and the 1s in the mask identify the don’t cares
in the suffix. For example, the suffix x0x1 may be represented by the simple suffix 0001
and the mask 1010, for example. The index field gives the index of the first non-simple
suffix. So, for example, if we have 2 simple suffixes and 3 non-simple suffixes in a type II
suffix node, the count field would be 5 and the index field would be 3.
Besides having type II suffix nodes to accommodate generalized prefixes, our suffix node structure groups the lengths of different suffixes together, so they can be retrieved parallely during lookup.

2.3.3.2 Our carving heuristic

To carve the normalized ternary trie into suffix nodes, we first compute the following values for each node \( y \) of the normalized trie.

1. \( y.numP \) \( \cdots \) number of prefixes stored in the subtrie rooted at \( y \). This is equivalent to the number of nodes in this subtrie that have a non-null next hop field. Let \( y.h = 0 \) if \( y.nextHop \) is null and 1 otherwise. It is easy to see that \( y.numP \) is the sum of the \( numP \) values for its up to 2 non-empty subtrees plus \( y.h \).

2. \( y.xNumP \) \( \cdots \) number of nodes in the subtree rooted at \( y \) that have a non-null next hop stored and the path from \( y \) to each of these nodes includes at least one \( x \)-child other than \( y \). Note that if \( y \) has an \( x \)-child it can have no other child and so \( y.xNumP \) is the \( numP \) value of this \( x \)-child. When \( y \) does not have an \( x \)-child, its \( xNumP \) value is the sum of the \( xNumP \) values of its children.

3. \( y.size \) \( \cdots \) number of bits needed to store the suffixes (together with suffix count, node type, index (if required), suffix lengths, masks (if required), and next hops) for the prefixes in the subtrie rooted at \( y \). Each such suffix is obtained by removing \( Q(p) \) from the \( y.numP \) prefixes in the subtrie rooted at \( y \). In case \( y.xNumP = 0 \), a type I suffix node is used. Otherwise, a type II suffix node is used. \( y.size \) also includes the bits needed to store the next hop for the covering prefix for \( y \) in case this is needed. When a covering prefix is needed, we store a suffix of length 0 along with the next hop associated with this covering prefix.

Figure 2-15 gives the \( numP \), \( xNumP \), and \( size \) values for each of the nodes of the normalized ternary trie of Figure 2-7, where \( size \) here includes only a next hop and suffix bits for simplicity. For the leaf nodes in this example, \( size \) is 12 since there is no suffix bit needed and under the assumption that it takes 12 bits to store the next hop. For the parent nodes to the leaves, \( size \) is the sum of bits required to store the nextHop s at the children and a suffix bit for each child. Thus the parameters \( numP \), \( xNumP \), and \( size \) are computed in a bottom-up manner.

To illustrate the computation of \( (numP, xNumP, size) \) for an internal node in the example of Figure 2-15, we consider the node \( N \) reached from the root by bit 0, and the
subtree rooted at $N$. There are three prefixes in the subtree (corresponding to nextHop s H1, H2, H3), so $numP$ is 3. Out of the three prefixes, the ones corresponding to next hop H2 and H3 have a ‘don’t care’ bit in them, so $xNumP$ is 2. Next we take each prefix one by one to determine $size$. The suffix to be stored at $N$ for the prefix with next hop H1, is “0”. So, number of bits needed for this prefix is $12+1=13$, where 12 bits are needed to store the next hop H1. The suffix to be stored at $N$ for the prefix with next hop H2 is “0x0” which degenerates to suffix bits “000” and mask bits “010”. Similarly suffix and mask bits for the prefix with next hop H3 are “001” and “010” respectively. The number of bits needed for the two prefixes H2 and H3 is thus: $(12 + 3 + 3) \times 2 = 36$. Hence, the total number of bits is $36+13 = 49$. Note that in this simplified example, we have omitted the bits needed to represent the number of suffixes, the suffix lengths or the type of the nodes - each of these parameters are allotted a fixed number of bits in real life, as given by Figures 2-13 and 2-14.

To carve a normalized ternary trie into suffix nodes that use at most $w$ bits per node, we perform a postorder traversal of the trie using the visit function of Figure 2-16, which differs from that of [21] in the manner in which $size$ is computed since carving is done on generalized prefixes. Although the visit function, as stated in Figure 2-16, does not make explicit use of $numP$ and $xNumP$, these values are indirectly included in the
computation of \( \text{size} \), and the \textit{carve} function uses \( xNumP \) of node to be carved to detect what kind of suffix node is to be created, Type I or II. Note that in Figure 2-16, the value of \( y.size \) is its value at the time \( y \) is visited and accounts for the fact that several of \( y \)'s original subtrees may have been carved out by this time.

\[
\text{Algorithm visit}(y)
\]

\[
\begin{align*}
\text{if} & \ (y.size < w) \ \text{return}; \\
\text{if} & \ (y.size == w) \ \{\text{carve}(y); \ \text{return}; \} \\
\text{// y.size} & \ > w \\
\text{if} & \ (y \ \text{has a single child z}) \ \{\text{carve}(z); \ \text{return}; \} \\
\text{// z could be 0-, 1- or x-child}
\end{align*}
\]

\[
\begin{align*}
\text{// y has both a 0-child u and a 1-child v} \\
\text{if} & \ (u.size \leq v.size) \ \{ \\
\text{carve} & \ (v); \ \\
\text{recompute} & \ y.size; \\
\text{if} & \ (y.size < w) \ \text{return}; \\
\text{if} & \ (y.size == w) \ \text{carve}(y); \\
\text{else} & \ \text{carve}(u); \\
\} \\
\text{else} & \ // u.size > v.size \\
\text{this is symmetric to the case u.size} & \leq v.size
\end{align*}
\]

Figure 2-16. Visit function for subtree carving heuristic[21]

2.3.4 Storing and Updating Prefixes in TCAM-SRAM

The prefixes and suffix nodes generated as a result of carving a normalized ternary trie could be inserted in length order in a 1-level TCAM-SRAM system for lookup to return correct results. However, PETCAM uses an additional index level which reduces power consumption during lookup by selectively activating a partition of the data level TCAM. We describe our PETCAM structure and an update procedure below.

2.3.4.1 PETCAM structure

The structure of PETCAM is illustrated in Figure 2-17. PETCAM comprises a 2 level structure - the index level (ITCAM and ISRAM) and the data level (DTCAM and DSRAM) as shown in the Figure 2-17. The entries in the DTCAM are grouped into fixed sized partitions or buckets, and the bold lines represent bucket boundaries. Both
ISRAM and DSRAM have wide words and store suffix-nodes. Whereas the suffix-nodes in DSRAM store next hop corresponding to a suffix, the suffix-nodes in ISRAM store a DTCAM bucket index for the DTCAM bucket to be activated.

We choose the M-12Wb and 1-12Wc prefix layout schemes discussed in [21] to enter prefixes in PETCAM and evaluate both in our experimental section. M-12Wb scheme is recommended in [21] for overall memory and power optimization and 1-12Wc scheme is recommended for power optimization. In 1-12Wc scheme, there is a one-one relationship between a suffix in ISRAM and a DTCAM bucket index. In M-12Wb scheme, there is a many-one relationship between suffixes in ISRAM and a DTCAM bucket index. Thus in M-12Wb scheme, all buckets except the last one are full with prefixes which leads to better utilization of space, whereas in the 1-12Wc scheme, a DTCAM bucket may not be full if there are not enough DTCAM prefixes to fill a bucket corresponding to an index prefix.

To perform a lookup, an input IP address is first searched in the ITCAM. The best matching entry is used to access the corresponding ISRAM word which contains multiple suffixes. The best matching suffix in the ISRAM word is identified and the DTCAM bucket pointed to by the suffix is activated for a search. The input IP address is supplied to the DTCAM to find the best matching prefix in the activated bucket. The DSRAM word corresponding to the best matching DTCAM prefix is then analyzed to
Figure 2-18. Relative efficiency for table compaction

extract the appropriate next hop. So for power consumption calculations we consider the following components: power to search the whole ITCAM, power to search a bucket in the DTCAM which depends on the bucket size that is chosen by the user, power to read a word in both ISRAM and DSRAM. It is to be noted that ITCAM is considerably smaller in size (fewer entries) compared to DTCAM.

The word size of the DTCAM in Figure 2-17 is 32 bits for IPv4 prefixes. For the ITCAM, we need a word size of 24 bits based on (1) the bit allocation scheme to suffix nodes in Figure 2-14, (2) suffix node size of 144 bits from QDRII SRAM in quad burst mode and (3) our choice of DTCAM bucket size of 128 prefixes based on power consumption results in [21].

2.3.4.2 PETCAM updates

PETCAM supports batch updates rather than incremental updates. In fact, batch updating strategy is more suitable to the related architectures [20, 21, 35], none of which provide efficient support for incremental updates.

2.4 Experimental Results

We implemented our PETCAM strategy in C++ and compared its performance with the power reduction schemes of [20, 21, 35]. The comparison was done using 24 IPv4
routing tables downloaded from [6] and [37]. Data sets AS65000 and rrc00 are from May 2008, AS6447 is from July 2008, and the data sets AS1221 and AS4637 are earlier than 2008 and were used in [21]. The remaining datasets are from February, 2009. Table 2-3 shows the number of prefixes in each of the data sets. Our experiments aim to measure the relative effectiveness of the scheme of Liu [20] (type I redundancy removal followed by mask extension) and the PETCAM scheme (using Compact in Figure 2-4) to compact the routing table as well as the overall relative performance of PETCAM, EaseCAM, and the method of Lu and Sahni [21] with respect to TCAM power and memory reduction. For our experiments we assume the SRAM word size, and hence the size of a suffix node, is 144 bits.

2.4.1 Compaction Efficiency

The compaction efficiency is measured by the number of prefixes after applying the compaction techniques. Figure 2-18 shows the the relative efficiency of prefix set compaction using PETCAM, Liu’s compaction scheme [20] and type II redundancy removal integrated in Liu’s compaction scheme (after removing type I and type II redundancies and applying mask extension).

In PETCAM, Step 1 of the Compact algorithm reduces the number of prefixes in the data sets by between 45% and 79%. Another approximately 5% reduction is achieved when Step 2 is executed on the optimal set of prefixes obtained in Step 1. So, overall PETCAM reduces the number of prefixes by about 50% to 84%.

We do not report the results of compaction using EaseCAM[35] because, as noted in Section 2.2.2, these enhancements do not guarantee compacted prefix sets equivalent to the input prefix set. For each of our data sets, PETCAM is more efficient in terms of prefix compaction, resulting in fewer generalized prefixes compared to the method of [20].
Table 2-3. Number of routing table prefixes in PETCAM

<table>
<thead>
<tr>
<th>Index</th>
<th>DataSet</th>
<th># of prefixes</th>
<th>Index</th>
<th>DataSet</th>
<th># of prefixes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>AS1221</td>
<td>281516</td>
<td>13</td>
<td>rrc11</td>
<td>277166</td>
</tr>
<tr>
<td>2</td>
<td>AS4637</td>
<td>210119</td>
<td>14</td>
<td>rrc12</td>
<td>278499</td>
</tr>
<tr>
<td>3</td>
<td>AS6447</td>
<td>275509</td>
<td>15</td>
<td>rrc13</td>
<td>284986</td>
</tr>
<tr>
<td>4</td>
<td>AS65000</td>
<td>259026</td>
<td>16</td>
<td>rrc14</td>
<td>276170</td>
</tr>
<tr>
<td>5</td>
<td>rrc00</td>
<td>266185</td>
<td>17</td>
<td>rrc15</td>
<td>284047</td>
</tr>
<tr>
<td>6</td>
<td>rrc01</td>
<td>276795</td>
<td>18</td>
<td>rrc16</td>
<td>282660</td>
</tr>
<tr>
<td>7</td>
<td>rrc03</td>
<td>283754</td>
<td>19</td>
<td>rviews2</td>
<td>294127</td>
</tr>
<tr>
<td>8</td>
<td>rrc04</td>
<td>288610</td>
<td>20</td>
<td>rviews4</td>
<td>275737</td>
</tr>
<tr>
<td>9</td>
<td>rrc05</td>
<td>280041</td>
<td>21</td>
<td>rviews.eqix</td>
<td>275736</td>
</tr>
<tr>
<td>10</td>
<td>rrc06</td>
<td>278744</td>
<td>22</td>
<td>rviews.isc</td>
<td>281095</td>
</tr>
<tr>
<td>11</td>
<td>rrc07</td>
<td>275097</td>
<td>23</td>
<td>rviews.linx</td>
<td>278196</td>
</tr>
<tr>
<td>12</td>
<td>rrc10</td>
<td>278898</td>
<td>24</td>
<td>rviews.wide</td>
<td>283569</td>
</tr>
</tbody>
</table>

2.4.2 Power Efficiency

We compared the power efficiency obtained using PETCAM with that of EaseCAM\[35\] and the schemes of Lu and Sahni\[21\]. Although the prefix preprocessing schemes in\[35\] are faulty, the two-level EaseCAM architecture, which is a specialization of the bit-selection architecture proposed by Zane et al. \[55\], may be employed in conjunction with any prefix set to reduce TCAM power as well as total TCAM memory. In EaseCAM, each TCAM word is 24 bits as the first 8 bits of each prefix are used in the level-1 index to the TCAM. Prefixes shorter than 8 bits are handled in a separate bucket and are ignored in our evaluation of EaseCAM.

We store prefixes in EaseCAM after modifying the prefix preprocessing scheme as follows. First, we remove type I and type II redundancies to reduce the size of the prefix table. The group of prefixes with the same first 8 bits is further divided into subgroups of prefixes having the same length and next hop. Each subgroup of prefixes is then minimized using Espresso. The minimized prefixes are placed in the TCAM according to the length of the prefixes in a subgroup. Thus logic minimization is done using the strategy of \[20\]. The difference with \[20\] is that the logic minimization is applied on subgroups of prefixes with the same first octet, length and next hop, whereas in \[20\] the
<table>
<thead>
<tr>
<th>DataSet</th>
<th>PTM</th>
<th>PTW</th>
<th>LuM</th>
<th>LuW</th>
<th>EaseCAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>AS1221</td>
<td>19.17</td>
<td>13.5</td>
<td>34.03</td>
<td>15.9</td>
<td>2041.62</td>
</tr>
<tr>
<td>AS4637</td>
<td>13.41</td>
<td>11.93</td>
<td>28.44</td>
<td>14.79</td>
<td>391.27</td>
</tr>
<tr>
<td>AS6447</td>
<td>19.62</td>
<td>13.57</td>
<td>32.77</td>
<td>15.64</td>
<td>662.38</td>
</tr>
<tr>
<td>AS65000</td>
<td>15.55</td>
<td>12.52</td>
<td>31.6</td>
<td>15.77</td>
<td>528.43</td>
</tr>
<tr>
<td>rrc00</td>
<td>16.32</td>
<td>12.72</td>
<td>34.74</td>
<td>15.7</td>
<td>563.25</td>
</tr>
<tr>
<td>rrc01</td>
<td>16.71</td>
<td>13.24</td>
<td>29.28</td>
<td>14.6</td>
<td>1710.7</td>
</tr>
<tr>
<td>rrc03</td>
<td>17.09</td>
<td>12.91</td>
<td>27.35</td>
<td>14.6</td>
<td>350.02</td>
</tr>
<tr>
<td>rrc04</td>
<td>16.84</td>
<td>12.91</td>
<td>27.6</td>
<td>14.53</td>
<td>317.47</td>
</tr>
<tr>
<td>rrc05</td>
<td>16.58</td>
<td>12.78</td>
<td>27.54</td>
<td>14.21</td>
<td>304.09</td>
</tr>
<tr>
<td>rrc06</td>
<td>14.64</td>
<td>12.33</td>
<td>25.54</td>
<td>13.76</td>
<td>231.18</td>
</tr>
<tr>
<td>rrc07</td>
<td>15.16</td>
<td>12.65</td>
<td>26.12</td>
<td>13.89</td>
<td>285.82</td>
</tr>
<tr>
<td>rrc10</td>
<td>17.42</td>
<td>13.1</td>
<td>28.51</td>
<td>14.98</td>
<td>346.31</td>
</tr>
<tr>
<td>rrc11</td>
<td>16.0</td>
<td>12.52</td>
<td>26.19</td>
<td>14.08</td>
<td>290.65</td>
</tr>
<tr>
<td>rrc12</td>
<td>16.77</td>
<td>12.91</td>
<td>27.35</td>
<td>14.15</td>
<td>323.46</td>
</tr>
<tr>
<td>rrc13</td>
<td>14.77</td>
<td>12.39</td>
<td>26.38</td>
<td>13.95</td>
<td>286.08</td>
</tr>
<tr>
<td>rrc14</td>
<td>16.51</td>
<td>12.97</td>
<td>27.41</td>
<td>14.4</td>
<td>368.87</td>
</tr>
<tr>
<td>rrc15</td>
<td>14.32</td>
<td>12.33</td>
<td>25.35</td>
<td>13.76</td>
<td>230.79</td>
</tr>
<tr>
<td>rrc16</td>
<td>16.45</td>
<td>12.72</td>
<td>27.28</td>
<td>14.15</td>
<td>323.2</td>
</tr>
<tr>
<td>rviews2</td>
<td>17.16</td>
<td>13.04</td>
<td>29.28</td>
<td>14.66</td>
<td>381.47</td>
</tr>
<tr>
<td>rviews4</td>
<td>15.1</td>
<td>12.52</td>
<td>26.06</td>
<td>14.02</td>
<td>297.97</td>
</tr>
<tr>
<td>rviews.eqix</td>
<td>15.42</td>
<td>12.46</td>
<td>25.93</td>
<td>13.95</td>
<td>316.76</td>
</tr>
<tr>
<td>rviews.isc</td>
<td>17.09</td>
<td>12.65</td>
<td>27.15</td>
<td>14.15</td>
<td>324.49</td>
</tr>
<tr>
<td>rviews.linx</td>
<td>16.19</td>
<td>12.59</td>
<td>26.25</td>
<td>13.95</td>
<td>337.09</td>
</tr>
<tr>
<td>rviews.wide</td>
<td>14.52</td>
<td>12.39</td>
<td>25.74</td>
<td>13.76</td>
<td>241.17</td>
</tr>
</tbody>
</table>

Logic minimization is applied on subgroups of prefixes with the same length and next hop.

To generate the data for PETCAM, we employ steps 1 and 2 of the PETCAM scheme and store the resulting generalized prefixes in a 2-level TCAM system using the M-12Wb layout. The sizes of a DTCAM bucket and a suffix node are set, respectively, to 128 TCAM prefixes and 144 SRAM bits for QDRII quad burst SRAMs.

Similarly, to obtain data for the schemes in [21], we use the M-12Wb layout or 1-12Wc layout after setting a DTCAM bucket size to 128 prefixes and a suffix node size to 144 bits.
Figure 2-19. Comparison of total power (TCAM + SRAM) among different schemes

We used CACTI [30] and a TCAM power estimation tool [1] on 70nm process, at 1.12V to estimate power consumption. The memories are assumed to have a single read-write port. This assumption works well for PETCAM, which uses two copies of the forwarding table, one for lookup and the other for updates. Even though EaseCAM permits incremental updates, the assumption is reasonable since the TCAM will be used for either lookup or updates at any time. Simultaneous lookup and update via dual ports will entail consistency issues which need to be addressed separately. Thus we consider memories with a single read-write port. Table 2-4 presents the total power for the TCAMs (e.g. ITCAM and DTCAM for PETCAM) assuming the TCAMs are being operated at 360MHz [36]. The columns marked PTM and PTW give the power consumed by PETCAM while using prefix layout schemes M-12Wb and 1-12Wc of [21], respectively. The columns marked LuM and LuW give the power consumed by the schemes in [21] while using prefix layout schemes M-12Wb and 1-12Wc respectively. For PTM, PTW, LuM and LuW the total TCAM power for a routing table is obtained by summing the dynamic and leakage power for a search in both ITCAM and DTCAM. The
Table 2-5. SRAM power (in mW) for the different schemes

<table>
<thead>
<tr>
<th>DataSet</th>
<th>PTM</th>
<th>PTW</th>
<th>LuM</th>
<th>LuW</th>
<th>EaseCAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>AS1221</td>
<td>16.88</td>
<td>11.17</td>
<td>20.73</td>
<td>13.21</td>
<td>82.39</td>
</tr>
<tr>
<td>AS4637</td>
<td>12.06</td>
<td>9.73</td>
<td>19.41</td>
<td>12.27</td>
<td>19.1</td>
</tr>
<tr>
<td>AS6447</td>
<td>17.25</td>
<td>11.23</td>
<td>22.95</td>
<td>13.0</td>
<td>28.74</td>
</tr>
<tr>
<td>AS65000</td>
<td>13.86</td>
<td>10.35</td>
<td>19.49</td>
<td>13.1</td>
<td>23.45</td>
</tr>
<tr>
<td>rrc00</td>
<td>14.5</td>
<td>10.51</td>
<td>21.2</td>
<td>13.05</td>
<td>24.84</td>
</tr>
<tr>
<td>rrc01</td>
<td>14.82</td>
<td>10.93</td>
<td>20.1</td>
<td>12.11</td>
<td>69.51</td>
</tr>
<tr>
<td>rrc03</td>
<td>15.14</td>
<td>10.67</td>
<td>18.51</td>
<td>12.11</td>
<td>17.95</td>
</tr>
<tr>
<td>rrc04</td>
<td>14.93</td>
<td>10.67</td>
<td>18.73</td>
<td>12.06</td>
<td>15.76</td>
</tr>
<tr>
<td>rrc05</td>
<td>14.71</td>
<td>10.56</td>
<td>18.67</td>
<td>11.76</td>
<td>15.28</td>
</tr>
<tr>
<td>rrc06</td>
<td>13.1</td>
<td>10.19</td>
<td>17.04</td>
<td>11.39</td>
<td>12.03</td>
</tr>
<tr>
<td>rrc07</td>
<td>13.53</td>
<td>10.46</td>
<td>17.51</td>
<td>11.49</td>
<td>14.49</td>
</tr>
<tr>
<td>rrc10</td>
<td>15.4</td>
<td>10.83</td>
<td>19.46</td>
<td>12.43</td>
<td>17.75</td>
</tr>
<tr>
<td>rrc11</td>
<td>14.23</td>
<td>10.35</td>
<td>17.56</td>
<td>11.65</td>
<td>14.72</td>
</tr>
<tr>
<td>rrc12</td>
<td>14.88</td>
<td>10.67</td>
<td>18.51</td>
<td>11.7</td>
<td>16.03</td>
</tr>
<tr>
<td>rrc13</td>
<td>13.21</td>
<td>10.25</td>
<td>17.72</td>
<td>11.55</td>
<td>14.49</td>
</tr>
<tr>
<td>rrc14</td>
<td>14.66</td>
<td>10.72</td>
<td>18.57</td>
<td>11.95</td>
<td>18.12</td>
</tr>
<tr>
<td>rrc15</td>
<td>12.84</td>
<td>10.19</td>
<td>16.88</td>
<td>11.39</td>
<td>12.03</td>
</tr>
<tr>
<td>rrc16</td>
<td>14.61</td>
<td>10.51</td>
<td>18.46</td>
<td>11.7</td>
<td>15.99</td>
</tr>
<tr>
<td>rviews2</td>
<td>15.19</td>
<td>10.78</td>
<td>20.1</td>
<td>12.16</td>
<td>18.65</td>
</tr>
<tr>
<td>rviews4</td>
<td>13.48</td>
<td>10.35</td>
<td>17.46</td>
<td>11.6</td>
<td>15.0</td>
</tr>
<tr>
<td>rviews.eqix</td>
<td>13.75</td>
<td>10.3</td>
<td>17.35</td>
<td>11.55</td>
<td>15.77</td>
</tr>
<tr>
<td>rviews.isc</td>
<td>15.14</td>
<td>10.46</td>
<td>18.36</td>
<td>11.7</td>
<td>16.08</td>
</tr>
<tr>
<td>rviews.linx</td>
<td>14.39</td>
<td>10.4</td>
<td>17.62</td>
<td>11.55</td>
<td>17.32</td>
</tr>
<tr>
<td>rviews.wide</td>
<td>13.0</td>
<td>10.25</td>
<td>17.19</td>
<td>11.39</td>
<td>12.5</td>
</tr>
</tbody>
</table>

Power consumption is the same for any lookup. For EaseCAM, the total TCAM power for a routing table is obtained by summing the dynamic and leakage power for a search in the TCAM at the first level and for a search in the largest partition in the TCAM at the second level, and therefore represents worst case power consumption. Both schemes in PETCAM consumed very low power compared to EaseCAM. Table 2-5 presents the total SRAM power consumed by a read access on each SRAM. Figure 2-19 plots the total power for each routing table, obtained by summing total TCAM and SRAM power for a lookup. From these figures, if we compare EaseCAM with PETCAM using M-12Wb scheme(PTM), we get 88% to 98% reduction in total power. Comparing EaseCAM with PETCAM using 1-12Wc scheme(PTW), we get 90% to 98% reduction in total power.
Both TCAM and SRAM power consumption in PTW is between 8% to 20% of that of LuW or Lu’s 1-12Wc layout. The power consumed by the SRAM in PETCAM is at most 1.4 times more than that of EaseCAM. For some instances, the SRAM power in PETCAM was actually smaller than that of EaseCAM. For example, in AS1221. We found that this is mainly due to the large number of SRAM bits activated during lookup, which resulted in a very high leakage power in EaseCAM.

Thus, PETCAM using 1-12Wc layout(PTW) is the most power efficient scheme on our datasets, the next best scheme being PETCAM with M-12Wb layout (PTM).
Table 2-7. SRAM area (in mm$^2$) for the different schemes

<table>
<thead>
<tr>
<th>DataSet</th>
<th>PTM</th>
<th>PTW</th>
<th>LuM</th>
<th>LuW</th>
<th>EaseCAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>AS1221</td>
<td>4.89</td>
<td>6.01</td>
<td>8.99</td>
<td>11.16</td>
<td>2.05</td>
</tr>
<tr>
<td>AS4637</td>
<td>1.55</td>
<td>1.95</td>
<td>6.78</td>
<td>8.6</td>
<td>0.6</td>
</tr>
<tr>
<td>AS6447</td>
<td>4.86</td>
<td>6.18</td>
<td>8.83</td>
<td>11.25</td>
<td>2.24</td>
</tr>
<tr>
<td>AS65000</td>
<td>2.79</td>
<td>3.57</td>
<td>8.35</td>
<td>10.52</td>
<td>1.15</td>
</tr>
<tr>
<td>rrc00</td>
<td>3.11</td>
<td>4.01</td>
<td>9.41</td>
<td>11.99</td>
<td>1.49</td>
</tr>
<tr>
<td>rrc01</td>
<td>3.82</td>
<td>4.88</td>
<td>6.74</td>
<td>8.41</td>
<td>2.05</td>
</tr>
<tr>
<td>rrc03</td>
<td>3.5</td>
<td>4.51</td>
<td>6.14</td>
<td>7.88</td>
<td>1.66</td>
</tr>
<tr>
<td>rrc04</td>
<td>3.41</td>
<td>4.37</td>
<td>6.2</td>
<td>7.93</td>
<td>1.56</td>
</tr>
<tr>
<td>rrc05</td>
<td>3.17</td>
<td>4.01</td>
<td>5.87</td>
<td>7.54</td>
<td>1.44</td>
</tr>
<tr>
<td>rrc06</td>
<td>2.27</td>
<td>2.81</td>
<td>4.81</td>
<td>6.04</td>
<td>0.92</td>
</tr>
<tr>
<td>rrc07</td>
<td>2.92</td>
<td>3.71</td>
<td>5.19</td>
<td>6.7</td>
<td>1.25</td>
</tr>
<tr>
<td>rrc10</td>
<td>4.12</td>
<td>5.27</td>
<td>6.77</td>
<td>8.65</td>
<td>1.73</td>
</tr>
<tr>
<td>rrc11</td>
<td>3.11</td>
<td>3.91</td>
<td>5.36</td>
<td>6.86</td>
<td>1.34</td>
</tr>
<tr>
<td>rrc12</td>
<td>3.61</td>
<td>4.61</td>
<td>5.84</td>
<td>7.57</td>
<td>1.58</td>
</tr>
<tr>
<td>rrc13</td>
<td>2.48</td>
<td>3.17</td>
<td>5.38</td>
<td>6.91</td>
<td>1.25</td>
</tr>
<tr>
<td>rrc14</td>
<td>3.59</td>
<td>4.65</td>
<td>5.87</td>
<td>7.68</td>
<td>1.57</td>
</tr>
<tr>
<td>rrc15</td>
<td>2.13</td>
<td>2.65</td>
<td>4.62</td>
<td>5.79</td>
<td>0.86</td>
</tr>
<tr>
<td>rrc16</td>
<td>3.42</td>
<td>4.29</td>
<td>5.74</td>
<td>7.48</td>
<td>1.5</td>
</tr>
<tr>
<td>rviews2</td>
<td>3.91</td>
<td>4.97</td>
<td>6.72</td>
<td>8.6</td>
<td>1.9</td>
</tr>
<tr>
<td>rviews4</td>
<td>2.79</td>
<td>3.57</td>
<td>5.19</td>
<td>6.62</td>
<td>1.17</td>
</tr>
<tr>
<td>rviews.eqix</td>
<td>2.97</td>
<td>3.69</td>
<td>5.25</td>
<td>6.75</td>
<td>1.28</td>
</tr>
<tr>
<td>rviews.isc</td>
<td>3.27</td>
<td>4.26</td>
<td>5.67</td>
<td>7.27</td>
<td>1.39</td>
</tr>
<tr>
<td>rviews.linx</td>
<td>3.23</td>
<td>4.04</td>
<td>5.54</td>
<td>7.05</td>
<td>1.41</td>
</tr>
<tr>
<td>rviews.wide</td>
<td>2.39</td>
<td>3.02</td>
<td>4.94</td>
<td>6.26</td>
<td>0.98</td>
</tr>
</tbody>
</table>

2.4.3 Area Efficiency

We obtained the area for the memories as follows. For TCAMs, we have followed the TCAM area estimation method used by Hashmi et al. in [16]. The area of a single TCAM cell is taken as $3.59 \mu m^2$ [31] for our calculations. We multiply the area of a single cell by the number of TCAM cells needed for storing each table in our dataset and then scale up the resulting area by 30% to take into account additional routing overhead. We scale down the area number for 70nm technology using a linear scaling model [7]. For SRAMs, we obtain the area numbers from the CACTI results. Table 2-6 presents the total TCAM area, which sums the area occupied by all TCAMs used in a scheme. Table 2-7 presents the total SRAM area, which is the sum of area occupied by
Table 2-8. TCAM size (in KB) for the different schemes

<table>
<thead>
<tr>
<th>DataSet</th>
<th>PTM</th>
<th>PTW</th>
<th>LuM</th>
<th>LuW</th>
<th>EaseCAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>AS1221</td>
<td>150.91</td>
<td>186.6</td>
<td>279.29</td>
<td>349.21</td>
<td>581.51</td>
</tr>
<tr>
<td>AS4637</td>
<td>46.15</td>
<td>58.53</td>
<td>211.05</td>
<td>267.16</td>
<td>170.77</td>
</tr>
<tr>
<td>AS6447</td>
<td>149.93</td>
<td>192.11</td>
<td>274.25</td>
<td>352.2</td>
<td>635.41</td>
</tr>
<tr>
<td>AS65000</td>
<td>84.75</td>
<td>109.06</td>
<td>258.69</td>
<td>328.71</td>
<td>326.85</td>
</tr>
<tr>
<td>rrc00</td>
<td>94.28</td>
<td>123.07</td>
<td>292.82</td>
<td>375.7</td>
<td>422.38</td>
</tr>
<tr>
<td>rrc01</td>
<td>116.8</td>
<td>150.59</td>
<td>275.74</td>
<td>352.2</td>
<td>582.51</td>
</tr>
<tr>
<td>rrc03</td>
<td>106.82</td>
<td>139.08</td>
<td>282.76</td>
<td>361.21</td>
<td>472.74</td>
</tr>
<tr>
<td>rrc04</td>
<td>103.8</td>
<td>134.58</td>
<td>286.79</td>
<td>365.72</td>
<td>443.89</td>
</tr>
<tr>
<td>rrc05</td>
<td>96.29</td>
<td>123.07</td>
<td>278.79</td>
<td>357.21</td>
<td>408.14</td>
</tr>
<tr>
<td>rrc06</td>
<td>68.71</td>
<td>85.55</td>
<td>277.74</td>
<td>355.69</td>
<td>259.68</td>
</tr>
<tr>
<td>rrc07</td>
<td>86.98</td>
<td>113.56</td>
<td>274.25</td>
<td>348.2</td>
<td>354.5</td>
</tr>
<tr>
<td>rrc10</td>
<td>126.33</td>
<td>163.08</td>
<td>277.74</td>
<td>353.71</td>
<td>490.98</td>
</tr>
<tr>
<td>rrc11</td>
<td>94.27</td>
<td>120.06</td>
<td>276.22</td>
<td>352.2</td>
<td>379.38</td>
</tr>
<tr>
<td>rrc12</td>
<td>110.3</td>
<td>142.08</td>
<td>277.24</td>
<td>354.21</td>
<td>448.52</td>
</tr>
<tr>
<td>rrc13</td>
<td>75.21</td>
<td>96.55</td>
<td>283.79</td>
<td>362.2</td>
<td>355.95</td>
</tr>
<tr>
<td>rrc14</td>
<td>109.79</td>
<td>143.58</td>
<td>274.74</td>
<td>351.2</td>
<td>447.54</td>
</tr>
<tr>
<td>rrc15</td>
<td>64.19</td>
<td>80.55</td>
<td>282.77</td>
<td>362.21</td>
<td>244.41</td>
</tr>
<tr>
<td>rrc16</td>
<td>104.29</td>
<td>132.07</td>
<td>281.78</td>
<td>361.21</td>
<td>425.04</td>
</tr>
<tr>
<td>rviews2</td>
<td>119.82</td>
<td>153.58</td>
<td>292.82</td>
<td>373.21</td>
<td>537.81</td>
</tr>
<tr>
<td>rviews4</td>
<td>84.73</td>
<td>109.06</td>
<td>274.75</td>
<td>350.21</td>
<td>333.0</td>
</tr>
<tr>
<td>rviews.eqix</td>
<td>90.24</td>
<td>113.06</td>
<td>274.74</td>
<td>350.7</td>
<td>363.63</td>
</tr>
<tr>
<td>rviews.isc</td>
<td>99.32</td>
<td>131.06</td>
<td>280.23</td>
<td>357.71</td>
<td>393.43</td>
</tr>
<tr>
<td>rviews.linx</td>
<td>98.28</td>
<td>124.06</td>
<td>277.25</td>
<td>353.2</td>
<td>400.25</td>
</tr>
<tr>
<td>rviews.wide</td>
<td>72.2</td>
<td>91.55</td>
<td>282.25</td>
<td>362.2</td>
<td>276.3</td>
</tr>
</tbody>
</table>

all SRAMs used in a scheme. The area occupied by the SRAMs in PETCAM (PTM) is at most 2.6 times that in EaseCAM. PETCAM uses about half or sometimes lesser SRAM than either of Lu's schemes. Tables 2-8 and 2-9 show the cost in terms of number of bytes.

PETCAM using M-12Wb layout (PTM) is the most space efficient scheme on our datasets. Thus, if the main design objective is power reduction, then the PETCAM scheme using 1-12Wc prefix layout should be used. On the other hand, for overall power and space optimization, PETCAM scheme with M-12Wb prefix layout should be chosen.
Table 2-9. SRAM size (in KB) for the different schemes

<table>
<thead>
<tr>
<th>DataSet</th>
<th>PTM</th>
<th>PTW</th>
<th>LuM</th>
<th>LuW</th>
<th>EaseCAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>AS1221</td>
<td>664.31</td>
<td>839.29</td>
<td>1231.09</td>
<td>1570.59</td>
<td>286.76</td>
</tr>
<tr>
<td>AS4637</td>
<td>201.93</td>
<td>263.26</td>
<td>930.46</td>
<td>1201.57</td>
<td>81.39</td>
</tr>
<tr>
<td>AS6447</td>
<td>658.99</td>
<td>864.04</td>
<td>1210.35</td>
<td>1584.08</td>
<td>313.71</td>
</tr>
<tr>
<td>AS65000</td>
<td>372.79</td>
<td>490.52</td>
<td>1140.49</td>
<td>1478.34</td>
<td>159.42</td>
</tr>
<tr>
<td>rrc00</td>
<td>414.43</td>
<td>553.53</td>
<td>1291.31</td>
<td>1689.83</td>
<td>207.19</td>
</tr>
<tr>
<td>rrc01</td>
<td>514.13</td>
<td>677.29</td>
<td>1216.1</td>
<td>1584.08</td>
<td>287.26</td>
</tr>
<tr>
<td>rrc03</td>
<td>470.14</td>
<td>625.53</td>
<td>1246.04</td>
<td>1624.59</td>
<td>232.37</td>
</tr>
<tr>
<td>rrc04</td>
<td>455.15</td>
<td>605.28</td>
<td>1265.97</td>
<td>1644.84</td>
<td>217.95</td>
</tr>
<tr>
<td>rrc05</td>
<td>422.49</td>
<td>553.53</td>
<td>1228.79</td>
<td>1606.59</td>
<td>200.07</td>
</tr>
<tr>
<td>rrc06</td>
<td>301.85</td>
<td>384.77</td>
<td>1223.69</td>
<td>1599.83</td>
<td>125.84</td>
</tr>
<tr>
<td>rrc07</td>
<td>390.35</td>
<td>510.78</td>
<td>1208.63</td>
<td>1566.08</td>
<td>173.25</td>
</tr>
<tr>
<td>rrc10</td>
<td>557.26</td>
<td>733.54</td>
<td>1224.8</td>
<td>1590.84</td>
<td>241.49</td>
</tr>
<tr>
<td>rrc11</td>
<td>413.81</td>
<td>540.02</td>
<td>1217.22</td>
<td>1584.08</td>
<td>185.69</td>
</tr>
<tr>
<td>rrc12</td>
<td>486.13</td>
<td>639.03</td>
<td>1224.22</td>
<td>1593.09</td>
<td>220.26</td>
</tr>
<tr>
<td>rrc13</td>
<td>331.0</td>
<td>434.27</td>
<td>1251.13</td>
<td>1629.08</td>
<td>173.97</td>
</tr>
<tr>
<td>rrc14</td>
<td>482.82</td>
<td>645.78</td>
<td>1212.6</td>
<td>1579.58</td>
<td>219.77</td>
</tr>
<tr>
<td>rrc15</td>
<td>280.91</td>
<td>362.27</td>
<td>1245.57</td>
<td>1629.09</td>
<td>118.21</td>
</tr>
<tr>
<td>rrc16</td>
<td>459.95</td>
<td>594.03</td>
<td>1240.9</td>
<td>1624.59</td>
<td>208.52</td>
</tr>
<tr>
<td>rviews2</td>
<td>527.27</td>
<td>690.78</td>
<td>1290.89</td>
<td>1678.59</td>
<td>264.9</td>
</tr>
<tr>
<td>rviews4</td>
<td>373.19</td>
<td>490.52</td>
<td>1210.74</td>
<td>1575.09</td>
<td>162.5</td>
</tr>
<tr>
<td>rviews.eqix</td>
<td>398.03</td>
<td>508.52</td>
<td>1211.04</td>
<td>1577.33</td>
<td>177.82</td>
</tr>
<tr>
<td>rviews.isc</td>
<td>436.88</td>
<td>589.53</td>
<td>1235.47</td>
<td>1608.84</td>
<td>192.71</td>
</tr>
<tr>
<td>rviews.linx</td>
<td>432.45</td>
<td>558.03</td>
<td>1222.04</td>
<td>1588.58</td>
<td>196.13</td>
</tr>
<tr>
<td>rviews.wide</td>
<td>317.26</td>
<td>411.77</td>
<td>1243.3</td>
<td>1629.08</td>
<td>134.15</td>
</tr>
</tbody>
</table>

2.4.4 PETCAMLite

Since the Step 2 (mask extension) of the algorithm Compact for PETCAM is quite time consuming, we investigate a light version, PETCAMLite, of PETCAM in which Step 2 is omitted. The structure of PETCAMLite is the same as PETCAM as given in Figure 2-17. Our experiments indicate that PETCAMLite requires 0% to 6% more TCAM power and 0.5% to 2% more TCAM memory than required by PETCAM. So, if Step 2 takes more computational resource than we wish to invest, we may use PETCAMLite and gain almost the same power and memory benefits as provided by PETCAM.

Table 2-10 shows the CPU time on a Sun4u Sparc SunOS 5.8 machine for executing steps 1 and 2. The step to map the reduced set of generalized prefixes 2-level TCAM...
takes between 1 and 4 seconds. So, if a Sun4u Sparc is used as the rebuild engine, the interval between successive rebuilds of the TCAM system will need to be at least 700 seconds for PETCAM but only about 10 seconds for PETCAMLite.

Table 2-10. Execution time in seconds for Steps 1 and 2 of Compact

<table>
<thead>
<tr>
<th>DataSet</th>
<th>Step 1(s)</th>
<th>Step 2(s)</th>
<th>DataSet</th>
<th>Step 1(s)</th>
<th>Step 2(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AS1221</td>
<td>5.38</td>
<td>642.83</td>
<td>rrc11</td>
<td>5.34</td>
<td>502.73</td>
</tr>
<tr>
<td>AS4637</td>
<td>3.7</td>
<td>296.62</td>
<td>rrc12</td>
<td>5.47</td>
<td>442.99</td>
</tr>
<tr>
<td>AS6447</td>
<td>5.14</td>
<td>347.25</td>
<td>rrc13</td>
<td>5.48</td>
<td>348.49</td>
</tr>
<tr>
<td>AS65000</td>
<td>4.57</td>
<td>600.55</td>
<td>rrc14</td>
<td>5.37</td>
<td>458.24</td>
</tr>
<tr>
<td>rrc00</td>
<td>4.78</td>
<td>407.05</td>
<td>rrc15</td>
<td>5.34</td>
<td>680.52</td>
</tr>
<tr>
<td>rrc01</td>
<td>5.45</td>
<td>329.04</td>
<td>rrc16</td>
<td>5.45</td>
<td>565.99</td>
</tr>
<tr>
<td>rrc03</td>
<td>5.56</td>
<td>399.9</td>
<td>rviews2</td>
<td>5.81</td>
<td>290.39</td>
</tr>
<tr>
<td>rrc04</td>
<td>5.71</td>
<td>581.65</td>
<td>rviews4</td>
<td>5.23</td>
<td>753.43</td>
</tr>
<tr>
<td>rrc05</td>
<td>5.42</td>
<td>561.9</td>
<td>rviews.eqix</td>
<td>5.22</td>
<td>656.25</td>
</tr>
<tr>
<td>rrc06</td>
<td>5.22</td>
<td>746.96</td>
<td>rviews.isc</td>
<td>5.5</td>
<td>565.7</td>
</tr>
<tr>
<td>rrc07</td>
<td>5.27</td>
<td>591.65</td>
<td>rviews.linx</td>
<td>5.61</td>
<td>507.44</td>
</tr>
<tr>
<td>rrc10</td>
<td>5.46</td>
<td>1274.07</td>
<td>rviews.wide</td>
<td>5.31</td>
<td>710.77</td>
</tr>
</tbody>
</table>

2.4.5 Implementation of Nexthop Computation

We designed a circuit using Verilog that implements a method for extracting nextHop information from a wide SRAM word. This design is used to estimate the associated costs in terms of timing, power and gate count. The design takes the following inputs: a wide SRAM word, the destination address being used in current lookup and a clock signal. It outputs the most appropriate nextHop. If suffix matching fails, then the default nextHop is returned. The circuit is a three stage pipeline, as illustrated in the Figure 2-20. The first stage determines the start position of nextHop and the start position of suffixes. For example, based on Figure 2-13, the nextHop start from bit number \((5+4+1+12+4 \times \text{suffixCount})\), since it starts past the fields - Match start position, Suffix count, Type, nextHop of S1, and consecutive 4-bit fields denoting the lengths of the different suffixes. This stage also extracts the destination bits for suffix matching. The bits are extracted from the destination address starting from position indicated by the ‘Match start position’ field in the suffix node up to a maximum of 15 bits (since we limit suffix length to a maximum of 15 bits). A set of 15-bit masks
Figure 2-20. Pipeline processing suffix node with variable suffix lengths

is also generated at this stage, one for each suffix, based on suffix length and this
mask is used during suffix matching. For example, a suffix of length 4 uses a mask
11110000000000000000 for matching with the destination bits. Suffix matching is done
parallely in the second stage for all suffixes in the SRAM word. In the third stage, the
next hop corresponding to the best matching suffix is sent to the output. We used
Synopsys Design Compiler to synthesize the logic using a 0.18 $\mu$m library [47, 48]
as well as a 45nm library [46] and found that our design successfully met the timing
constraints with 500MHz and 1600MHz clock respectively for the two technologies.
The results are presented in the Table 2-11. The throughput is represented in terms of

<table>
<thead>
<tr>
<th>Process</th>
<th>Time (ns)</th>
<th>Throughput (Mtps)</th>
<th>Voltage (V)</th>
<th>Power (mW)</th>
<th>Gate Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.18$\mu$m</td>
<td>2.00</td>
<td>500</td>
<td>1.8</td>
<td>17.40</td>
<td>5124</td>
</tr>
<tr>
<td>45nm</td>
<td>0.62</td>
<td>1600</td>
<td>1.1</td>
<td>10.92</td>
<td>6560</td>
</tr>
</tbody>
</table>

million searches per second (Mtps). An example of a TCAM with a speed of 143MHz
(effectively, 143 Mtps) can be found in [31], which uses 0.13$\mu$m technology. Though
we couldn’t find a target library for synthesis using 0.13$\mu$m, it is expected that the delay
overhead and throughput of our design will improve compared to that with 0.18$\mu$m
technology. In other words, our pipelined design can operate at the same speed of TCAM and therefore our suffix node processing will not introduce any additional lookup performance overhead.

There is a tradeoff between TCAM usage and SRAM search time/additional hardware complexity as the SRAM word size is increased. The graph in Figure 2-21 represents this tradeoff.

![Figure 2-21. Tradeoff graph](image)

**2.4.6 Performance Analysis**

PETCAM takes one clock cycle for a TCAM lookup (see Section 2.4.5) and hence its packet forwarding performance is comparable to that of the existing schemes using TCAMs.

For comparing update performance, note that Liu’s scheme and EaseCAM use incremental updates, whereas PETCAM uses batch updates and the latter strategy is also apt for Lu and Sahni’s schemes in [21]. Incremental updates, in general, are more efficient than batch updates. However, incremental updates are complex and inefficient for schemes using prefix compaction. For example, it is likely that a prefix (to be deleted, say) may not be present in the TCAM at all since the prefix had been optimized away during the compaction process. To ensure correct prefix matching, TCAM lookup must be stalled while the updates are incorporated. This affects lookup performance. In
contrast, schemes that use batch updates do not need to stall lookup since updates are incorporated in parallel on a different copy of the forwarding table. As noted in Section 2.4.4, update time in PETCAMLite is shorter by orders of magnitude, so when faster updates are needed PETCAMLite should be used.

In this chapter, we have pointed out some of the shortcomings of the power reduction methods for TCAM lookup tables proposed in [20, 35]. By starting with an optimal prefix set for the given routing table prefix set, we can achieve much better power reduction and TCAM memory requirement than when we use the compaction schemes suggested in [20, 35]. This is true regardless of whether we use the EaseCAM [35] architecture or the architecture of [21]. Compared to EaseCAM, the worst case power reduced between 87% and 98% while TCAM memory is reduced between 62% and 73%. The power and memory reduction relative to the architecture of [21] are 8% to 20% and 45% to 78% respectively. Although there is an increase in SRAM usage compared to EaseCAM, the reduction in TCAM usage has greater impact on area and power characteristics. PETCAM, just as the schemes in [21], requires additional hardware to extract the most appropriate next hop from a wide SRAM word. We have proposed two memory and power efficient TCAM lookup systems – PETCAM and PETCAMLite. While PETCAM has slightly better memory and power characteristics than does PETCAMLite, the rebuild time for PETCAM is two orders-of-magnitude larger than that for PETCAMLite. PETCAMLite supports acceptable rebuild times using modest computational resources. On our data sets, the power and memory penalty using PETCAMLite are at most 6% and at most 2%, respectively.

In the next chapter we will present an architecture that lets us develop incremental updates while many of the power saving strategies discussed in this chapter are still applicable.
CHAPTER 3
DUO: DUAL TCAM ARCHITECTURE FOR FORWARDING TABLES WITH INCREMENTAL UPDATE

In this chapter, we present three versions of a TCAM architecture for packet forwarding tables along with advanced memory management schemes for performing efficient and consistent incremental updates are presented. The first version of the architecture is DUOS – dual TCAM with simple SRAM, where both the TCAMs have a simple associated SRAM that is used for storing next hops. The second version of the architecture is DUOW – dual TCAM with wide SRAM, where one or both the TCAMs have wide associated SRAMs that are used to store suffixes as well as next hops. The third version is IDUOW – indexed dual TCAM with wide SRAM, in which either or both TCAMs have an associated index TCAM. The advantages of the dual TCAM architecture and the memory management schemes presented in this chapter are:

1. Support for incremental updates that require very few rule moves.
2. Reduced usage of TCAM memory and power.

The chapter is organized as follows. Section 3.1 presents related research work. The DUOS architecture and our memory management schemes are described in Section 3.2, DUOW is described in Section 3.3, and IDUOW is described in Section 3.4. An experimental evaluation of DUO is presented in Section 3.5, and we conclude in Section ??

3.1 Background and Related Work

Since our focus in this chapter is to develop router architectures that have efficient support for incremental updates, we present work related to TCAM incremental updates in some detail.

Shah and Gupta [41] describe incremental update algorithms for TCAMs using two different strategies to place prefixes in the TCAM. In PLO_OPT, the prefixes are placed in the TCAM in decreasing order of length. Unused TCAM slots/words are in the middle of the TCAM. So, prefixes of length $W$, $\cdots$, $W/2 + 1$ are above the free slots and the
remaining prefixes are below the free slots, where $W = 32$ for IPv4. An insert or delete requires at most $W/2$ prefix moves in PLO_OPT. In CAO_OPT, the prefixes are placed in the TCAM so that if two prefixes are nested, the longer prefix precedes the shorter one. If we start with the binary trie representation of the prefixes of the routing table, the prefixes along any path from the trie root to a trie leaf are nested. So, every root to leaf path in the trie defines a chain of nested prefixes. In CAO_OPT, the prefixes on every chain appear in reverse order in the TCAM. This placement ensures that the first prefix in the TCAM that matches a destination address is the longest matching prefix. The TCAM free slots are in the middle of the TCAM. If the maximum number of prefixes in a nested chain is $q$, then at most $\lceil q/2 \rceil$ prefixes of a chain are above the free slots. An insert or delete in CAO_OPT requires at most $\lceil q/2 \rceil = W/2$ moves. Since $q$ is about 6 in practical routing tables, CAO_OPT gives a performance improvement over PLO_OPT in practice (though the worst-case performance of both is the same).

Wang et al. [54] define a consistent rule table to be a rule table in which the rule matched (including the action associated with the rule) by a look up operation performed in the data plane is either the rule (including action) that would be matched just before or just after any ongoing update operation in the control plane. Wang et al. [54] develop a scheme for consistent table update without locking the TCAM at any time, essentially allowing a search to proceed while the table is being updated. Consistency is ensured by avoiding overwriting of a TCAM entry. Their CoPTUA algorithm can be applied to the PLO_OPT and CAO_OPT schemes of [41] so that rule updates can be carried out without locking the table for data plane lookups under suitable assumptions for TCAM operation [54].

Wang and Tzeng [53] also propose a consistent TCAM scheme. Their scheme, MIPS, however delays data plane lookups that match TCAM slots whose next hop information is being updated. In MIPS, the TCAM stores a set of independent prefixes (i.e., disjoint). This set of independent prefixes is obtained from the original set of
prefixes by using the leaf pushing technique [45] followed by a compression step. Since the prefixes in the TCAM are independent, at most one prefix matches any given destination address. Hence, the independent prefixes may be placed in the TCAM in any order and we may dispense with the priority encoder logic of the TCAM, which results in a reduction in TCAM lookup latency by about 50% [3]. Further, a new prefix may be inserted into any free slot of the TCAM and an old prefix deleted by simply setting the associated slot's valid bit to 0. While the use of an independent prefix set simplifies table management, leaf pushing replicates a prefix many times. In the worst case, an insert or delete, requires changes to $\Omega(n)$ TCAM entries, where $n$ the number of independent prefixes in the TCAM (Figure 3-1). Furthermore, the number of independent prefixes that result from leaf pushing and compression can be quite large as, in the worst-case, the compression step may fail to do any reduction in the prefix set following leaf pushing. Experimental results presented in [53] suggest, however, that, on practical rule sets, leaf expansion and compression actually reduce the number of prefixes by 20% to 68% because of the prevalence of a large number of redundant prefixes in practical rule sets. Further, each update operation results in between one and two accesses to the TCAM on average. Wang and Zheng [53] do not use any memory management scheme to keep track of the free slots in the TCAM and instead rely on a TCAM search operation to find an empty slot when such a slot is needed. Since a TCAM cannot perform a data plane search concurrent with a control plane search, update operations delay data plane lookups. In practice, since the number of updates per second is quite small and since each routing table update results in only one or two TCAM update operations (on average) the delay caused by control plane lookups on data plane lookups is quite small. As TCAM lookups consume a significant amount of energy relative to that consumed by TCAM read/write operations, using lookups to locate free TCAM slots increases total energy consumption for updates significantly.
Zane et al. [55] propose an indexed TCAM scheme to reduce the total TCAM power used to search routing tables of a given size. The indexed TCAM schemes of [55], however, increase the total TCAM size needed relative to non-indexed TCAMs. Lu and Sahni [21] couple indexed TCAMs with wide SRAMs to reduce both power and TCAM memory by a significant amount. Although the strategies of [21] are power and memory efficient, they are not well suited to incremental update. Similarly the prefix compaction methods of [14, 20, 27], while resulting in power and memory reduction, do not lend themselves well to incremental update. Chang [9] proposes a TCAM partitioning and indexing scheme in which the TCAM index is stored in a pivot prefix SRAM and an index SRAM. In Chang’s scheme [9], the TCAM index is searched using a binary search that makes \( O(\log K) \) SRAM accesses to determine the TCAM bucket that is to be searched. On the other hand, the scheme of Zane et al. [55] stores its index in a TCAM enabling the determination of the bucket for further search by a query on the index TCAM. As a result, a lookup takes 2 TCAM searches when the scheme of [55] is used and take 1 TCAM search plus \( O(\log K) \) SRAM accesses when the scheme of [9] is used.

### 3.2 Simple Dual TCAM – DUOS

DUOS uses any reasonably efficient data structure to store the routing-table rules in the control plane. For example, a simple data structure such as a binary trie or 1-bit
trie stored in a 100ns DRAM, permits about 300K IPv4 lookups, inserts, and deletes per second. This performance is quite adequate for the anticipated tens of thousands of control plane operations. For concreteness, we assume that a binary trie is used, in the control plane, to store the routing-table rules. Additionally, DUOS uses two TCAMs each with an associated SRAM. The TCAMs are labeled ITCAM (Interior TCAM) and LTCAM (Leaf TCAM) in Figure 3-2. The associated SRAMs are similarly labeled. Prefixes stored in leaf (non leaf or interior) nodes of the control plane trie are stored also in the LTCAM (ITCAM) and their associated next hops are stored in the LSRAM (ISRAM). Since the LTCAM stores only leaf prefixes, the prefixes in the LTCAM are disjoint and at most one may match a given destination address. Consequently, the LTCAM prefixes, even though of varying length, may be stored in any order. Further, the LTCAM does not require a priority encoder and, as a result, the latency of an LTCAM search is up to 50% less than that of a search in a TCAM with a priority encoder [3]. A data plane lookup is performed by doing a search for the packet’s destination address in both ITCAM and LTCAM. The ITCAM search yields the next hop associated with the longest matching non-leaf prefix while the LTCAM search yields the next hop associated with at most one leaf prefix that matches the destination address. Additional logic shown in Figure 3-2 returns the next hop (if any) from the LTCAM search; the next hop from the ITCAM search is returned only if the LTCAM search found no match. Note that since the LTCAM has no priority encoder, its search completes sooner than that in the ITCAM. The combining logic of Figure 3-2 can take advantage of this and abort the ITCAM search whenever the LTCAM search is successful, thereby reducing average lookup time. The correctness of the lookup is readily established. Figure 3-3 shows a 4-prefix forwarding table together with its corresponding binary trie that is stored in the control plane as well as the content of the two TCAMs and the two SRAMs of DUOS.

Each node of the control plane trie has fields such as prefix, slot, nexthop and length in which the prefix (if any) stored at this node is recorded along with the ITCAM
or LTCAM slot in which the prefix is stored and the nexthop and length of the prefix.

Functions for basic operations on the control plane trie (hereinafter simply referred to as trie) are assumed (see Figure 3-4).

As the control plane will modify the ITCAM, LTCAM, ISRAM, and LSRAM while the data plane performs lookups, the TCAMs need to be dual ported. Specifically, we make the following assumptions:

1. Each TCAM has two ports, which can be used to simultaneously access the TCAM from the control plane and the data plane.

2. Each TCAM entry-slot is tagged with a valid bit, that is set to 1 if the content for the entry is valid, and to 0 otherwise. A TCAM lookup engages only those slots whose valid bit is 1. The TCAM slots engaged in a lookup are determined at the start of a lookup to be those slots whose valid bits are 1 at that time. Changing a valid bit from 1 to 0 during a data plane lookup does not disengage that slot from the ongoing lookup. Similarly, changing a valid bit from 0 to 1 during a data plane lookup does not engage that slot until the next lookup.
Figure 3-3. DUOS for an example 4-prefix forwarding table. Note that prefixes in ITCAM are stored in length order, whereas those in LTCAM are stored arbitrarily since the prefixes are disjoint.

Function: Trie.insert
(a, b) = Trie.insert(prefix, length, nextHop);
This function inserts a prefix given its length and next hop into the control-plane binary trie. It returns the trie node a which stores the new prefix and a's nearest ancestor node b that contains a prefix.

Function: Trie.delete
(a, b) = Trie.delete(prefix, length);
This function deletes a prefix from the control plane trie and returns the trie node a that used to store the prefix just deleted and a's nearest ancestor node b that contains a prefix.

Function: Trie.change
a = Trie.change(prefix, length, newHop);
This function changes the next hop associated with a prefix and returns the trie node a that contains the prefix.

Figure 3-4. Table of control-plane trie functions

We assume the availability of the function waitWriteValidate which writes to a TCAM slot and sets the valid bit to 1. In case the TCAM slot being written to is the subject of ongoing data plane lookup, the write is delayed till this lookup completes. During the write, the TCAM slot being written to is excluded from data plane lookups\(^1\). This is

---

\(^1\) A possible mechanism to accomplish this exclusion is to set the valid bit to 0 before commencing the write and to change this bit to 1 when the write completes.
equivalent to the requirement that “After a rule is matched, resetting the valid bit has no effect on the action return process” [54], and to setting the valid entry to “hit” [53]. Similarly, we assume the availability of the function invalidateWaitWrite, which sets the valid bit of a TCAM slot to 0 and then writes an address to the associated SRAM word in such a way that the outcome of the ongoing lookup is unaffected.

We note that waitWriteValidate may, at times, write the prefix and nexthop information in the TCAM and associated SRAM slot and validate it, without any wait. This happens, for example, when the writing is to be done to a TCAM slot that is not the subject of the ongoing data plane lookup. The wait component of the function waitWriteValidate is said to be null in this case.

Figure 3-5 lists the various update algorithms we define later in this section for DUOS and its associated ITCAM and LTCAM. The indentation represents the hierarchy of function calls. A function at one level of indentation calls one or more functions below it at the next level of indentation or at the same level of indentation.

3.2.1 DUOS Incremental Update Algorithms

3.2.1.1 Insert

Figure 3-6 gives the algorithm to insert a new prefix $p$ of length $l$ and nexthop $h$. For simplicity, we assume that $p$ is, in fact new (i.e., $p$ is not already in the rule table). First, $p$ is inserted into the trie using the trie insertion algorithm, which returns nodes $m$ and $n$, where $m$ is the trie node storing $p$ and $n$ is the nearest ancestor (if any) of $m$ that has a prefix. When $m$ is a leaf of the trie, there is a possibility that the insertion of $p$ transformed a prefix that was previously a leaf prefix into a non-leaf prefix. If so, this prefix is moved from the LTCAM to the ITCAM. Regardless, $p$ is inserted into the LTCAM. When $m$ is not a leaf, $p$ is inserted into the ITCAM. Figure 3-7, 3-8 and 3-9 illustrate the insertion of rules P5, P6 and P7 respectively starting with the initial prefix trie in Figure 3-3.
dual-TCAM:
insert
delete
change

ITCAM (with simple SRAM):
insert
delete
change
getSpace
freeSpace
movesFromAbove
movesFromBelow
getFromAbove
getFromBelow

LTCAM (with simple SRAM)
insert
delete
change

LTCAM (with wide SRAM)
insert
delete
change

freeSpace
carve

generated by
addSuffix
split
dual
Figure 3-5. Table of functions used for incremental update

Algorithm: insert \((p, l, h)\)
\((m, n) = \text{Trie.insert}(p, l, h)\)
if \(m\) is a leaf then begin
if \(n\) exists and \(n\rightarrow\text{prefix}\) was a leaf prefix then
\(slot = \text{ITCAM.insert}(n\rightarrow\text{prefix}, n\rightarrow\text{nexthop}, n\rightarrow\text{length});\) // \(n\rightarrow\text{prefix}\) is no longer a leaf
\(\text{LTCAM.delete}(n\rightarrow slot);\)
\(n\rightarrow slot = slot;\)
endif
\(m\rightarrow slot = \text{LTCAM.insert}(p, h, l);\)
else \(m\rightarrow slot = \text{ITCAM.insert}(p, h, l);\)
endif

Figure 3-6. Algorithm to insert into DUOS
Figure 3-7. Insert rule P5 - \{1^*, H5\} to the initial table in Figure 3-3. P5 is a leaf and hence is added to the LTCAM.

Figure 3-8. Insert rule P6 - \{011^*, H6\} to the prefixes in Figure 3-7. P6 is added to the LTCAM, while P3, which is no longer a leaf, is deleted from LTCAM and added to ITCAM.

Figure 3-9. Insert rule P7 - \{0^*, H7\} to the prefixes in Figure 3-8. P7 is added to the ITCAM since it involves an intermediate prefix.
Algorithm: delete \((p, l)\)

\((m, n) = \text{Trie.delete}(p, l)\)

If \(m\) is a leaf then

\(\text{LTCAM.delete}(m \rightarrow \text{slot})\)

If \(n\) exists and \(n\) is now a leaf then

\(\text{slot} = \text{LTCAM.insert}(n \rightarrow \text{prefix}, n \rightarrow \text{nexthop}, n \rightarrow \text{length})\)

\(\text{ITCAM.delete}(n \rightarrow \text{slot}, n \rightarrow \text{length})\) // since \(n\) is now a leaf prefix

\(n \rightarrow \text{slot} = \text{slot}^\prime;\)

endif

else

\(\text{ITCAM.delete}(m \rightarrow \text{slot}, m \rightarrow \text{length})\)

endif

Figure 3-10. Algorithm to delete from DUOS

3.2.1.2 Delete

Figure 3-10 gives the algorithm to delete the prefix \(p\) from DUOS. For simplicity, we assume that \(p\) is, in fact, present in the rule table and so may be deleted. First, \(p\) is deleted from the trie. The trie deletion function returns nodes \(m\) and \(n\), where \(m\) is the trie node where \(p\) was stored and \(n\) is the nearest ancestor (if any) of \(m\) that has a prefix. If \(m\) was a leaf, then \(p\) is to be deleted from the LTCAM. In this case, the prefix (if any) in \(n\) may become a leaf prefix. If so, the prefix in \(n\) is to be moved from the ITCAM to the LTCAM. When \(m\) is not a leaf, \(p\) is deleted from the ITCAM. Figure 3-11, 3-12, 3-13 illustrate the delete procedure of prefixes P7, P4 and P5 respectively starting with the prefix trie in Figure 3-9.
Figure 3-11. Delete rule P7 - \{0*, H7\} from the prefixes in Figure 3-9. P7 is deleted from ITCAM.

Figure 3-12. Delete rule P4 - \{000*, H4\} from the prefixes in Figure 3-11. P4 is deleted from LTCAM. P2 is inserted to LTCAM and deleted from ITCAM as P2 is now a leaf.

Figure 3-13. Delete rule P5 - \{1*, H5\} from the prefixes in Figure 3-12. P5 is deleted from LTCAM.
Algorithm: change \((p, length, newH)\)

\[
m = \text{Trie.change}(p, l, newH)
\]

If \(m\) is a leaf then

\[
m\rightarrow\text{slot} = \text{LTCAM.change}(p, m\rightarrow\text{slot}, newH);
\]

else

\[
m\rightarrow\text{slot} = \text{ITCAM.change}(p, m\rightarrow\text{slot}, newH, length);
\]

Figure 3-14. Algorithm to change a next hop in DUOS

3.2.1.3 Change

To change the next hop of an existing prefix to \(newH\), we first change the next hop of the prefix in the trie and return the node \(m\) that contains \(p\). Then, depending on whether \(m\) is a leaf or non leaf, we invoke the change function for the corresponding TCAM. Figure 3-14 gives the algorithm.

3.2.2 ITCAM Algorithms

The prefixes in the ITCAM are stored in such a manner as to support determining the longest matching prefix (i.e., in any topological order that conforms to the precedence constraints defined by the binary trie—\(p_1\) must come before \(p_2\) whenever \(p_1\) is a descendent of \(p_2\) [41]). Decreasing order of length is a commonly used ordering.

The function \(\text{getSlot}(length)\) returns an ITCAM slot such that insertion of the new prefix into this slot satisfies the ordering constraint in use provided the new prefix has the specified length; the function \(\text{freeSlot}(slot, length)\) frees a slot previously occupied by a prefix of the specified length and makes this slot available for reuse later. These functions, which are described in Section 3.2.4, are used in our ITCAM insert, delete, and change algorithms (Figure 3-15), which are self explanatory.

Notice that following the first step of the change algorithm, the prefix whose next hop is being changed is in two valid slots of the ITCAM—\(oldSlot\) and \(slot\). This duplication does not affect correctness of data plane lookups as whichever one is matched by the ITCAM, we return the next hop that is valid either before or after the change operation. On the other hand, if we attempted to change the next hop in \(ISRAM[oldSlot]\) directly, an ongoing lookup may return a garbled next hop. Similarly,
Algorithm: insert(prefix, nexthop, length)
    slot = getSlot(length);
    ITCAM.waitWriteValidate(slot, prefix, nexthop);
    return slot;

Algorithm: delete(slot, length)
    freeSlot(slot, length);

Algorithm: change(prefix, oldSlot, nexthop, length)
    slot = insert(prefix, nexthop, length);
    delete(oldSlot, length);
    return slot;

Figure 3-15. ITCAM algorithms

if we delete first and then insert, lookups that take place between the delete and the
insert may return a next hop that doesn’t correspond to the routing table state either
before or after the change. If a waitWriteValidate is used to change ISRAM[oldSlot] to
nexthop, oldSlot becomes unavailable for data plane lookups during the write operation
and inconsistent results are returned in case the prefix in TCAM[oldSlot] is the longest
matching prefix.

3.2.3 LTCAM Algorithms

The prefixes in the LTCAM are disjoint and so may be stored in any order. The
unused (or free) slots of the LTCAM/LSRAM are linked together into a chain using the
words of the LSRAM to build this chain. We use AV to store the index of the first LSRAM
word on the chain. So, the free slots are AV, LSRAM[AV], LSRAM[LSRAM[AV]],
and so on. The last free slot on the AV chain has LSRAM[last] = −1. The LTCAM
algorithms to insert, delete, and change are given in Figure 3-16. These algorithms are
self explanatory.

3.2.4 ITCAM Memory Management

In this section, we describe four possible memory management schemes for
an ITCAM. The description of each memory management scheme includes an
implementation of the getSlot and freeSlot functions used in Section 3.2.2 to get
and free ITCAM slots. The implementations employ the function move (Figure 3-17)
Algorithm: insert(prefix, nexthop, length)
   if (AV == -1) throw NoSlotException;
   slot = AV;
   AV = LSRAM[slot];
   LTCAM.waitWriteValidate(slot, prefix, nexthop);
   return slot;

Algorithm: delete(slot)
   LTCAM.invalidateWaitWrite(slot, AV); // AV is stored in LSRAM[slot] after waiting
   // for an ongoing lookup to complete
   AV = slot;

Algorithm: change(prefix, oldSlot, nexthop, length)
   slot = insert(prefix, nexthop, length);
   delete (oldSlot);
   return slot;

Figure 3-16. LTCAM algorithms

Algorithm: move (src, dest)
   ITCAM.waitWriteValidate(dest, ITCAM[src], ISRAM[src]);

Figure 3-17. Move from ITCAM[src] to ITCAM[dest]

that moves the content of an in-use ITCAM slot to a free ITCAM slot in such a way as to
maintain data plane lookup consistency. Our memory management algorithms maintain
the invariant that an ITCAM slot has its valid bit set to 0 iff that slot wasn’t matched by
the ongoing data plane lookup (if any); that is, iff the slot isn’t involved in the ongoing
data plane lookup.

3.2.4.1 Memory management scheme 1

This scheme, which is the PLO_OPT scheme of [41], is shown in Figure 3-18(a),
the ITCAM slots are indexed 0 through N. The prefixes are stored in decreasing order
of length in the TCAM, which ensures that the longest matching prefix is returned as
the first matching prefix. The pool of free slots is kept at the logical center of the TCAM,
that is, the first free slot in the pool appears after all blocks of prefixes of length \( W/2 + 1 \)
or more and the last free slot appears before all blocks of prefixes of length \( W/2 \) or
less, where \( W \) is the width of the IP address (32 in the case of IPv4). As noted in [41],
this scheme requires at most \( W/2 \) moves for each getSlot and freeSlot request. Our
contribution is to provide an implementation that maintains consistency of data plane lookups.

Figure 3-18. Prefix arrangement in ITCAM for Scheme 1 for IPv4. The free space pool is indicated by hatched lines. Numbers 1, 2 by the curved arrow correspond to the first and second move, respectively.

Our lookup consistent implementation of \textit{getSlot} and \textit{freeSlot} employ the following variables:

\begin{itemize}
  \item \( W = \text{prefix length (32 for IPv4)} \)
  \item \( \text{top}[i] = \text{first slot used by block } i, \ 1 \leq i \leq W/2 \)
  \item \( \text{bot}[i] = \text{last slot used by block } i, \ W/2 + 1 \leq i \leq W \)
\end{itemize}

The following invariants are maintained:

\begin{itemize}
  \item \( \text{top}[i] = \text{top}[i-1] \) iff block \( i \) is empty, \( 1 \leq i \leq W/2 \)
  \item \( \text{bot}[i] = \text{bot}[i+1] \) iff block \( i \) is empty, \( W/2 + 1 \leq i \leq W \)
\end{itemize}

Initially, all blocks are empty and \( \text{top}[0 : W/2] = N+1 \) and \( \text{bot}[W/2 + 1 : W + 1] = -1 \) (recall that the ITCAM slots are indexed 0:N). Figures 3-19 and 3-20, respectively, give the \textit{getSlot} and \textit{freeSlot} algorithms for Scheme 1. Their correctness and the fact that data plane lookup consistency is preserved are easily established.
Algorithm: getSlot(len)
//len: length of prefix to be inserted.
// returns free slot for prefix insertion
if (bot[W/2 + 1] == top[W/2] - 1) throw NoSpaceException;
if (len ≥ W/2 + 1)
    d = ++bot[W/2 + 1];
    for (i = W/2 + 2; i ≤ len; ++i)
        if (bot[i] == d-1) // block i - 1 is empty
            bot[i] = bot[i - 1];
        else // move from top of i - 1 to d
            s = ++bot[i];
            move(s, d);
            d = s;
    endif
else
    d = --top[W/2];
    for (i = W/2 - 1; i ≥ len; --i)
        if (top[i] == d+1) // block i + 1 is empty
            top[i] = top[i + 1];
        else // move from bottom of i + 1 to d
            s = --top[i];
            move(s, d);
            d = s;
    endif
endif

Figure 3-19. Scheme 1 algorithm to get a free slot to insert a prefix whose length is len
Algorithm: freeSlot(slot, len)
// free ITCAM[slot] which had a prefix of length len
if (len \geq W/2 + 1) // free space from the top half.
    if (slot \neq bot[len])
        move (bot[len], slot); slot = bot[len];
    endif
    bot[len]--;
    for (i = len - 1; i > W/2; --i)
        if (bot[i] \neq slot) // block i is not empty
            move (bot[i], slot); slot = bot[i];
        endif
        bot[i]--;
    endfor
else // free space from the bottom half.
    if (slot \neq top[len])
        move (top[len], slot); slot = top[len];
    endif
    top[len]++;
    for (i=len+1; i\leq W/2; ++i)
        if (top[i] \neq slot) // block i is not empty
            move (top[i], slot); slot = top[i];
        endif
        top[i]++;
    endfor
    ITCAM[slot].valid = 0;
endif

Figure 3-20. Scheme 1 algorithm to free a slot previously occupied by a prefix of length len
### 3.2.4.2 Memory management scheme 2

This scheme is a variation of Scheme 1 in which the free slots are in the boundary between two prefix blocks (Figure 3-21). This scheme is also called DFS_PLO (Distributed Free Space with Prefix Length Ordering Constraint). At the time the ITCAM is initialized, the available free slots are distributed in proportion to the number of prefixes in a block with the caveat that an empty block gets 1 free slot at its boundary. In this scheme, top[$i$] is the slot where the first prefix of length $i$ is stored and bot[$i$] is the slot where the last prefix of length $i$ is stored, $0 \leq i \leq W$ (i.e., these variables define the start and end of block $i$). Note that top[$i$] $\leq$ bot[$i$] for a non-empty block $i$ and top[$i$] $>$ bot[$i$] for an empty block. For convenience, we define top[0]=bot[0]=N + 1 and top[$W + 1]=bot[W + 1] = -1. For an empty ITCAM, top[$i$] = N + 1 for $1 \leq i \leq W$; bot[$i$] = -1 for $1 \leq i \leq W$.

![Figure 3-21. ITCAM layout for Scheme 2 (DFS_PLO)](image)

Our getSlot algorithm (Figure 3-22) provides a free slot from either block boundary when there is a free slot on the block boundary. Otherwise, it moves a free slot from the nearest block boundary that has a free slot. The algorithm to free a slot (Figure 3-23) simply moves the slot to be freed to the block boundary unless this slot is at the
Algorithm: getSlot\((len)\)
  \[
  \begin{align*}
  ma &= \text{movesFromAbove}(len, \&aPos); \\
  mb &= \text{movesFromBelow}(len, \&bPos); \\
  &\text{if } (ma < mb) \\
  &\quad d = \text{getFromAbove}(len, aPos); \\
  &\quad \text{if } (\text{top}[len] > \text{bot}[len]) \text{ bot}[len] = d; \\
  &\quad \text{top}[len] = d; \\
  &\text{else} \\
  &\quad \text{if } (mb == W + 1) \text{ throw NoSpaceException; } \\
  &\quad d = \text{getFromBelow}(len, bPos); \\
  &\quad \text{if } (\text{top}[len] > \text{bot}[len]) \text{ top}[len] = d; \\
  &\quad \text{bot}[len] = d; \\
  &\text{endif} \\
  &\text{return } d;
  \end{align*}
\]

Figure 3-22. Scheme 2 algorithm to get a free slot to insert a prefix whose length is \(len\)

Algorithm: freeSlot\((d, len)\)
  \[
  \begin{align*}
  &\text{if } (\text{top}[len] == d) \text{ ITCAM[top[len]+].valid = 0;} \\
  &\text{else if } (\text{bot}[len] == d) \text{ ITCAM[bot[len]-].valid = 0;} \\
  &\text{else} \\
  &\quad \text{move(} \text{bot[len]}, d)\text{;} \\
  &\quad \text{ITCAM[bot[len]-].valid = 0; } \\
  &\text{endif}
  \end{align*}
\]

Figure 3-23. Scheme 2 algorithm to free a slot

boundary to begin with. Again, correctness and consistency are established easily.
Although the worst-case performance of the Scheme 2 algorithms is the same as
that of the Scheme 1 algorithms, we expect the Scheme 2 algorithms to have better
performance on average.

The getSlot algorithm utilizes several supporting algorithms that are given in
Figure 3-24. The algorithm movesFromAbove (movesFromBelow) returns the number of
prefix moves that are required to get the nearest free slot from above (below) the block
where it is needed and getFromAbove and getFromBelow, respectively, get the nearest
free slot above or below the block where the free slot is needed.
Algorithm: movesFromAbove\((len, *pos)\) // returns number of moves needed to acquire free space from above the block of length \(len\)
moves=0;
for (p=len; top[p] > bot[p]; p—); // find max \(p \leq len\) such that block \(p\) is not empty
for (c=len+1; c<=W + 1; c++) // find min \(c > len\) with space just below it
    if (top[c] \leq bot[c]) // not empty
        if (bot[c]+1 < top[p]) *pos = p; return moves; endif
        moves++; 
        p = c;
    endif
return \(W + 1\);
Algorithm: movesFromBelow\((len, *pos)\) // returns number of moves needed to acquire free space from below the block of length \(len\)
moves=0;
for (p=len; top[p] > bot[p]; p++); // find min \(p \geq len\) such that block \(p\) is not empty
for (c=len-1; c>=0; c—) // find min \(c > len\) with space just below it
    if (top[c] \leq bot[c]) // not empty
        if (top[c]-1 > bot[p]) *pos = p; return moves; endif
        moves++; 
        p = c;
    endif
return \(W + 1\);
Algorithm: getFromAbove\((len, pos)\) // get free space from above
\(d = top[\text{pos}]-1\);
for (c=pos; c> len; c—)
    if (top[c] \leq bot[c])
        \(d = bot[c]\
        move(bot[c]=--, --top[c]);
    endif
return \(d\);
Algorithm: getFromBelow\((len, pos)\) // get free space from below
\(d = bot[\text{pos}]+1\);
for (c=pos; c< len; c++)
    if (top[c] \leq bot[c])
        \(d = top[c]\
        move(top[c]=++, ++bot[c]);
    endif
return \(d\);

Figure 3-24. Supporting algorithms used by the algorithm of Figure 3-22
3.2.4.3 Memory management scheme 3

This is an enhancement of Scheme 2 in which we maintain a doubly-linked list of free slots within each block in addition to contiguous free slots at the block boundaries (Figure 3-25). This scheme is also called DLFS_PLO (Distributed and Linked Free Space with Prefix Length Ordering Constraint). The lists of free slots within a block enable us to avoid the move that is done by the Scheme 2 freeSlot algorithm of Figure 3-23. The forward links, called next[], of the doubly-linked list are maintained using the ISRAM words corresponding to the free ITCAM slots with AV[i] recording the first slot on the list for the i\(^{th}\) block. The backward links, called prev[], are maintained in these ISRAM words in case an ISRAM word is large enough to accommodate two links and in the control plane memory otherwise. All variables, including the array AV[], are, of course, stored in the control plane memory.

The scheme 3 getSlot algorithm (Figure 3-26) first attempts to make available a slot from the doubly-linked list for the desired block. When this list is empty, the algorithm behaves like the getSlot algorithm for Scheme 2 and the supporting algorithms of

The algorithm to free a slot (Figure 3-27) differs from that for Scheme 2 in that when the slot being freed is inside a block it is added to the doubly-linked list of free slots. Again, correctness and consistency are established easily. Although the worst-case performance of the Scheme 3 algorithms is the same as that of the algorithms for the first two schemes, we expect the Scheme 3 algorithms to have better performance on average.

Figures 3-28 and 3-29 are similar to the corresponding supporting algorithms for Scheme 3.
Figure 3-25. ITCAM layout for Scheme 3, with moves for insert and delete. The curved arrows on the right show the forward links in the list of free spaces.
Algorithm: **getSlot(len)**

```plaintext
aP = 0; bP = 0; aC = 0; bC = 0;
if (AV[len] == -1) // AV[len] stores the first free space in block of length len
    ma = movesFromAbove(len, &aP, &aC);
    mb = movesFromBelow(len, &bP, &bC);
    if (ma < mb)
        d = getFromAbove(len, aP, aC);
        if (top[len] > bot[len]) bot[len] = d;
        top[len] = d;
    else
        if (mb == W + 1) throw NoSpaceException; // no space
        d = getFromBelow(len, bP, bC);
        if (top[len] > bot[len]) top[len] = d;
        bot[len] = d;
    endif
else
    d = AV[len];
    AV[len] = next[d];
endif
return d;
```

Figure 3-26. Scheme 3 algorithm to get a free slot to insert a prefix whose length is `len`.

Algorithm: **freeSlot(d, len)**

```plaintext
if (top[len] == d) ITCAM[top[len]+].valid = 0;
else if (bot[len] == d) ITCAM[bot[len]-].valid = 0;
else
    ITCAM.invalidateWaitWrite(d, AV[len]); // AV[len] is stored in ISRAM[d].
    if (AV[len] != -1) prev[AV[len]] = d;
    AV[len] = d;
endif
```

Figure 3-27. Scheme 3 algorithm to free a slot.
Algorithm: movesFromAbove(len, *pos, *cur)
    moves=0;
    for (p=len; top[p] > bot[p]; p--); // find max p ≤ len such that block p is not empty
    for (c=len+1; c≤ W+1; c++) // find min c > len with space just below it
        if (top[c] ≤ bot[c]) // not empty
            if (bot[c]+1 < top[p] || !valid[bot[c]])
                *cur = c; *pos = p;
                return moves;
            endif
            moves++;
            if (AV[c] >= 0) *pos = c; *cur = c; return moves; endif
            p = c;
        endif
    return W + 1;

Algorithm: getFromAbove(len, p, c)
    if (top[p] > bot[c]+1) d = top[p]-1; c = p;
    else
        if (!valid[bot[c]])
            d = bot[c]--;
            if (d == AV[c]) AV[c] = next[AV[c]];
            else
                next[prev[d]] = next[d];
                if (next[d] != -1) prev[next[d]] = prev[d];
            endif
        else
            d = AV[c];
            AV[c] = next[AV[c]];
            move(bot[c], d);
            d = bot[c]--;
        endif
        c --;
    endif
    for (; c > len; c --)
        if (top[c] ≤ bot[c])
            move(bot[c]--, --top[c]);
            d = bot[c]+1;
        endif
    return d;

Figure 3-28. Algorithms to bring in a slot to a target block from above (lower addresses of the TCAM). These algorithms are used by getSlot of Figure 3-26
Algorithm: movesFromBelow\((len, *pos, *cur)\)

\[
moves=0;
\]
for (\(p=len; top[p] > bot[p]; p++\)); // find min \(p \geq len\) such that block \(p\) is not empty

\[
\text{for } (c=len-1; c>=0; c--) // find min \(c > len\) with space just below it
\]
if (top[c] \(\leq\) bot[c]) // not empty
  if (top[c]-1 > bot[p] || !valid[top[c]])
    \(*pos = p; *cur = c;
\)
  return moves;
  moves++;
  if (AV[c] \(\geq\) 0) \(*pos = c; *cur = c;\) return moves; endif
  p = c;
endif

return \(W + 1\);

Algorithm: getFromBelow\((len, p, c)\)

if (top[c]-1 > bot[p]) \(d = bot[p]+1; c = p;\)
else
  if (!valid[top[c]])
    \(d = top[c]+1;\)
  if (d == AV[c]) AV[c] = next[AV[c]];
  else
    next[prev[d]] = next[d];
    if (next[d] != -1) prev[next[d]] = prev[d];
  endif
else
  \(d = AV[c];\)
  AV[c] = next[AV[c]];
  move(top[c], d);
  \(d = top[c]+1;\)
endif
\(c++;\)
endif
for (; c < len; c++)
  if (top[c] \(\leq\) bot[c])
    move\((top[c]+1, ++bot[c])\);
  \(d = top[c]-1;\)
  endif
return \(d;\)

Figure 3-29. Algorithms to bring in a slot to a target block from below (higher addresses of the TCAM). These algorithms are used by getSlot of Figure 3-26
3.2.4.4 Scheme 4

This is the CAO_OPT scheme presented in [41]. Here, prefixes are arranged in chain order, with the free space pool in the middle of the ITCAM. Figures 3-30–3-32 give the necessary algorithms. The interfaces are different from those used by the first 3 schemes. The input to getSlot is \( p \), which is the node in the trie where the prefix being inserted is stored. Each trie node stores \( wt, wt\_ptr, hchild\_ptr, lchild, rchild \), which are explained in [41]. In addition to these we use the following variables:

- \( slot \): address of ITCAM slot in which prefix is entered. If prefix has not yet been entered, then this variable is set to -1.
- \( firstFree \): first free space
- \( lastFree \): last free space
- \( shift[0:W/2] \): temporary array of nodes

Also needed is an array of nodes, say \( nodeMap[0:N] \) for ITCAM[0:N] that contains the node address of each valid prefix in the ITCAM, so that they can be located in the trie.
Algorithm: getSlot(p)

d = isTopHeavy(p) ? lastFree : firstFree++;
if (parent(p) and p→parent(p)→slot < firstFree) // Case I: Insert prefix on top.
    c = parent(p); i=0;
    while (c and c→slot < firstFree)
        shift[i++] = c;
        c = parent(c);
    endwhile
    for (j=i-1; j>=0; –j)
        tmp = shift[j]→slot;
        move(shift[j]→slot, d);
        d = tmp;
    endfor
else if (child(p) and child(p)→slot > lastFree) // Case II: Insert prefix on bottom.
    c = child(p); i=0;
    while (c and c→slot > lastFree)
        shift[i++] = c;
        c = child(p);
    endwhile
    for (j=i-1; j>=0; –j)
        tmp = shift[j]→slot;
        move(shift[j]→slot, d);
        d = tmp;
endfor
endif
return d;

Figure 3-30. Scheme 4 getSlot algorithm
Algorithm: freeSlot(d)
if (d < firstFree)
c = nodeMap[firstFree-1]; i=0;
while (c and c->slot > d)
    shift[i++] = c;
c = child(c);
endwhile
for (j=i-1; j>=0; –j)
    tmp = shift[j]->slot;
    move(shift[j]->slot, d);
    d = tmp;
endfor
firstFree --;
else
    c = nodeMap[lastFree+1]; i=0;
while (c and c->slot < d)
    shift[i++] = c;
c = parent(c);
endwhile
for (j=i-1; j>=0; –j)
    tmp = shift[j]->slot;
    move(shift[j]->slot, d);
    d = tmp;
endfor
lastFree++;
endif
ITCAM[d].valid = 0;

Figure 3-31. Scheme 4 freeSlot algorithm
Algorithm: `isTopHeavy(p)`

```plaintext
top=bot=0;
for (c = parent(p); c != NULL; c = parent(c))
    if (c→slot > lastFree) bot++;
    else top++;
for (c = p→wt_ptr; c != NULL; c = c→wt_ptr)
    if (c stores a prefix)
        if (c→slot > -1) // prefix is already placed in TCAM
            if (c→slot > lastFree) bot++;
            else top++;
        else k++;
    endif
n = top+bot+k;
return (top > n/2) ? 1 : 0
```

Algorithm: `parent(p)`
```
c = p→parentNode;
while (c and !c→valid) c = c→parentNode;
return c;
```

Algorithm: `child(p)`
```
c = NULL;
// don’t return a LTCAM prefix as child.
if (p→hcld_ptr and p→hcld_ptr→tcam == 1) c = p→hcld_ptr;
return c;
```

**Figure 3-32.** Supporting control plane trie algorithms used by the Scheme 4 `getSlot` and `freeSlot` algorithms
3.3 Wide Dual TCAM–DUOW

In this section we extend our DUOS scheme to the case when wide SRAMs (say, 144-bit words or larger) are in use. We describe the extension only for the case when the LSRAM is wide. The case when the ISRAM is wide uses techniques almost identical to those used in [21] while for a wide LSRAM, we need to modify these techniques. As in [21], a wide LSRAM word is used to store a subtree of the binary trie of a forwarding table. However, instead of beginning with the binary trie for all prefixes as is done in [21], we begin with the binary trie, leaf trie, for only the leaf prefixes. When a subtree of the leaf trie is stored in an LSRAM word, that subtree is removed from (or carved out of) the leaf trie before another subtree is identified for carving. Let $N$ be the root of the subtree being carved and let $Q(N)$ be the prefix defined by the path from the root of the trie to $N$. $Q(N)$ is stored in the LTCAM, and $|P_i| - |Q(N)|$ suffix bits, of each prefix $P_i$ in the carved subtree rooted at $N$, are stored in the LSRAM word. Note that each suffix stored in the LSRAM word is a suffix of a leaf prefix that begins with $Q(N)$. By repeating this carving process, all leaf prefixes are allocated to the LTCAM and LSRAM. To obtain the mapping of leaf prefixes to the LTCAM and LSRAM, we need a carving algorithm that ensures that the $Q(N)$s stored in the LTCAM are disjoint. Since the carving algorithm of [21] does not ensure disjointedness, a new carving algorithm is needed. As an example, consider the binary trie of Figure 3-33(a), which has been carved using a carving algorithm that ensures that each carved subtree has at most 2 leaf prefixes. The LTCAM will need to store $Q(N1)$, $Q(N2)$ and $Q(N3)$. Even though the prefixes in the binary trie are disjoint, the $Q(N)$s in the LTCAM are not disjoint (e.g., $Q(N1)$ is a descendant of $Q(N2)$ and so $Q(N2)$ matches all IP addresses matched by $Q(N1)$). To retain much of the simplicity of the LTCAM management scheme of DUOS it is necessary to carve the leaf trie in such a way that all $Q(N)$s in the LTCAM are disjoint.

As in [21], we carve via a postorder traversal of the binary trie. However, we use the visit algorithm of Figure 3-35 to do the carving. In this algorithm, $w$ is the number
of bits in an LSRAM word and $x$-size is the number of bits needed to store (1) the suffix bits corresponding to prefixes in the subtrie rooted at $x$, (2) the length of each suffix, (3) the next hop for each suffix, (4) the number of suffixes in the word, and (5) the length of $Q(x)$, which is the corresponding prefix stored in the LTCAM. Algorithm $splitNode(q)$ (not specified in this chapter) does the actual carving of the subtree rooted at node $q$. The basic idea in our carving algorithm is to forbid carving at two nodes that have an ancestor-descendent relationship. This ensures that the $Q(N)$s are disjoint.

Figure 3-33(b) shows the subtrees carved by our algorithm. As can be seen, $Q(N1)$, $Q(N2)$, $Q(N3)$ are disjoint. Although our carving algorithm generally results in more $Q(N)$s than when the carving algorithm of [21] is used, our carving algorithm allows us to retain the flexibility to store the $Q(N)$s in any order in the LTCAM as the $Q(N)$s are independent.

The LTCAM algorithms to insert, delete, change, and necessary support algorithms are given in Figures 3-36–3-41.

The function $carve$ is invoked by both the insert and delete algorithms under different contexts that we analyze below. When a prefix is deleted, the LSRAM word storing its suffix (corresponding to the LTCAM word for $Q(cNode)$) may have remaining suffixes that can be merged with another LSRAM word. This merge is accomplished...
by the carve function, by carving the trie at \( tNode \), which is the nearest ancestor with two children, of \( cNode \). Thus carve helps to reduce the LTCAM entries by one. When a prefix is inserted, it may be possible to add the suffix bits of the new prefix in the LSRAM word that corresponds to the LTCAM slot for \( Q(cNode) \). If there is no \( cNode \) in the path between the new prefix node and the root, then we try to carve at \( tNode \), which is the nearest degree 2 ancestor of the new prefix node, and therefore includes the new prefix along with other existing prefixes. So, in this case, using carve we prevent the addition of a new LTCAM entry for the new prefix.

Next we show that \( tNode \) is indeed an appropriate node to carve and the algorithm preserves the property of carving at only one node along any path from the root. \( tNode \) is carved only if the number of bits needed to store all suffixes in the subtree rooted at \( tNode \) is less than the size of an LSRAM word. In this case there is a single \( otherNode \) that is a descendant of \( tNode \) and for which \( Q(otherNode) \) is in the LTCAM. To see that there cannot be more than one \( otherNode \), suppose there are \( q \) such nodes with \( Q(q) \) in the LTCAM. All of these \( q \) nodes must be in the subtree of \( tNode \) that does not contain the target node, which is \( cNode \) for a delete and the new prefix node for an insert. This is because, if there was one carved node \( t \) among the \( q \) nodes in the subtree of \( cNode \), for a delete, then \( t \) must occur either in the path between \( cNode \) and \( tNode \), or as a descendant of \( cNode \), given that \( tNode \) is the nearest ancestor of \( cNode \) with two children. In either case, \( t \) violates the property of a single carving along any path from the root. Similarly for an insert, if there were a carved node \( t \) in the same subtree that contained the newly added prefix, then \( t \) would have served as the \( cNode \) and we would not have started the carve algorithm in the first place. Since all \( q \) nodes must appear in the same subtree rooted at either the left or right child of \( tNode \), and the sum of their sizes is small enough to fit in an LSRAM word, our carving algorithm would have carved that child of \( tNode \). Thus there is only one \( otherNode \). Since we delete \( Q(cNode) \) and
\( Q(\text{otherNode}) \) right after adding \( Q(\text{tNode}) \), the property of carving only once along any path is maintained.

Figure 3-34 shows a possible assignment of the 5-prefix example in Figure 3-7. The intermediate prefixes P1 and P2 are stored in the ITCAM, while the leaf prefixes P3, P4 and P5 are stored in the LTCAM using a wide LSRAM. The suffix nodes begin with the prefix length field of 2 bits in this example followed by the suffix count field of 2 bits. Next comes the (length, suffix, nexthop) triplet for each prefix encoded in the suffix node, the number of allocated bits being (2bits, 4 bits, 6 bits) respectively for the three fields in the triplet.

![Figure 3-34. Assignment of prefixes of Figure 3-7 to TCAMs in the dual TCAM architecture](image-url)
Algorithm: visit_postorder(x)
if (!x) return 0;
isSplit = visit_postorder(y);
isSplit | = visit_postorder(z);
if (isSplit || x->size > w) then
    splitNode(y);
splitNode(z); // where y and z are children of x
    return 1;
else if (x->size == w) then
    splitNode(x);
    return 1;
endif
return 0;

Figure 3-35. Algorithm to carve a leaf trie to obtain disjoint $Q(N)$s

Algorithm: insert(node, cNode, tNode)
// node: node in leaf trie for new prefix to be inserted.
// cNode: nearest carved ancestor in leaf trie of node (may be NULL).
// tNode: nearest degree 2 ancestor of node.
if (cNode)
d = cNode->slot;
addSuffix(d, cNode, node);
LTCAM.invalidateWaitWrite(d, AV);
AV = d;
else if (!carve(tNode, node)) // create new suffix node with 1 suffix
    d = AV;
    AV = next[d];
    LTCAM.waitWriteValidate(d, LTCAM[slot], LSRAM[slot] + suffix);
    node->slot = d;
endif

Figure 3-36. DUOW algorithm to insert a prefix into the LTCAM

Algorithm: addSuffix(slot, cNode, node)
if (suffix does not fit in LSRAM[slot]) // need another suffix node
    split(cNode);
cNode->slot = -1;
else // add suffix to LSRAM[slot]
d = AV;
    AV = next[d];
    LTCAM.waitWriteValidate(d, LTCAM[slot], LSRAM[slot] + suffix);
cNode->slot = d;
endif

Figure 3-37. Algorithm to add a suffix to a wide LSRAM word

96
Algorithm: deletePrefixes\((node)\)

- bucketIndex = node\(\rightarrow bIndex\);
- len = length of deleteList;
- for (i=0; i<len; ++i)
  - slot = deleteList\[i\];
  - DLTCAM.\(\text{invalidateWaitWrite}\)(slot, AV[bucketIndex]);
  - AV[bucketIndex] = slot;
  - incrementRoom(bucketIndex);
- endfor
- clear deleteList;

Figure 3-38. Delete prefixes

Algorithm: split \((cNode)\)

- if \((cNode\rightarrow y \text{ and } cNode\rightarrow z)\) // carve at children y & z of cNode
  - cNode\(\rightarrow y\rightarrow slot = AV;\)
  - cNode\(\rightarrow z\rightarrow slot = \text{next}[AV];\)
  - AV = next[next[AV]];
  - LTCAM.\(\text{waitWriteValidate}\)(cNode\(\rightarrow y\rightarrow slot, Q(cNode\rightarrow y), \text{suffixes}(cNode\rightarrow y));\)
  - LTCAM.\(\text{waitWriteValidate}\)(cNode\(\rightarrow z\rightarrow slot, Q(cNode\rightarrow z), \text{suffixes}(cNode\rightarrow z));\)
- else if (cNode\(\rightarrow y) \) split (cNode\(\rightarrow y));
- else split(cNode\(\rightarrow z));
- endif

Figure 3-39. Algorithm to split a wide LSRAM word into two
Algorithm: delete(node, cNode, tNode)
// node: node in leaf trie corresponding to the prefix to be deleted
// cNode: nearest carved ancestor of node cannot be NULL
// tNode: nearest degree 2 ancestor node of cNode.
oldSlot = cNode→slot;
p = number of suffixes in LSRAM[oldSlot];
if (p > 1 and carve(tNode, cNode)) // delete suffix from its suffix node
  d = AV;
  AV = next[d];
  LTCAM.waitWriteValidate(d, LTCAM[oldSlot], LSRAM[oldSlot] - suffix);
  cNode→slot = d;
else cNode→slot = -1;
endif
LTCAM.invalidateWaitWrite(oldSlot, AV);
AV = oldSlot;

Algorithm: carve(tNode, cNode)
if (!tNode) return 0;
if (suffixes(tNode) fit in a suffix node) // carve at tNode
  d = AV;
  AV = next[d];
  LTCAM.waitWriteValidate(d, Q(tNode), suffixes(tNode));
  tNode→slot = d;
  otherNode = the carvedNode in subtree rooted at tNode that is not cNode;
  LTCAM.invalidateWaitWrite(otherNode→slot, AV);
  AV = otherNode→slot;
  otherNode→slot = -1;
  return 1;
endif
return 0;

Figure 3-40. DUOW algorithm to delete a leaf prefix

Algorithm: change(prefix, cNode, nexthop)
oldSlot = cNode→slot;
d = AV;
AV = next[d];
newWord = LSRAM[oldSlot] with next hop for cNode→prefix set to nexthop;
LTCAM.waitWriteValidate(d, prefix, newWord);
cNode→slot = d;
LTCAM.invalidateWaitWrite(oldSlot, AV);
AV = oldSlot;

Figure 3-41. DUOW algorithm to change the next hop of a leaf prefix
3.4 Indexed DUOW–IDUOW

Zane et al. [55] introduced the concept of an indexed TCAM that reduces significantly the power consumed by a TCAM lookup. This concept was refined by Lu and Sahni [21] to reduce both the TCAM power and space requirements substantially. We show how to incorporate an index TCAM in conjunction with an LTCAM that uses a wide LSRAM (i.e., an index for the LTCAM of DUOW). Adding an index to the ITCAM of DUOW follows easily from [21]. When the LTCAM is indexed, we have two TCAMs replacing the LTCAM—a data TCAM referred to as DLTCAM and an index TCAM referred to as ILTCAM. The associated SRAMs are DLSRAM and ILSRAM.

We consider the two most effective index TCAM strategies of [21]—1-12Wc and M-12Wb. The former is best for power whereas the latter is the best overall scheme consuming least TCAM space and low power for lookups [21]. Both 1-12Wc and M-12Wb organize the DLTCAM into fixed size buckets that are indexed using the ILTCAM and ILSRAM, which also is a wide SRAM that stores suffixes and associated information.

3.4.1 Memory Management for DLTCAM and ILTCAM

Each DLTCAM bucket is assigned a unique number between 0 and $totalSlots/bucketSize$, where $totalSlots$ is the total number of DLTCAM slots. The unique number so assigned to a bucket is called its index. A bucket index is stored in the trie node (in field $bIndex$) that is carved and represents an index prefix enclosing the DLTCAM prefixes in the bucket. The free slots in a bucket are linked through the associated DLSRAM. The first several bits (32 should be enough) of a DLSRAM word store the address of the next free DLTCAM slot in the same bucket. The last free slot in a bucket stores -1 in bits 0-31 of the corresponding DLSRAM word. For each bucket we keep one free slot at all times. This free slot is used for consistent updates, to copy the new prefix before deleting the old one. The first free slot in a bucket is stored in an array $AV$ indexed by the bucket index. The array $AV$ is initialized and maintained in the control plane. A list of
free buckets is maintained in the DLSRAM using additional bits of each DLSRAM word (12 bits are sufficient when the number of buckets is at most 4096). The first available slot in a free bucket stores the bucket index of the next free bucket in the DLSRAM bits and so on. The free bucket chain is terminated by a –1 in the bits used to store the index of the next free bucket. The variable bucketAV keeps track of the first bucket on the free bucket chain. In our algorithms we use the array nextBucket to represent the forward links in the bucket list.

When the prefixes in an ILTCAM are disjoint, we may use the simple memory management scheme used for the LTCAM of DUOS and when these prefixes are not disjoint, they must be ordered and any of the memory management schemes proposed for the ITCAM of DUOS in Section 3.2.4 may be used.

The update algorithms (Figures 3-42–3-38) are almost identical for 1-12Wc and M-12Wb. We explain the differences in the next two subsections.
Algorithm: insert(node, cNode, tNode, isramNode, itcamNode, itNode)
// node: node in leaf trie for new prefix to be inserted.
// cNode: nearest carved ancestor in leaf trie of node (may be NULL).
// tNode: nearest degree 2 ancestor of node.
// isramNode: index node enclosing cNode
// itcamNode: node whose prefix exists in ILTCAM.
// itNode: nearest degree 2 ancestor of isramNode.

if (cNode)
    bucketIndex = isramNode→bIndex;
    d = cNode→slot;
    addSuffix(d, cNode, node, isramNode, itcamNode, itNode);
    DLTCAM.invalidateWaitWrite(d, AV[bucketIndex]);
    AV[bucketIndex] = d;
else if (!carve(tNode, node, bucketIndex)) // create new suffix node with 1 suffix
    if (isramNode) then
        bucketIndex = isramNode→bIndex;
        d = AV[bucketAV];
        AV[bucketAV] = next[d];
        if (AV[bucketAV] == -1) then
            splitBucket(isramNode, itcamNode, itNode);
            if (node→slot == -1) // slot has not been assigned in splitBucket
                N = descendant of isramNode pointing to a DTCAM bucket and enclosing node.
                newd = AV[N→bIndex];
                AV[N→bIndex] = next[newd];
                AV[bucketAV] = d;
                d = newd;
        endif
    endif
    if (node→slot == -1)
        DLTCAM.waitWriteValidate(d, Q(node), suffix);
        decrementRoom(bucketIndex);
    endif
else
    assignNewBucket(node);
    d = AV[node→bIndex];
    AV[node→bIndex] = next[d];
    DLTCAM.waitWriteValidate(d, Q(node), suffix);
    node→slot = d;
    ILTCAM.insert(node, NULL, NULL);
endif
endif

Figure 3-42. DLTCAM insert algorithm
Algorithm: addSuffix(slot, cNode, node, isramNode, itcamNode, itNode)
if (suffix does not fit in DLSRAM[slot]) // need another suffix node
  split(cNode, isramNode, itcamNode, itNode);
cNode→slot = -1;
else // add suffix to DLSRAM[slot]
  bucketIndex = isramNode→bIndex;
d = AV[bucketIndex];
  AV[bucketIndex] = next[d];
  DLTCAM.waitWriteValidate(d, DLTCAM[slot], DLSRAM[slot] + suffix);
cNode→slot = d;
endif

Figure 3-43. Add a suffix to a DLSRAM word
Algorithm: split \((cNode, isramNode, itcamNode, itNode)\)

if \((cNode\to y \text{ and } cNode\to z)\) // carve at children y & z of cNode
bucketIndex = \(isramNode\to bIndex\);
if (next[AV[bucketIndex]] == -1 || next[next[AV[bucketIndex]]] == -1)
    splitBucket(isramNode, itcamNode, itNode); // AV[bucketIndex] should get reset here
if (isramNode is not the same as cNode) then
    if (cNode\to y\to slot == -1) then
        N = descendant of isramNode pointing to a DTCAM bucket and enclosing node.
        cNode\to y\to slot = AV[N\to bIndex];
        cNode\to z\to slot = next[AV[N\to bIndex]];
        AV[N\to bIndex] = next[next[AV[N\to bIndex]]];
        decrementRoom(N\to bIndex, 2);
    endif
    incrementRoom(bucketIndex, 1);
else
    if (cNode\to y\to slot == -1)
        cNode\to y\to slot = AV[isramNode\to child[0]\to bIndex];
        AV[isramNode\to child[0]\to bIndex] = next[cNode\to y\to slot];
        decrementRoom(isramNode\to child[0]\to bIndex, 1);
    endif
    if (cNode\to z\to slot == -1)
        cNode\to z\to slot = AV[isramNode\to child[1]\to bIndex];
        AV[isramNode\to child[1]\to bIndex] = next[cNode\to z\to slot];
        decrementRoom(isramNode\to child[1]\to bIndex, 1);
    endif
    incrementRoom(bucketIndex, 1);
else
    cNode\to y\to slot = AV[bucketIndex];
    cNode\to z\to slot = next[AV[bucketIndex]];
    AV[bucketIndex] = next[next[AV[bucketIndex]]];
    decrementRoom(bucketIndex, 1);
endif
DLTCAM.waitWriteValidate(cNode\to y\to slot, Q(cNode\to y), suffixes(cNode\to y));
DLTCAM.waitWriteValidate(cNode\to z\to slot, Q(cNode\to z), suffixes(cNode\to z));
else if (cNode\to y) split (cNode\to y);
else split(cNode\to z);
endif

Figure 3-44. Split a DLSRAM word
Algorithm: delete(node, cNode, tNode, isramNode, itcamNode, itNode)
// node: node in leaf trie corresponding to the prefix to be deleted
// cNode: nearest carved ancestor of node, cannot be NULL
// tNode: nearest degree 2 ancestor node of cNode.
// isramNode: index node enclosing cNode
// itcamNode: node whose prefix exists in ILTCAM.
// itNode: nearest degree 2 ancestor of itcamNode.

bucketIndex = isramNode→bIndex;
oldSlot = cNode→slot;
p = number of suffixes in DLSRAM[oldSlot];
if (p > 1 and !carve(tNode, cNode, bucketIndex)) // delete suffix from its suffix node
  d = AV[bucketIndex];
  AV[bucketIndex] = next[d];
  DLSRAM[d] = DLSRAM[oldSlot] - suffix;
  DLTCAM[d].prefix = DLTCAM[oldSlot].prefix;
  DLTCAM[d].valid = 1;
  cNode→slot = d;
else
  cNode→slot = -1;
  if (isramNode→size == 0) then // bucket becomes empty
    deleteBucket(isramNode, itcamNode, itNode);
  endif
endif
DLCAM.invalidateWrite(oldSlot, AV[bucketIndex]);
AV[bucketIndex] = oldSlot;

Algorithm: carve(tNode, cNode, bucketIndex)
if (!tNode) return 0;
if (suffixes(tNode) fit in a suffix node) // carve at tNode
  d = AV;
  AV = next[d];
  DLSRAM[d] = suffixes(tNode);
  DLTCAM[d].prefix = Q(tNode);
  DLTCAM[d].valid = 1;
  tNode→slot = d;
  otherNode = the carvedNode in subtree rooted at tNode that is not cNode;
  DLTCAM.invalidateWrite(otherNode→slot, AV[bucketIndex]);
  AV[bucketIndex] = otherNode→slot;
  otherNode→slot = -1;
  return 1;
endif
return 0;

Figure 3-45. Delete a leaf prefix
Algorithm: change(prefix, cNode, nexthop, isramNode)
oldSlot = cNode→slot;
bucketIndex = isramNode→bIndex;
d = AV[bucketIndex];
AV[bucketIndex] = next[d];
newWord = DLSRAM[oldSlot] with next hop for cNode→prefix set to nexthop;
DLTCAM.waitWriteValidate(d, prefix, newWord);
cNode→slot = d;
DLTCAM.invalidateWaitWrite(oldSlot, AV[bucketIndex]);
AV[bucketIndex] = oldSlot;

Algorithm: deleteBucket(isramNode, itcamNode, itNode)
bucketIndex = isramNode→bIndex;
nextBucket[bucketIndex] = bucketAV;
bucketAV = bucketIndex;
isramNode→bIndex = -1;
ILTCAM.delete(isramNode, itcamNode, itNode);

Algorithm: splitBucket (isramNode, itcamNode, itNode)
if (isramNode→y and isramNode→z) // carve at children y & z of isramNode
   // We want to move the split child that contains fewer prefixes.
   if (isramNode→y contains fewer prefixes)
      isramNode→y→bIndex = isramNode→bIndex;
      assignNewBucket(isramNode→z);
      node = isramNode→z;
   else
      assignNewBucket(isramNode→y);
      isramNode→z→bIndex = isramNode→bIndex;
      node = isramNode→y;
   endif
   ILTCAM.insert(isramNode→y, itcamNode, itNode);
   ILTCAM.insert(isramNode→z, itcamNode, itNode);
   ILTCAM.delete(isramNode, itcamNode, itNode);
deletePrefixes(node);
else if (isramNode→y) splitBucket(isramNode→y, itcamNode, itNode);
else splitBucket(isramNode→z, itcamNode, itNode);
endif

Figure 3-46. Change the next hop of a leaf prefix
3.4.2 1-12Wc

This two-level TCAM organization in [21] employs wide SRAMs in association with both the data and index TCAMs as shown in the Figure 3-47. The strategy adopted in [21] to fill up the TCAMs and the SRAMs is summarized as follows. Firstly, suffix nodes are created for prefixes in the 1-bit trie, as described in Section 3.3, using Lu’s carving heuristic. Secondly, every \( Q(N) \) to be entered in the data TCAM, is treated as a prefix and the subtree split algorithm [21] is applied to carve index nodes in the trie. The carving is done so that the number of data TCAM prefixes enclosed by the node being carved, is less than or equal to the size \( b \) of a data TCAM bucket. A new bucket is assigned to every index node. An enclosed data TCAM prefix and the corresponding suffix node are entered in a new entry in the bucket. When an index node encloses fewer than \( b \) prefixes, the remaining entries in the bucket are padded with null prefixes. Finally, the index nodes are treated as prefixes, the algorithm to create suffix nodes is run on the trie containing only index prefixes. The newly carved index \( Q(N) \) prefixes and the corresponding suffix nodes are entered in the index TCAM and the associated wide SRAM respectively. Using this strategy, the bucket numbers corresponding to the suffixes in an index SRAM suffix node, happen to be consecutive. Hence, the index SRAM omits the bucket number for all suffixes except the starting suffix, as shown in the Figure 3-47.

![Figure 3-47. 1-12Wc configuration in [21]](image-url)
During incremental updates, if a bucket overflows then assigning a new bucket immediately next to the overflowing bucket may require a large number of moves. Hence, the suffix node format in IDUOW stores the bucket number for each suffix, which makes it possible to assign any empty bucket in case of an overflow. The suffix node format for the ILSRAM for 1-12Wc is shown in Figure 3-48. Also, in keeping with the main idea of storing independent prefixes in the LTCAM, the visit_postorder algorithm is used instead of the subtree split algorithm in [21] while filling out the TCAMs. The prefix assignment algorithm for 1-12Wc is given below.

1. Suffix nodes corresponding to prefixes in the forwarding table are created using the visit_postorder algorithm on the 1-bit leaf prefix trie as shown in Section 3.3.

2. Each $Q(N)$ prefix resulting from Step 1 is to be entered into DLTCAM and is marked as a DLTCAM prefix in the trie.

3. The visit_postorder algorithm is applied to carve the index prefix nodes. The symbols used in the visit_postorder algorithm have slightly different meaning now: $x \rightarrow \text{size}$ represents the number of DLTCAM prefixes enclosed by node $x$, and $w$ is $b - 1$, where $b$ is the size of a DLTCAM bucket with one free slot for consistent updates. As an index node is carved, the enclosed DLTCAM prefixes are entered in a new DLTCAM bucket, and the bucket index is stored in the trie node, corresponding to the index, in field $bIndex$.

4. Each $Q(N)$, for the index nodes carved in Step 3, is marked as an index prefix in the trie.

5. Suffix nodes are created for the index prefixes using the visit_postorder algorithm on the 1-bit trie containing the index prefixes. The $Q(N)$ prefixes corresponding to the carved nodes are entered in the ILTCAM. Suffixes for the index prefixes are
Algorithm: `assignNewBucket(node)`

```java
node->bIndex = bucketAV;
if (bucketAV == -1) throw NoBucketsException;
bucketAV = nextBucket[bucketAV];
```

Figure 3-49. Assign a new bucket in 1-12Wc

entered in ILSRAM along with their bucket indexes, in the ILSRAM suffix node format as shown in the Figure 3-48.

The functions `incrementRoom` and `decrementRoom` are not relevant for 1-12Wc and are null functions. The `assignNewBucket` function is outlined in Figure 3-49.

The 1-12Wc scheme loses space efficiency as we carve out independent index prefix nodes and use a single bucket to store the DLTCAM prefixes enclosed by a single index prefix. The M-12Wb scheme doesn’t have this deficiency as DLTCAM prefixes from index prefixes are stored in the same bucket.

### 3.4.3 M-12Wb

The characteristic of the many-1 schemes in [21] is that all DTCAM buckets, except the last one, can be completely filled. Thus multiple index nodes use the same bucket to store their enclosed data TCAM prefixes. The configuration for M-12Wb is shown in Figure 3-50. The algorithm for carving and prefix assignment follows:

![Figure 3-50. M-12Wb configuration in [21]](image)

1. **Step 1: [Seed the DLTCAM buckets]**
   Run `feasibleST2(T, b - 1)[n/(b - 1)]` times. \( b - 1 \), since one free slot is needed in a bucket for consistent updates.
   Each time call `splitNode` to carve the found `bestST` from `T` (thereby updating `T`) and pack `bestST` into a new DLTCAM bucket.
The function \textit{splitNode} adds one or more prefixes to the ILTCAM.

2. **Step 2: [Fill the buckets]**
   While there is a DLTCAM bucket that is not full and \( T \) is not empty, repeat Step 3.

3. **Step 3: [Add to a bucket]**
   Let \( B \) be the DLTCAM bucket with the fewest number of prefixes.
   Let \( s \) be the number of prefixes in \( B \).
   Run \( \text{feasibleST}^2(T, b - s) \).
   Using \textit{splitNode} carve the found \textit{bestST} from \( T \) (thereby updating \( T \)) and pack \textit{bestST} into \( B \).
   The function \textit{splitNode} adds one or more prefixes to the ILTCAM.

4. **Step 4: [Use additional buckets as needed]**
   While \( T \) is not empty, fill a new DLTCAM bucket by making repeated invocations of \( \text{feasibleST}^2(T, q) \), where \( q \) is the remaining capacity of the bucket.
   Add ILTCAM prefixes as needed.

There are three main differences between this algorithm and the PS2 algorithm in [21].

- The first difference is reflected in the \textit{visit}2 algorithm (invoked by \textit{feasibleST}2) in that covering prefixes are not stored in the TCAMs.
- The second difference is in supplying \( b - 1 \) as available space in an empty bucket of size \( b \), reserving one free slot for consistent updates.
- The third difference is in the use of carving function \textit{splitNode} which helps to create independent prefixes for IDUOW.

Apart from the data structures already defined for the two-level indexing schemes, the M-12Wb requires a doubly linked list of used buckets to keep track of the buckets and the available spaces in them. An instance of a class BList is maintained in the control plane which contains the doubly linked list of buckets as well as an array to get to the right bucket quickly using a bucket index. Each bucket in the list has fields \textit{room} to indicate available bucket slots and \textit{index} to indicate the index of the bucket. The room in a bucket decreases from \textit{head} to \textit{tail} of the list. BList uses function \textit{add} to add a new bucket to the list and the array and \textit{getBucket} to get the appropriate bucket based on bucket index.
**Algorithm: visit2(x)**

\[
d = \text{count}(x); // \text{return}\text{the number of DLTCAM prefixes.}
\]
if \((d \leq q\ \text{and}\ \ d > \text{bestCount})\)
   \[
   \text{bestST} = \text{ST}(x);
   \text{bestCount} = d;
   \]
endif
// check \(T - \text{ST}(x)\)
\[
d = \text{count}(\text{root}(T)) - \text{count}(x);
\]
if \((d \leq q\ \text{and}\ \ d > \text{bestCount})\)
   \[
   \text{bestST} = T - \text{ST}(x);
   \text{bestCount} = d;
   \]
endif

Figure 3-51. Visit algorithm

**Algorithm: splitNode(\(N, NoN\))**

\(NoN\) is trie node \(x\) if \(\text{bestST} = T - \text{ST}(x)\), otherwise \(NoN\) is passed as NULL.
if \(! N \ || \ N == NoN\) return;
if \((N \rightarrow \text{istouched} == 0)\)
   \[
   N \rightarrow \text{istouched} = 1;
   \]
   \[N \rightarrow \text{bIndex} = \text{BList.head} \rightarrow \text{index};\]
   fill bucket with DLTCAM prefixes in \(N\).
   Let \(s\) = number of DLTCAM prefixes in \(N\).
   \[
   \text{BList.head} \rightarrow \text{room} = \text{BList.head} \rightarrow \text{room} - s.
   \]
endif
splitNode\((N \rightarrow y)\);
splitNode\((N \rightarrow z)\);

Figure 3-52. Split a node

**Algorithm: assignNewBucket(\(node\))**

Let \(s\) = number of DLTCAM prefixes in \(node\).
if \((s < \text{BList.head} \rightarrow \text{room})\)
   \[
   \text{node} \rightarrow \text{bIndex} = \text{BList.head} \rightarrow \text{index};
   \]
   \[
   \text{BList.head} \rightarrow \text{room} -= s;
   \]
else if \((\text{bucketAV} > -1)\)
   \[
   \text{node} \rightarrow \text{bIndex} = \text{bucketAV};
   \]
   \[
   \text{BList.add}(\text{bucketSize, bucketAV});
   \]
   \[
   \text{BList.head} \rightarrow \text{room} -= s;
   \]
   \[
   \text{bucketAV} = \text{nextBucket[bucketAV]};
   \]
else
   if \((\text{BList.head} \rightarrow \text{room} == 1)\) throw NoSpaceException;
   Run step 3 in PS2 while \(node\) still has a prefix;
endif

Figure 3-53. Assign a new bucket
Algorithm: incrementRoom(bucketIndex)
    \[b = \text{BList.getBucket(bucketIndex)};\]
    \[b->room++;\]
    if \(b->\text{prev} \text{ and } b->\text{room} > b->\text{prev}->\text{room}\) then relocate \(b\) to restore order.

Algorithm: decrementRoom(bucketIndex)
    \[b = \text{getBucket(bucketIndex)};\]
    \[b->room--;\]
    if \(b->\text{next} \text{ and } b->\text{room} < b->\text{next}->\text{room}\) then relocate \(b\) to restore order.

Figure 3-54. Increment and decrement room
3.5 Experimental Results

We evaluated the performance of the different versions of DUO using 21 IPv4 routing tables and update sequences downloaded from [6] and [37]. Table 3-1 gives the characteristics of these datasets. The update sequences for the first 20 routing tables were captured from files storing update announcements from 12am on February 1, 2009 for the stated number of hours; the update sequence for the last routing table rrc00May20 was captured from files storing eight hours of activity starting from 12am on May 20, 2008. The columns labeled \#RawInserts, \#RawDeletes and \#RawChanges, respectively, give the number of insert, delete, and change next hop requests in the update sequences. Using consistent updates, a next hop change request is implemented (see Figure 3-15 for example) as an insert (of the prefix with the new next hop) followed by a delete (of the prefix with the old nexthop). Therefore, all results
henceforth are in terms of the effective inserts and deletes. Note that the number of effective inserts (#Inserts) and deletes (#Deletes) is given by the following equations.

\[
#\text{Inserts} = #\text{RawInserts} + #\text{RawChanges};
\]

\[
#\text{Deletes} = #\text{RawDeletes} + #\text{RawChanges};
\]

3.5.1 Evaluation of Memory Management Schemes

We first ran a set of experiments on the simple TCAM [21] to compare our memory management schemes—Schemes 1-4. The simple TCAM we instantiated for the experiments has 300,000 slots. Tables 3-2 and 3-3, respectively, give the total and average number of prefix moves (i.e., number of invocations of \textit{move()}) required for an insert (includes raw inserts change next hop inserts) and a delete in our test update sequences (the data in Table 3-3 is obtained from that in Table 3-2 by dividing by #Inserts or #Deletes). Note that the theoretical worst-case number of moves for an insert/delete in IPv4 for the four memory management schemes is, respectively, 16, 32, 32 and 16. Tables 3-4 and 3-5, respectively, give the maximum number of moves per insert/delete and the standard deviation. From our experiments, we make the following observations:

1. Scheme 1 (PLO\_OPT) required the maximum number of moves (sum of moves for inserts and deletes) for all our test sets and Scheme 3 required the least. In fact, the disparity among the 4 schemes is very significant with Scheme 3 requiring a total number of moves that is orders of magnitude less than that required by the remaining schemes. Schemes 2 is comparable to Scheme 4 and Scheme 1 requires 10 times (or more) as many moves as required by Schemes 2 and 4.

2. The number of moves due to inserts in Scheme 2 is lower than those in Scheme 4 (CAO\_OPT) by orders of magnitude. For some of our test sets, inserts required no moves when Scheme 2 was used.

3. The number of moves due to deletes in Scheme 2 is comparable to that in Scheme 4 (CAO\_OPT).
4. The number of moves due to inserts in Scheme 3 is lower than that in Scheme 4 (CAO_OPT) by orders of magnitude. For the inserts in some of our test sets, Scheme 3 required no moves at all.

5. The number of moves due to deletes is 0 in Scheme 3 because in this scheme the slot within a block, freed by a delete is simply appended to the free space list for the block.

6. The maximum number of moves per insert/delete about the same for Schemes 2, 3 and 4, and about half that for Scheme 1. We note that Scheme 4 has a better worst-case performance for inserts than Schemes 2 and 3 but is worse for deletes.

7. The standard deviation is very small for Schemes 2 and 3. The number of moves, needed for an insert operation using Scheme 3, has low average and standard deviation values. So, the number of TCAM moves for any insert operation is, with a good probability, very low as well when Scheme 3 is used. The number of moves, needed for a delete operation using Scheme 3, has zero average and standard deviation values since Scheme 3 does not involve any move for a delete operation.

We also note that for Schemes 2 and 4, the number of moves due to deletes is much more than that due to inserts. For Scheme 4 this is because a delete rarely occurs adjacent to either of the two boundaries of the free space pool and non-boundary deletes require at least one move to shift the empty slot to the free space pool. However, since the prefix trie is shallow and the free space pool cuts each root to leaf path in the middle, many of the inserts in an update sequence are expected to occur at a boundary of the free space pool. So, inserts take much less than 1 move, on average, when Scheme 4 is used. Similarly, when Scheme 2 is used, most deletes are from within a block rather than at a block boundary. These non-boundary deletes require 1 move each. However, an insert requires no moves if there is a free slot at the top or bottom of its block, a likely occurrence.

Table 3-6 shows the number of \textit{waitWrites} (sum of invocations of \textit{waitWriteValidate()} and \textit{invalidateWaitWrite()}) which is the equal to the sum of inserts, deletes and moves for the simple TCAM and reflects the update performance for the four memory management schemes. As expected, Scheme 3 requires the least number of operations, due to the small number of moves. For Scheme 3, the average number of \textit{waitWrites} per
insert and delete (number of $\text{waitWrites}/(\#\text{Inserts} + \#\text{Deletes})$) ranged from a low of 1 for rrc01, rrc07, rrc16, route-views.linx to a high of 1.0072 for rrc03. Figure 3-55(a) shows the normalized average number of moves for each scheme on a logarithmic scale. For this figure, we computed the average number of moves per Insert/Delete for each data set. Then the average of these averages was computed and normalized by the average of averages for Scheme 3. Figure 3-55(b) shows the normalized average $\text{waitWrites}$ invoked by the different schemes. For this figure, we computed the average number of $\text{waitWrites}$ per Insert/Delete for each data set, then computed the average of these averages for each memory management scheme and finally normalized by the average of the averages for Scheme 3.

Table 3-2. Number of moves for the simple TCAM

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Scheme 1</th>
<th></th>
<th>Scheme 2</th>
<th></th>
<th>Scheme 3</th>
<th></th>
<th>Scheme 4</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>insert</td>
<td>delete</td>
<td>insert</td>
<td>delete</td>
<td>insert</td>
<td>delete</td>
<td>insert</td>
<td>delete</td>
</tr>
<tr>
<td>rrc00</td>
<td>2527899</td>
<td>2938315</td>
<td>395</td>
<td>404839</td>
<td>401</td>
<td>0</td>
<td>28621</td>
<td>405733</td>
</tr>
<tr>
<td>rrc01</td>
<td>3106800</td>
<td>3641827</td>
<td>0</td>
<td>531397</td>
<td>0</td>
<td>0</td>
<td>57619</td>
<td>529312</td>
</tr>
<tr>
<td>rrc03</td>
<td>1933994</td>
<td>2254451</td>
<td>4622</td>
<td>317445</td>
<td>4630</td>
<td>0</td>
<td>35144</td>
<td>321432</td>
</tr>
<tr>
<td>rrc04</td>
<td>1260636</td>
<td>1467502</td>
<td>0</td>
<td>208142</td>
<td>2</td>
<td>0</td>
<td>61507</td>
<td>221368</td>
</tr>
<tr>
<td>rrc05</td>
<td>2765874</td>
<td>3210264</td>
<td>543</td>
<td>455214</td>
<td>541</td>
<td>0</td>
<td>48677</td>
<td>461485</td>
</tr>
<tr>
<td>rrc06</td>
<td>2785323</td>
<td>3228450</td>
<td>8</td>
<td>435997</td>
<td>8</td>
<td>0</td>
<td>11452</td>
<td>439501</td>
</tr>
<tr>
<td>rrc07</td>
<td>973836</td>
<td>1153713</td>
<td>0</td>
<td>179987</td>
<td>0</td>
<td>0</td>
<td>35256</td>
<td>206623</td>
</tr>
<tr>
<td>rrc10</td>
<td>2090263</td>
<td>2444704</td>
<td>658</td>
<td>347529</td>
<td>671</td>
<td>0</td>
<td>30037</td>
<td>355326</td>
</tr>
<tr>
<td>rrc11</td>
<td>2100218</td>
<td>2449182</td>
<td>266</td>
<td>343898</td>
<td>245</td>
<td>0</td>
<td>17726</td>
<td>342096</td>
</tr>
<tr>
<td>rrc12</td>
<td>2657748</td>
<td>3101916</td>
<td>4665</td>
<td>438759</td>
<td>4659</td>
<td>0</td>
<td>44243</td>
<td>448979</td>
</tr>
<tr>
<td>rrc13</td>
<td>1784545</td>
<td>2090102</td>
<td>1035</td>
<td>304541</td>
<td>989</td>
<td>0</td>
<td>53433</td>
<td>560835</td>
</tr>
<tr>
<td>rrc14</td>
<td>1517650</td>
<td>1778785</td>
<td>4</td>
<td>255964</td>
<td>4</td>
<td>0</td>
<td>18265</td>
<td>255381</td>
</tr>
<tr>
<td>rrc15</td>
<td>1682880</td>
<td>1940303</td>
<td>2986</td>
<td>266885</td>
<td>2769</td>
<td>0</td>
<td>22314</td>
<td>286126</td>
</tr>
<tr>
<td>rrc16</td>
<td>71864</td>
<td>67329</td>
<td>0</td>
<td>9777</td>
<td>0</td>
<td>0</td>
<td>580</td>
<td>11075</td>
</tr>
<tr>
<td>route-views2</td>
<td>4140653</td>
<td>4844342</td>
<td>14</td>
<td>691240</td>
<td>14</td>
<td>0</td>
<td>92948</td>
<td>697924</td>
</tr>
<tr>
<td>route-views4</td>
<td>3584127</td>
<td>4177510</td>
<td>141</td>
<td>590235</td>
<td>141</td>
<td>0</td>
<td>39481</td>
<td>586756</td>
</tr>
<tr>
<td>route-views.eqix</td>
<td>1813054</td>
<td>2115841</td>
<td>33</td>
<td>300259</td>
<td>33</td>
<td>0</td>
<td>14235</td>
<td>301296</td>
</tr>
<tr>
<td>route-views.isc</td>
<td>2003320</td>
<td>2338493</td>
<td>12</td>
<td>331570</td>
<td>12</td>
<td>0</td>
<td>13537</td>
<td>326232</td>
</tr>
<tr>
<td>route-views.linx</td>
<td>2440442</td>
<td>2848138</td>
<td>0</td>
<td>404276</td>
<td>0</td>
<td>0</td>
<td>39136</td>
<td>403168</td>
</tr>
<tr>
<td>route-views.wide</td>
<td>2918481</td>
<td>3402684</td>
<td>1</td>
<td>462801</td>
<td>1</td>
<td>0</td>
<td>18559</td>
<td>466995</td>
</tr>
<tr>
<td>rrc00May20</td>
<td>311588</td>
<td>361323</td>
<td>19</td>
<td>50512</td>
<td>22</td>
<td>0</td>
<td>6380</td>
<td>50946</td>
</tr>
</tbody>
</table>
Table 3-3. Average number of moves for the simple TCAM

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Scheme 1</th>
<th></th>
<th>Scheme 2</th>
<th></th>
<th>Scheme 3</th>
<th></th>
<th>Scheme 4</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>insert</td>
<td>delete</td>
<td>insert</td>
<td>delete</td>
<td>insert</td>
<td>delete</td>
<td>insert</td>
<td>delete</td>
</tr>
<tr>
<td>rrc00</td>
<td>6.20</td>
<td>7.20</td>
<td>0.000969</td>
<td>0.9921</td>
<td>0.000984</td>
<td>0.0</td>
<td>0.0702</td>
<td>0.994</td>
</tr>
<tr>
<td>rrc01</td>
<td>5.8</td>
<td>6.8</td>
<td>0.0</td>
<td>0.9945</td>
<td>0.0</td>
<td>0.0</td>
<td>0.1078</td>
<td>0.9906</td>
</tr>
<tr>
<td>rrc03</td>
<td>6.04</td>
<td>7.04</td>
<td>0.0144</td>
<td>0.9909</td>
<td>0.0145</td>
<td>0.0</td>
<td>0.1098</td>
<td>1.0033</td>
</tr>
<tr>
<td>rrc04</td>
<td>6.01</td>
<td>7.01</td>
<td>0.0</td>
<td>0.9941</td>
<td>0.00001</td>
<td>0.0</td>
<td>0.293</td>
<td>1.0573</td>
</tr>
<tr>
<td>rrc05</td>
<td>6.01</td>
<td>7.01</td>
<td>0.00118</td>
<td>0.994</td>
<td>0.00118</td>
<td>0.0</td>
<td>0.1058</td>
<td>1.0078</td>
</tr>
<tr>
<td>rrc06</td>
<td>6.23</td>
<td>7.23</td>
<td>0.000018</td>
<td>0.976</td>
<td>0.00002</td>
<td>0.0</td>
<td>0.0256</td>
<td>0.9836</td>
</tr>
<tr>
<td>rrc07</td>
<td>5.41</td>
<td>6.41</td>
<td>0.0</td>
<td>0.9996</td>
<td>0.0</td>
<td>0.0</td>
<td>0.1958</td>
<td>1.1476</td>
</tr>
<tr>
<td>rrc10</td>
<td>6.00</td>
<td>7.00</td>
<td>0.00189</td>
<td>0.9952</td>
<td>0.00193</td>
<td>0.0</td>
<td>0.0862</td>
<td>1.0176</td>
</tr>
<tr>
<td>rrc11</td>
<td>6.02</td>
<td>7.01</td>
<td>0.000762</td>
<td>0.9854</td>
<td>0.0007</td>
<td>0.0</td>
<td>0.0508</td>
<td>0.9802</td>
</tr>
<tr>
<td>rrc12</td>
<td>5.99</td>
<td>6.99</td>
<td>0.0105</td>
<td>0.9881</td>
<td>0.0105</td>
<td>0.0</td>
<td>0.0997</td>
<td>1.0111</td>
</tr>
<tr>
<td>rrc13</td>
<td>5.78</td>
<td>6.77</td>
<td>0.00335</td>
<td>0.9874</td>
<td>0.0032</td>
<td>0.0</td>
<td>0.1731</td>
<td>1.0351</td>
</tr>
<tr>
<td>rrc14</td>
<td>5.82</td>
<td>6.82</td>
<td>0.000015</td>
<td>0.9816</td>
<td>0.000015</td>
<td>0.0</td>
<td>0.0701</td>
<td>0.979</td>
</tr>
<tr>
<td>rrc15</td>
<td>6.03</td>
<td>7.01</td>
<td>0.0107</td>
<td>0.963</td>
<td>0.0099</td>
<td>0.0</td>
<td>0.08</td>
<td>1.0333</td>
</tr>
<tr>
<td>rrc16</td>
<td>5.87</td>
<td>6.84</td>
<td>0.0</td>
<td>0.9943</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0474</td>
<td>1.1263</td>
</tr>
<tr>
<td>route-views2</td>
<td>5.97</td>
<td>6.97</td>
<td>0.00002</td>
<td>0.9951</td>
<td>0.00002</td>
<td>0.0</td>
<td>0.1341</td>
<td>1.0047</td>
</tr>
<tr>
<td>route-views4</td>
<td>6.01</td>
<td>7.01</td>
<td>0.000236</td>
<td>0.99</td>
<td>0.000236</td>
<td>0.0</td>
<td>0.0663</td>
<td>0.9843</td>
</tr>
<tr>
<td>route-views.eqix</td>
<td>5.94</td>
<td>6.94</td>
<td>0.000108</td>
<td>0.98523</td>
<td>0.000108</td>
<td>0.0</td>
<td>0.0467</td>
<td>0.9886</td>
</tr>
<tr>
<td>route-views.isc</td>
<td>5.95</td>
<td>6.94</td>
<td>0.000036</td>
<td>0.9846</td>
<td>0.000036</td>
<td>0.0</td>
<td>0.0402</td>
<td>0.969</td>
</tr>
<tr>
<td>route-views.linx</td>
<td>5.98</td>
<td>6.98</td>
<td>0.0</td>
<td>0.9914</td>
<td>0.0</td>
<td>0.0</td>
<td>0.096</td>
<td>0.9887</td>
</tr>
<tr>
<td>route-views.wide</td>
<td>6.15</td>
<td>7.15</td>
<td>0.000002</td>
<td>0.9725</td>
<td>0.000002</td>
<td>0.0</td>
<td>0.0392</td>
<td>0.9807</td>
</tr>
<tr>
<td>rrc00May20</td>
<td>6.12</td>
<td>7.10</td>
<td>0.00037</td>
<td>0.99308</td>
<td>0.000431</td>
<td>0.0</td>
<td>0.1253</td>
<td>1.0016</td>
</tr>
</tbody>
</table>

Figure 3-55. Comparison of performance between different memory management schemes
Table 3-4: Maximum number of moves for the simple TCAM

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Scheme 1</th>
<th>Scheme 2</th>
<th>Scheme 3</th>
<th>Scheme 4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>insert</td>
<td>delete</td>
<td>insert</td>
<td>delete</td>
</tr>
<tr>
<td>rrc00</td>
<td>15</td>
<td>16</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>rrc01</td>
<td>15</td>
<td>16</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>rrc03</td>
<td>14</td>
<td>15</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>rrc04</td>
<td>14</td>
<td>15</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>rrc05</td>
<td>14</td>
<td>15</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>rrc06</td>
<td>11</td>
<td>12</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>rrc07</td>
<td>11</td>
<td>12</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>rrc10</td>
<td>15</td>
<td>16</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>rrc11</td>
<td>14</td>
<td>15</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>rrc12</td>
<td>15</td>
<td>16</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>rrc13</td>
<td>15</td>
<td>16</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>rrc14</td>
<td>15</td>
<td>16</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>rrc15</td>
<td>14</td>
<td>15</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>rrc16</td>
<td>14</td>
<td>13</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>route-views2</td>
<td>15</td>
<td>16</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>route-views4</td>
<td>13</td>
<td>14</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>route-views.eqix</td>
<td>14</td>
<td>15</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>route-views.isc</td>
<td>15</td>
<td>16</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>route-views.linx</td>
<td>15</td>
<td>16</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>route-views.wide</td>
<td>14</td>
<td>14</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>rrc00May20</td>
<td>15</td>
<td>16</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

**Effect of TCAM Size on Memory Management Schemes**

The number of moves required by an update sequence is independent of the size of the TCAM (provided there are enough slots to accommodate all prefixes) when Schemes 1 and 4 are used. This, however, is not the case for Schemes 2 and 3.

Because of the relatively poor performance of Scheme 2 in our earlier test (Table 3-2), we did not study the impact of TCAM size on the number of moves using this scheme. Table 3-7 gives the number of moves required by the inserts (effective) in each of our test update sequences for varying TCAM size. The column labeled #Prefixes gives the initial number of prefixes in the routing table while that labeled #MaxPrefixes gives the maximum size attained by the routing table during the course of the update sequence. The TCAM occupancy is defined to be #MaxPrefixes/(TCAM size)*100%. For our experiment, we selected TCAM size so as to have occupancies of 80%, 90%, 95%,
Table 3-5. Standard deviation in number of moves for the simple TCAM

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Scheme 1</th>
<th>Scheme 2</th>
<th>Scheme 3</th>
<th>Scheme 4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>insert</td>
<td>delete</td>
<td>insert</td>
<td>delete</td>
</tr>
<tr>
<td>rrc00</td>
<td>1.582</td>
<td>1.578</td>
<td>0.064</td>
<td>0.088</td>
</tr>
<tr>
<td></td>
<td>0.064</td>
<td>0.0</td>
<td>0.276</td>
<td>0.196</td>
</tr>
<tr>
<td>rrc01</td>
<td>1.957</td>
<td>1.951</td>
<td>0.0</td>
<td>0.0735</td>
</tr>
<tr>
<td></td>
<td>0.0</td>
<td>0.0</td>
<td>0.314</td>
<td>0.212</td>
</tr>
<tr>
<td>rrc02</td>
<td>1.750</td>
<td>1.747</td>
<td>0.311</td>
<td>0.095</td>
</tr>
<tr>
<td></td>
<td>0.311</td>
<td>0.0</td>
<td>0.337</td>
<td>0.212</td>
</tr>
<tr>
<td>rrc03</td>
<td>2.434</td>
<td>2.421</td>
<td>0.076</td>
<td>0.003</td>
</tr>
<tr>
<td></td>
<td>0.0</td>
<td>0.0</td>
<td>0.554</td>
<td>0.276</td>
</tr>
<tr>
<td>rrc04</td>
<td>1.675</td>
<td>1.669</td>
<td>0.077</td>
<td>0.077</td>
</tr>
<tr>
<td></td>
<td>0.0</td>
<td>0.0</td>
<td>0.333</td>
<td>0.170</td>
</tr>
<tr>
<td>rrc05</td>
<td>1.457</td>
<td>1.453</td>
<td>0.004</td>
<td>0.154</td>
</tr>
<tr>
<td></td>
<td>0.004</td>
<td>0.0</td>
<td>0.163</td>
<td>0.274</td>
</tr>
<tr>
<td>rrc06</td>
<td>2.168</td>
<td>2.168</td>
<td>0.019</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>0.0</td>
<td>0.0</td>
<td>0.412</td>
<td>0.407</td>
</tr>
<tr>
<td>rrc07</td>
<td>1.888</td>
<td>1.882</td>
<td>0.067</td>
<td>0.069</td>
</tr>
<tr>
<td></td>
<td>0.069</td>
<td>0.0</td>
<td>0.288</td>
<td>0.246</td>
</tr>
<tr>
<td>rrc08</td>
<td>1.699</td>
<td>1.697</td>
<td>0.04</td>
<td>0.12</td>
</tr>
<tr>
<td></td>
<td>0.012</td>
<td>0.0</td>
<td>0.223</td>
<td>0.262</td>
</tr>
<tr>
<td>rrc09</td>
<td>1.896</td>
<td>1.89</td>
<td>0.27</td>
<td>0.1083</td>
</tr>
<tr>
<td></td>
<td>0.27</td>
<td>0.0</td>
<td>0.307</td>
<td>0.281</td>
</tr>
<tr>
<td>rrc10</td>
<td>2.144</td>
<td>2.1365</td>
<td>0.117</td>
<td>0.111</td>
</tr>
<tr>
<td></td>
<td>0.111</td>
<td>0.0</td>
<td>0.449</td>
<td>0.245</td>
</tr>
<tr>
<td>rrc11</td>
<td>1.834</td>
<td>1.833</td>
<td>0.005</td>
<td>0.134</td>
</tr>
<tr>
<td></td>
<td>0.005</td>
<td>0.0</td>
<td>0.286</td>
<td>0.287</td>
</tr>
<tr>
<td>rrc12</td>
<td>2.066</td>
<td>2.0428</td>
<td>0.216</td>
<td>0.186</td>
</tr>
<tr>
<td></td>
<td>0.216</td>
<td>0.0</td>
<td>0.279</td>
<td>0.310</td>
</tr>
<tr>
<td>rrc13</td>
<td>1.7722</td>
<td>1.8366</td>
<td>0.0</td>
<td>0.9943</td>
</tr>
<tr>
<td></td>
<td>0.0</td>
<td>0.0</td>
<td>0.215</td>
<td>0.406</td>
</tr>
<tr>
<td>route-views2</td>
<td>1.746</td>
<td>1.744</td>
<td>0.005</td>
<td>0.07</td>
</tr>
<tr>
<td></td>
<td>0.005</td>
<td>0.0</td>
<td>0.371</td>
<td>0.14</td>
</tr>
<tr>
<td>route-views4</td>
<td>1.756</td>
<td>1.754</td>
<td>0.034</td>
<td>0.098</td>
</tr>
<tr>
<td></td>
<td>0.034</td>
<td>0.0</td>
<td>0.251</td>
<td>0.218</td>
</tr>
<tr>
<td>route-views.eqix</td>
<td>1.722</td>
<td>1.720</td>
<td>0.0107</td>
<td>0.1206</td>
</tr>
<tr>
<td></td>
<td>0.0107</td>
<td>0.0</td>
<td>0.213</td>
<td>0.243</td>
</tr>
<tr>
<td>route-views.isc</td>
<td>1.691</td>
<td>1.686</td>
<td>0.006</td>
<td>0.1232</td>
</tr>
<tr>
<td></td>
<td>0.006</td>
<td>0.0</td>
<td>0.1992</td>
<td>0.2627</td>
</tr>
<tr>
<td>route-views.linx</td>
<td>1.933</td>
<td>1.928</td>
<td>0.0</td>
<td>0.092</td>
</tr>
<tr>
<td></td>
<td>0.0</td>
<td>0.0</td>
<td>0.306</td>
<td>0.212</td>
</tr>
<tr>
<td>route-views.wide</td>
<td>1.502</td>
<td>1.5</td>
<td>0.0015</td>
<td>0.164</td>
</tr>
<tr>
<td></td>
<td>0.0015</td>
<td>0.0</td>
<td>0.198</td>
<td>0.266</td>
</tr>
<tr>
<td>rrc00May20</td>
<td>1.7479</td>
<td>1.7286</td>
<td>0.0284</td>
<td>0.0829</td>
</tr>
<tr>
<td></td>
<td>0.0284</td>
<td>0.0</td>
<td>0.3417</td>
<td>0.228</td>
</tr>
</tbody>
</table>

97%, and 99%. As can be seen, even with an occupancy of 99%, our Scheme 3 does very well. In fact, its nearest competitor, Scheme 4 (CAO_OPT), requires between 72 and 241879 times as many moves (for inserts and deletes combined) as required by Scheme 3 (see Table 3-2 for the number of moves required by Scheme 4).

3.5.2 Evaluation of DUOS

In DUOS, each prefix in the forwarding table occupies a slot in either the ITCAM or the LTCAM. Columns 2 and 5 of Table 3-8 give the initial prefix distribution between the 2 TCAMs of DUOS. Columns 3 and 6 give the distribution of the inserts (i.e., number of non-leaf inserts and number of leaf inserts) while columns 4 and 7 give the distribution of the deletes. We note that a leaf insert/delete may trigger additional insert and/or delete operations on the TCAMS of DUOS. These additional inserts/deletes are accounted for in Table 3-8. As a result,
Table 3-6. Number of \textit{waitWrites} for the simple TCAM

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Scheme 1</th>
<th>Scheme 2</th>
<th>Scheme 3</th>
<th>Scheme 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>rrc00</td>
<td>6281844</td>
<td>1220864</td>
<td>816031</td>
<td>1249984</td>
</tr>
<tr>
<td>rrc01</td>
<td>7816937</td>
<td>1599707</td>
<td>1068310</td>
<td>1655241</td>
</tr>
<tr>
<td>rrc03</td>
<td>4828969</td>
<td>962591</td>
<td>645154</td>
<td>997100</td>
</tr>
<tr>
<td>rrc04</td>
<td>3146985</td>
<td>626989</td>
<td>418849</td>
<td>701722</td>
</tr>
<tr>
<td>rrc05</td>
<td>6893993</td>
<td>1373612</td>
<td>918396</td>
<td>1428017</td>
</tr>
<tr>
<td>rrc06</td>
<td>6907413</td>
<td>1329645</td>
<td>893648</td>
<td>1344593</td>
</tr>
<tr>
<td>rrc07</td>
<td>2487684</td>
<td>540122</td>
<td>360135</td>
<td>602014</td>
</tr>
<tr>
<td>rrc10</td>
<td>5232500</td>
<td>1045720</td>
<td>698204</td>
<td>1082896</td>
</tr>
<tr>
<td>rrc11</td>
<td>5247135</td>
<td>1041899</td>
<td>697980</td>
<td>1057557</td>
</tr>
<tr>
<td>rrc12</td>
<td>6647360</td>
<td>1331120</td>
<td>892355</td>
<td>1380918</td>
</tr>
<tr>
<td>rrc13</td>
<td>4491700</td>
<td>922629</td>
<td>618042</td>
<td>989752</td>
</tr>
<tr>
<td>rrc14</td>
<td>3817753</td>
<td>777286</td>
<td>521322</td>
<td>794964</td>
</tr>
<tr>
<td>rrc15</td>
<td>4178985</td>
<td>825673</td>
<td>558571</td>
<td>864242</td>
</tr>
<tr>
<td>rrc16</td>
<td>161260</td>
<td>31844</td>
<td>22067</td>
<td>33722</td>
</tr>
<tr>
<td>route-views2</td>
<td>10372629</td>
<td>2078888</td>
<td>1387648</td>
<td>2178506</td>
</tr>
<tr>
<td>route-views4</td>
<td>8953622</td>
<td>1782361</td>
<td>1192126</td>
<td>1818222</td>
</tr>
<tr>
<td>route-views.eqix</td>
<td>4538451</td>
<td>909848</td>
<td>609589</td>
<td>925087</td>
</tr>
<tr>
<td>route-views.isc</td>
<td>5015189</td>
<td>1004958</td>
<td>673388</td>
<td>1013145</td>
</tr>
<tr>
<td>route-views.linx</td>
<td>6103818</td>
<td>1219514</td>
<td>815238</td>
<td>1257542</td>
</tr>
<tr>
<td>route-views.wide</td>
<td>7270918</td>
<td>1412555</td>
<td>949754</td>
<td>1435007</td>
</tr>
<tr>
<td>rrc00May20</td>
<td>774709</td>
<td>152329</td>
<td>101820</td>
<td>159124</td>
</tr>
</tbody>
</table>

ITCAM.\#.inserts + LTCAM.\#.inserts ≥ \#Inserts \hspace{1cm} (3–3)

It is interesting to note that more than 90% of the prefixes in each data set are leaf prefixes and that more than 90% of the inserts and deletes in each update sequence are directed at the LTCAM.

Given the distribution of the prefixes and insert and delete operations, we instantiated an LTCAM with 300,000 slots and an ITCAM with 28,000 slots for our DUOS experiments. Since the performance of DUOS is determined by the number of \textit{waitWrite} operations, we measure this quantity for our datasets. In addition, since the number of moves directly impacts the number of \textit{waitWrite} operations, we measured the number of moves separately so to compare the effect of the four memory management schemes for ITCAM. Table 3-9 gives the number of ITCAM moves for inserts and deletes. The
Table 3-7. Number of Scheme 3 moves for inserts

<table>
<thead>
<tr>
<th>Dataset</th>
<th>#Prefixes</th>
<th>#MaxPrefixes</th>
<th>80%</th>
<th>90%</th>
<th>95%</th>
<th>97%</th>
<th>99%</th>
</tr>
</thead>
<tbody>
<tr>
<td>rrc00</td>
<td>294098</td>
<td>294318</td>
<td>0</td>
<td>0</td>
<td>96</td>
<td>227</td>
<td>554</td>
</tr>
<tr>
<td>rrc01</td>
<td>276795</td>
<td>277002</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>31</td>
<td>307</td>
</tr>
<tr>
<td>rrc03</td>
<td>283754</td>
<td>284464</td>
<td>3412</td>
<td>4311</td>
<td>4641</td>
<td>4774</td>
<td>4916</td>
</tr>
<tr>
<td>rrc04</td>
<td>288610</td>
<td>288915</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>5</td>
<td>1144</td>
</tr>
<tr>
<td>rrc05</td>
<td>280041</td>
<td>282223</td>
<td>180</td>
<td>383</td>
<td>584</td>
<td>696</td>
<td>810</td>
</tr>
<tr>
<td>rrc06</td>
<td>278744</td>
<td>279202</td>
<td>0</td>
<td>5</td>
<td>11</td>
<td>17</td>
<td>97</td>
</tr>
<tr>
<td>rrc07</td>
<td>275097</td>
<td>275130</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>rrc10</td>
<td>278898</td>
<td>280158</td>
<td>381</td>
<td>490</td>
<td>798</td>
<td>1044</td>
<td>1355</td>
</tr>
<tr>
<td>rrc11</td>
<td>277166</td>
<td>277391</td>
<td>39</td>
<td>168</td>
<td>362</td>
<td>514</td>
<td>668</td>
</tr>
<tr>
<td>rrc12</td>
<td>278499</td>
<td>279155</td>
<td>2588</td>
<td>4174</td>
<td>4966</td>
<td>5212</td>
<td>5448</td>
</tr>
<tr>
<td>rrc13</td>
<td>284986</td>
<td>285621</td>
<td>120</td>
<td>592</td>
<td>973</td>
<td>1107</td>
<td>1272</td>
</tr>
<tr>
<td>rrc14</td>
<td>276170</td>
<td>276385</td>
<td>0</td>
<td>2</td>
<td>14</td>
<td>51</td>
<td>146</td>
</tr>
<tr>
<td>rrc15</td>
<td>284047</td>
<td>286467</td>
<td>1652</td>
<td>2392</td>
<td>2736</td>
<td>2838</td>
<td>2982</td>
</tr>
<tr>
<td>rrc16</td>
<td>282660</td>
<td>285170</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>rviews2</td>
<td>294127</td>
<td>294598</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>28</td>
</tr>
<tr>
<td>rviews4</td>
<td>275737</td>
<td>276035</td>
<td>12</td>
<td>106</td>
<td>178</td>
<td>201</td>
<td>222</td>
</tr>
<tr>
<td>rviews.eqix</td>
<td>275736</td>
<td>276230</td>
<td>12</td>
<td>29</td>
<td>97</td>
<td>166</td>
<td>269</td>
</tr>
<tr>
<td>rviews.isc</td>
<td>281095</td>
<td>281430</td>
<td>0</td>
<td>5</td>
<td>14</td>
<td>20</td>
<td>131</td>
</tr>
<tr>
<td>rviews.linx</td>
<td>278196</td>
<td>278283</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>34</td>
<td>348</td>
</tr>
<tr>
<td>rviews.wide</td>
<td>283569</td>
<td>284569</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>19</td>
</tr>
<tr>
<td>rrc00May20</td>
<td>266185</td>
<td>267344</td>
<td>13</td>
<td>32</td>
<td>101</td>
<td>173</td>
<td>430</td>
</tr>
</tbody>
</table>

The number of moves shown in Table 3-9 includes the ITCAM moves resulting from ITCAM operations triggered by LTCAM inserts and deletes as well (for example, when inserting a leaf prefix, we insert into the LTCAM and delete its parent prefix (if any) from the LTCAM and reinsert this parent prefix into the ITCAM). The relative performance of the 4 memory management schemes for ITCAM is quite similar to that observed for a simple TCAM organization and Scheme 3 outperforms the remaining schemes handily. Table 3-10 shows the number of waitWrites generated in the ITCAM and we find that Scheme 3 is the best for this metric as expected from the smaller number of moves required by Scheme 3. Table 3-11 gives the number of LTCAM moves required by the
Table 3-8. Distribution of prefixes, inserts, and deletes for DUOS

<table>
<thead>
<tr>
<th>Dataset</th>
<th>ITCAM</th>
<th>LTCAM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#Prefixes</td>
<td>#inserts</td>
</tr>
<tr>
<td>rrco0</td>
<td>27381</td>
<td>27454</td>
</tr>
<tr>
<td>rrco1</td>
<td>24787</td>
<td>39796</td>
</tr>
<tr>
<td>rrco3</td>
<td>26116</td>
<td>30461</td>
</tr>
<tr>
<td>rrco4</td>
<td>25137</td>
<td>20549</td>
</tr>
<tr>
<td>rrco5</td>
<td>25375</td>
<td>36852</td>
</tr>
<tr>
<td>rrco6</td>
<td>25207</td>
<td>36518</td>
</tr>
<tr>
<td>rrco7</td>
<td>24441</td>
<td>15946</td>
</tr>
<tr>
<td>rrco10</td>
<td>24832</td>
<td>26364</td>
</tr>
<tr>
<td>rrco11</td>
<td>24787</td>
<td>29399</td>
</tr>
<tr>
<td>rrco12</td>
<td>24894</td>
<td>35725</td>
</tr>
<tr>
<td>rrco13</td>
<td>26320</td>
<td>33025</td>
</tr>
<tr>
<td>rrco14</td>
<td>24485</td>
<td>27455</td>
</tr>
<tr>
<td>rrco15</td>
<td>26184</td>
<td>26356</td>
</tr>
<tr>
<td>rrco16</td>
<td>25586</td>
<td>1368</td>
</tr>
<tr>
<td>route-views2</td>
<td>26883</td>
<td>64285</td>
</tr>
<tr>
<td>route-views4</td>
<td>24543</td>
<td>49773</td>
</tr>
<tr>
<td>route-views.eqix</td>
<td>24423</td>
<td>28188</td>
</tr>
<tr>
<td>route-views.isc</td>
<td>25459</td>
<td>36595</td>
</tr>
<tr>
<td>route-views.linx</td>
<td>25072</td>
<td>36247</td>
</tr>
<tr>
<td>route-views.wide</td>
<td>26410</td>
<td>37266</td>
</tr>
<tr>
<td>rrc00May20</td>
<td>24407</td>
<td>3738</td>
</tr>
</tbody>
</table>

As expected, the number of LTCAM moves is zero. The total number of moves for the simple TCAM is between 14-24 times that for DUOS using Scheme 1 (PLO_OPT), between 9-15 times using Scheme 2, 7-227 times using Scheme 3, and 9-16 times using Scheme 4 (CAO_OPT). Thus there is a reduction of more than 90% in the total number of moves for any scheme. This is due to the DUO architecture, as the reduction is observed for PLO_OPT and CAO_OPT also. Note that the number of \textit{wait\text{\textbar}writes} in an LTCAM equals the number of inserts and deletes on the LTCAM and \textit{wait\text{\textbar}write\text{\textcharp}validates} in an LTCAM, have null wait as no invalid slot is involved in an ongoing lookup. This is ensured by using \textit{invalidate\text{\textbar}wait\text{\textbar}write} to free a slot. Note that

\footnote{Recall that, in an LTCAM, an insert may be done in any free slot and a slot freed by a delete is simply linked to the free space list.}
invalidateWaitWrite waits till an ongoing lookup is complete and then invalidates the slot. Since updates are done serially in the control plane, invalidateWaitWrite from an LTCAM delete must complete before the next update operation begins.

Table 3-9. Number of moves for inserts and deletes in the ITCAM of DUOS

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Scheme 1</th>
<th>Scheme 2</th>
<th>Scheme 3</th>
<th>Scheme 4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>insert</td>
<td>delete</td>
<td>insert</td>
<td>delete</td>
</tr>
<tr>
<td>rrc00</td>
<td>102529</td>
<td>129102</td>
<td>11</td>
<td>26064</td>
</tr>
<tr>
<td>rrc01</td>
<td>125384</td>
<td>164002</td>
<td>1</td>
<td>138410</td>
</tr>
<tr>
<td>rrc03</td>
<td>115788</td>
<td>144517</td>
<td>604</td>
<td>27691</td>
</tr>
<tr>
<td>rrc04</td>
<td>57004</td>
<td>76867</td>
<td>0</td>
<td>19686</td>
</tr>
<tr>
<td>rrc05</td>
<td>119429</td>
<td>153774</td>
<td>8</td>
<td>35401</td>
</tr>
<tr>
<td>rrc06</td>
<td>124810</td>
<td>156191</td>
<td>0</td>
<td>29826</td>
</tr>
<tr>
<td>rrc07</td>
<td>42336</td>
<td>58283</td>
<td>0</td>
<td>15919</td>
</tr>
<tr>
<td>rrc10</td>
<td>85401</td>
<td>111943</td>
<td>3</td>
<td>25892</td>
</tr>
<tr>
<td>rrc11</td>
<td>104737</td>
<td>131628</td>
<td>9</td>
<td>26124</td>
</tr>
<tr>
<td>rrc12</td>
<td>126858</td>
<td>159951</td>
<td>605</td>
<td>31608</td>
</tr>
<tr>
<td>rrc13</td>
<td>119635</td>
<td>151585</td>
<td>7</td>
<td>31627</td>
</tr>
<tr>
<td>rrc14</td>
<td>81599</td>
<td>107038</td>
<td>5</td>
<td>24795</td>
</tr>
<tr>
<td>rrc15</td>
<td>88563</td>
<td>110812</td>
<td>112</td>
<td>23127</td>
</tr>
<tr>
<td>rrc16</td>
<td>4502</td>
<td>3374</td>
<td>0</td>
<td>794</td>
</tr>
<tr>
<td>route-views2</td>
<td>213075</td>
<td>276846</td>
<td>5</td>
<td>62422</td>
</tr>
<tr>
<td>route-views4</td>
<td>135689</td>
<td>182428</td>
<td>0</td>
<td>45788</td>
</tr>
<tr>
<td>route-views.eqix</td>
<td>87653</td>
<td>113690</td>
<td>4</td>
<td>25349</td>
</tr>
<tr>
<td>route-views.isc</td>
<td>118409</td>
<td>152221</td>
<td>5</td>
<td>32703</td>
</tr>
<tr>
<td>route-views.linx</td>
<td>135956</td>
<td>170540</td>
<td>3</td>
<td>33718</td>
</tr>
<tr>
<td>route-views.wide</td>
<td>136375</td>
<td>172235</td>
<td>3</td>
<td>32563</td>
</tr>
<tr>
<td>rrc00May20</td>
<td>13182</td>
<td>16674</td>
<td>16</td>
<td>3458</td>
</tr>
</tbody>
</table>

3.5.3 Evaluation of DUOW

In evaluating DUOW, we used a wide SRAM in conjunction with the LTCAM only as the ITCAM has relatively few (about 10%) prefixes. We instantiated an LTCAM with 100,000 slots and used the same configuration for the ITCAM as used in our evaluation of DUOS. For the DUOW evaluation, we used only Scheme 3 for memory management in the ITCAM. Table 3-12 gives the number of LTCAM prefixes carved by Lu's carving heuristic [21] and our carving heuristic of Section 3.3. The carving by both methods is done only on the trie of leaf prefixes as only leaf prefixes are stored in the LTCAM and its associated wide SRAM. Surprisingly, the number of prefixes that result when
Table 3-10. Number of \textit{waitWrites} in the ITCAM of DUOS

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Scheme 1</th>
<th>Scheme 2</th>
<th>Scheme 3</th>
<th>Scheme 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>rcc00</td>
<td>286568</td>
<td>81012</td>
<td>54949</td>
<td>83948</td>
</tr>
<tr>
<td>rcc01</td>
<td>368971</td>
<td>117996</td>
<td>79586</td>
<td>122687</td>
</tr>
<tr>
<td>rcc03</td>
<td>321230</td>
<td>89220</td>
<td>61539</td>
<td>92625</td>
</tr>
<tr>
<td>rcc04</td>
<td>174955</td>
<td>60770</td>
<td>41084</td>
<td>64562</td>
</tr>
<tr>
<td>rcc05</td>
<td>346573</td>
<td>108779</td>
<td>73378</td>
<td>112833</td>
</tr>
<tr>
<td>rcc06</td>
<td>353972</td>
<td>102797</td>
<td>72971</td>
<td>113443</td>
</tr>
<tr>
<td>rcc07</td>
<td>132509</td>
<td>47809</td>
<td>31890</td>
<td>50076</td>
</tr>
<tr>
<td>rcc10</td>
<td>250228</td>
<td>78779</td>
<td>52887</td>
<td>82620</td>
</tr>
<tr>
<td>rcc11</td>
<td>295146</td>
<td>84914</td>
<td>58790</td>
<td>87429</td>
</tr>
<tr>
<td>rcc12</td>
<td>358247</td>
<td>103651</td>
<td>72065</td>
<td>108545</td>
</tr>
<tr>
<td>rcc13</td>
<td>337230</td>
<td>97644</td>
<td>66019</td>
<td>102846</td>
</tr>
<tr>
<td>rcc14</td>
<td>243520</td>
<td>79683</td>
<td>54888</td>
<td>83733</td>
</tr>
<tr>
<td>rcc15</td>
<td>251663</td>
<td>75527</td>
<td>52390</td>
<td>79383</td>
</tr>
<tr>
<td>rcc16</td>
<td>10081</td>
<td>2999</td>
<td>2205</td>
<td>3154</td>
</tr>
<tr>
<td>route-views2</td>
<td>618746</td>
<td>191252</td>
<td>128831</td>
<td>195879</td>
</tr>
<tr>
<td>route-views4</td>
<td>417640</td>
<td>145311</td>
<td>99523</td>
<td>149081</td>
</tr>
<tr>
<td>route-views.eqix</td>
<td>257668</td>
<td>81678</td>
<td>56329</td>
<td>83475</td>
</tr>
<tr>
<td>route-views.isc</td>
<td>343790</td>
<td>105868</td>
<td>73164</td>
<td>108043</td>
</tr>
<tr>
<td>route-views.linx</td>
<td>379023</td>
<td>106248</td>
<td>72532</td>
<td>109375</td>
</tr>
<tr>
<td>route-views.wide</td>
<td>383443</td>
<td>107396</td>
<td>74833</td>
<td>114688</td>
</tr>
<tr>
<td>rcc00May20</td>
<td>37316</td>
<td>10934</td>
<td>7479</td>
<td>11081</td>
</tr>
</tbody>
</table>

our method is used is fewer than when the method of [21] is used. This is surprising because our method carves out independent prefixes while the method of [21] may carve any set of prefixes. The approximately 1% drop in the number of prefixes when our carving method is used results from the observation that when our method is used we do not need to supplement the carving prefixes with covering prefixes while covering prefixes need to be added to the set of carving prefixes generated by the method of [21]. Since covering prefixes account for approximately 8% of the prefixes generated by the method of [21], a 1% drop in the total number of prefixes when our method is used implies a roughly 7% increase in carving prefixes before accounting for covering prefixes.

Table 3-13 gives the number of inserts and deletes applied on the LTCAM of DUOW as well as the number \textit{waitWrites}. We observe that the number of \textit{waitWrites}
for the LTCAM of DUOW is more than the number of inserts and deletes done in the LTCAM. This is in contrast to DUOS where the number of waitWrites is the same as the number of inserts and deletes. This is because additional writes are needed in DUOW to maintain lookup consistency when the contents of an SRAM word are split or merged or when a suffix is added to or deleted from an existing SRAM word.

We note that the number of ITCAM inserts and deletes as well as the number of ITCAM waitWrites are unaffected by the coupling of a wide SRAM to the LTCAM. So, the numbers shown in Table 3-9 are valid for the DUOW ITCAM as well as for the DUOS ITCAM.

### 3.5.4 Evaluation of IDUOW

As was the case for our DUOW evaluation, for IDUOW too, we used a wide SRAM only in conjunction with the LTCAM. Further, an index TCAM (ILTCAM) with
Table 3-12. Number of prefixes to be stored in the LTCAM and associated wide SRAM

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Lu [21]</th>
<th>Our</th>
</tr>
</thead>
<tbody>
<tr>
<td>rrc00</td>
<td>68876</td>
<td>68196</td>
</tr>
<tr>
<td>rrc01</td>
<td>65068</td>
<td>64672</td>
</tr>
<tr>
<td>rrc03</td>
<td>66567</td>
<td>66060</td>
</tr>
<tr>
<td>rrc04</td>
<td>67895</td>
<td>67327</td>
</tr>
<tr>
<td>rrc05</td>
<td>65726</td>
<td>65319</td>
</tr>
<tr>
<td>rrc06</td>
<td>65411</td>
<td>65014</td>
</tr>
<tr>
<td>rrc07</td>
<td>64737</td>
<td>64322</td>
</tr>
<tr>
<td>rrc10</td>
<td>65566</td>
<td>65199</td>
</tr>
<tr>
<td>rrc11</td>
<td>65187</td>
<td>64766</td>
</tr>
<tr>
<td>rrc12</td>
<td>65564</td>
<td>65133</td>
</tr>
<tr>
<td>rrc13</td>
<td>66832</td>
<td>66366</td>
</tr>
<tr>
<td>rrc14</td>
<td>64955</td>
<td>64575</td>
</tr>
<tr>
<td>rrc15</td>
<td>66544</td>
<td>65982</td>
</tr>
<tr>
<td>rrc16</td>
<td>66353</td>
<td>65859</td>
</tr>
<tr>
<td>rviews2</td>
<td>68939</td>
<td>68300</td>
</tr>
<tr>
<td>rviews4</td>
<td>64839</td>
<td>64435</td>
</tr>
<tr>
<td>rviews.eqix</td>
<td>64881</td>
<td>64466</td>
</tr>
<tr>
<td>rviews.isc</td>
<td>66079</td>
<td>65664</td>
</tr>
<tr>
<td>rviews.linx</td>
<td>65372</td>
<td>64957</td>
</tr>
<tr>
<td>rviews.wide</td>
<td>66319</td>
<td>65910</td>
</tr>
<tr>
<td>rrc00May20</td>
<td>62638</td>
<td>62014</td>
</tr>
</tbody>
</table>

an associated wide SRAM was added only to the LTCAM. Our instantiated DLTCAM and ILTCAM had 200,000 and 20,000 slots, respectively. The DLTCAM bucket size was set to 512 slots for both schemes discussed in Section 3.4. Tables 3-14 and 3-15 give the number of inserts and deletes as well as the number of waitWrites for the ILTCAM and DLTCAM using 1-12Wc while Tables 3-16 and 3-17 give these numbers for the M-12Wb indexing scheme. As can be seen, the 1-12Wc scheme required between 203 to 227 buckets, thereby using up between 103936 and 116224 DLTCAM slots. The number of moves resulting from bucket splits varied from 0 to 1085. The M-12Wb scheme is more space efficient requiring between 128 and 153 buckets, thereby using up between 65536 and 78336 DLTCAM slots. However, the number of moves is between 800 and 16603 when M-12Wb is used. (We shall see later that the worst-case number of moves for these two schemes is comparable). Just as in DUOW, the number of waitWrites is more...
Table 3-13. Number of *waitWrites* in the LTCAM of DUOW

<table>
<thead>
<tr>
<th>Dataset</th>
<th>#inserts</th>
<th>#deletes</th>
<th>#waitWrites</th>
</tr>
</thead>
<tbody>
<tr>
<td>rrc00</td>
<td>391398</td>
<td>391867</td>
<td>840029</td>
</tr>
<tr>
<td>rrc01</td>
<td>507076</td>
<td>507379</td>
<td>1071516</td>
</tr>
<tr>
<td>rrc03</td>
<td>302359</td>
<td>302568</td>
<td>641880</td>
</tr>
<tr>
<td>rrc04</td>
<td>194212</td>
<td>194117</td>
<td>412859</td>
</tr>
<tr>
<td>rrc05</td>
<td>429065</td>
<td>427408</td>
<td>884675</td>
</tr>
<tr>
<td>rrc06</td>
<td>442145</td>
<td>442208</td>
<td>1121870</td>
</tr>
<tr>
<td>rrc07</td>
<td>164213</td>
<td>164186</td>
<td>328698</td>
</tr>
<tr>
<td>rrc10</td>
<td>329469</td>
<td>330166</td>
<td>693637</td>
</tr>
<tr>
<td>rrc11</td>
<td>336674</td>
<td>336954</td>
<td>750563</td>
</tr>
<tr>
<td>rrc12</td>
<td>423898</td>
<td>424286</td>
<td>892330</td>
</tr>
<tr>
<td>rrc13</td>
<td>282676</td>
<td>282509</td>
<td>594906</td>
</tr>
<tr>
<td>rrc14</td>
<td>251034</td>
<td>251273</td>
<td>577811</td>
</tr>
<tr>
<td>rrc15</td>
<td>268558</td>
<td>266942</td>
<td>667692</td>
</tr>
<tr>
<td>rrc16</td>
<td>11297</td>
<td>9427</td>
<td>23902</td>
</tr>
<tr>
<td>route-views2</td>
<td>635210</td>
<td>636625</td>
<td>1290565</td>
</tr>
<tr>
<td>route-views4</td>
<td>572678</td>
<td>572828</td>
<td>1252536</td>
</tr>
<tr>
<td>route-views.eqix</td>
<td>289812</td>
<td>289825</td>
<td>648771</td>
</tr>
<tr>
<td>route-views.isc</td>
<td>315905</td>
<td>316093</td>
<td>694299</td>
</tr>
<tr>
<td>route-views.linx</td>
<td>379967</td>
<td>380210</td>
<td>789002</td>
</tr>
<tr>
<td>route-views.wide</td>
<td>466474</td>
<td>468214</td>
<td>1070171</td>
</tr>
<tr>
<td>rrc00May20</td>
<td>48036</td>
<td>47982</td>
<td>104402</td>
</tr>
</tbody>
</table>

than the number of inserts and deletes and for DLTCAM there is an additional source for writes—prefix moves resulting from bucket overflows.

3.5.5 Comparison with MIPS and CAO_OPT

MIPS [53] and an update consistent version of CAO_OPT [41] obtained using the method of [54] are the competitors of DUO. In this section, we compare the consistent update TCAM schemes MIPS, CAO_OPT, and DUO. In MIPS, a data plane lookup is delayed if the lookup matches a TCAM slot whose next hop information is being updated. To avoid this delay while changing the nexthop of a prefix, we first insert a new entry with latest nexthop, and then delete the existing entry, in our experiments for MIPS. This ensures that data plane lookups are consistent and correct and are not delayed by control plane operations. Also as noted earlier, the MIPS scheme as described in [53] uses no memory management scheme and free slots are determined using TCAM
lookups that delay data plane lookups. To avoid these data plane lookup delays, for our experiments, we augmented the MIPS scheme of [53] with the memory management scheme employed by us for the LTCAM (Section 3.3). For the ITCAM of DUO, memory management is done using Scheme 3. Since the performance of the 3 TCAM schemes is characterized by the total number of the \textit{waitWrite} operations required by an update sequence as well as the maximum number of operations for an individual update request, our experiments measured these quantities.

Table 3-18 gives the total number of \textit{waitWrites} required to perform our test update sequences. We see that our DUO architecture requires fewer write operations than MIPS and CAO.OPT. The average number of \textit{waitWrites} per operation (Insert or Delete) ranged from a low of 1.565 to a high of 6.72 for MIPS, from 1.505 to 6.51 for CAO.OPT, from 1.000039 to 1.0641 for DUOS, from 1.00126 to 1.33705 for DUOW, from 1.00128

<table>
<thead>
<tr>
<th>Dataset</th>
<th>#inserts</th>
<th>#deletes</th>
<th>#waitWrites</th>
</tr>
</thead>
<tbody>
<tr>
<td>rrc00</td>
<td>10</td>
<td>5</td>
<td>34</td>
</tr>
<tr>
<td>rrc01</td>
<td>8</td>
<td>4</td>
<td>28</td>
</tr>
<tr>
<td>rrc03</td>
<td>4</td>
<td>2</td>
<td>14</td>
</tr>
<tr>
<td>rrc04</td>
<td>10</td>
<td>5</td>
<td>37</td>
</tr>
<tr>
<td>rrc05</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>rrc06</td>
<td>15</td>
<td>7</td>
<td>48</td>
</tr>
<tr>
<td>rrc07</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>rrc10</td>
<td>2</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>rrc11</td>
<td>6</td>
<td>3</td>
<td>21</td>
</tr>
<tr>
<td>rrc12</td>
<td>4</td>
<td>2</td>
<td>15</td>
</tr>
<tr>
<td>rrc13</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>rrc14</td>
<td>4</td>
<td>2</td>
<td>15</td>
</tr>
<tr>
<td>rrc15</td>
<td>6</td>
<td>3</td>
<td>20</td>
</tr>
<tr>
<td>rrc16</td>
<td>10</td>
<td>5</td>
<td>34</td>
</tr>
<tr>
<td>route-views2</td>
<td>2</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>route-views4</td>
<td>2</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>route-views.eqix</td>
<td>4</td>
<td>2</td>
<td>15</td>
</tr>
<tr>
<td>route-views.isc</td>
<td>4</td>
<td>2</td>
<td>15</td>
</tr>
<tr>
<td>route-views.linx</td>
<td>6</td>
<td>3</td>
<td>20</td>
</tr>
<tr>
<td>route-views.wide</td>
<td>2</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>rrc00May20</td>
<td>4</td>
<td>2</td>
<td>15</td>
</tr>
</tbody>
</table>
Table 3-15. Statistics for the DLTCAM of IDUOW using 1-12Wc

<table>
<thead>
<tr>
<th>Dataset</th>
<th>#inserts</th>
<th>#deletes</th>
<th>numBuckets</th>
<th>#waitWrites</th>
</tr>
</thead>
<tbody>
<tr>
<td>rrc00</td>
<td>391398</td>
<td>391867</td>
<td>221</td>
<td>849709</td>
</tr>
<tr>
<td>rrc01</td>
<td>507076</td>
<td>507379</td>
<td>215</td>
<td>1081031</td>
</tr>
<tr>
<td>rrc03</td>
<td>302359</td>
<td>302568</td>
<td>215</td>
<td>649739</td>
</tr>
<tr>
<td>rrc04</td>
<td>194212</td>
<td>194117</td>
<td>227</td>
<td>417069</td>
</tr>
<tr>
<td>rrc05</td>
<td>429065</td>
<td>427408</td>
<td>214</td>
<td>887978</td>
</tr>
<tr>
<td>rrc06</td>
<td>442145</td>
<td>442208</td>
<td>219</td>
<td>1151447</td>
</tr>
<tr>
<td>rrc07</td>
<td>164213</td>
<td>164186</td>
<td>209</td>
<td>328706</td>
</tr>
<tr>
<td>rrc10</td>
<td>329469</td>
<td>330166</td>
<td>218</td>
<td>696321</td>
</tr>
<tr>
<td>rrc11</td>
<td>336674</td>
<td>336954</td>
<td>215</td>
<td>760370</td>
</tr>
<tr>
<td>rrc12</td>
<td>423898</td>
<td>424286</td>
<td>213</td>
<td>902959</td>
</tr>
<tr>
<td>rrc13</td>
<td>282676</td>
<td>282509</td>
<td>215</td>
<td>600789</td>
</tr>
<tr>
<td>rrc14</td>
<td>251034</td>
<td>251273</td>
<td>213</td>
<td>585663</td>
</tr>
<tr>
<td>rrc15</td>
<td>268558</td>
<td>266942</td>
<td>218</td>
<td>683039</td>
</tr>
<tr>
<td>rrc16</td>
<td>11297</td>
<td>9427</td>
<td>219</td>
<td>26241</td>
</tr>
<tr>
<td>route-views2</td>
<td>635210</td>
<td>636625</td>
<td>223</td>
<td>1295688</td>
</tr>
<tr>
<td>route-views4</td>
<td>572678</td>
<td>572828</td>
<td>211</td>
<td>1268931</td>
</tr>
<tr>
<td>route-views.eqix</td>
<td>289812</td>
<td>289825</td>
<td>213</td>
<td>656627</td>
</tr>
<tr>
<td>route-views.isc</td>
<td>315905</td>
<td>316093</td>
<td>213</td>
<td>705386</td>
</tr>
<tr>
<td>route-views.linx</td>
<td>379967</td>
<td>380210</td>
<td>215</td>
<td>796007</td>
</tr>
<tr>
<td>route-views.wide</td>
<td>466474</td>
<td>468214</td>
<td>218</td>
<td>1089092</td>
</tr>
<tr>
<td>rrc00May20</td>
<td>48036</td>
<td>47982</td>
<td>203</td>
<td>105453</td>
</tr>
</tbody>
</table>

to 1.3702 for IDUOW with 1-12Wc and from 1.00583 to 2.3966 for IDUOW with M-12Wb. Since the various DUO schemes require a similar number of writes, M-12Wb is to be preferred because of its lower TCAM memory and power requirement. Figure 3-56(a) shows the normalized average \( \text{waitWrites} \) for the different architectures. For this figure, we first computed the average number of \( \text{waitWrites} \) per Insert/Delete for each dataset. Then, the average of the averages was computed for each architecture and normalized by the average of the averages for DUOS.

Table 3-19 gives the maximum number of write operations required by an insert or delete in our test update sequences. As can be seen, MIPS uses a larger number of writes in the worst case than any of the remaining schemes. We notice that the worst-case number of writes for rrc00May20 is particularly large for MIPS. This is because the update sequence for rrc00May20 contains announcements and withdrawals.
Table 3-16. Statistics for the ILTCAM of IDUOW using M-12Wb

<table>
<thead>
<tr>
<th>Dataset</th>
<th>#inserts</th>
<th>#deletes</th>
<th>#waitWrites</th>
</tr>
</thead>
<tbody>
<tr>
<td>rrc00</td>
<td>170</td>
<td>85</td>
<td>533</td>
</tr>
<tr>
<td>rrc01</td>
<td>174</td>
<td>87</td>
<td>550</td>
</tr>
<tr>
<td>rrc03</td>
<td>134</td>
<td>67</td>
<td>424</td>
</tr>
<tr>
<td>rrc04</td>
<td>176</td>
<td>88</td>
<td>564</td>
</tr>
<tr>
<td>rrc05</td>
<td>210</td>
<td>105</td>
<td>656</td>
</tr>
<tr>
<td>rrc06</td>
<td>284</td>
<td>142</td>
<td>893</td>
</tr>
<tr>
<td>rrc07</td>
<td>20</td>
<td>10</td>
<td>63</td>
</tr>
<tr>
<td>rrc10</td>
<td>182</td>
<td>91</td>
<td>574</td>
</tr>
<tr>
<td>rrc11</td>
<td>214</td>
<td>108</td>
<td>673</td>
</tr>
<tr>
<td>rrc12</td>
<td>150</td>
<td>75</td>
<td>472</td>
</tr>
<tr>
<td>rrc13</td>
<td>224</td>
<td>117</td>
<td>693</td>
</tr>
<tr>
<td>rrc14</td>
<td>154</td>
<td>77</td>
<td>485</td>
</tr>
<tr>
<td>rrc15</td>
<td>298</td>
<td>152</td>
<td>936</td>
</tr>
<tr>
<td>rrc16</td>
<td>224</td>
<td>112</td>
<td>702</td>
</tr>
<tr>
<td>route-views2</td>
<td>158</td>
<td>79</td>
<td>498</td>
</tr>
<tr>
<td>route-views4</td>
<td>276</td>
<td>138</td>
<td>868</td>
</tr>
<tr>
<td>route-views.eqix</td>
<td>270</td>
<td>135</td>
<td>848</td>
</tr>
<tr>
<td>route-views.isc</td>
<td>156</td>
<td>78</td>
<td>493</td>
</tr>
<tr>
<td>route-views.linx</td>
<td>260</td>
<td>130</td>
<td>816</td>
</tr>
<tr>
<td>route-views.wide</td>
<td>265</td>
<td>134</td>
<td>835</td>
</tr>
<tr>
<td>rrc00May20</td>
<td>104</td>
<td>52</td>
<td>323</td>
</tr>
</tbody>
</table>

Figure 3-56. Comparison of TCAM performance and power consumption between MIPS, CAO_OPT, DUO
Table 3-17. Statistics for the DLTCAM of IDUOW using M-12Wb

<table>
<thead>
<tr>
<th>Dataset</th>
<th>#inserts</th>
<th>#deletes</th>
<th>#numBuckets</th>
<th>#waitWrites</th>
</tr>
</thead>
<tbody>
<tr>
<td>rrc00</td>
<td>391398</td>
<td>391867</td>
<td>149</td>
<td>867908</td>
</tr>
<tr>
<td>rrc01</td>
<td>507076</td>
<td>507379</td>
<td>142</td>
<td>1099815</td>
</tr>
<tr>
<td>rrc03</td>
<td>302359</td>
<td>302568</td>
<td>144</td>
<td>664596</td>
</tr>
<tr>
<td>rrc04</td>
<td>194212</td>
<td>194117</td>
<td>148</td>
<td>436057</td>
</tr>
<tr>
<td>rrc05</td>
<td>429065</td>
<td>427408</td>
<td>146</td>
<td>912990</td>
</tr>
<tr>
<td>rrc06</td>
<td>442145</td>
<td>442208</td>
<td>149</td>
<td>1180372</td>
</tr>
<tr>
<td>rrc07</td>
<td>164213</td>
<td>164186</td>
<td>128</td>
<td>330282</td>
</tr>
<tr>
<td>rrc10</td>
<td>329469</td>
<td>330166</td>
<td>142</td>
<td>716458</td>
</tr>
<tr>
<td>rrc11</td>
<td>336674</td>
<td>336954</td>
<td>145</td>
<td>784772</td>
</tr>
<tr>
<td>rrc12</td>
<td>423898</td>
<td>424286</td>
<td>142</td>
<td>920560</td>
</tr>
<tr>
<td>rrc13</td>
<td>282676</td>
<td>282509</td>
<td>149</td>
<td>624757</td>
</tr>
<tr>
<td>rrc14</td>
<td>251034</td>
<td>251273</td>
<td>140</td>
<td>602928</td>
</tr>
<tr>
<td>rrc15</td>
<td>268558</td>
<td>266942</td>
<td>153</td>
<td>714549</td>
</tr>
<tr>
<td>rrc16</td>
<td>11297</td>
<td>9427</td>
<td>147</td>
<td>49978</td>
</tr>
<tr>
<td>route-views2</td>
<td>635210</td>
<td>636625</td>
<td>149</td>
<td>1313783</td>
</tr>
<tr>
<td>route-views4</td>
<td>572678</td>
<td>572828</td>
<td>148</td>
<td>1299622</td>
</tr>
<tr>
<td>route-views.eqix</td>
<td>289812</td>
<td>289825</td>
<td>147</td>
<td>685458</td>
</tr>
<tr>
<td>route-views.isc</td>
<td>315905</td>
<td>316093</td>
<td>142</td>
<td>723400</td>
</tr>
<tr>
<td>route-views.linx</td>
<td>379967</td>
<td>380210</td>
<td>149</td>
<td>823707</td>
</tr>
<tr>
<td>route-views.wide</td>
<td>466474</td>
<td>468214</td>
<td>153</td>
<td>1120072</td>
</tr>
<tr>
<td>rrc00May20</td>
<td>48036</td>
<td>47982</td>
<td>131</td>
<td>117622</td>
</tr>
</tbody>
</table>

of routes for prefixes of small lengths, such as 2 and 4. Each of these translates into a very large number of inserts/deletes of independent prefixes.

Our DUOS and DUOW architectures have better worst-case performance (on a per update basis) than MIPS. DUOS is generally better than CAO_OPT and DUOW, while inferior to CAO_OPT, is often competitive. Even though, the worst-case number of writes with IDUOW is more than that for CAO_OPT, the number of writes is bounded by the size of a bucket. Thus, the worst-case writes may be reduced by using a smaller bucket size than the 512 size used in our experiments. For example, when the bucket size as 32, the maximum number of write operations in DLTCAM of IDUOW is also 32. This is because when an index node is split, we relocate the split node that has the smaller number of DLTCAM prefixes. Thus at most 16 prefixes are moved, and hence there are 32 write operations at most.
Table 3-18. Total number of TCAM *waitWrite* operations

<table>
<thead>
<tr>
<th>Dataset</th>
<th>MIPS [53]</th>
<th>CAO.OPT [41]</th>
<th>DUOS</th>
<th>DUOW (1-12Wc)</th>
<th>IDUOW (M-12Wb)</th>
</tr>
</thead>
<tbody>
<tr>
<td>rrc00</td>
<td>1442078</td>
<td>1249984</td>
<td>831630</td>
<td>894978</td>
<td>904692</td>
</tr>
<tr>
<td>rrc01</td>
<td>1798445</td>
<td>1655241</td>
<td>1083395</td>
<td>1151102</td>
<td>1160645</td>
</tr>
<tr>
<td>rrc03</td>
<td>1159357</td>
<td>997100</td>
<td>655262</td>
<td>703419</td>
<td>711292</td>
</tr>
<tr>
<td>rrc04</td>
<td>887877</td>
<td>701722</td>
<td>425587</td>
<td>453943</td>
<td>458190</td>
</tr>
<tr>
<td>rrc05</td>
<td>1436610</td>
<td>1428017</td>
<td>924947</td>
<td>958053</td>
<td>961356</td>
</tr>
<tr>
<td>rrc06</td>
<td>2074384</td>
<td>1344593</td>
<td>950924</td>
<td>1194841</td>
<td>1224466</td>
</tr>
<tr>
<td>rrc07</td>
<td>783637</td>
<td>602014</td>
<td>360149</td>
<td>360588</td>
<td>360596</td>
</tr>
<tr>
<td>rrc10</td>
<td>1168964</td>
<td>1082896</td>
<td>703412</td>
<td>746524</td>
<td>749215</td>
</tr>
<tr>
<td>rrc11</td>
<td>1352758</td>
<td>1057557</td>
<td>715786</td>
<td>809353</td>
<td>819181</td>
</tr>
<tr>
<td>rrc12</td>
<td>1602375</td>
<td>1380918</td>
<td>908971</td>
<td>964395</td>
<td>975039</td>
</tr>
<tr>
<td>rrc13</td>
<td>1191824</td>
<td>989752</td>
<td>630400</td>
<td>660925</td>
<td>666808</td>
</tr>
<tr>
<td>rrc14</td>
<td>993155</td>
<td>794964</td>
<td>536465</td>
<td>632699</td>
<td>640566</td>
</tr>
<tr>
<td>rrc15</td>
<td>1208090</td>
<td>864424</td>
<td>585012</td>
<td>720082</td>
<td>735449</td>
</tr>
<tr>
<td>rrc16</td>
<td>45895</td>
<td>33722</td>
<td>22919</td>
<td>26107</td>
<td>28480</td>
</tr>
<tr>
<td>route-views2</td>
<td>2242123</td>
<td>2178506</td>
<td>1397932</td>
<td>1419396</td>
<td>1424525</td>
</tr>
<tr>
<td>route-views4</td>
<td>2304065</td>
<td>1818222</td>
<td>1225347</td>
<td>1352059</td>
<td>1368462</td>
</tr>
<tr>
<td>route-views.eqix</td>
<td>1278271</td>
<td>925087</td>
<td>624700</td>
<td>705100</td>
<td>712971</td>
</tr>
<tr>
<td>route-views.isc</td>
<td>1172542</td>
<td>1013145</td>
<td>695496</td>
<td>767463</td>
<td>778565</td>
</tr>
<tr>
<td>route-views.linx</td>
<td>1306298</td>
<td>1257542</td>
<td>826755</td>
<td>861534</td>
<td>868559</td>
</tr>
<tr>
<td>route-views.wide</td>
<td>1988152</td>
<td>1435007</td>
<td>988721</td>
<td>1145004</td>
<td>1163932</td>
</tr>
<tr>
<td>rrc00May20</td>
<td>683608</td>
<td>663306</td>
<td>102817</td>
<td>111881</td>
<td>112947</td>
</tr>
</tbody>
</table>

Theoretically, it is possible for each update in MIPS to require a number of TCAM writes equal to the number of prefixes in the table. This happens for example when there is a trie in which no leaf prefix has a sibling after the leaf pushing and prefix compression steps, and to that trie if a default prefix of length 0 is inserted or deleted (see Figure 3-1). On the other hand, CAO.OPT requires at most $W/2$ moves per update ($W = 32$ for IPv4). Hence, CAO.OPT requires $W/2$ writes per update in the worst case. For DUOS, the worst case writes occur when a prefix is to be inserted to LTCAM and this requires a prefix deletion from LTCAM and a prefix insertion at ITCAM. The two LTCAM operations require 2 writes, whereas the ITCAM operation requires $W$ writes in the worst case using Scheme 3. Thus DUOS requires $(W + 2)$ writes in the worst case. For DUOW, the worst case scenario is same as that for DUOS, except that a LTCAM insert can require 3 writes when a SRAM word is split (1 delete to remove the split word
and 2 inserts for the new words). Similarly, a LTCAM delete can also require 3 writes when a SRAM word is merged (2 deletes for the two words merged and 1 insert for the new word). Thus, DUOW requires \((W + 6)\) writes in the worst case. For IDUOW, the worst case combination involves the ITCAM, ILTCAM and DLTCAM. IDUOW requires at most \(W\) writes for ITCAM and 6 writes for ILTCAM and \(\text{bucketSize}\) writes for DLTCAM, with a maximum of \((W + \text{bucketSize} + 6)\) writes for a single update.

Table 3-19. Maximum number of TCAM writes for a single raw insert/delete

<table>
<thead>
<tr>
<th>Dataset</th>
<th>MIPS [53]</th>
<th>CAO_OPT [41]</th>
<th>DUOS</th>
<th>DUOW</th>
</tr>
</thead>
<tbody>
<tr>
<td>rrc00</td>
<td>266</td>
<td>5</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>rrc01</td>
<td>296</td>
<td>5</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>rrc03</td>
<td>1186</td>
<td>5</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>rrc04</td>
<td>2682</td>
<td>6</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>rrc05</td>
<td>383</td>
<td>5</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>rrc06</td>
<td>1278</td>
<td>5</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>rrc07</td>
<td>6389</td>
<td>5</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>rrc10</td>
<td>304</td>
<td>6</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>rrc11</td>
<td>546</td>
<td>5</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>rrc12</td>
<td>7099</td>
<td>6</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>rrc13</td>
<td>1071</td>
<td>5</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>rrc14</td>
<td>306</td>
<td>6</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>rrc15</td>
<td>5938</td>
<td>4</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>rrc16</td>
<td>198</td>
<td>5</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>route-views2</td>
<td>568</td>
<td>5</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>route-views4</td>
<td>377</td>
<td>5</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>route-views.eqix</td>
<td>260</td>
<td>7</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>route-views.isc</td>
<td>386</td>
<td>5</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>route-views.linx</td>
<td>306</td>
<td>6</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>route-views.wide</td>
<td>278</td>
<td>4</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>rrc00May20</td>
<td>102249</td>
<td>6</td>
<td>5</td>
<td>7</td>
</tr>
</tbody>
</table>

Table 3-20 gives the power consumption characteristics of MIPS, CAO_OPT and DUO in terms of the number of entries enabled during a search operation. The TCAM entries are counted based on the initial layout of prefixes for the input routing table. MIPS, CAO_OPT, DUOS and DUOW enable all valid TCAM entries during a search operation. IDUOW, on the other hand, enables all valid TCAM entries for ITCAM and ILTCAM, and only a bucket of entries for DLTCAM. Column 2 gives the number of
enabled entries for MIPS, while column 3 gives the number of enabled entries for CAO\_OPT on the simple TCAM and also for DUOS which is obtained by summing up the number of ITCAM and LTCAM entries. Both CAO\_OPT and DUOS have the same number of entries in TCAM since they store each prefix in a single TCAM entry. Column 4 gives the number of enabled entries for DUOW, which is obtained as the sum of valid ITCAM and LTCAM entries. Columns 5 and 6 give the number of enabled entries for IDUOW with 1-12Wc and M-12Wb, respectively. This number is obtained as the sum of valid entries in ITCAM, ILTCAM and the number of entries in a bucket in DLTCAM (fixed to 512 for our experiments). We observe that for MIPS, the leaf pushing and prefix compression steps have reduced the number of TCAM entries, and hence the power compared to CAO\_OPT and DUOS. MIPS requires about 1.5 to 2 times the power required by DUOW for all the tests, except rrc06 and rrc15. In the case of rrc06, MIPS requires about 7% more power than DUOW while it requires about 7% less power on rrc15. MIPS consumes between 3 to 10 times the power consumed by IDUOW.

Figure 3-56(b) shows the normalized average power for the different schemes. For this figure, we first computed the average number of enabled entries for every TCAM search for each architecture. Then, the average was normalized by the average number of enabled entries for IDUOW with 1-12Wc. Note that the power requirement for DUOW can be reduced further by using a wider SRAM than the 144 bit wide SRAM used for our experiments. The power requirements for IDUOW may be reduced by increasing SRAM width and by adding an index TCAM and a wide SRAM to the ITCAM. For example, the power consumed by DLTCAM and ILTCAM of IDUOW was less than 560 for the 1-12Wc scheme and less than 630 for the M-12Wb scheme. When an index TCAM and wide SRAM is added to the ITCAM to our experimental IDUOW, the power requirement for the ITCAM is expected to approximate that for the LTCAM (assuming the same bucket size is used). So, the IDUOW power requirement would drop to about 1120 for 1-12Wc and about 1260 for M-12Wb. So, with the addition of an index TCAM and a wide SRAM.
to the ITCAM of IDUOW, the power required by MIPS is between 68 to 248 times that required by IDUOW.

Table 3-20. A comparison of power consumed by MIPS, CAO_OPT and DUO in performing TCAM search

<table>
<thead>
<tr>
<th>Dataset</th>
<th>MIPS</th>
<th>CAO_OPT or DUO</th>
<th>DUOW</th>
<th>IDUOW (1-12Wc)</th>
<th>IDUOW (M-12Wb)</th>
</tr>
</thead>
<tbody>
<tr>
<td>rrc00</td>
<td>245875</td>
<td>294098</td>
<td>95577</td>
<td>27938</td>
<td>27989</td>
</tr>
<tr>
<td>rrc01</td>
<td>200733</td>
<td>276795</td>
<td>89459</td>
<td>25343</td>
<td>25387</td>
</tr>
<tr>
<td>rrc03</td>
<td>272046</td>
<td>283754</td>
<td>92176</td>
<td>26672</td>
<td>26714</td>
</tr>
<tr>
<td>rrc04</td>
<td>203375</td>
<td>288610</td>
<td>92464</td>
<td>25701</td>
<td>25759</td>
</tr>
<tr>
<td>rrc05</td>
<td>261067</td>
<td>280041</td>
<td>90694</td>
<td>25933</td>
<td>25987</td>
</tr>
<tr>
<td>rrc06</td>
<td>96479</td>
<td>278744</td>
<td>90221</td>
<td>25762</td>
<td>25839</td>
</tr>
<tr>
<td>rrc07</td>
<td>188373</td>
<td>275097</td>
<td>88763</td>
<td>24997</td>
<td>25055</td>
</tr>
<tr>
<td>rrc10</td>
<td>178987</td>
<td>278898</td>
<td>90031</td>
<td>25390</td>
<td>25434</td>
</tr>
<tr>
<td>rrc11</td>
<td>188527</td>
<td>277166</td>
<td>89553</td>
<td>25343</td>
<td>25399</td>
</tr>
<tr>
<td>rrc12</td>
<td>203440</td>
<td>278499</td>
<td>90027</td>
<td>25450</td>
<td>25493</td>
</tr>
<tr>
<td>rrc13</td>
<td>234053</td>
<td>284986</td>
<td>92686</td>
<td>26877</td>
<td>26935</td>
</tr>
<tr>
<td>rrc14</td>
<td>172096</td>
<td>276170</td>
<td>89060</td>
<td>25041</td>
<td>25084</td>
</tr>
<tr>
<td>rrc15</td>
<td>85463</td>
<td>284047</td>
<td>92166</td>
<td>26741</td>
<td>26813</td>
</tr>
<tr>
<td>rrc16</td>
<td>212282</td>
<td>282660</td>
<td>91445</td>
<td>26143</td>
<td>26198</td>
</tr>
<tr>
<td>rviews2</td>
<td>277560</td>
<td>294127</td>
<td>95183</td>
<td>27439</td>
<td>27502</td>
</tr>
<tr>
<td>rviews4</td>
<td>140962</td>
<td>275737</td>
<td>88978</td>
<td>25099</td>
<td>25145</td>
</tr>
<tr>
<td>rviews.eqix</td>
<td>175659</td>
<td>275736</td>
<td>88889</td>
<td>24980</td>
<td>25054</td>
</tr>
<tr>
<td>rviews.isc</td>
<td>193800</td>
<td>281095</td>
<td>91123</td>
<td>26018</td>
<td>26091</td>
</tr>
<tr>
<td>rviews.linx</td>
<td>202254</td>
<td>278196</td>
<td>90029</td>
<td>25631</td>
<td>25688</td>
</tr>
<tr>
<td>rviews.wide</td>
<td>150427</td>
<td>283569</td>
<td>92320</td>
<td>26966</td>
<td>27037</td>
</tr>
<tr>
<td>rrc00May20</td>
<td>220067</td>
<td>266185</td>
<td>86421</td>
<td>24961</td>
<td>25001</td>
</tr>
</tbody>
</table>

In this chapter, we have proposed a dual TCAM architecture-DUO-for routing tables. Four memory management schemes also have been evaluated extensively for the ITCAM of DUO. Of these memory management schemes, Scheme 2 and Scheme 3 are the ones proposed by us. Our experiments showed that Scheme 3 is far better than any of the other schemes in terms of the number of moves per update operation.

DUO provides incremental update facility to the low power lookup schemes in [21], without locking the TCAM at any time for performing updates. Supporting incremental updates to the schemes in [21] was problematic since each prefix in the TCAM stored a corresponding covering prefix in the wide SRAM, and such covering...
prefixes could be shared by a number of TCAM entries. The covering prefixes are a subset of the intermediate prefixes, and by putting all intermediate prefixes in a separate TCAM (ITCAM) and enabling a parallel match on ITCAM and LTCAM, as in DUO, we completely bypass the problem with covering prefixes in the performance of incremental updates.

DUO is fast and power efficient when it comes to incorporating the updates. The speed and power efficiency of DUO are due to both its architecture and its memory management schemes. From our datasets we found that over 90% of the updates are directed to the LTCAM of DUO, which stored disjoint prefixes. Thus, no prefix move in involved for most of the updates. The less than 10% of updates that are directed to the ITCAM involve very small number of moves when the memory management Scheme 3 is used. Memory management Scheme 3, proposed by us, requires between 1/74000 and 1/93 times the number of moves required by CAO.OPT. The low average values for the number of moves using Scheme 3 are backed by very low standard deviation for all the tests in our dataset.

Our DUO architectures, like those based on the CoPTUA [54], provide for consistent data-plane lookups and incremental control-plane updates that do not delay data-plane lookups. While the MIPS architecture of [53] provides consistent data-plane lookups, these lookups may encounter delays by ongoing control-plane operations that, for example, change the next hop associated with a prefix. These delays may be eliminated by implementing a next hop change as an insert followed by a delete as suggested in [53]. Delays caused by control-plane operations that require a free slot to be found may be eliminated using one of our proposed memory management schemes, preferably Scheme 3. Making these two modifications to MIPS results in a delay-free MIPS.

Experiments with delay-free MIPS and a consistent lookup version of CAO.OPT indicate that these two architectures make, on average, between 1.5 and 5 times as
many TCAM writes as made by any our DUO architectures to perform control-plane updates. In terms of the worst-case number of writes needed for an insert or delete, MIPS requires as many writes as prefixes in the table while CAO_OPT requires 16 for IPv4, DUOS requires 34, DUOW requires 38, and IDUOW requires 38+bucketSize. On our test data, MIPS required up to 102,249 writes for a single insert/delete while CAO_OPT required at most 7 writes, DUOS required at most 10 writes, DUOW required at most 11 writes, and IDUOW required at most 512 writes. The maximum number of writes for IDUOW may be reduced by reducing the bucket size. The very large number of worst-case writes for MIPS is a serious problem as this makes the router very susceptible to malicious users who inject a stream of worst-case inserts/deletes into the update stream. While this also is an issue, though to a lesser extent, for IDUOW, IDUOW offers power advantages over the remaining DUO schemes.

On our test data, MIPS reduced power consumption for a TCAM search by 4% to 69% relative to CAO_OPT and DUOS, which take the same amount of power. However, MIPS generally required between 1.5 and 2 times the power required by DUOW and between 3 and 10 times that required by our experimental version of IDUOW. However, by adding an index TCAM and a wide SRAM to the ITCAM of IDUOW, the power required by MIPS is between 68 and 248 times that required by the enhanced IDUOW. Further reduction in power required by DUOW and IDUOW result from using a wider SRAM than the 144-bit wide SRAM used in our experiments.

Between DUOW and IDUOW, IDUOW is recommended for least power consumption during lookups whereas DUOW is recommended for a lower worst-case delay in incorporating the updates to the forwarding table while still providing significant power benefits during a TCAM lookup.

In the next chapter, we will further analyze consistency issues in a very detailed way.
CHAPTER 4
CONSIST: CONSISTENT UPDATES FOR PACKET CLASSIFIERS

Rules in packet classifiers and forwarding tables are always in a flux of change, with new rules being added and existing rules being deleted or changed. In this chapter, we formalize the consistency properties of incrementally updated classifiers when updates arrive in clusters and present a heuristic to obtain a near optimal batch consistent sequence as a topological ordering of the precedence graph. Here, optimality is defined with respect to the increase in size of the intermediate rule table, where an optimal sequence guarantees minimum increase in the maximum table size.

We begin with Section 4.1 by reviewing background and related work. In Section 4.2 we present the consistency requirements and analysis. An experimental evaluation with update traces for forwarding tables at various routers and synthetic update traces for packet classifiers is presented in Section 4.3 and we conclude the chapter with Section 4.4.

4.1 Background and Related Work

It is often found that a router or a packet classifier is faced with a number of update requests that arrive at the same time. For example, if we consider update messages received by a router under Border Gateway Protocol (BGP) from a BGP update file [6], we see multiple route announcement and withdrawal notices in the same message, and many such messages having the same timestamp of receipt. The announcement and withdrawal messages result in insertion, deletion or changes in rules in the forwarding table of the router.

The updates received in a cluster can be processed either incrementally, or in a batch. Most routers perform updates one at a time (i.e., incrementally) in the control plane concurrent with lookup operations in the data plane. By incrementally performing the updates in a cluster in a carefully selected order, it is possible for an incrementally updated router to behave exactly like one that is batch updated. Note that in a batch
updated router, packets are routed to a next hop determined either by the rule table before any update is done or by the rule table following the incorporation of all updates.

![Diagram](image)

**Figure 4-1.** Prefixes in forwarding table before and after applying updates

Informally, a rule table is consistent when every lookup returns the action that would be returned if the lookup were done just before a cluster of updates is applied or just after the update cluster is completed. For example, suppose a forwarding table contains rules: \((00*, H2), (*, H0)\). Figure 4-1(a) illustrates these rules on a 4-bit address space. Now suppose that the following updates are received in a cluster: delete\((00*)\), insert\((0*, H1)\). Figure 4-1(b) gives the prefixes in the forwarding table after the updates are applied. The next hop for a data packet with destination address 0010 is shown in Figure 4-1 with a bold line. If no update has been processed yet, then from Figure 4-1(a) next hop \(H2\) is returned, using shortest range matching (which is equivalent to LMP, as prefix \(00*\) matches). If on the other hand, the cluster of updates is completely processed, then from Figure 4-1(b) the returned next hop is \(H1\). Thus, as we apply the updates we must ensure that the packets are forwarded to a hop from the set \(\{H1, H2\}\). For example, if we apply the updates in the following sequence: insert\((0*, H1)\), delete\((00*)\), then consistency is maintained at every step as we see from Figure 4-2(a), since the hop is picked from set \(\{H1, H2\}\). On the other hand, if we
apply the updates as they come, then consistency is not maintained (see Figure 4-2(b)) because the returned hop is $H_0 \not\in \{H_1, H_2\}$ following the operation delete(00*). This example deals with batch consistency, which we will define later.

As updates are received in clusters, it is possible to arrange the updates in such a way that the size of the intermediate rule table is the minimum. In general, we use functions insert, delete and change to represent the insertion, deletion and change, respectively, of a rule. For example, consider the cluster of updates: insert(rule10), insert(rule11), delete(rule3), delete(rule4), delete(rule5). If the updates are done in this order, then the number of rules in the table first increases by two due to the inserts and finally decreases as the deletes are performed. On the other hand, if the deletes are done first, then there is no temporary increase in the number of entries in the table.

If the table size is tightly constrained then picking the deletes first could be helpful in avoiding overflow situations.

With the Internet being under a constant flux of change, there has been a significant amount of work in the recent past dealing with how to incorporate the changes to the
forwarding table and packet classifier. The two broad categories of work in this area include one in which the updates are applied incrementally [4, 41, 42, 45, 54] and the other in which the updates are applied in batches [12, 27, 52].

To apply the updates in batches, typically a duplicate copy of the forwarding table is maintained so that when one copy is involved in lookup the other can be updated and after the update is complete, the lookups are done in the updated copy. Although batch updates require double the memory footprint, they are useful for situations when incremental updates are compute-intensive. Batch updates have been used in high-performance router design [34]. Incremental updates, on the other hand, are applied to the same copy of the forwarding table that is used for lookups. Update and lookup operations could use time multiplexing to share the common memory that stores the prefix database [4, 41, 42]. Thus the database is locked when the update operations are carried out, to prevent a lookup operation from taking place at the same time.

Z. Wang et. al in CoPTUA [54] developed an approach to perform classifier updates without locking the policy filter (PF) table (stored in a TCAM) from search key matching while ensuring that the searches return consistent and non-erroneous results. They define a consistent rule table to be a rule table in which the rule matched (including the action associated with the rule) during a search key matching is either the rule (including action) that would be matched just before or just after any ongoing update operation. Wang et al. [54] developed a scheme for consistent table update without locking the TCAM at any time, essentially allowing a search to proceed while the table is being updated. The concept of consistent rule table is easily extended to a forwarding table such that the table is not locked while updates are processed and lookups return next hops that would be returned just before or just after an ongoing update operation. Other TCAM designs that support consistent incremental updates on a per operation basis without locking for lookups are [26, 53].
The authors in [53] maintain a leaf-pushed prefix trie. As a result, any insert, delete or change operation to a non-leaf trie node could cause multiple leaf prefixes to be changed, deleted or inserted. In fact, in the worst case, when there is an operation on the default prefix (to be applied to the root node of the trie) there could be \(O(n)\) extra operations [26]. The authors in [54] show how multiple policy filter rules get affected by the introduction or deletion of a filter rule and how in the worst case \(N_r/2\) rule moves may be required for updating a filter table consisting of \(N_r\) rules. Both [54] and [53] suggest handling a cluster of updates by doing the deletes first and then the inserts. This strategy, however, does not ensure table consistency.

Compared to the existing works [54], [26, 53], our work is different in that we focus on the ordering of the update operations that can be readily applied to the rule table at a given time. The ordering of updates matter when the rule table is updated incrementally. For example, if the updates are applied in any order, it could result in matching rules with lower priority during lookup even as higher priority rules are waiting in the queue to be incorporated.

Basu et al. in [5] present an algorithm to reduce the number of writes to the prefix trie by suppressing the “redundant” writes. A redundant write is one that is succeeded by another write and after the succeeding write is applied, the state of the prefix database is the same irrespective of whether the redundant write was applied. In this chapter, we offer a detailed algorithm for computing a reduction of a given update sequence based on the “insert”, “delete” and “change” functions.
4.2 Consistent Updates

Figure 4-4 shows the overall flow of our methodology for obtaining a consistent sequence of updates from a given cluster of updates that leads to near-optimal growth of the intermediate table size. The first phase is the reduction phase where reduction, in the context of update operations, is presented in Definition 7. In the second phase a precedence graph is built with the update operations as the vertices. Section 4.2.2.1 discusses about the generation of the graph. The third phase deals with generation of a consistent update sequence as a topological ordering of the precedence graph that leads to near-optimal increase in intermediate size of the rule table. Section 4.2.2.2 presents a heuristic for the third phase.

4.2.1 Definitions and Properties

We first define the following terms: reduction of an update sequence, a batch consistent sequence and an incremental consistent sequence. Later we state and prove some of the interesting properties of reduced, batch and incremental consistent sequences.

**Definition 7.** Let $U = u_1, \ldots, u_r$ be an update sequence; each $u_i$ is an insert, delete, or change operation. The update sequence $V(U)$ (or simply $V$) derived from the update sequence $U$ in the following manner is called the **reduction** of $U$.

Examine the update operations in the order $u_1, u_5, \ldots$. Let $F$ be the field tuple associated with the operation $u_i$ being examined. If $F$ occurs next in $u_j$, $j > i$, do the following:

1. If $u_i = \text{change}(F, \text{newA})$ and $u_j = \text{change}(F, \text{newA}')$, remove $u_i$ from $U$. If $\text{newA}'$ is the same as the existing action for $F$ in the rule table, then remove $u_j$ from $U$ as well.

2. If $u_i = \text{change}(F, \text{newA})$ and $u_j = \text{delete}(F)$, remove $u_i$ from $U$

3. If $u_i = \text{delete}(F)$ and $u_j = \text{insert}(F, A)$, remove $u_i$ from $U$ and replace $u_j$ by $\text{change}(F, A)$. ($u_j$ may also be removed from $U$ when action $A$ equals the current action associated with $F$ in the classifier.)
F: a symbol for a filter representing a d-tuple (F[1], F[2], ..., F[d]), where F[i] is a range specified for an attribute in the packet header such as destination address, source address, source port range, etc. When there are multiple filters, they are represented as F1, F2, ...

f: a tuple constructed using destination address, source address, source port range, etc. values from packet header.

A: action corresponding to a filter F. Similarly, A0, A1, A2, ...

newA, newA’: new action

(P, H): forwarding table rule, where P is a prefix and H is the next hop for P.

LMP: longest matching prefix

HPM: highest priority matching rule

U: an update sequence

u: an operation, which could be an insert, delete or change that is a part of U.

V: a reduced update sequence.

v: an operation that is part of the reduced sequence V.

S: a reduced and batch consistent update sequence. In another instance it is used to represent an arbitrary sequence. S is also used to represent the overlapping portion of two filters.

s: an operation that is part of sequence S.

r: number of operations in original update sequence.

m: number of operations in reduced update sequence.

insert(F, A): insert operation, represented also as I, I1, I2 etc.

delete(F): delete operation, represented also as D, D1, D2, etc.

change(F, newA): change operation, represented also as C, C1, C2, etc.

T0, R: packet classifier

Ti(U): packet classifier obtained after applying i operations from update sequence U, starting from the first operation.

action(f, R): action corresponding to the highest priority matching rule for filter f in packet classifier R.

priority(F, A): priority of rule (F, A) in the packet classifier.

|T0|: number of rules in a classifier table initially.

|Ti(S)|: number of rules in classifier table after i updates from sequence S has been applied.

max0≤i≤m|Ti(S)|: maximum increase in rule table size as each update numbered 0 through m in the batch update sequence S is applied on the classifier table.

optB(T0, U): maximum increase in rule table size as all the updates numbered 0 through m in an optimal batch consistent update sequence S (corresponding to original update sequence U) are applied sequentially to the classifier table.

optI(T0, U): maximum increase in rule table size as all the updates numbered 0 through m in an optimal incremental consistent update sequence corresponding to U are applied sequentially to the classifier table.

#inserts(U), #deletes(U): Number of inserts and deletes, in the update sequence U.

G: precedence graph for the update sequence V.

E(G): set of directed edges of G.

Q: set of pairs of (a, b) values corresponding to different update sequences.

σ(Q): permutation of (a, b) pairs.

B(i): Sum of b values in a permutation σ(Q) of (a, b) pairs.

A(i): Sum of b values till the (i-1) pair and the a value for the ith pair.

Δ: Increase in table size as two sequences are merged.

Figure 4-3. Notation used in this chapter
4. If \( u_i = \text{insert}(F, A) \) and \( u_j = \text{change}(F, \text{new}A) \), remove \( u_i \) from \( U \) and replace \( u_j \) by \( \text{insert}(F, \text{new}A) \).

5. If \( u_i = \text{insert}(F, A) \) and \( u_j = \text{delete}(F) \), remove \( u_i \) and \( u_j \) from \( U \).

Note that the remaining four possibilities for \( u_i \) and \( u_j \) ((change, insert), (delete, change), (delete, delete), and (insert, insert)) are invalid.

For example, the reduction of \( U = \text{insert}(F_1, A_1), \text{insert}(F_2, A_2), \text{delete}(F_1, A_1) \) is \( V = \text{insert}(F_2, A_2) \). It is easy to see that a field tuple \( F \) may be associated with at most one operation in the reduction of \( U \).

**Definition 8.** Let \( U = u_1, \ldots, u_r \) be an update sequence; each \( u_i \) is an insert, delete, or change operation. Let \( T_0 \) be a packet classifier and let \( T_i(U) \) be the state of this classifier after the operations \( u_1, \ldots, u_i \) have been performed, in this order, on \( T_0 \).

Let \( T_0(U) = T_0 \) and let \( \text{action}(f, R) \) be the action associated with the highest priority matching rule for field tuple \( f \) in packet classifier \( R \). Let \( S = s_1, \ldots, s_m \) be another
**update sequence.** 

_S is **batch consistent** with respect to \( T_0 \) and \( U \) iff_

\[
T_r(U) = T_m(S) \land \forall i \forall f [action(f, T_i(S))] \in \{ action(f, T_0), \\
action(f, T_r(U)) \}
\]

_S is **incremental consistent** with respect to \( T_0 \) and \( U \) iff_

\[
T_r(U) = T_m(S) \land \forall i \forall f [action(f, T_i(S))] \in \{ action(f, T_0), \\
\cdots, action(f, T_r(U)) \}
\]

Note that two tables are equal iff they contain exactly the same rules. Further, although \( U \) is always incremental consistent with respect to itself, it is generally not batch consistent with respect to itself (i.e., \( S = U \)) and a table \( T_0 \). For example, suppose \( U = insert(F1, A1), delete(F1, A1) \) and \( T_0 = \{(*, A0)\} \), where ‘*’ stands for the default rule containing * for all fields and which therefore any field tuple would match. Further, let \( A0 \neq A1 \) and \( priority(F1, A1) > priority(*, A0) \). Then, \( T_2(U) = T_0 \) and \( action(f, T_0) = action(f, T_2) = A0 \) for all \( f \). However, even though \( T_r(U) = T_m(S), action(f, T_1(S)) = A1 \neq A0 \) for every \( f \) that matches classifier rule \( (F1, A1) \).

Note that the reduced update sequence \( V(U) \) may be neither batch nor incremental consistent with respect to \( U \) and \( T_0 \). For example, suppose that \( T_0 = \{ (*, H0) \} \) and \( U = insert(00*, H1), insert(0*, H2), insert(000*, H3), delete(00*) \). Figure 4-5(a) shows \( T_0, T_1(U), \cdots, T_4(U) \). Batch consistency requires the next hops for destination addresses matched by 000* to be in set \( Hb = \{ H0, H3 \} \) while incremental consistency requires these next hops to be in set \( Hi = \{ H0, H1, H3 \} \) as illustrated in Figure 4-5(a). Figure 4-5(b) shows \( T_1(V) \) and \( T_2(V) \) for the reduced sequence \( V(U) = insert(0*, H2), insert(000*, H3) \). We see that \( nextHop(d, T_1(V)) = H2 \) and \( H2 \notin Hb, H2 \notin Hi \) for addresses \( d \) matched by 000*. So, \( V(U) \) is neither batch nor incremental consistent with respect to \( U \) and \( T_0 \).
Theorem 4.1 establishes the existence of a batch consistent update sequence for every classifier $T_0$ and every update sequence $U$. Note that the existence of an incremental consistent update sequence follows from our earlier observation that $U$ is incremental consistent with respect to itself and every $T_0$. This follows also from Theorem 4.1 as every batch consistent update sequence is incremental consistent as well. An incremental consistent sequence may not, however, be batch consistent.

**Theorem 4.1.** For every classifier $T_0$ and update sequence $U$, there exists a batch consistent update sequence $S$.

**Proof.** Let $S = s_1, \cdots, s_m$ be derived from $U = u_1, \cdots, u_r$ as below.

**Step 1** Let $V = v_1, \cdots, v_m$ be the reduction of $U$.

**Step 2** Reorder the operations of $V$ so that the inserts are at the front and in decreasing order of priority; followed by the change operations in any order;

Figure 4-5. Actual and reduced update sequences
followed by the deletes in increasing order of priority. Call the resulting sequence $S$.

We shall show that $S$ is batch consistent with respect to $U$ and every $T_0$. First, it is easy to see that $T_r(U) = T_m(V) = T_m(S)$. So, we need only to show

$$\forall i \forall f [\text{action}(f, T_i(S))] \in \{ \text{action}(f, T_0), \text{action}(f, T_m(S)) \}$$

The proof is by induction on $i$. For the induction base, $i = 0$, $T_0(S) = T_0$, and so

$$\forall f [\text{action}(f, T_0(S)) = \text{action}(f, T_0)].$$

For the induction hypothesis, assume that

$$\forall f [\text{action}(f, T_j(S)) \in \{ \text{action}(f, T_0), \text{action}(f, T_m(S)) \}]$$

for some $j$, $0 \leq j < m$. In the induction step, we show

$$\forall f [\text{action}(f, T_{j+1}(S)) \in \{ \text{action}(f, T_0), \text{action}(f, T_m(S)) \}] \quad (4-1)$$

If $\forall f [\text{action}(f, T_{j+1}(S)) = \text{action}(f, T_j(S))]$, then Equation 4–1 follows from the induction hypothesis. So, suppose there is an $f$ such that

$$\text{action}(f, T_{j+1}(S)) \neq \text{action}(f, T_j(S)) \quad (4–2)$$

There are three cases to consider for $s_{j+1} = \text{insert}(F, A)$, $\text{change}(F, \text{new}A)$, and $\text{delete}(F)$. When $s_{j+1} = \text{insert}(F, A)$ or $s_{j+1} = \text{change}(F, \text{new}A)$, it must be that $\text{HPM}(f, T_{j+1}(S)) = F$ (assuming overlapping rules have different priority). Because of the ordering of operations in $S$, the remaining operations $s_{j+2}, \ldots$ do not change either the highest priority matching rule for $f$ or the action associated with this matching rule. So,

$$\text{HPM}(f, T_m(S)) = F \text{ and } \text{action}(f, T_{j+1}(S)) = \text{action}(f, T_m(S)) \text{ (see Figure 4-6).}$$

When $s_{j+1} = \text{delete}(F)$, it must be that $\text{HPM}(f, T_{j+1}(S)) = F'$, where $F'$ is the highest priority tuple in $T_{j+1}(S)$ that matches $f$. Note that $\text{priority}(F') < \text{priority}(F)$. Because of the ordering of operations in $S$, the remaining operations $s_{j+2}, \ldots$ do not change either the
highest priority matching rule for \( f \) or the action associated with this matching rule. So,
\[
HPM(f, T_m(S)) = HPM(f, T_{j+1}(S)) = F' \text{ and } action(f, T_{j+1}(S)) = action(f, T_m(S)).
\]

Figure 4-6. \( T_{j+1} \) for a forwarding table. \( F \) is newly inserted/changed prefix, \( f \) is the destination address on a packet.

From the proof of Theorem 4.1, we see that for an update sequence to be batch consistent, it is necessary only that whenever an operation changes the action for a tuple \( f \), that change be reflected to the action in the final table \( T_r(U) \). Using this observation, it is possible to construct additional batch consistent update sequences. For example, we could partition the operations in the reduction of \( U \) so that the field tuples in one partition are disjoint from those covered by other partitions. Then, for each partition, we can order the operations as in the construction of Theorem 4.1 and concatenate these orderings to obtain a batch consistent update sequence.

Different batch consistent update sequences may result in intermediate router tables of different size. As an example, consider the reduced update sequence of Figure 4-7 for a forwarding table. The outermost prefix and all prefixes marked D are in \( T_0 \). Those marked I are prefixes that are to be inserted and those marked D are to be deleted. The batch consistent sequence constructed in the proof of Theorem 4.1 will do all the inserts first and then the deletes. If there are \( a \) inserts, the table size increases by \( a \) following the last insert and then decreases back to the size of \( T_0 \) as the deletes complete. For the update sequence to succeed (when inserts/deletes are done incrementally, i.e., one at a time), we must have \( a \) units of additional table capacity. An alternative batch consistent update sequence follows each insert with the deletion of its enclosed prefix that is labeled D in the figure. Using this sequence, only one additional unit of table
capacity is required and this is optimal for the given example as it is not possible to do a delete before its enclosing insert and maintain batch consistency (unless, of course, the table is locked for lookup from the start of a delete to the completion of its enclosing insert).

![Figure 4-7. A bad example for sequence of Theorem 4.1](image)

We next prove that a reduced sequence obtained using the Definition 7 has the smallest number of operations and it is not possible to reduce it further.

**Theorem 4.2.** For every classifier $T_0$ and update sequence $U = u_1, \cdots, u_r$, the reduced sequence $V(U)$ has the smallest number of operations needed to transform $T_0$ to $T_r(U)$.

**Proof.** Consider any sequence $S$ that transforms $T_0$ to $T_r(U)$. Let $V(S)$ be the reduction of $S$. Clearly, $|V(S)| \leq |S|$ and $V(S)$ also transforms $T_0$ to $T_r$. We shall show that $|V(U)| \leq |V(S)|$, thereby proving the theorem.

Consider any $v_i \in V(U)$. If $v_i = \text{insert}(F, A)$, then $F \not\in T_0$ (follows from the correctness of $U$ with respect to $T_0$ and the definition of $V(U)$) and $F \in T_r(U)$. Consequently, $\text{insert}(F, A)' \in V(S)$. Since $F$ appears only once in $V(S)$, $A' = A$.

So, $v_i \in V(S)$. If $v_i = \text{delete}(F)$, then $F \in T_0$ and $F \not\in T_r(U)$. So, $\text{delete}(F) \in V(S)$.

Similarly, when $v_i = \text{change}(F, \text{new}A)$, $v_i \in V(S)$. So, $V(U) \subseteq V(S)$ and $|V(U)| \leq |V(S)|$.

**Definition 9.** A batch (incremental) consistent sequence $S = s_1, \cdots, s_m$ for $U$ and $T_0$ is optimal iff it minimizes $\max_{0 \leq i \leq m}(|T_i(S)|)$ relative to all batch (incremental) consistent sequences.
We next present a theorem that establishes a relationship on the maximum growth of table size as a batch consistent update sequence is applied and as an incrementally consistent update sequence is applied on rule table $T_0$.

**Theorem 4.3.** Let $S$ be an optimal batch consistent sequence for $U$ and $T_0$. Let $\text{optB}(T_0, U) = \max_{0 \leq i \leq m}\{|T_i(S)|\}$. Let $\text{optI}(T_0, U)$ be the corresponding quantity for an optimal incremental consistent sequence. $\text{optB}(T_0, U) - m/2 \leq \text{optI}(T_0, U) \leq \text{optB}(T_0, U)$ and both upper and lower bounds on $\text{optI}(T_0, U)$ are tight.

**Proof.** $\text{optI}(T_0, U) \leq \text{optB}(T_0, U)$ follows from the observation that every batch consistent sequence also is incremental consistent. To see that this is a tight upper bound, consider the case when $U$ is comprised only of change (or only of insert or only delete) operations. Now, $\text{optI}(T_0, U) = \text{optB}(T_0, U)$.

To establish $\text{optB}(T_0, U) - m/2 \leq \text{optI}(T_0, U)$, we note that $\text{optI}(T_0, U) \geq |T_0| + \max\{0, \#\text{inserts}(U) - \#\text{deletes}(U)\}$ and $\text{optB}(T_0, U) \leq |T_0| + \#\text{inserts}(U)$. So, $\text{optB}(T_0, U) - \text{optI}(T_0, U)$ is maximum when $\#\text{inserts}(U) = \#\text{deletes}(U)$. Since, $\#\text{inserts}(U) + \#\text{deletes}(U) \leq m$, $\text{optB}(T_0, U) - \text{optI}(T_0, U)$ is maximum when $\#\text{inserts}(U) = \#\text{deletes}(U) = m/2$. At this time, $\text{optB}(T_0, U) - \text{optI}(T_0, U) = m/2$. Hence, $\text{optB}(T_0, U) - m/2 \leq \text{optI}(T_0, U)$. For the tightness of this bound, consider a sequence $U$ of $m/2$ deletes followed by $m/2$ inserts as in Figure 4-8 (one delete that encloses $m/2$ inserts and $m/2-1$ deletes all of which are independent). Since $U$ is incremental consistent with itself, $\text{optI}(T_0, U) = |T_0|$. Batch consistency limits us to permutations of $U$ in which the inserts precede all the deletes. So, $\text{optB}(T_0, U) = |T_0| + m/2$. Hence, $\text{optB}(T_0, U) - m/2 = \text{optI}(T_0, U)$. \[\square\]
4.2.2 Batch Consistent Sequences

When performing the updates \( U = u_1, \ldots, u_r \) in a batch consistent manner, our primary objective is to perform the fewest possible inserts/deletes/changes to transform \( T_0 \) to \( T_r \) and our secondary objective is to perform these fewest updates in a batch consistent order that minimizes the maximum size of an intermediate table. The primary objective is met by using the reduction \( V(U) \) of \( U \) (Theorem 4.2). For the secondary objective, we construct a precedence graph. Section 4.2.2.1 shows how a batch consistent update sequence can be obtained from a precedence graph and in Section 4.2.2.2 we describe a heuristic for producing a batch consistent update sequence that results in near-optimal growth in the size of the intermediate rule table.

4.2.2.1 Graph representation of updates

We construct an \( m \) vertex digraph \( G \) from \( V = v_1, \ldots, v_m \). Vertex \( i \) of \( G \) represents the update operation \( v_i \). Let \((F_i, A_i)\) be the rule associated with update \( v_i \), \( 1 \leq i \leq m \). There is a directed edge between vertices \( i \) and \( j \) iff (a) all fields in tuples \( F_i \) and \( F_j \) overlap, that is \( F_i \cap F_j = S, S \neq \emptyset \), where \( S \) is a tuple built from fields representing overlapping regions of \( F_i \) and \( F_j \), (b) there is no rule \((F_k, A_k)\) such that \( F_k \cap S \neq \emptyset \) and priority of \((F_k, A_k)\) lies between those of rules \((F_i, A_i)\) and \((F_j, A_j)\), and (c) one of the following relationships between \( v_i \) and \( v_j \) hold good, assuming without loss of generality that \( \text{priority}(F_i, A_i) > \text{priority}(F_j, A_j) \).

1. \( v_i \) and \( v_j \) are inserts \( \Rightarrow (i, j) \in E(G) \), where \( E(G) \) is the set of directed edges of \( G \).
2. \( v_i \) is an insert and \( v_j \) is a delete \( \Rightarrow (i, j) \in E(G) \).
3. \( v_i \) and \( v_j \) are deletes \( \Rightarrow (j, i) \in E(G) \).
4. \( v_i \) is a delete and \( v_j \) is an insert \( \Rightarrow (j, i) \in E(G) \).
5. \( v_i \) is an insert and \( v_j \) is a change \( \Rightarrow (i, j) \in E(G) \).
6. \( v_i \) is a delete and \( v_j \) is a change \( \Rightarrow (j, i) \in E(G) \).
Definition 10. *i* is an immediate predecessor of *j* in *G* iff \((i, j) \in E(G)\). *i* is a predecessor of *j* iff there is a directed path from *i* to *j* in *G*.

We assign a weight of 1, 0, or −1 to vertex *i* of precedence graph *G* depending on whether *v* is an insert, change, or delete. Figure 4-9 gives an example reduced update set *V* and its corresponding digraph *G*. One may verify that a permutation of a reduced update set *V* is batch consistent iff it corresponds to a topological ordering of the vertices of *G*. Further, for every topological ordering, \(|T_i| - |T_0|\) equals the sum of the weights of the first *i* vertices in the ordering.

Definition 11. For a given topological order, we define \(w_i\) to be the sum of the weights of the first *i* vertices. The max weight of a topological order is \(\max\{w_i\}\). An optimal topological ordering is a topological ordering that has minimum max weight.

Although we have been unable to develop an efficient algorithm to find an optimal topological order, we propose an efficient heuristic (Figure 4-10). Notice that our secondary objective is met by an optimal topological ordering.

4.2.2.2 Constructing a near-optimal batch consistent sequence

We propose an efficient heuristic in Figure 4-10 in which a topological order is constructed in several rounds. In each round, one of the remaining deletes is selected to be the next delete in the topological order being constructed. In case no delete remains in *G*, any topological ordering of the remaining vertices may be concatenated to the ordering so far constructed to complete the overall topological order. Assume at
Algorithm: computeOrder(G)
create candidateDeletes from G, by listing all the delete vertices in G that have no predecessor deletes.
for each delete vertex \( d \in \text{candidateDeletes} \)
    compute \((a(d), b(d))\) where
    \( a(d) = \) number of insert predecessors of \( d \);
    \( b(d) = a - 1 - \) number of delete successors of \( d \) without
    insert predecessors;
endfor
nextDelete = NULL;
while there is a delete in candidateDeletes
    if there is only one delete \( d \) in candidateDeletes
        nextDelete = \( d \);
    else
        \( \text{b}_{\text{smallest}} = \) smallest \( b \) among the \( b \) values for all deletes
        in candidateDeletes.
        if \( \text{b}_{\text{smallest}} < 0 \) then
            nextDelete = \( d \in \text{candidateDeletes} \) s.t. \( b(d)<0 \) and
            \( a(d) \) is smallest
        else if \( \text{b}_{\text{smallest}} == 0 \) then
            nextDelete = \( d \in \text{candidateDeletes} \) s.t. \( b(d)==0 \)
        else
            nextDelete = \( d \in \text{candidateDeletes} \) s.t. \( a(d)-b(d) \) is largest
        endif
    endif
append(nextDelete); // append to topological order and delete appended vertices from G.
if any new deletes are found whose predecessor deletes have
been appended and hence they have no remaining
predecessor deletes then
    compute (a, b) values for all such freed deletes and
    add them to candidateDeletes.
endif
endwhile
concatenate the remaining vertices in any topological ordering

Figure 4-10. Algorithm to compute a near-optimal topological order

least one delete remains in \( G \). Only deletes that remain in \( G \) and that have no delete
predecessors are candidates for the next delete. Each candidate delete \( d \) is assigned an
\((a, b)\) value where \( a \) is the number of insert predecessors of \( d \) and \( b = a - \) the number of
delete successors of \( d \) (including \( d \)) that have no insert predecessors that are not also
predecessors of \( d \). From the candidate deletes, we select one using the following rule.

1. If the least \( b \) is less than 0, from among the candidate deletes that have negative \( b \),
   select one with least \( a \).

2. If the least \( b \) equals 0, select any one of the candidate deletes that have \( b = 0 \).
3. If the least \( b \) is more than 0, from among the candidate deletes, select one with largest \( a - b \).

Once the next delete for the topological ordering is selected, we concatenate its remaining predecessor inserts and changes (these inserts and changes are first put into topological order) to the topological ordering being constructed followed by the selected delete \( d \) followed (in topological order) by the delete successors (and the remaining change predecessors of these delete successors) of \( d \) that have no remaining insert predecessors. All newly added vertices to the topological ordering being constructed (together with incident edges) are deleted from \( G \) before commencing the next round selection. Our heuristic is motivated by the following two theorems.

**Theorem 4.4.** For every \( G \), there exists an optimal topological ordering in which between any two successive deletes \( d_i \) and \( d_{i+1} \) we have only the predecessor inserts and changes of \( d_{i+1} \) that are not predecessor inserts and changes of any of the deletes \( d_1, \ldots, d_i \). Here \( d_1, \ldots \) are the deletes of \( V \) indexed in the order they appear in the topological ordering.

**Proof.** Consider an optimal topological ordering of \( G \). Examine the deletes left to right. Let \( d_{i+1} \) be the first delete such that there is an insert or change between \( d_i \) and \( d_{i+1} \) that is not a predecessor of \( d_{i+1} \). All inserts and changes between \( d_i \) and \( d_{i+1} \) that are not predecessors of \( d_{i+1} \) may be moved from their present location in the topological ordering (without changing their relative ordering) to just after \( d_{i+1} \). This relocation of the inserts and changes yields a new topological ordering that also is optimal. Repeating this transformation a finite number of times results in an optimal topological ordering that satisfies the theorem.

For the second theorem that motivates our heuristic, let \( S \) be a sequence of inserts and deletes to be performed (in the given order) on a forwarding table. The \( a \) value of the sequence \( S \) is the maximum increase in table size when the sequence of inserts and deletes is done in the given order and \( b \) is the increase in table size following
the last insert/delete in $S$. For example, when $S = I_1 I_2 D_1 I_3 D_2 D_3$, $a = 2$ and $b = 0$.

Suppose we are given $n$ sequences $S_1, S_2, \cdots, S_n$ of inserts and deletes and we wish to concatenate these into a single sequence. Every permutation of $S_1, S_2, \cdots, S_n$ defines a legal concatenation. However, different permutations have different $(a, b)$ values. For example, when $n = 2$, $S_1 = I_1 I_2 I_3 I_4 D_1 D_2 D_3 D_4$ and $S_2 = I_5 I_6 D_5 D_6 D_7 D_8$, the permissible concatenations/permissions are $S_1 S_2$ and $S_2 S_1$. The $(a, b)$ values for $S_1, S_2, S_1 S_2$ and $S_2 S_1$ are, respectively, $(4, 0), (2, -2), (4, -2)$ and $(2, -2)$. The permutation $S_2 S_1$ result in the smallest increase in table size and is therefore the optimal permutation.

We introduce the following notation:

1. $Q = \{(a_1, b_1), (a_2, b_2), \cdots, (a_n, b_n)\}$ is a set of $(a, b)$ values corresponding to $n$ update sequences $S_1, S_2, \cdots, S_n$.

2. $\sigma(Q)$ is a permutation of the pairs of $Q$ and $\sigma(Q, i)$ is the $i$th pair in this permutation. For simplicity, $\sigma(Q)$ and $\sigma(Q, i)$ will be abbreviated to $\sigma$ and $\sigma(i)$.

3. $B(\sigma(Q), i) = \sum_{j=1}^{i} b_{\sigma(j)}$, $B(\sigma(Q), i)$ will be abbreviated to $B(i)$. Note that $B(i)$ is the sum of the second coordinates (or $b$ values) of the first $i$ pairs of $\sigma$.

4. $A(\sigma(Q), i) = \max_{1 \leq j \leq i} \{B(j - 1) + a_{\sigma(j)}\}$ and is abbreviated $A(i)$.

$\sigma(Q)$ is an optimal permutation iff it minimizes $A(n)$. Next, we state and prove a theorem to construct an optimal permutation of a collection of update sequences.

**Theorem 4.5.** Let $\sigma(Q)$ be such that:

1. The pairs with negative $b$ come first followed by those with zero $b$ followed by those with positive $b$.

2. The pairs with negative $b$ are in increasing (non-decreasing) order of $a$.

3. The pairs with zero $b$ are in any order.

4. The pairs with positive $b$ are in decreasing (non-increasing) order of $a - b$.

$\sigma(Q)$ is an optimal sequence.

**Proof.** First, we show that permutations that violate one of the listed conditions cannot have a smaller $A(n)$ than those that satisfy all conditions. Consider a permutation that
does not satisfy the conditions of the theorem. Suppose that the first violation of these conditions is at position \(i\) of the permutation (i.e., pairs \(i\) and \(i + 1\) of the permutation violate one of the conditions). Let \((a_i, b_i)\) be the \(i\)th pair and \((a_{i+1}, b_{i+1})\) be the \(i + 1\)st pair. Let \(\Delta = \max\{a_i, b_i + a_{i+1}\}\) and \(\Delta' = \max\{a_{i+1}, b_{i+1} + a_i\}\). We shall show that \(\Delta' \leq \Delta\). This together with the observation that \(A(i+1) = \max\{A(i-1), B(i-1) + a_i, B(i-1) + b_i + a_{i+1}\}\) imply that swapping the pairs \(i\) and \(i + 1\) does not increase \(A(i + 1)\). By repeatedly performing these violation swaps a finite number of times, we obtain a permutation that satisfies the conditions of the theorem and that has an \(A(n)\) value no larger than that of the original permutation. Hence, a permutation that violates a listed condition cannot have a smaller \(A(n)\) than one that satisfies all conditions.

To show \(\Delta' \leq \Delta\), we consider the four possible cases for a violation of the conditions of the theorem—(a) \(b_i \geq 0\) and \(b_{i+1} < 0\) (violation of condition 1), (b) \(b_i > 0\) and \(b_{i+1} = 0\) (violation of condition 1), (c) \(a_i > a_{i+1}, b_i < 0,\) and \(b_{i+1} < 0\) (violation of condition 2), and (d) \(a_i - b_i < a_{i+1} - b_{i+1}, b_i > 0,\) and \(b_{i+1} > 0\) (violation of condition 4). Note that condition 3 cannot be violated as this condition permits arbitrary ordering of pairs with zero \(b\). In fact, we see that when \(b_i = b_{i+1} = 0, \Delta = \Delta' = \max\{a_i, a_{i+1}\}\) and swapping the pairs \(i\) and \(i + 1\) does not affect \(A(i + 1)\).

**Case (a):** We see that \(b_i + a_{i+1} \geq a_{i+1} \) and \(b_{i+1} + a_i < a_i\). So, \(\Delta' \leq \Delta\).

**Case (b):** Now, \(\Delta' = \max\{a_{i+1}, b_{i+1} + a_i\} \leq \max\{a_{i+1}, a_i\} = \Delta\).

**Case (c):** Now, \(\Delta' < a_i = \Delta\).

**Case (d):** From \(a_i - b_i < a_{i+1} - b_{i+1}\), it follows that \(a_i + b_{i+1} < b_i + a_{i+1}\). From this and \(b_{i+1} > 0\), we get \(a_i < b_i + a_{i+1}\). Hence, \(\Delta = b_i + a_{i+1}\). If \(a_{i+1} \geq b_{i+1} + a_i\), \(\Delta' = a_{i+1} < \Delta\). If \(a_{i+1} < b_{i+1} + a_i\), \(\Delta' = b_{i+1} + a_i < b_i + a_{i+1} = \Delta\).

To complete the proof, we need to show that all \(\sigma\)s that satisfy the conditions of the theorem have the same value of \(A(n)\). Specifically, we need to show that \(\Delta = \Delta'\) whenever (c') \(a_i = a_{i+1}, b_i < 0,\) and \(b_{i+1} < 0\) (tie in condition 2), and (d') \(a_i - b_i = a_{i+1} - b_{i+1}, b_i > 0,\) and \(b_{i+1} > 0\) (tie in condition 4). This is easily established. \(\square\)
Since the vertex weights in $G$ are 1, 0, and $-1$, the max weight of a topological ordering cannot exceed $m$, the number of vertices in $G$ and cannot be less than $-1$. The maximum number of table entries occurs, for example, when all $v_i$ are inserts and the minimum happens, for example, when all $v_i$ are deletes. So, a topological ordering may have a max weight that exceeds the minimum max weight by $O(m)$. Our heuristic may produce topological orderings whose max weight is $\Omega(m)$ more than that of the optimal ordering. For example, consider the digraph of Figure 4-11 that has two components. The first component is comprised of a delete $d_1$ that has $m/3 - 2$

![Diagram of the first component](image1)

and $m/3 - 4$ immediate successor deletes. Deletes $d_1$ and $d_2$ are the candidate deletes during the first round of our heuristic. Their $(a, b)$ values are $(m/3 - 2, -1)$ and $(2, 1)$, respectively.

The second component has a delete $d_2$ that has 2 immediate predecessor inserts and a successor delete $d_3$ that also has 2 immediate predecessor inserts and $m/3 - 1$ immediate successor deletes. Deletes $d_1$ and $d_2$ are the candidate deletes during the first round of our heuristic. Their $(a, b)$ values are $(m/3 - 2, -1)$ and $(2, 1)$, respectively.
Delete $d_1$ is selected by our heuristic and the partial topological ordering constructed has $m/3 - 2$ inserts followed by $m/3 - 3$ deletes. In the next round $d_2$ preceded by its 2 predecessor inserts is added to the ordering. Finally, in the third round, $d_3$ preceded by its 2 predecessor inserts and followed by its $m/3 - 1$ successor deletes is added. The max weight of the constructed topological ordering is $m/3 - 2$ (assume $m/3 \geq 4$). In an optimal ordering, the first component appears after the second and the max weight is 3. So, the heuristic ordering has a max weight that is $m/3 - 5 = O(m)$ more than optimal.

![Diagram of a delete star]

Figure 4-12. A delete star

Whenever each component of $G$ is a delete star as in Figure 4-12, our heuristic finds an optimal ordering. Note that in a delete star, we have a delete vertex all of whose predecessors are inserts and/or changes and all of whose successors are deletes that have no additional predecessor inserts. This follows from Theorem 4.5 and the observation that each component has only one delete that ever becomes a candidate for selection by our heuristic. In general, whenever no component of $G$ has two deletes that become candidates for selection, our heuristic obtains an optimal topological ordering. Although this property doesn’t hold for the $G$ that arise in practice, the $G$ that arise in practice have a sufficiently simple structure that our heuristic obtains optimal topological orderings. Figure 4-13 shows some of the more complex components in the $Gs$ of trace update data for forwarding tables.

4.2.3 Incremental Consistent Sequences

When performing the updates $U = u_1, \cdots, u_r$ in an incremental consistent manner, the primary and secondary objectives are the same as those for batch consistency. The
primary objective is to perform the fewest possible inserts/deletes/changes to transform $T_0$ to $T_r$. The secondary objective is to perform these fewest updates in an incremental consistent order that minimizes the maximum size of an intermediate table. The primary objective is met by using the reduction $V(U)$ of $U$ (Theorem 4.2). Note that since $V(U)$ has a batch consistent ordering, it has also an incremental consistent ordering. For the secondary objective, however, there is no digraph $H(V)$ whose topological orderings correspond to the permissible incremental consistent orderings of $V(U)$. To see this, consider a forwarding table $T_0 = \{(*, H0), (00\ast, H1)\}$ and $U = u_1, u_2, u_3 = delete(00\ast), insert(000\ast, H2), insert(0\ast, H3)$. For this example, $V(U) = U$ and the incremental consistent orderings are $u_1u_2u_3$, $u_2u_1u_3$, $u_2u_3u_1$, and $u_3u_2u_1$. The remaining two orderings $u_1u_3u_2$ and $u_3u_1u_2$ are not incremental consistent. To see that $u_1u_3u_2$, for example, is not incremental consistent, note that following $u_3$, the next hop for destination addresses that are of the form $000\ast$ is $H3$ whereas in the original ordering $u_1u_2u_3$ these destination addresses have next hop $H1$ initially, $H0$ following $u_1$, and $H2$ following both $u_2$ and $u_3$. 

Figure 4-13. Some of the complex digraph components of the trace update data.
Any $H(V)$ that disallows the topological ordering $u_1u_3u_2$ must have at least one of the directed edges $(u_3, u_1)$, $(u_2, u_1)$, and $(u_2, u_3)$. However, the presence of any one of these edges in $H(V)$ also invalidates one of the four permissible orderings. So, no $H(V)$, whose topological orderings coincide exactly with the set of permissible orderings, exists.

We can formulate meta-heuristic (e.g., simulated annealing, genetic, etc.) based algorithms to determine near optimal incremental consistent orderings of $V(U)$. However, given the rather simple relationships that exist in our test update sequences, it is unlikely these will provide incremental orderings that are significantly superior to the batch consistent orderings generated by our algorithm of Figure 4-10.

4.3 Experiments

The benchmarks used for the experiments are described in Section 4.3.1 and the results are presented in Section 4.3.2.

4.3.1 Benchmarks

We have applied our heuristic to obtain near optimal batch consistent sequences on two sets of benchmarks. The first set of benchmarks consist of 21 datasets derived from BGP update sequences of various routers [6, 37]. Table 4-1 gives the details of these datasets, with the first and second columns showing the name and the total number of prefixes in the initial forwarding table for each dataset. The third column gives the period for which the data has been collected. Columns four to seven, respectively, give the number of “insert”, “delete” and “change” operations, and the total number of operations for each dataset. All the datasets except rrc00Jan25 were collected starting from the zeroth hour on February 1, 2009. The last one, rrc00Jan25, is for the 3 hour period from 5:30am to 8:30am on January 25, 2003, which corresponds to the SQL Slammer worm attack [17].

The “insert”, “delete” and “change” operations in Table 4-1 are derived from BGP update messages received by the router. BGP is essentially an incremental protocol in
which a router generates update messages only when there is a change in its routing state. Such a change could be in the network topology (for example when a router fails or comes up after failure or is added to the network) or in the routing policy. The BGP update messages consisting of route announcements and withdrawals are sent over semi-permanent TCP connections to the neighboring routers.

A route announcement advertises a new route to a prefix. Upon receiving an announcement, a router compares the newly advertised route with the existing ones for the same prefix in the routing information base (RIB or routing table). If there are no existing routes then a new prefix is inserted to the forwarding table along with the next hop as IP address of the BGP peer from which the announcement was received. If, on the other hand, there are existing routes, then the new route is compared with the existing ones by applying the BGP selection rules. If the new route is superior to the best among the existing routes, then the rule corresponding to the route is changed in the forwarding table by changing the next hop to point to the BGP peer that sent the new route announcement. If the new route is inferior to the best existing route, then the announcement has no effect on the forwarding table and hence the routing policy. The new route is stored in the RIB in any case. We generated the “insert” and “change” next hop operations corresponding to route announcements in our experiments in keeping with the forwarding table update strategy of BGP.

A route withdrawal message, similarly, triggers a number of actions at a router. The router removes the route from the RIB, and then checks if there are existing routes from different peers to the same prefix. If there are no such routes, then the forwarding rule for the route is deleted from the forwarding table. On the other hand, if there are more routes and the withdrawn route was the best among them, then the next best route is picked from the remaining routes by applying the BGP selection rules and the forwarding table is updated by changing appropriately the next hop of the rule corresponding to the route. Otherwise, if the withdrawn route is not the best among the existing routes, then
the forwarding table is left unchanged. Just as for route announcements, we generated the “delete” and “change” next hop operations corresponding to the route withdrawals in line with the BGP update strategy for forwarding tables. Therefore, some of the withdrawals may not lead to any operations, just as some of the announcements.

We note that the IETF recommends that BGP routers use minRouteAdver with a timer value of 30 seconds. This, together with route flap dampening mechanisms will eliminate redundancies in BGP update clusters. For update sequences that are guaranteed to be free of redundancies, the redundancy removal step of Figure 4-4 may be eliminated.

Note that our algorithm changes the order in which updates (received in a batch) are applied to the forwarding table. Thus, the forwarding table is the only entity that is affected, which in turn, affects packet forwarding. Our algorithms do not affect the routing table (RIB) or any BGP message being sent out from the router. Thus, our algorithms do not affect the various nuances of BGP including the BGP convergence time.

The second set of benchmarks consist of 24 synthetic classifiers generated using ClassBench [50] as shown in Table 4-2. The first column in this figure presents the names of the classifiers, the second column shows the number of rules in each of the classifiers, and columns three to six give the number of inserts, deletes, and change operations in the update traces as well as the total number of update operations for each dataset. We used 12 seed files based on access control lists (acl), firewalls (fw) and IP chains (ipc) to generate 24 classifiers. Each rule consists of the fields: source address, destination address, source port range, destination port range, protocol. We created an update trace from the classifiers by marking rules for insertion/deletion/change randomly, and later removing the rules marked for insertion. The corresponding insert, delete and change operations are shuffled and then written to the update file.
Table 4-1. Datasets used in the experiments

<table>
<thead>
<tr>
<th>DataSet</th>
<th>#Prefixes</th>
<th>Time (hours)</th>
<th>#Inserts</th>
<th>#Deletes</th>
<th>#Changes</th>
<th>#Total-updates</th>
</tr>
</thead>
<tbody>
<tr>
<td>rrc00</td>
<td>294098</td>
<td>75.7</td>
<td>39553</td>
<td>40051</td>
<td>368013</td>
<td>447617</td>
</tr>
<tr>
<td>rrc01</td>
<td>276795</td>
<td>75.2</td>
<td>41692</td>
<td>41988</td>
<td>492315</td>
<td>575995</td>
</tr>
<tr>
<td>rrc03</td>
<td>283754</td>
<td>42.7</td>
<td>27702</td>
<td>27914</td>
<td>292454</td>
<td>348070</td>
</tr>
<tr>
<td>rrc04</td>
<td>288610</td>
<td>17.0</td>
<td>16086</td>
<td>15977</td>
<td>193392</td>
<td>225455</td>
</tr>
<tr>
<td>rrc05</td>
<td>280041</td>
<td>103.0</td>
<td>20276</td>
<td>18285</td>
<td>439647</td>
<td>478208</td>
</tr>
<tr>
<td>rrc06</td>
<td>278744</td>
<td>235.0</td>
<td>157549</td>
<td>157547</td>
<td>289272</td>
<td>604368</td>
</tr>
<tr>
<td>rrc07</td>
<td>275097</td>
<td>0.417</td>
<td>247</td>
<td>218</td>
<td>179835</td>
<td>180300</td>
</tr>
<tr>
<td>rrc10</td>
<td>278898</td>
<td>105.0</td>
<td>21620</td>
<td>22473</td>
<td>326720</td>
<td>370813</td>
</tr>
<tr>
<td>rrc11</td>
<td>277166</td>
<td>80.2</td>
<td>58115</td>
<td>58378</td>
<td>290621</td>
<td>407114</td>
</tr>
<tr>
<td>rrc12</td>
<td>278499</td>
<td>62.3</td>
<td>33196</td>
<td>33572</td>
<td>410464</td>
<td>477232</td>
</tr>
<tr>
<td>rrc13</td>
<td>284986</td>
<td>57.8</td>
<td>23920</td>
<td>23713</td>
<td>284710</td>
<td>332343</td>
</tr>
<tr>
<td>rrc14</td>
<td>276170</td>
<td>83.6</td>
<td>56598</td>
<td>56810</td>
<td>203955</td>
<td>317363</td>
</tr>
<tr>
<td>rrc15</td>
<td>284047</td>
<td>134.0</td>
<td>95790</td>
<td>93750</td>
<td>183131</td>
<td>372671</td>
</tr>
<tr>
<td>rrc16</td>
<td>282660</td>
<td>672.0</td>
<td>3338</td>
<td>937</td>
<td>8896</td>
<td>13171</td>
</tr>
<tr>
<td>rv2</td>
<td>294127</td>
<td>56.5</td>
<td>13882</td>
<td>15552</td>
<td>679100</td>
<td>708534</td>
</tr>
<tr>
<td>rv4</td>
<td>275737</td>
<td>95.0</td>
<td>69627</td>
<td>69754</td>
<td>526302</td>
<td>665683</td>
</tr>
<tr>
<td>rv.eqix</td>
<td>275736</td>
<td>70.3</td>
<td>51104</td>
<td>51066</td>
<td>253693</td>
<td>355863</td>
</tr>
<tr>
<td>rv.isc</td>
<td>281095</td>
<td>68.2</td>
<td>44286</td>
<td>44444</td>
<td>292323</td>
<td>381053</td>
</tr>
<tr>
<td>rv.linx</td>
<td>278196</td>
<td>49.1</td>
<td>23137</td>
<td>23413</td>
<td>384344</td>
<td>430894</td>
</tr>
<tr>
<td>rv.wide</td>
<td>283569</td>
<td>174.0</td>
<td>101821</td>
<td>103862</td>
<td>372035</td>
<td>577718</td>
</tr>
<tr>
<td>rrc00Jan25</td>
<td>228281</td>
<td>3.0</td>
<td>18275</td>
<td>19455</td>
<td>272115</td>
<td>309845</td>
</tr>
</tbody>
</table>

4.3.2 Results

The two following subsections present the results of applying our methodology (Figure 4-4) on forwarding tables as well as on the synthetically generated packet classifiers.

4.3.2.1 Forwarding tables

Table 4-3 gives the total number of update operations that remain after applying our reduction algorithm to each update batch of Table 4-1. All updates with the same timestamp define a batch. When reduction is applied to batches of updates

1 As noted earlier, BGP routers implement minRouteAdver timer and route flap dampening mechanisms. Thus although we see redundancies in our data set, ideally, BGP update sequences will not have redundancies
Table 4-2. Synthetic classifiers and update traces used in the experiments

<table>
<thead>
<tr>
<th>Data Set</th>
<th>#Rules</th>
<th>#Inserts</th>
<th>#Deletes</th>
<th>#Changes</th>
<th>#Total-updates</th>
</tr>
</thead>
<tbody>
<tr>
<td>acl1_100k</td>
<td>51522</td>
<td>47839</td>
<td>28703</td>
<td>19135</td>
<td>95677</td>
</tr>
<tr>
<td>acl1_200k</td>
<td>156418</td>
<td>42119</td>
<td>21114</td>
<td>15835</td>
<td>105297</td>
</tr>
<tr>
<td>acl2_100k</td>
<td>60278</td>
<td>15835</td>
<td>21114</td>
<td>15835</td>
<td>52784</td>
</tr>
<tr>
<td>acl2_200k</td>
<td>100276</td>
<td>20457</td>
<td>40915</td>
<td>40915</td>
<td>102287</td>
</tr>
<tr>
<td>acl3_100k</td>
<td>86610</td>
<td>12868</td>
<td>21114</td>
<td>8578</td>
<td>42893</td>
</tr>
<tr>
<td>acl3_200k</td>
<td>138046</td>
<td>60756</td>
<td>75946</td>
<td>15189</td>
<td>151891</td>
</tr>
<tr>
<td>acl4_100k</td>
<td>62165</td>
<td>37138</td>
<td>27854</td>
<td>27854</td>
<td>92846</td>
</tr>
<tr>
<td>acl4_200k</td>
<td>175451</td>
<td>22902</td>
<td>11451</td>
<td>22902</td>
<td>57255</td>
</tr>
<tr>
<td>acl5_100k</td>
<td>88171</td>
<td>9550</td>
<td>2387</td>
<td>11938</td>
<td>23875</td>
</tr>
<tr>
<td>acl5_200k</td>
<td>147411</td>
<td>52589</td>
<td>35059</td>
<td>87649</td>
<td>175297</td>
</tr>
<tr>
<td>fw1_100k</td>
<td>76947</td>
<td>11185</td>
<td>16778</td>
<td>27963</td>
<td>55926</td>
</tr>
<tr>
<td>fw1_200k</td>
<td>106740</td>
<td>50631</td>
<td>40504</td>
<td>10126</td>
<td>101261</td>
</tr>
<tr>
<td>fw2_100k</td>
<td>87757</td>
<td>8289</td>
<td>41446</td>
<td>33157</td>
<td>82892</td>
</tr>
<tr>
<td>fw2_200k</td>
<td>91664</td>
<td>99980</td>
<td>36356</td>
<td>45445</td>
<td>181781</td>
</tr>
<tr>
<td>fw3_100k</td>
<td>46770</td>
<td>36016</td>
<td>13097</td>
<td>16371</td>
<td>65484</td>
</tr>
<tr>
<td>fw3_200k</td>
<td>105870</td>
<td>40235</td>
<td>57479</td>
<td>17243</td>
<td>114957</td>
</tr>
<tr>
<td>fw4_100k</td>
<td>60876</td>
<td>22952</td>
<td>25502</td>
<td>2550</td>
<td>51004</td>
</tr>
<tr>
<td>fw4_200k</td>
<td>98108</td>
<td>67952</td>
<td>52851</td>
<td>30201</td>
<td>151004</td>
</tr>
<tr>
<td>fw5_100k</td>
<td>73202</td>
<td>10753</td>
<td>23656</td>
<td>8602</td>
<td>43011</td>
</tr>
<tr>
<td>fw5_200k</td>
<td>118636</td>
<td>31255</td>
<td>43757</td>
<td>50008</td>
<td>125020</td>
</tr>
<tr>
<td>ipc1_100k</td>
<td>52130</td>
<td>47103</td>
<td>12846</td>
<td>25692</td>
<td>85641</td>
</tr>
<tr>
<td>ipc1_200k</td>
<td>116826</td>
<td>81634</td>
<td>24490</td>
<td>57144</td>
<td>163268</td>
</tr>
<tr>
<td>ipc2_100k</td>
<td>60068</td>
<td>39932</td>
<td>11979</td>
<td>27952</td>
<td>79863</td>
</tr>
<tr>
<td>ipc2_200k</td>
<td>138945</td>
<td>61055</td>
<td>47487</td>
<td>27135</td>
<td>135677</td>
</tr>
</tbody>
</table>

with the same timestamp, the reduction of the number of updates varied between 0.1% (rrc07) and 23% (route-views.linx) with an average of 6.64%. After applying reduction to remove the redundant operations, the remaining operations are arranged in a consistent sequence using our near optimal heuristic in Figure 4-10 and the fifth column in Table 4-3 gives the maximum growth in the forwarding table.

### 4.3.2.2 Packet classifiers

The update traces for the classifier benchmarks (Table 4-2) were synthetically generated to be free from redundancies. Table 4-4 gives the growth in the packet classifiers compared to the initial table size. The first column presents the names of each dataset, whereas the other columns indicate the maximum increase in size of the
Table 4-3. Total number of updates before and after applying reduction and the maximum increase in table size

<table>
<thead>
<tr>
<th>DataSets</th>
<th>Before reduction</th>
<th>After reduction</th>
<th>Percent decrease in #updates</th>
<th>Maximum increase in table size</th>
</tr>
</thead>
<tbody>
<tr>
<td>rrc00</td>
<td>431867</td>
<td>393691</td>
<td>8.84</td>
<td>219</td>
</tr>
<tr>
<td>rrc01</td>
<td>574879</td>
<td>517974</td>
<td>9.90</td>
<td>206</td>
</tr>
<tr>
<td>rrc03</td>
<td>339241</td>
<td>294147</td>
<td>13.29</td>
<td>709</td>
</tr>
<tr>
<td>rrc04</td>
<td>225454</td>
<td>224848</td>
<td>0.27</td>
<td>301</td>
</tr>
<tr>
<td>rrc05</td>
<td>473136</td>
<td>436699</td>
<td>7.70</td>
<td>2181</td>
</tr>
<tr>
<td>rrc06</td>
<td>604367</td>
<td>602135</td>
<td>0.37</td>
<td>457</td>
</tr>
<tr>
<td>rrc07</td>
<td>180139</td>
<td>179956</td>
<td>0.10</td>
<td>32</td>
</tr>
<tr>
<td>rrc10</td>
<td>370806</td>
<td>366235</td>
<td>1.23</td>
<td>1259</td>
</tr>
<tr>
<td>rrc11</td>
<td>403140</td>
<td>382377</td>
<td>5.15</td>
<td>224</td>
</tr>
<tr>
<td>rrc12</td>
<td>475843</td>
<td>422334</td>
<td>11.25</td>
<td>655</td>
</tr>
<tr>
<td>rrc13</td>
<td>329200</td>
<td>314652</td>
<td>4.42</td>
<td>634</td>
</tr>
<tr>
<td>rrc14</td>
<td>317436</td>
<td>308546</td>
<td>2.80</td>
<td>214</td>
</tr>
<tr>
<td>rrc15</td>
<td>349469</td>
<td>344706</td>
<td>1.36</td>
<td>2439</td>
</tr>
<tr>
<td>rrc16</td>
<td>12363</td>
<td>12298</td>
<td>0.53</td>
<td>2534</td>
</tr>
<tr>
<td>rv2</td>
<td>690339</td>
<td>570229</td>
<td>17.40</td>
<td>470</td>
</tr>
<tr>
<td>rv4</td>
<td>653511</td>
<td>593864</td>
<td>9.13</td>
<td>305</td>
</tr>
<tr>
<td>rv.eqix</td>
<td>318019</td>
<td>304375</td>
<td>4.29</td>
<td>493</td>
</tr>
<tr>
<td>rv.isc</td>
<td>380630</td>
<td>336328</td>
<td>11.64</td>
<td>334</td>
</tr>
<tr>
<td>rv.linx</td>
<td>430355</td>
<td>331347</td>
<td>23.01</td>
<td>86</td>
</tr>
<tr>
<td>rv.wide</td>
<td>577715</td>
<td>572963</td>
<td>0.82</td>
<td>995</td>
</tr>
<tr>
<td>rrc00Jan25</td>
<td>53113</td>
<td>49912</td>
<td>6.03</td>
<td>1158</td>
</tr>
</tbody>
</table>

classifier as the updates are clustered as indicated, and are subjected to consistent sequencing using our heuristic. The components of the precedence graph are simplistic and our heuristic in Figure 4-10 will produce optimal update sequences for all the classifier benchmarks used in our experiments. When we apply our algorithms to the entire set of updates, then we see a remarkable drop in the maximum growth to zero for most datasets in Table 4-4. This can be explained by the large number of deletes in the update sequences and the availability of more deletes that require no prior inserts, as all the updates are considered. These deletes are put first in the new sequence freeing up enough space in the rule table to hold the newly inserted rules without any increase in the size compared to the initial table.
Table 4-4. Maximum increase in intermediate classifier rule table size

<table>
<thead>
<tr>
<th>DataSet</th>
<th>acl1_100k</th>
<th>acl1_200k</th>
<th>acl2_100k</th>
<th>acl2_200k</th>
<th>acl3_100k</th>
<th>acl3_200k</th>
<th>acl4_100k</th>
<th>acl4_200k</th>
<th>acl5_100k</th>
<th>acl5_200k</th>
<th>fw1_100k</th>
<th>fw1_200k</th>
<th>fw2_100k</th>
<th>fw2_200k</th>
<th>fw3_100k</th>
<th>fw3_200k</th>
<th>fw4_100k</th>
<th>fw4_200k</th>
<th>fw5_100k</th>
<th>fw5_200k</th>
<th>ipc1_100k</th>
<th>ipc1_200k</th>
<th>ipc2_100k</th>
<th>ipc2_200k</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=100</td>
<td>245</td>
<td>411</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>112</td>
<td>89</td>
<td>154</td>
<td>492</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>22</td>
<td>0</td>
<td>27</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>233</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>x=200</td>
<td>240</td>
<td>403</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>112</td>
<td>89</td>
<td>143</td>
<td>492</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>22</td>
<td>0</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>233</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>x=300</td>
<td>240</td>
<td>407</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>112</td>
<td>68</td>
<td>139</td>
<td>473</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>22</td>
<td>0</td>
<td>27</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>176</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>x=400</td>
<td>234</td>
<td>401</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>90</td>
<td>89</td>
<td>125</td>
<td>450</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>22</td>
<td>0</td>
<td>27</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>176</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>x=500</td>
<td>200</td>
<td>403</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>108</td>
<td>50</td>
<td>143</td>
<td>473</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>22</td>
<td>0</td>
<td>27</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>141</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

A major challenge in efficient router design is how to perform fast and consistent updates. We presented a methodology for performing incremental updates in classifier tables, when updates arrive together in clusters. By ordering the updates in a consistent manner, we ensure that the data packets are handled properly in terms of being forwarded to appropriate next hops, or being applied the proper action (e.g. accept/deny/drop). In the next chapter we will discuss about power efficient TCAM architectures for packet classifiers.

A major challenge in efficient router design is how to perform fast and consistent updates. In this chapter, we presented a methodology for performing incremental
updates in classifier tables, when updates arrive together in clusters. By ordering the updates in a consistent manner, we ensure that the data packets are handled properly in terms of being forwarded to appropriate next hops, or being applied the proper action (e.g. accept/deny/drop). Two types of consistency, namely, batch consistency and incremental consistency have been introduced and a reduction method is presented which identifies and removes the redundant updates from a given batch of updates. We showed that the insert/delete/change operations that arrive in a cluster may be conveniently represented as a precedence graph. Every topological ordering of the vertices in the precedence graph gives us a batch consistent sequence. Among all the batch consistent sequences it is desirable to have the one that leads to minimum growth of the rule table at the time of incorporating the updates. We presented an efficient heuristic that builds a near optimal batch consistent sequence for practical datasets. For incremental consistent orderings, we showed that there is no precedence graph whose topological orderings correspond to all the permissible incremental consistent orderings.
CHAPTER 5
PC-DUOS: FAST TCAM LOOKUP AND UPDATE FOR PACKET CLASSIFIERS

The main goal of storing packet classifiers in TCAMs is to enable high speed table lookup. It is possible to implement a software solution for packet classification, but it is difficult to achieve the search speed requirement in gigabit routers using a purely software solution [10, 15]. In this chapter, we explore a dual TCAM architecture for packet classifiers which enable faster updates and lookup.

The chapter is organized as follows. Section 5.1 presents background and related research work, while Section 5.2 presents the methodology of storing and updating packet classifiers in PC-DUOS, and Section 5.3 presents experimental results. We conclude in Section ??

5.1 Background and Related Work

PC-DUOS (Packet Classifier - DUOS) is an extension of DUOS [26], which has been proposed for packet forwarding, and described in Chapter 3. DUOS, as shown in Figure 3-2, has two TCAMs, labeled as the ITCAM (Interior TCAM) and the LTCAM (Leaf TCAM). DUOS also employs a binary trie in the control plane of the router to represent the prefixes in the forwarding table. The prefixes found in the leaf nodes of the trie are stored in the LTCAM, and the remaining prefixes are stored in the ITCAM. The prefixes stored in the LTCAM are independent and therefore at most one LTCAM prefix can match a specified destination address. Hence the LTCAM doesn’t need a priority encoder. Prefix lookup works in parallel on both the TCAMs. If a match is found in the LTCAM then that is guaranteed to be the longest matching prefix and the corresponding next hop is returned. At the same time the ongoing lookup process on the ITCAM (which takes longer due to the priority resolution step) is aborted. Thus, if a match is found on the LTCAM, the overall lookup time is shortened by about 50% [3]. The final stage logic in Figure 3-2 that chooses between the two next hops could be moved ahead and placed between the TCAM and SRAM stages. In that case, the logic receives one
“matching index” input from the LTCAM and another from the ITCAM. If a match is found in the LTCAM, the index from LTCAM input is used to access the LSRAM, otherwise, the ITCAM index is used to access the ISRAM. Further, if a match is found in the LTCAM, the ITCAM lookup is aborted.

To support lock-free updates, so the TCAMs can be updated without locking them from lookups, DUOS implements consistent update operations that rule out incorrect matches or erroneous next hops during lookup. For consistent updates, it is assumed that:

1. Each TCAM has two ports, which can be used to simultaneously access the TCAM from the control plane and the data plane.
2. Each TCAM entry/slot is tagged with a valid bit, that is set to 1 if the content for the entry is valid, and to 0 otherwise. A TCAM lookup engages only those slots whose valid bit is 1. The TCAM slots engaged in a lookup are determined at the start of a lookup to be those slots whose valid bits are 1 at that time. Changing a valid bit from 1 to 0 during a data plane lookup does not disengage that slot from the ongoing lookup. Similarly, changing a valid bit from 0 to 1 during a data plane lookup does not engage that slot until the next lookup.

Additionally, the availability of the function `waitWriteValidate` is assumed which writes to a TCAM slot and sets the valid bit to 1. In case the TCAM slot being written to is the subject of an ongoing data plane lookup, the write is delayed till this lookup completes. During the write, the TCAM slot being written to is excluded from data plane lookups. Similarly, the availability of the function `invalidateWaitWrite`, is assumed. This function sets the valid bit of a TCAM slot to 0 and then writes an address to the associated SRAM word in such a way that the outcome of the ongoing lookup is unaffected. All these assumptions for DUOS are also made by our PC-DUOS architecture.

The problem of incorporating updates to packet classifiers stored in TCAMs has been studied in [54] and [43]. The authors in [54] present a method for consistent updates when the classifier updates arrive in a batch. All deletes in an update batch are first performed to create empty slots in the TCAM. Then the relative priority of the relevant rules (for example rules overlapping with a new rule being inserted) is
determined and the existing rules are moved accordingly to reflect any change in priority ordering as the entire batch of updates is applied. Following the ordering of existing rules, new rules are inserted in appropriate locations. A problem with the algorithm of [54] is that it performs the deletes in the update batch first. This could lead to temporary inconsistencies in lookup [25].

Given a packet classifier, a naive approach is to store it in a TCAM by entering each rule sequentially as they appear in the classifier and distribute all the empty slots between rules. As mentioned in [43], this approach could lead to high power consumption during lookup as the whole TCAM has to be searched including the empty entries. On the other hand, if the empty entries are kept together at the higher addresses of the TCAM, then those may be excluded from lookups. However, if the empty spaces are kept at one end of the TCAM, then it would require a large number of rule moves to create an empty slot at a given location. Specifically, all the rules in the TCAM, below the slot to be emptied must be moved below.

We use a simple TCAM (STCAM) architecture for performance comparison. The STCAM is a modification over the naive TCAM in that the rules are grouped by block numbers, which reduces the number of required moves when a free slot is needed. The required number of moves is now bounded by the total number of blocks. The block numbers are assigned to the rules using the algorithm presented in [43], based on a priority graph. In this method a subset of the rules is identified such that within the subset, each rule overlaps with every other rule. Each rule in the subset is assigned a different block number based on its priority. Block numbers can be reused for different non-overlapping rule subsets. Thus, rules with the same block number are all non-overlapping or independent. Two rules are independent iff there is no packet that matches both the rules. Filters are grouped based on their assigned block numbers. The group with the lowest block number is of highest priority and these rules are stored in the lowest memory addresses of the TCAM.
Song and Turner [43] describe a fast TCAM update scheme on packet classifiers. In their method, the classifier rules are entered arbitrarily in the TCAM and are not arranged according to decreasing order of priority. They ensure that the action corresponding to the highest priority matching rule is returned by performing multiple searches on the TCAM. Specifically, they assign a priority (which we call block number here) to each rule and encode the block number as a TCAM field and allow the highest priority TCAM match to be found using $\log_2 n$ searches, where $n$ is the total number of block values assigned in the classifier. The highest priority match corresponds to the rule with the minimum block number. The rule and its assigned block number are entered in the TCAM. Even though this method does not incur TCAM writes due to rule moves for maintaining consistent block numbers for overlapping rules or to create an empty slot at the right place for inserting a new rule, this method involves a number of TCAM writes as the assigned block numbers of rules change due to inserts or deletes. Moreover, lookup speed is slowed down since multiple TCAM searches are required and these searches cannot be pipelined as they take place on the same TCAM. Our PC-DUOS architecture performs lookup using a single TCAM search.

### 5.2 PC-DUOS: Methodology

PC-DUOS uses the same two TCAM architecture as used in DUOS (Figure 3-2 in Chapter 3). Lookup also works in the same way as for DUOS. That is, the LTCAM and ITCAM are searched in parallel using the packet header information. In case a match is found in the LTCAM, the ongoing search in the ITCAM is aborted. When the ITCAM search is aborted, lookup time is reduced by about 50% [3], because the LTCAM has no priority encoder. For this lookup strategy to yield correct results, the following requirements must hold:

**REQ1)** No packet is matched by more than one rule in the LTCAM.

**REQ2)** When a packet is matched by a rule in the LTCAM, the matched rule must be the highest priority matching rule.
The algorithms used for storing and updating rules in the TCAMs are discussed in detail in this section.

5.2.1 Storing Rules in TCAMs

Figure 5-1 shows the overall flow of our methodology of storing rules in the ITCAM and LTCAM of PC-DUOS. The first phase involves storing all the rules in a multi-dimensional trie maintained on the control plane of the classifier. This is further discussed in Section 5.2.1.1. The second phase in our methodology consists of traversing the multi-dimensional trie and identifying independent rules for inclusion in the LTCAM as discussed in Section 5.2.1.2. In the third phase, rules not stored in the LTCAM are stored in the ITCAM in priority order. This is discussed in Section 5.2.1.3.

```
Packet Classifier, Empty TCAMs
  Store in control-plane trie
    Store some rules in LTCAM
      Store remaining rules in ITCAM
        Filled TCAMs
```

Figure 5-1. Flow diagram for storing packet classifiers in TCAMs

5.2.1.1 Representing classifier rules

The classifier rules are represented in a multi-dimensional trie, where each dimension represents one field of the rule. The fields in a filter rule appear in the following order in the trie: `< destination, source, protocol, source port range, destination port range >`. We assume that the destination and source fields of the filters are specified as prefixes. So, these are represented in a trie in the standard way with the
left child of a node representing a 0 and the right child a 1. Ranges may be handled in one of many ways. In this chapter, we consider the DIRPE scheme of [18] that requires the use of a multibit trie. Our methodology may also be applied to other range encoding schemes [8, 11].

5.2.1.2 Storing rules in the LTCAM

Recall that two rules are independent iff no packet is matched by both rules. For the LTCAM we are interested in identifying the largest set of rules that are pairwise independent. Note that every independent rule set satisfies the first requirement, \textbf{REQ1}, for a lookup to work correctly. To find an independent rule set in acceptable computing time, we relax the “largest set” requirement and instead look for a large set of independent rules. It is easy to see that the rules in the nodes obtained by traversing the multidimensional trie from the root to the leaves of the outer level trie and then from these leaves into their attached next level tries and so on until we reach leaves of the innermost level tries are independent. We call this set of independent rules the \textit{leaves of leaves set}.

![Diagram of a two-dimensional trie storing three rules: R1, R2, and R3]

**Figure 5-2.** Example of a two-dimensional trie storing three rules: R1, R2, and R3

Figure 5-2 shows an example classifier, with three rules. The outer level trie (level 1) is on the destination prefix and the inner level (level 2) is on the source prefix. Rules R2 and R3 belong to the leaves-of-leaves set, because they are stored at leaves of their
source trie and the source trie itself hangs off a leaf in the destination trie. On the other hand, R1 does not belong to the leaves-of-leaves set even though it is stored in a leaf of its source trie because this source trie hangs off a non-leaf node in the destination trie.

It is easy to see that the leaves-of-leaves set of rules are independent. Although the rules in the leaves of leaves set are uniquely determined by the multi-dimensional trie, by changing the mapping between filter tuple components and trie dimension, we get different leaves of leaves sets of possibly different size. For example, when the dimension ordering is \(<\text{destination, source, protocol, source port range, destination port range}>\), the leaves of leaves set is different from when the ordering \(<\text{source, destination, protocol, source port range, destination port range}>\) is used.

The algorithm of Figure 5-3 partitions the rule set into two lists – leafList and otherList, where leafList is the leaves of leaves set, and otherList contains the remaining rules. In the algorithm, the trie is traversed in postorder fashion, starting with the root of the multi-dimensional trie. Upon hitting a leaf node, we jump to a trie for the next field values stored at that leaf node. For this trie too, we traverse the trie till we hit a leaf, and then switch to the trie stored at that leaf. We continue this way till the last field. In the example of Figure 5-2, R2 and R3 are added to the leafList and R1 is added to the otherList that is to contain the remaining rules.

Our rule placement strategy places all rules in otherList in the ITCAM. Additionally, it may be necessary to place some of the rules in leafList in the ITCAM as well so as to ensure that the rules which will be placed in the LTCAM, satisfy the second requirement for lookup correctness. Once the leaves-of-leaves set of rules has been identified and put into leafList, then for each rule \(R\) in leafList it is checked if there exists a higher priority rule in otherList (containing rules for insertion in ITCAM) that overlaps with \(R\). If there is, then this rule is added to the ITCAM. In Figure 5-2, R3, which is in leafList, is added to the ITCAM instead of the LTCAM, since it has lower priority compared to the otherList rule R1. Thus, in our three rule example of Figure 5-2, R2 is added to
Algorithm: assignRules(node)
Inputs:
node: a trie node, set to root of multi-dimensional trie, initially.
Output:
leafList: a list of rules consisting of potential candidates for LTCAM.
for each child i of node
    assignRules(node→child[i]);
endfor
if (node→valid) // true if node contains a rule or a root of a trie for the next field.
    if (node contains root of a trie)
        if (node is a leaf) // true if node has no children.
            assignRules(node→trie→root);
        else // node belongs to trie for the last field and contains a rule
            if (node is a leaf)
                append node (as representative of rule) to leafList
            endif
        endif
    endif
endif

Figure 5-3. Selecting rules for insertion into LTCAM

the LTCAM and R1 and R3 are added to the ITCAM. Since the rules assigned to the
LTCAM are independent, they can be entered in any order and there is no need for a
priority encoder. Our LTCAM rule selection process guarantees that a match found in
the LTCAM during lookup will be of the highest priority. Thus, during lookup, if a match
is found in the LTCAM, then the corresponding action can be returned, without waiting
for the ITCAM lookup to finish. The ITCAM lookup must go through a priority encoder to
resolve multiple matches. This makes lookups that find a match in the LTCAM quicker by
about 50% [3].

5.2.1.3 Storing rules in the ITCAM

A priority graph is first created for each rule to be stored in the ITCAM. There is a
directed edge (u, v) from vertex u to vertex v iff (a) the rules corresponding to u and v
overlap (i.e., at least one packet matches both rules) and (b) the priority of u is more
than that of v (we assume that overlapping rules have different priority). For the directed
dge (u, v), we say that u is the parent of v and v is the child of u. The priority graph is
used to assign block numbers to rules/vertices as follows [43]. All vertices with in-degree
0 are assigned the block number 1. Each remaining vertex v is assigned a block number
equal to
\[ 1 + \max_{(u,v) \in E} \{ \text{block number of } u \} \]

where \( E \) is the set of edges in the priority graph. Thus a child of any vertex is assigned a block number that is at least one more than the block number of this vertex.

The first step in building the priority graph, is an iteration over the rules designated for the ITCAM and identifying, for each ITCAM rule, its overlapping ITCAM rules. We traverse the trie to identify the overlapping rules instead of following a simple \( O(n^2) \) algorithm of comparing every rule with every other rule. This, along with the fact that about 50% of the rules end up in the ITCAM (for most cases as shown by our experiments) give us significant performance benefit during the initial assignment of rules to the ITCAM, compared to the case when all the rules are stored in a STCAM.

The algorithm to determine the rules that overlap a given rule uses trie based traversal and is given in Figure 5-4. For simplicity, the algorithm is specified for the case when rules have only two fields - destination and source prefix. Its extension to rules with a larger number of fields is straight forward. Given a rule, the algorithm first extracts the values for the different fields for the rule, and traverses the trie along these prefix paths until all overlapping rules are found. For each overlapping rule found, a directed edge is added to the priority graph. The priority graph is a directed acyclic graph and block numbers may be assigned using an iterative process that assigns the next block number to all vertices whose in-degree is zero and then symbolically eliminates these vertices from the priority graph.

5.2.2 Update Algorithms

The first step in processing a classifier update is to perform the update operation on the multi-dimensional trie. Section 5.2.2.1 describes how this is done. In the second step, existing rules are moved either in the same TCAM or between TCAMs to preserve the priorities and properties of rules and the actual update is done. This is discussed in Section 5.2.2.2.
Algorithm: findOverlappingRules\texttt{(ruleInstance)}

\textbf{Inputs:}
\texttt{ruleInstance}: a trie node representing a rule and storing its action.

\textbf{Output:}
list: a list of rules overlapping with the input rule

get destination prefix \texttt{Dest}, source prefix \texttt{Src} from \texttt{ruleInstance}

\texttt{nodeD} = root of destination trie;

\textbf{for} \texttt{(i=0; i<length of destination prefix; ++i)}

\textbf{if} (root of a source trie is stored at \texttt{nodeD})

\texttt{nodeS} = root of source trie

\textbf{for} \texttt{(j=0; j<length of source prefix and nodeS; ++j)}

\textbf{if} nodeS stores a rule \texttt{R}

append \texttt{R} to list.

\texttt{branchBitS} = \texttt{Src[j]};

\texttt{nodeS} = \texttt{nodeS} \texttt{!} \texttt{child[branchBitS]};

\textbf{endfor}

\textbf{if} (\texttt{nodeS} != NULL) \texttt{then}

visit all nodes in subtrie rooted at \texttt{nodeS}

\textbf{if} (any node stores a rule \texttt{R})

append \texttt{R} to list.

\textbf{endif}

\textbf{endif}

\textbf{endif}

\texttt{branchBitD} = \texttt{Dest[i]};

\texttt{nodeD} = \texttt{nodeD} \texttt{!} \texttt{child[branchBitD]};

\textbf{endfor}

visit all nodes in subtrie rooted at \texttt{nodeD}

\textbf{if} (any node stores a rule \texttt{R})

append \texttt{R} to list.

\textbf{endif}

\textbf{Figure 5-4.} Find overlapping rules by trie traversal

\textbf{5.2.2.1 Update the trie}

This is the first step in the update process. The multi-dimensional trie is updated with the help of functions as described in Figure 5-5. As the trie is updated, a number of TCAM update requests are generated that are needed to maintain the two requirements for a rule to be in the LTCAM. For example, as a result of inserting a new rule, some existing rules may no longer be leaves-of-leaves rules. These rules can be obtained from the list of nodes \texttt{b} returned by \texttt{Trie.insert} of Figure 5-5. For these rules, ITCAM insert requests are generated to insert them to the ITCAM and LTCAM delete requests.
Function: Trie.insert
(a, b, isLeaf) = Trie.insert(rule, action);
This function inserts a rule and its action into the control-plane multi-dimensional trie. It
returns the trie node \( a \) which stores the new rule and a list \( b \) of nodes that were leaves
in different dimensions prior to the insertion of the rule, and a boolean variable isLeaf
which is true if the inserted rule is stored in a leaf of leaves.

Function: Trie.delete
(a, b) = Trie.delete(rule);
This function deletes a rule from the control plane trie and returns the trie node \( a \) that
used to store the rule just deleted and \( a \)’s nearest ancestor node \( b \) that will be reached
upon deleting the trie nodes.

Function: Trie.change
a = Trie.change(rule, action);
This function changes the action associated with a prefix and returns the trie node \( a \)
that stores the rule.

Figure 5-5. Table of control-plane trie functions

are generated to delete them from the LTCAM. The ITCAM inserts must happen first so
that a TCAM search returns consistent results.

Similarly, when a rule is deleted from the trie, some existing rules may now become
leaves-of-leaves rules. The nearest ancestor node \( b \) returned by Trie.delete may be
traversed to obtain rules that now become leaves-of-leaves rules. For these rules it is
further checked if they are the highest priority rules in the set of overlapping rules. If they
are, then the second requirement for membership in LTCAM gets satisfied and for these
rules a number of LTCAM insert and ITCAM delete requests are generated.

Thus, as a result of updating the trie a number of ITCAM and LTCAM insert and
delete requests are generated. These requests still remain in the control-plane and are
processed further as described below, before they are applied to the TCAMs.

5.2.2.2 Other control-plane operations for updates

At this stage, the TCAM requests generated as a result of updating the multi-dimensional
trie are further processed and run through the memory management algorithms for the
ITCAM and the LTCAM. The memory management schemes from DUOS may be used
here. For the ITCAM of PC-DUOS we implemented the DLFS_PLO scheme, as its the
most efficient scheme known to us for moving free slots to a desired location in a TCAM. Also, for operations on the ITCAM, we use dynamically sized arrays for \( \text{top}, \text{bot}, \text{AV} \) and \( \text{prev} \) variables of the DLFS\_PLO scheme, since rule blocks are randomly added and deleted – unlike same length prefix blocks in packet forwarding that results in a maximum of 32 blocks for IPv4. In the DLFS\_PLO initial rule placement scheme, free slots are kept in the region between two blocks. Additionally, there may be free slots \textit{within} a block. So a list of free slots is maintained for each block on the TCAM, with the list being empty initially. As rules are deleted from a block, the freed slots are added to the list for that block.

\textbf{ITCAM.insert:}

To insert a new rule in the ITCAM, firstly, the rule is added to the priority graph and a block number is assigned to it. It may be required to adjust the block numbers of existing rules overlapping with the new rule. Next, DLFS\_PLO insert and delete requests are generated for the existing rules, whose block numbers are changed, to move them to the corresponding rule blocks. Secondly, a new DLFS\_PLO insert request is generated for the new rule to insert it in the desired rule block. The method to assign and adjust block numbers during updates is explained in [43]. A DLFS\_PLO insert or delete request would invoke a number of \textit{move} operations to create a free slot at a specific block or to manage freed slots.

Thirdly, to ensure that the second requirement for membership in LTCAM still holds for all the LTCAM rules, the priority of the new rule is compared with that of the LTCAM rules. All overlapping LTCAM rules with priority lower than that of the new rule, must be inserted in the ITCAM using the first and second steps above and then deleted from the LTCAM. Overlapping LTCAM rules are identified by traversing the trie using the algorithm of Figure 5-4. Thus, at the end of this step a number of ITCAM insert and LTCAM delete requests are generated.

\textbf{ITCAM.delete:}
The steps for deleting a rule from the ITCAM are as follows. Firstly, the rule is deleted from the priority graph, following which a DLFS_PLO delete request is generated to delete the rule from the ITCAM. Secondly, every ITCAM rule that now becomes leaf of leaves and has the highest priority among other ITCAM rules that overlap, are moved to the LTCAM.

**LTCAM.insert and LTCAM.delete:**

The memory management scheme for LTCAM is relatively simple as all the rules in the LTCAM are independent so a new rule can be inserted anywhere in the TCAM. However, we still need to locate a free slot. The LTCAM memory management algorithm of DUOS creates a linked list of the free slots. When a free slot is needed, a slot is obtained from the head of the free slot list. PC-DUOS uses the memory management algorithm for DUOS for its LTCAM (see Chapter 3).

For an LTCAM insert, the $AV$ variable in the control plane is used to access an available slot at the head of the free slot list and an insert request is generated for the new rule at that slot.

Similarly, to delete a rule from the LTCAM, a delete request is generated to set the valid bit for the slot holding the rule to 0 and to set the $AV$ variable on the corresponding SRAM word and then update $AV$ to the address of the slot currently being freed.

### 5.3 Experimental Results

We evaluated the performance of PC-DUOS in updating packet classifiers using a set of synthetic IPv4 classifiers generated using Classbench [50]. Our experiments were run on an x86 Linux box with 64bit, 1200MHz CPU. Table 5-1 shows the details of these datasets. The first column in this figure presents the names of the classifiers, the second column shows the seed files in Classbench from which these tests were derived, the third column shows the number of rules in each of the classifiers, and columns four to six give the number of insert and delete operations in the update traces as well as the total number of update operations for each dataset. We used 7 seed files based
on access control lists (acl), firewalls (fw) and IP chains (ipc) to generate 7 classifiers.
Each rule consists of the fields: source address, destination address, source port range,
destination port range, protocol. We created an update trace from the classifiers by
marking rules for insertion/deletion randomly, and later removing the rules marked for
insertion. The corresponding insert, and delete operations are shuffled and then written
to the update file. To update the classifier in a consistent fashion, change operations
are implemented as a succession of insert and delete commands, that is insert the new,
changed rule first and then delete the existing rule being changed. Hence, in our update
simulation we did not add change operations.

Table 5-1. Synthetic classifiers and update traces used in the experiments

<table>
<thead>
<tr>
<th>DataSet</th>
<th>seed_file</th>
<th>#Rules</th>
<th>#Inserts</th>
<th>#Deletes</th>
</tr>
</thead>
<tbody>
<tr>
<td>test1</td>
<td>acl1_seed</td>
<td>30075</td>
<td>69300</td>
<td>29700</td>
</tr>
<tr>
<td>test2</td>
<td>fw1_seed</td>
<td>7989</td>
<td>28800</td>
<td>7200</td>
</tr>
<tr>
<td>test3</td>
<td>ipc1_seed</td>
<td>15338</td>
<td>34300</td>
<td>14700</td>
</tr>
<tr>
<td>test4</td>
<td>acl2_seed</td>
<td>53970</td>
<td>45000</td>
<td>45000</td>
</tr>
<tr>
<td>test5</td>
<td>fw5_seed</td>
<td>5571</td>
<td>45900</td>
<td>5100</td>
</tr>
<tr>
<td>test6</td>
<td>acl4_seed</td>
<td>34254</td>
<td>5000</td>
<td>5000</td>
</tr>
<tr>
<td>test7</td>
<td>ipc1_seed</td>
<td>5165</td>
<td>94050</td>
<td>4950</td>
</tr>
</tbody>
</table>

The smoothness and scope parameters in ClassBench used to generate the
packet classifiers are as follows- smoothness: 2; address scope: 0.5, application scope:
-0.1. From the description of these parameters in [50], the values used ensure that
the generated files contain a fair amount of overlapping rules. Note that the number
of overlapping rules directly impact the update performance as the overlapping rules
have to be rearranged to maintain priority ordering in the ITCAM. If we look at the
priority graphs for the tests, we find that the number of levels of nodes in these graphs is
between 27 and 73. This indicates a close resemblance to real life packet classifiers in
terms of the maximum length of a chain of overlapping updates [43].

We have run two sets of experiments on our datasets. The first set uses prefix
expansion of ranges[18, 43], whereas the second set uses DIRPE [18] for representing
source and destination port ranges. DIRPE is implemented by introducing multi-bit tries
for source and destination port ranges. For experiments with DIRPE, we assume that 36 bits are available for encoding each port range in a rule. With this assumption, we use strides 223333 for both source and destination port range tries in our experiments, which give us minimum expansion of the rules. The stride value 223333 indicates that for a given port number (16 bits), the root of the port range trie will use the first two bits to branch to one of its four possible child nodes at level 1. Each node at level 1 uses the next two bits to branch to one among its four possible child nodes at level 2. A node at the level 2, on the other hand, uses the next 3 bits to branch to one among its eight possible child nodes at the level 3, and so on. Thus, all the 16 bits ($2 + 2 + 3 + 3 + 3 + 3 = 16$) are used to traverse the trie and arrive at the last node (at the 6th level) representing the port number.

We compare our results with those from a single TCAM setup (STCAM) as is commonly used today for packet classification. In this setup, all rules are entered into the TCAM in priority order. The ordering is needed only for rules that overlap. If two rules do not overlap, their relative ordering does not matter. We use a priority graph for the whole set of rules to track the block numbers of the rules as well as to compute adjustments to block numbers as new rules are inserted. The memory management scheme DLFS_PLO is used for the STCAM to allot a free slot for rule insertion or to manage a freed up slot following rule deletion. We do not compare PC-DUOS’ update performance with that of the work in [43], since PC-DUOS’ lookup performance is far superior to the worst case of [43], which is at least 4 times slower in the worst case, on our datasets (obtained as logarithm of the number of blocks).

Tables 6-3 and 5-3 show the number of TCAM writes and the total computation time for processing the updates starting from the first to the last update using STCAM and PC-DUOS architectures. The processing time includes everything except the actual time to perform the TCAM writes. The write ratio is obtained by dividing the number of writes in STCAM, by the number of writes (sum of ITCAM and LTCAM writes) in PC-DUOS.
Table 5-2. Number of TCAM writes in PC-DUOS and STCAM using prefix representation scheme

<table>
<thead>
<tr>
<th>Data-Sets</th>
<th>#Rules</th>
<th>PC-DUOS</th>
<th>STCAM</th>
<th>write ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>#ITCAM writes</td>
<td>#LTCAM writes</td>
<td>Time(s)</td>
</tr>
<tr>
<td>test1</td>
<td>30075</td>
<td>1301</td>
<td>131731</td>
<td>10</td>
</tr>
<tr>
<td>test2</td>
<td>7989</td>
<td>120146</td>
<td>72209</td>
<td>1635</td>
</tr>
<tr>
<td>test3</td>
<td>15338</td>
<td>47702</td>
<td>54398</td>
<td>291</td>
</tr>
<tr>
<td>test4</td>
<td>53970</td>
<td>98768</td>
<td>115130</td>
<td>1366</td>
</tr>
<tr>
<td>test5</td>
<td>5571</td>
<td>181379</td>
<td>54357</td>
<td>3693</td>
</tr>
<tr>
<td>test6</td>
<td>34254</td>
<td>32062</td>
<td>12412</td>
<td>47</td>
</tr>
<tr>
<td>test7</td>
<td>5165</td>
<td>231437</td>
<td>108681</td>
<td>1611</td>
</tr>
</tbody>
</table>

Table 5-3. Number of TCAM writes in PC-DUOS and STCAM using DIRPE[18]

<table>
<thead>
<tr>
<th>Data-Sets</th>
<th>#Rules</th>
<th>PC-DUOS</th>
<th>STCAM</th>
<th>write ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>#ITCAM writes</td>
<td>#LTCAM writes</td>
<td>Time(s)</td>
</tr>
<tr>
<td>test1</td>
<td>30075</td>
<td>1301</td>
<td>115092</td>
<td>8</td>
</tr>
<tr>
<td>test2</td>
<td>7989</td>
<td>78628</td>
<td>27238</td>
<td>928</td>
</tr>
<tr>
<td>test3</td>
<td>15338</td>
<td>44055</td>
<td>44681</td>
<td>265</td>
</tr>
<tr>
<td>test4</td>
<td>53970</td>
<td>59067</td>
<td>89660</td>
<td>921</td>
</tr>
<tr>
<td>test5</td>
<td>5571</td>
<td>82656</td>
<td>30702</td>
<td>2012</td>
</tr>
<tr>
<td>test6</td>
<td>34254</td>
<td>12391</td>
<td>9639</td>
<td>31</td>
</tr>
<tr>
<td>test7</td>
<td>5165</td>
<td>207274</td>
<td>89389</td>
<td>1449</td>
</tr>
</tbody>
</table>

We observe from Tables 5-3 and 5-3 that there is an improvement in the write ratio with PC-DUOS. The improvement is in the range [1.2, 2.81] for the prefix representation scheme and [1.19, 2.82] for the DIRPE based scheme. A significant improvement in update processing time is seen for the test1, which is faster by 329 times for the prefix representation scheme and 308 times faster that the STCAM scheme when DIRPE is used. The test1 is derived from a seed file for access control lists. It contains a large percentage of very specific rules for access control. For example, following is a less specific rule - block access to a resource for packets originating from any source address and any port number. Such rules are stored in the ITCAM of PC-DUOS and for test1, only 2% of the rules were assigned to ITCAM. ITCAM updates are more CPU intensive as they require updating the priority graph and invoking the memory management scheme to create empty slots at the right place. LTCAM updates, on the
other hand, are faster since rules are independent and can be added to any free slot and hence a simpler memory management scheme.

Thus, the performance of PC-DUOS depends on the number of rules in the ITCAM and also on the number of rules that overlap with a new rule being inserted. For example, if a newly inserted rule overlap with a large number of rules, it may require the overlapping rules to be moved to different blocks to maintain priority relationship. These two factors are primarily responsible for the differences in improvement in processing time between PC-DUOS and STCAM. The varied total time numbers for the tests that we see on the columns marked Time, are due to the two factors above as well as due to differences in the total number of updates being processed for each of these tests.

Assuming the number of lookup requests satisfied by the LTCAM is proportional to the number of rules stored in it, we can estimate the improvement in average lookup time on our datasets. For a single lookup finding a match in the LTCAM, the improvement is 50%

Assuming the number of lookup requests satisfied by the LTCAM is proportional to the number of rules stored in it, we can estimate the improvement in average lookup time on our datasets. For a single lookup finding a match in the LTCAM, the improvement is 50%\[3\]. The best case in our dataset is test1, which has 98% of the rules in the LTCAM. The improvement in average lookup time for this test is 49%. The worst case is test2 with DIRPE, which contains only 40% of the rules in the LTCAM, and for this test the improvement in average lookup time is 20%.

A new scheme PC-DUOS has been proposed, in this chapter, for packet classifier lookup and update. Two TCAMs are used which have been named as LTCAM and ITCAM. PC-DUOS stores high priority independent rules in the LTCAM. The remaining rules are stored in the ITCAM. During lookup for highest priority rule matching, both ITCAM and LTCAM are searched in parallel. Since the LTCAM stores independent rules, at most one rule may match during lookup in the LTCAM and a priority encoder is not needed. If a match is found in the LTCAM during lookup, it is guaranteed to be the highest priority match and the corresponding action can be returned immediately yielding a 50% [3] improvement in TCAM search time. The average improvement in lookup time is estimated to be between 20% to 48% for the tests in our data set. The
distribution of rules to the two TCAMs makes updates faster by reducing the number of TCAM writes by up to 2.82 times and reducing the control-plane processing time by up to 329 times. The maximum reduction in control-plane processing time is observed for ACL lists.
CHAPTER 6
PC-DUOS+: AN ENHANCED DUAL-TCAM ARCHITECTURE FOR PACKET CLASSIFIERS

PC-DUOS+ is an enhanced version of PC-DUOS with improved update algorithms. The chapter is organized as follows. Section 6.1 describes the PC-DUOS+ scheme of storing packet classifiers in TCAMs. An experimental evaluation of PC-DUOS+ is done in Section 6.2 and we conclude in Section ??.

6.1 PC-DUOS+: Methodology

PC-DUOS+ uses the same TCAM architecture as described for PC-DUOS and requires that \text{REQ1} and \text{REQ2} outlined in Section 5.2 of Chapter 5, are satisfied for lookups to work correctly.

6.1.1 Storing Rules in TCAMs

Figure 6-1 shows the overall flow of storing rules in the ITCAM and the LTCAM. The first phase involves creating a priority graph, in addition to, a multi-dimensional trie for the rules in the classifier. This is further discussed in Section 6.1.1.1. The second phase in our methodology consists of identifying a set of highest priority independent rules and storing these in the LTCAM, which is discussed in Section 6.1.1.2. In the third phase, the remaining rules are stored in the ITCAM in priority order. This is discussed in Section 6.1.1.3.

PC-DUOS+, differs from PC-DUOS, in the way the selection of rules for the LTCAM is made. PC-DUOS filters the leaves of leaves set in a multi-dimensional trie to keep only the highest priority rules among all overlapping rules. The rules in the filtered leaves of leaves set is then entered in the LTCAM. PC-DUOS+, on the other hand, uses a priority graph to select rules for the LTCAM. Moreover, PC-DUOS+ use enhanced algorithms for ITCAM rule insertion which require fewer moves to rearrange rules for priority based adjustments.
6.1.1.1 Representing classifier rules

The classifier rules are represented in a priority graph as well as in a multi-dimensional trie. A priority graph contains one vertex for each rule in the classifier. There is a directed edge between two vertices iff the two rules overlap and the direction of the edge is from the higher to the lower priority rule. Two rules overlap iff there exists at least one packet that matches both the rules.

6.1.1.2 Storing rules in the LTCAM

For PC-DUOS+, we identify a large set of independent rules as the rules in the vertices of the priority graph with in-degree 0. This satisfies REQ1. Further, these rules are also the highest priority rules among all rules that overlap with them. This satisfies the second requirement (REQ2) for a lookup to work correctly. Hence, we choose to enter these rules into the LTCAM. All remaining rules are entered in the ITCAM.

6.1.1.3 Storing rules in the ITCAM

The rules to be stored in the ITCAM, are initially assigned block numbers in the same way as described for PC-DUOS in Section 5.2.1.3.
In the block assignment scheme, rules that are assigned the same block number are independent and hence grouped together in a single block. These blocks are entered in the TCAM in increasing order of the assigned block numbers. In our implementation for PC-DUOS+, each vertex $v$ in the priority graph has a field $v \rightarrow hpri$ which stores a pseudo priority associated with the block number of the vertex. While $v \rightarrow hpri$ equals the block number of $v$ in PC-DUOS, in PC-DUOS+, $priorityMap(v \rightarrow hpri)$ is the block number for rule $v$. When the priority graph is constructed for the initial classifier, $v \rightarrow hpri$ equals the block number of $v$ and $priorityMap$ is an identity mapping. However, as we insert and delete rules, $v \rightarrow hpri$ may no longer equal the block number of $v$ (in fact, $v \rightarrow hpri$ may not be an integer) and $priorityMap$ is no longer an identity mapping.

To build the priority graph, we first iterate over the classifier rules and for each rule, identify all rules that overlap with it. A trie-based algorithm to determine the rules that overlap a given rule is presented in Figure 5-4 of Chapter 5.

Even though in the worst case all the trie nodes have to be explored for finding overlapping rules (this happens, for example, when $ruleInstance$ is the root of the multi-dimensional trie and thus represents a classifier rule with wild-carded fields) this approach works well on average and, in fact, it makes the computation in PC-DUOS+ (and also in PC-DUOS) scalable during the initial setup as well as while processing the updates. In contrast, the simple approach of iterating over all the rules of the classifier to compare overlaps and priorities, quickly becomes a performance bottleneck as the number of rules in the classifier increases.

6.1.2 Update Algorithms

When an update request is received, the priority graph and the multi-dimensional trie are updated. Section 6.1.2.1 describes how this is done. Next the existing ITCAM rules that overlap with the rule involved in the update are rearranged to ensure that the highest priority rules are still matched after the update is complete. Rules may also be
moved from the ITCAM to the LTCAM or vice versa as a result of the updates. This step is discussed in Section 6.1.2.2.

6.1.2.1 Update the priority graph and the trie

This is the first step in the update process. The multi-dimensional trie is updated with the help of functions as described in Figure 6-2.

**Function: Trie.insert**

Trie.insert(rule, action);
This function inserts a rule and its action into the control-plane multi-dimensional trie.

**Function: Trie.delete**

Trie.delete(rule);
This function deletes a rule from the control plane trie.

**Function: Trie.change**

Trie.change(rule, action);
This function changes the action associated with a prefix.

Figure 6-2. Table of control-plane trie functions

The priority graph is updated next. If the update is a delete request, then the vertex for the rule to be deleted (together with incident edges) is removed from the priority graph and rules corresponding to vertices whose in-degree becomes 0 are moved from the ITCAM to the LTCAM. Each rule that is to be so moved is first inserted into the LTCAM and then deleted from the ITCAM using insert/delete procedures described in Section 6.1.2.2. If the update is an insert, then a new vertex is added to the priority graph. All rules overlapping with the new rule are found, and a new edge is added for each overlapping rule. Overlapping rules are identified by traversing the trie using the algorithm of Figure 5-4. After adding a new vertex v to the priority graph, \( v \rightarrow h\text{pri} \) is calculated. If v has no incoming edges \( v \rightarrow h\text{pri} \) is set to 1 and the new rule in v is placed in the LTCAM. Otherwise, v is placed in the ITCAM.

If v is placed in the ITCAM then \( v \rightarrow h\text{pri} \) is set either by moving v’s ancestors upward or its descendants downward or by moving neither descendants nor ancestors. These three possibilities are shown in Figures 6-3(c), (d) and (e). Figure 6-3(a) depicts a portion of the original graph. The number next to each vertex shows the \( h\text{pri} \) value on
that vertex. The newly added vertex $v$ is colored black in Figure 6-3(b). In Figure 6-3(c), $v \rightarrow hpri$ is set based on $v$’s parent $hpri$ so that $v$ will be placed in the ITCAM block below that of its parent. Note that the $hpri$ of $v$’s child must be updated too and the child is moved one block downward, thus avoiding $v$ and its child being placed in the same ITCAM block. Such updates propagate to all descendants. In Figure 6-3(d), $v \rightarrow hpri$ is set based on the $hpri$ of $v$’s child so that $v$ will be placed in the block above that of its child. The $hpri$ of $v$’s parent is updated so that the parent is moved one block upward and these updates propagate to all ancestors. Figure 6-3(e) shows a case where a new block is inserted between the parent block and the child block, and the $hpri$ associated
Algorithm: insertRule(v)

Input: Rule stored in vertex v in the priority graph.

1. \( \text{maxP} = \max(\text{parent} \rightarrow hpri) \) of ITCAM parents of v; \( \text{minC} = \min(\text{child} \rightarrow hpri) \) of children of v;
2. // Default values are maxP: -1 and minC: infinity
3. childMoves = parentMoves = 0;
4. if \((\text{maxP} < \text{minC})\) then
5. compute childMoves to push descendants down and
6. parentMoves to push ancestors up according to Figure 6-6(b).
7. endif
8. // Get block BC for minC. If v has no outgoing edges, then \( BC - 1 \) is the last block
9. \( BC = \text{priorityMap}(\text{minC}); \) \( BP = \text{priorityMap}(\text{maxP}); \)
10. if \((v \text{ has a parent vertex in the ITCAM and parentMoves < childMoves} \) and
11. childMoves > 50) then // Move ancestors upwards
12. targetBlock = \( BC - 1 \);
13. if \((BC - 1 == BP \text{ and parentMoves > 50})\) targetBlock = new block between \( BP \) and \( BC \).
14. endif
15. // Function reversePriorityMap returns pseudo-priority corresponding to targetBlock.
16. \( v \rightarrow hpri = \text{reversePriorityMap}(\text{targetBlock}); \) assign slot in targetBlock for v;
17. if \((\text{v has no parent in the ITCAM} \) or \( \text{parentMoves > 50} \)) begin
18. sort the parent vertices in a decreasing order of \( hpri \);
19. for each parent of v if \((\text{v} \rightarrow hpri > \text{parent} \rightarrow hpri))
20. if (parent is in ITCAM) moveParentUp(parent);
21. endif
22. else // Move descendants downwards
23. // Initially, all highest priority rules in ITCAM have \( hpri=2 \). So, targetBlock is set to that block.
24. targetBlock = priorityMap(2);
25. if \((v \text{ has no parent in the ITCAM})\) then
26. if (there exists a block \( BC - 1 \)) targetBlock = \( BC - 1 \);
27. else if (childMoves > 50) targetBlock = new block on top of \( BC \).
28. endif
29. \( v \rightarrow hpri = \text{reversePriorityMap}(\text{targetBlock}); \)
30. assign slot in targetBlock for v;
31. if \((\text{v} \rightarrow hpri < \text{minC})\) begin
32. sort the descendant vertices in an increasing order of \( hpri \)
33. for each child of v if \((\text{v} \rightarrow hpri < \text{child} \rightarrow hpri))\) moveChildDown(child);
34. endif
35. endif
36. // Process nodeList from moveParentUp/moveChildDown
37. for each w in nodeList starting from the last one
38. slotW = current TCAM slot occupied by the rule of w;
39. write the rule of w in the assigned slot; free slotW;
40. endfor
41. write the rule of v in the assigned slot.

Figure 6-4. Insert a rule in the ITCAM
Algorithm: moveChildDown(child)
Input: Rule stored in vertex 'child' in the priority graph.

mP = find max(parent→hpri) from all parents of child
mC = find min(child→hpri) from all children of child
if (mP < child→hpri and child→hpri < mC) return;
block = priorityMap(maxP) + 1;
child→hpri = reversePriorityMap(block);
assign a slot in block for child; append child to nodeList;
if (!(child→hpri < mC)) begin
    sort the descendant vertices in an increasing order of hpri
    for each childi of child
        if (!(child→hpri < childi→hpri)) moveChildDown(childi);
endif

Figure 6-5. Moving descendants downward in the ITCAM

with the new block is 3.5. Thus v→hpri is set to 3.5, and neither the descendants nor the
ancestors of the new block are moved.

Figure 6-4 shows the algorithm to set v→hpri. Figure 6-5 shows how the descendants
are moved downwards. In Figure 6-4, we first calculate the number of moves to set
v→hpri when descendants are moved downwards (childMoves) and when the ancestors
are moved upwards (parentMoves). These calculations are based on the flow diagram
in Figure 6-6(b). Suppose v→hpri is set by moving descendants downwards, and the
block number corresponding to the maximum hpri of the parent vertices is B. Then v is
assigned to a block B + 1 and no child vertex of v can be in a block lower than B + 2.
If a child vertex is found to be in a block lower than B + 2 by mapping the child's hpri,
then that child must be moved to an appropriate block, which could be either block B + 2
or some higher block such as B + 3, B + 4, etc. Such updating happens recursively
for all descendants as shown in Figure 6-5. The algorithm to set v→hpri by moving
ancestors upwards is similar. Moving either the descendants or the ancestors to adjust
priorities is computationally intensive, with a worst case complexity of \(O(NL)\), where \(N\)
is the number of vertices in the priority graph and \(L\) is number of vertices on the longest
path. \(L\) is also referred to as the maximum chain length of the priority graph. The worst
case happens when each vertex is connected to every other vertex. In that case, to find the minimum and the maximum hpri (the first two lines of Figures 6-4 and 6-5) the algorithms must touch all the vertices.

Calculating the number of moves is a compute intensive task too, with the same complexity of $O(NL)$ since the same algorithms are used, without actually moving the rules. So, to avoid a performance bottleneck, we perform these calculations selectively. Further, a maxLimit is set so that as soon as the number of moves exceeds maxLimit we stop further calculations. The flowchart in Figure 6-6(a) shows an unoptimized decision diagram that causes significant performance degradation. In this case, the actual number of moves is computed for both the cases when the descendants and the ancestors are moved. Whichever direction results in a lower number of moves, the priorities are adjusted for that direction.

The flowchart in the Figure 6-6(b) shows an optimized decision diagram, that breaks up the process into three stages and focuses on relative instead of actual number of moves. In the first stage of this flow, we calculate childMoves which is the number of moves needed to shift the descendants downward, with maxLimit set to 500. If childMoves is less than 50, we go ahead and move the descendants downwards without calculating parentMoves, which is the number of moves required to shift the ancestors upward. However, if it takes more than 50 childMoves, then we are at the second stage where the parentMoves are calculated with maxLimit set to (childMoves +100), which could potentially be a number up to 600. If parentMoves is less than childMoves at this stage, then we move the ancestors upwards. If parentMoves is more than 500 then we are at the third stage where the exact number of childMoves is first calculated (by setting maxLimit to infinity) and then a relative number of parentMoves is calculated (by setting maxLimit to (childMoves +100)). Descendants are moved downwards if childMoves is smaller, otherwise ancestors are moved upwards. This flow gives an acceptable update
A Inefficient flow

- Adjust ancestors
  - No
  - Set maxLimit to infinity
    - Compute childMoves
    - Compute parentMoves
  - parentMoves > childMoves?
    - Yes
    - Adjust descendants
  - No
    - Compute childMoves
    - Compute parentMoves
    - parentMoves > childMoves?
      - Yes
      - Adjust descendants
      - No

B Efficient flow

- maxLimit = 500
  - Compute childMoves
  - childMoves > 50?
    - No
    - Move descendants
    - Yes
      - Set maxLimit to infinity
      - Compute childMoves
      - maxLimit = childMoves + 100
      - Compute parentMoves

- maxLimit = childMoves + 100
  - Compute parentMoves
  - parentMoves > 500?
    - No
    - Compare with childMoves and move the descendants or the ancestors accordingly
    - Yes
      - Set maxLimit to infinity
      - Compute childMoves
      - maxLimit = childMoves + 100
      - Compute parentMoves

Figure 6-6. Decision diagrams for priority adjustment of descendants vs. ancestors

Performance on our datasets, since very few updates involve over 500 moves in either or both the directions.

We use another optimization in the ITCAM rule placement strategy, where a new block is inserted into the TCAM between two existing blocks as shown in Figure 6-3(e) and on lines 13 and 30 of Figure 6-4. If the maximum block number of the parents of \( v \) is \( B \) and the minimum block number of its children is \( B + 1 \), then instead of moving all children in block \( B + 1 \) to \( B + 2 \) or all parents in block \( B \) to \( B - 1 \), a new block is created in the ITCAM between the blocks \( B \) and \( B + 1 \) and \( v \rightarrow \text{hpri} \) is set to the average of the hpri-s of the two blocks (i.e. \( \text{hpri}_\text{of}(B) + \text{hpri}_\text{of}(B + 1) \) /2). The new rule for \( v \) is then added to the new block. If the new rule is to be added on top of the topmost ITCAM block as on line 27 of Figure 6-4, then \( v \rightarrow \text{hpri} \) is set to \( (1 + \text{hpri}_\text{of}(B) /2) \). Recall that the vertices with in-degree 0 are assigned an block number 1. So, we add 1 in this expression to ensure that no hpri becomes less than 1. Addition of a new block must
be done judiciously, since it requires an extra move while bringing in a free slot to a particular block $B$ when the newly inserted block is between the free space pool and $B$. So, we add new blocks only if the number of moves was calculated to be over 50. Figure 6-3(e) shows that $v \rightarrow hpri$ for the new vertex $v$ is set to 3.5. A new block is added between the parent and the child blocks in this case.

For consistent updates [25, 54], if the vertices are to be moved downwards, then the moves may be executed in increasing order of priority starting from the lowest priority rule and after all the descendants are moved, the new rule is added. If the vertices are moved upwards, then the moves may be executed in decreasing order of priority, starting from the highest priority rule. After all the ancestors are moved, the new rule is added. Lines 40-43 of Figure 6-4 ensure that nodes are moved to their assigned slots in the reverse order of visiting them. Thus, the node last visited for updating $hpri$ is the first to be moved to its assigned slot. This preserves update consistency for both the cases when the descendants are moved downwards and the parents upwards. The new rule is added at the end (Line 44).

### 6.1.2.2 Updating the TCAMs

TCAM updates are generated after updating the priority graph. Rules may be moved from the ITCAM to the LTCAM or vice versa or they may be moved within the ITCAM for rearrangement of overlapping rules. To insert or move a rule in a TCAM we need a free slot at an appropriate location. This slot can be obtained efficiently using memory management algorithms. In particular, the memory management schemes from DUOS may be used here. For the ITCAM of PC-DUOS+, we implemented the DLFS_PLO scheme, just like we did for PC-DUOS. For the LTCAM too, PC-DUOS and PC-DUOS+ use the same memory management algorithms.

Since the blocks grow both ways, up as well as down, PC-DUOS+ has a modified initial rule placement policy as shown in Figure 6-7 where 25% of the free slots (represented by white blocks) are placed on the top of the TCAM (that is, covering
the lowest addresses) and another 25% are kept at the bottom of the TCAM (covering the highest addresses). The remaining 50% of the free slots are distributed to the region between the blocks in proportion to the number of rules in a block.

**ITCAM.insert**: To insert a new rule in the ITCAM, a free slot is first made available at the desired block. A free slot may be present in the same block in which case no moves are needed to get it from the free slot list of the block. If there is no free slot in the block, then a free slot may be obtained from the inter-block region on the top or the bottom of the block. No moves are needed in this case too. If there is no free slot in the inter-block region adjacent to the block, then a free slot is moved from the nearest neighboring block where its available.

To insert a new block between two blocks in the ITCAM, it is first checked if there is a free slot between the top and bottom blocks. If there are free slots in the region between the top and the bottom blocks, then the rule in the new block is inserted there in such a way that there are some free slots above and below the new block. Otherwise, free slots for the new block are moved in from the nearest neighboring block that has free slots.
**ITCAM.delete**: After deleting the vertex corresponding to the rule in the priority graph, the valid bit on the corresponding TCAM slot is set to 0. DLFS_PLO frees up the block if the rule deleted is the last rule in the block. Otherwise, the freed slot is prepended to the head of the list of free slots in the block.

**ITCAM.change**: Suppose the specified change is with respect to the fields of a rule, then such a change is implemented as an insert followed by a delete. The insert adds the changed rule to the same block as the old rule, while the delete removes the old rule from this block. If the change is in the priority of the rule, then, we revisit all the incoming and outgoing edges of the corresponding vertex \( v \) in the priority graph and reverse the edges appropriately to maintain the edge direction from the higher to the lower priority rule. Then the block number is freshly calculated for \( v \), and the rule is moved to a block at a higher address (if the priority was lowered) or to a block at a lower address (if the priority of the rule was increased) in the ITCAM. If the vertex \( v \) does not have any incoming edge following the update, it is moved to the LTCAM.

**LTCAM.insert, LTCAM.delete and LTCAM.change**: To insert a new rule in the LTCAM, a free slot is obtained from the head of the LTCAM free slot list. If a rule is deleted from the LTCAM, then the valid bit of the slot is set to 0 and the freed up slot is prepended to the head of the free slot list.

For incorporating a changed rule, if the change is with respect to the fields of a rule, then the changed rule is simply inserted in the LTCAM and the old rule deleted. If the change is in the priority of a rule in such a way that the corresponding vertex now has an incoming edge, then the rule is moved to the ITCAM. Otherwise, if the rule continues to be the highest priority rule among all overlapping rules even after the change, then nothing needs to be done.
6.2 Experimental Results

The experimental setup is first described in Section 6.2.1. We analyze the results based on two perspectives – improvement in lookup performance and improvement in update performance in Sections 6.2.2 and 6.2.2.1 respectively.

6.2.1 Setup

6.2.2 Lookup Performance

Figure 6-8. Percentage of rules stored in the LTCAM and percentage improvement in lookup time compared to STCAM architecture

Recall that during a lookup, if a match is found in LTCAM of PC-DUOS+ then the corresponding action is returned faster. Having a large number of rules in the LTCAM makes the probability of finding a match in the LTCAM, higher. Figure 6-8(b) shows
the improvement in average lookup time. Since the tests 1 (acl1) and 9 (acl5) had 99% of their rules in the LTCAM, almost all the lookups found a hit in the LTCAM, and consequently, the improvement in average lookup time on these tests was almost 50%. On the other hand, the test fw1 had least hits in the LTCAM, and showed an improvement of about 19% in the average lookup time. Figure 6-1 presents the details on the number of rules in the ITCAM and LTCAM and the percentage improvement in lookup performance. The first three columns give the dataset index, its name and the number of rules respectively. The fourth and fifth columns give the number of rules entered in the ITCAM and LTCAM, respectively. The sixth and seventh columns give, respectively, the number of lookups performed and the percentage improvement in average lookup time.

The percentage improvement values for lookup performance statistically depend on the number of rules in the LTCAM. Thus the actual improvement, obtained based on the distribution of lookups in a trace served by the ITCAM and the LTCAM, may deviate from the expected improvement. For example, from Figures 6-8(a) and (b), test 13 shows better improvement in lookup time compared to test 12, even though both the tests contain about the same percentage of rules in the LTCAM.

Table 6-1. Number of rules in ITCAM and LTCAM of PC-DUOS+ and improvement in lookup time relative to STCAM

<table>
<thead>
<tr>
<th>Index</th>
<th>Dataset</th>
<th>#Rules</th>
<th>#ITCAM</th>
<th>#LTCAM</th>
<th>#Lookups</th>
<th>%Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>acl1</td>
<td>30075</td>
<td>305</td>
<td>29731</td>
<td>120301</td>
<td>49.6</td>
</tr>
<tr>
<td>2</td>
<td>fw1</td>
<td>7989</td>
<td>4885</td>
<td>3068</td>
<td>103857</td>
<td>19.2</td>
</tr>
<tr>
<td>3</td>
<td>ipc1</td>
<td>15338</td>
<td>3504</td>
<td>11834</td>
<td>107618</td>
<td>40.6</td>
</tr>
<tr>
<td>4</td>
<td>acl2</td>
<td>53970</td>
<td>8875</td>
<td>45095</td>
<td>107940</td>
<td>42.6</td>
</tr>
<tr>
<td>5</td>
<td>fw5</td>
<td>5571</td>
<td>2689</td>
<td>2796</td>
<td>105430</td>
<td>30.8</td>
</tr>
<tr>
<td>6</td>
<td>acl4</td>
<td>34254</td>
<td>5882</td>
<td>28372</td>
<td>103104</td>
<td>41.1</td>
</tr>
<tr>
<td>7</td>
<td>ipc2</td>
<td>5165</td>
<td>1476</td>
<td>3689</td>
<td>98136</td>
<td>38.7</td>
</tr>
<tr>
<td>8</td>
<td>acl3</td>
<td>19745</td>
<td>6737</td>
<td>13007</td>
<td>102851</td>
<td>31.4</td>
</tr>
<tr>
<td>9</td>
<td>acl5</td>
<td>19492</td>
<td>260</td>
<td>19209</td>
<td>97460</td>
<td>49.4</td>
</tr>
<tr>
<td>10</td>
<td>fw2</td>
<td>16668</td>
<td>4739</td>
<td>11929</td>
<td>100008</td>
<td>40.6</td>
</tr>
<tr>
<td>11</td>
<td>fw3</td>
<td>16841</td>
<td>5688</td>
<td>10986</td>
<td>103794</td>
<td>33.0</td>
</tr>
<tr>
<td>12</td>
<td>fw4</td>
<td>12882</td>
<td>5004</td>
<td>7878</td>
<td>103266</td>
<td>20.2</td>
</tr>
<tr>
<td>13</td>
<td>ipc3</td>
<td>20000</td>
<td>8027</td>
<td>11973</td>
<td>100163</td>
<td>34.9</td>
</tr>
</tbody>
</table>
6.2.2.1 Update performance

Figure 6-9 shows the number of TCAM writes needed to process the test update sequence by PC-DUOS+, PC-DUOS and STCAM, normalized with respect to that of PC-DUOS+. A noticeable improvement in the number of writes is observed with respect to STCAM for almost all the tests except for tests 9 (acl5) and 13 (ipc3). Test 8 (acl3) requires up to 3.72 times more writes using an STCAM compared to PC-DUOS+,
while test6 (acl4) requires up to 1.5 times the number of writes using PC-DUOS versus PC-DUOS+.

Figure 6-10 shows the time taken to process the updates by PC-DUOS+, PC-DUOS and STCAM. These times have been normalized with respect to PC-DUOS+. Tests 1 (acl1) and 9 (acl5) show the maximum improvement in runtime compared to STCAM, the improvement being 247 and 188 times respectively. This is related to the fact that over 99% of the rules in these tests are entered in the LTCAM of PC-DUOS+. The LTCAM offers a fast and light-weight update mechanism compared to the ITCAM. Note that the ITCAM has a similar update mechanism as STCAM. In fact, from Figures 6-10 and 6-8(a), we see that the improvement in runtime is closely related to the number of rules that are in the LTCAM. Figure 6-10 shows that compared to PC-DUOS, there is an improvement in the runtime too, for all tests except test 1 (acl1).

![Figure 6-11. Ratio of edges to vertices of graph](image)

From Figure 6-9 we observe that tests 9 (acl5) and 13 (ipc3) need almost similar number of writes in all the three setups, namely, PC-DUOS+, PC-DUOS and STCAM. The priority graph for test 9 (acl5) has a very small number of edges. In fact, the ratio of the edges to the vertices for this graph is only 0.018 (Figure 6-11), and the length of the maximum chain is just 3 (Figure 6-12). Thus, STCAM needs a single write for most of the inserts. The priority graph for test 13 (ipc3), on the other hand, is a well-structured
Figure 6-12. Maximum chain length in graph before processing updates

Figure 6-13. A small graph representing test ipc3

Figure 6-14. Percentage of updates that require 1 write, ≤ 3 and ≤ 10 writes
graph with three distinct types of vertices. The first type is for rules with very specific source and destination prefix, which are specified up to 32 bits for most cases. The second type is for rules with very specific source address prefix, but generic destination prefix (0 or 1 bit long), and the third type is for rules with very specific destination and generic source address prefix. As a result, the vertices of a particular type are sparsely connected to each other as they fail to match on the source or destination prefix field that is specified up to 32 bits. Figure 6-13(a) represents a small example of such a graph. Here the rules at the top level are placed in block number 1, the rules at the next level are placed in block number 2 and the rules on the last level are placed in block number 3 in the TCAM. Now suppose an insert request for a new rule is received. Figure 6-13(b) shows a new vertex corresponding to the rule, and an updated priority graph. As can be seen, the highest block number for a parent is 1, and the lowest block number for a child of the new vertex is 3, which makes the new vertex a perfect fit in block number 2. The graph for ipc3 is close to this example, with only 6 blocks and 100% of the rules are placed in the right block with just one TCAM write. The fact that 100% of the rules need just 1 TCAM write can be seen from Figure 6-14, which shows the percentage of rules requiring 1 TCAM write and the percentage of rules requiring at most 3 and 10 TCAM writes. Thus, ipc3 produces similar results for PC-DUOS+, PC-DUOS as well as the STCAM. It may be noticed that a common feature of tests acl5 and ipc3 is that both of them have a small maximum chain length.

Table 6-2 gives the average and the worst case TCAM writes for PC-DUOS+ and STCAM. The average writes for PC-DUOS+ are lower than the corresponding numbers for STCAM. The worst case writes for PC-DUOS+ is lower than those for STCAM for all tests except test 10 (fw2). The number of TCAM writes in the worst case for PC-DUOS+ is quite high, even though we observed that more than 99% of the rules require at most 10 writes.
Table 6-2. Average and worst case TCAM writes for PC-DUOS+

<table>
<thead>
<tr>
<th>Index</th>
<th>DataSet</th>
<th>PC-DUOS+</th>
<th>STCAM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>#Average writes</td>
<td>#Worst case writes</td>
</tr>
<tr>
<td>1</td>
<td>acl1</td>
<td>1.18</td>
<td>31</td>
</tr>
<tr>
<td>2</td>
<td>fw1</td>
<td>2.07</td>
<td>3971</td>
</tr>
<tr>
<td>3</td>
<td>ipc1</td>
<td>1.52</td>
<td>1900</td>
</tr>
<tr>
<td>4</td>
<td>acl2</td>
<td>1.56</td>
<td>4274</td>
</tr>
<tr>
<td>5</td>
<td>fw5</td>
<td>1.6</td>
<td>5167</td>
</tr>
<tr>
<td>6</td>
<td>acl4</td>
<td>1.43</td>
<td>418</td>
</tr>
<tr>
<td>7</td>
<td>ipc2</td>
<td>2.04</td>
<td>5152</td>
</tr>
<tr>
<td>8</td>
<td>acl3</td>
<td>1.92</td>
<td>651</td>
</tr>
<tr>
<td>9</td>
<td>acl5</td>
<td>1.12</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>fw2</td>
<td>1.71</td>
<td>9</td>
</tr>
<tr>
<td>11</td>
<td>fw3</td>
<td>1.55</td>
<td>2603</td>
</tr>
<tr>
<td>12</td>
<td>fw4</td>
<td>2.83</td>
<td>1962</td>
</tr>
<tr>
<td>13</td>
<td>ipc3</td>
<td>1.0</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 6-3. Total TCAM writes in PC-DUOS+, PC-DUOS and STCAM

<table>
<thead>
<tr>
<th>Index</th>
<th>Data-Sets</th>
<th>PC-DUOS+</th>
<th>PC-DUOS</th>
<th>STCAM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>#TCAM writes</td>
<td>Time(s)</td>
<td>#TCAM writes</td>
</tr>
<tr>
<td>1</td>
<td>acl1</td>
<td>116418</td>
<td>10</td>
<td>116393</td>
</tr>
<tr>
<td>2</td>
<td>fw1</td>
<td>76792</td>
<td>385</td>
<td>105866</td>
</tr>
<tr>
<td>3</td>
<td>ipc1</td>
<td>74059</td>
<td>128</td>
<td>88736</td>
</tr>
<tr>
<td>4</td>
<td>acl2</td>
<td>139397</td>
<td>193</td>
<td>148727</td>
</tr>
<tr>
<td>5</td>
<td>fw5</td>
<td>82044</td>
<td>969</td>
<td>113358</td>
</tr>
<tr>
<td>6</td>
<td>acl4</td>
<td>14357</td>
<td>8</td>
<td>22030</td>
</tr>
<tr>
<td>7</td>
<td>ipc2</td>
<td>201082</td>
<td>557</td>
<td>296663</td>
</tr>
<tr>
<td>8</td>
<td>acl3</td>
<td>11736</td>
<td>16</td>
<td>11798</td>
</tr>
<tr>
<td>9</td>
<td>acl5</td>
<td>28104</td>
<td>0.88</td>
<td>27366</td>
</tr>
<tr>
<td>10</td>
<td>fw2</td>
<td>51250</td>
<td>161</td>
<td>51402</td>
</tr>
<tr>
<td>11</td>
<td>fw3</td>
<td>77516</td>
<td>767</td>
<td>92955</td>
</tr>
<tr>
<td>12</td>
<td>fw4</td>
<td>57822</td>
<td>76</td>
<td>69958</td>
</tr>
<tr>
<td>13</td>
<td>ipc3</td>
<td>30000</td>
<td>217</td>
<td>30000</td>
</tr>
</tbody>
</table>

Table 6-3 shows the actual number of TCAM writes for inserting or deleting rules in the different datasets and the time taken to perform these updates in PC-DUOS+, PC-DUOS and STCAM.

PC-DUOS+, which is an enhancement of PC-DUOS [28] and an extension of DUOS[26], is proposed in this chapter for packet classifier lookup and update. Two TCAMs named LTCAM and ITCAM are used. PC-DUOS+ stores the highest priority
independent rules in the LTCAM. The remaining rules are stored in the ITCAM. During lookup for highest priority rule matching, both the ITCAM and the LTCAM are searched in parallel. Since the LTCAM stores independent rules, at most one rule may match during lookup in the LTCAM and a priority encoder is not needed. If a match is found in the LTCAM during lookup, it is guaranteed to be the highest priority match and the corresponding action can be returned immediately yielding up to 50% improvement in TCAM search time relative to STCAM. The average improvement in lookup time is found to be between 19% to 49% for the tests in our dataset. The distribution of rules to the two TCAMs makes updates faster by reducing the average number of TCAM writes by up to 3.72 times (for acl3) and reducing the control-plane processing time by up to 247 times (for acl1). The maximum reduction in control-plane processing time is observed for the ACL tests.
CHAPTER 7
PC-TRIO: AN INDEXED TCAM ARCHITECTURE FOR PACKET CLASSIFIERS

PC-TRIO is an indexed, triple TCAM architecture that supports faster and low power lookups, in addition to, efficient incremental updates. The chapter is organized as follows. Section 7.1 presents background and related work in the field of packet classification. Section 7.1.1 describes the PC-TRIO architecture and associated algorithms and Section 7.4 presents experimental results. We conclude in Section ??.

7.1 Background and Related Work

We will describe the research on TCAM based packet classifiers in Section 7.1.1, and discuss the main problems in having an indexed TCAM architecture for packet classifiers in Section 7.1.2 and then in Section 7.1.3 show how to overcome these problems.

7.1.1 Packet Classifiers

The work on packet classifiers in TCAMs, targets three main problems: port range expansion, power consumption and updates. The first two problems are inter-related as reducing port range expansion also reduces the power consumption in a TCAM. Various approaches have been proposed in the literature to alleviate the range expansion problem. The schemes in [8, 11, 18, 20, 22, 33] encode the ranges and store modified rules in the TCAM. As a packet arrives, an encoded search key is created from the packet header fields using the encoding algorithm and the TCAM is searched using the encoded search key. Spitznagel et al. [44] proposed enhancements to the TCAM hardware to include range comparison. With such an enhanced TCAM circuit, each rule occupies a single entry in the TCAM.

Compressing packet classifiers by removing redundancies is an effective strategy to reduce TCAM power consumption. The approaches in [13, 14, 19, 24, 49] present algorithms that transform an input classifier to an equivalent smaller classifier. These algorithms quite naturally contain port range expansions. While these approaches
bring about significant reductions in classifier size, they are generally not suitable for incremental updates, since a rule to be deleted, for instance, may not be present in the transformed classifier.

Song and Turner [43] describe an algorithm for fast incremental filter updates. An explicit priority value (which we call block number in this paper) is calculated for each rule based on the rule’s implicit priority, which is derived from the position of the rule in the classifier, and the implicit priority values of the overlapping rules. The block number so computed is stored along with the rule in the TCAM using unused TCAM bits. A new rule may be placed anywhere in the TCAM. This relieves the TCAM of moving existing rules to maintain priority ordering. Instead, during lookup, multiple lookups per packet are performed to identify the best matching rule. Mishra, Sahni and Seetharaman in PC-DUOS [28] and PC-DUOS+ [29] use dual TCAMs for representation and incremental update of classifiers.

7.1.2 Problems in Storing a Classifier in an Indexed TCAM

We have seen how packet forwarding tables are stored in indexed TCAMs associated with wide SRAMs in Chapter 2, thereby resulting in significant decrease in lookup power. It is expected that storing packet classifiers in indexed TCAMs with wide SRAMs would yield power savings in a similar manner. However, there are two problems in mapping a packet classifier to an indexed TCAM architecture with wide SRAMs. Recall that during a TCAM lookup, the contents in the SRAM word corresponding to the first matching rule is returned. A constraint on the size of a wide SRAM word (and also that on the size of a TCAM bucket), makes it impossible to guarantee that the first matching word will contain the highest priority rule matching the packet. For example, consider the classifier with 4 rules in Figure 7-1, where each rule has two fields - a destination, and a source. The classifier is mapped to the indexed TCAM in Figure 7-2. The data TCAM has two buckets and the index TCAM uses bits from the destination prefix of each rule, to index into the buckets of the data TCAM. In this
Figure 7-1. An example classifier

<table>
<thead>
<tr>
<th>Filter</th>
<th>Action</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>Destination</td>
<td>Source</td>
<td></td>
</tr>
<tr>
<td>00*</td>
<td>1000</td>
<td>A1</td>
</tr>
<tr>
<td>0*</td>
<td>0101</td>
<td>A2</td>
</tr>
<tr>
<td>000*</td>
<td>01*</td>
<td>A3</td>
</tr>
<tr>
<td>*</td>
<td>*</td>
<td>A4</td>
</tr>
</tbody>
</table>

Figure 7-2. Classifier rules stored in a indexed TCAM

setup, assuming that addresses are 4 bits, suppose a packet arrives with destination and source addresses as 0000 and 0101 respectively. The best matching rule from Figure 7-2 is the second rule on the first bucket of the data TCAM and A3 is returned as the action to be applied on the packet. However, from the table in Figure 7-1, A2 is the desired action. Thus if there are multiple matching rules on a TCAM, then all the corresponding SRAM words must be processed to return the action of the matching rule with the highest priority, and this will take more than one TCAM clock cycle to finish a search. This is the first problem.

The second problem is about the covering rules of a wide SRAM word or a data TCAM bucket. A covering prefix [21, 55], in the context of packet forwarding tables, is a default prefix for a TCAM bucket. The presence of covering prefixes in a TCAM bucket makes every search in the TCAM bucket return at least one match. In a packet classifier, covering rules similarly guarantee that a search on a TCAM bucket matches at least one rule. The fourth rule in Figure 7-1 is a covering rule and hence entered in both the TCAM buckets in Figure 7-2. A packet classifier may have several covering rules for a TCAM bucket. Further, different TCAM buckets may need the same covering
rules which makes it necessary to store a single rule multiple times in the TCAM, once in every TCAM bucket for which it is a covering rule. Having a rule replicated as such in the TCAM, is unacceptable specially considering the fact that the replicated rules themselves may undergo range expansion.

### 7.1.3 Overcoming the Problems

The dual TCAM architecture presented for PC-DUOS in Chapter 5 and PC-DUOS+ in Chapter 6, as well as the PC-TRIO architecture presented in this chapter, makes it possible to get around both the problems mentioned about using wide SRAMs and index TCAMs with a TCAM for packet classifiers. The LTCAM (Leaf TCAM) of PC-DUOS stores independent rules. Two rules are independent iff no packet matches both the rules. Storing a set of independent rules in a TCAM, ensures that at most one TCAM entry matches during a search and we simply process the corresponding SRAM word. The ITCAM (Interior TCAM) of PC-DUOS stores all the remaining rules which includes the covering rules. During a lookup both TCAMs are searched in parallel, and in case there is no match on the LTCAM, the ITCAM returns the action for the matching rule with the highest priority. Note that the LTCAM of PC-DUOS is a suitable candidate for augmenting with wide SRAM words and an index TCAM, since at most one TCAM entry matches during a search. The rules in the ITCAM, on the other hand, are not independent and hence multiple TCAM entries will match during a search. Thus, the ITCAM is not a suitable candidate for using with it a wide SRAM or an index TCAM.

### 7.2 PC-TRIO

The PC-TRIO architecture is presented in Section 7.2.1. The algorithms for storing and updating the TCAMs are discussed in Sections 7.2.2 and 7.2.3.

#### 7.2.1 The Architecture

Figure 7-3 illustrates the PC-TRIO architecture. It primarily consists of three TCAMs, the ITCAM (Interior TCAM), the LTCAM1 (Leaf TCAM) and the LTCAM2. The corresponding associated SRAMs are: ISRAM, LSRAM1 and LSRAM2, respectively.
The LTCAMs store independent rules, hence both the TCAMs are augmented with wide SRAMs and index TCAMs. ILTCAM1 and ILTCAM2 are the index TCAMs for LTCAM1 and LTCAM2, respectively. The index TCAMs also have wide associated SRAMs, namely, ILSRAM1 and ILSRAM2. Since the rules stored in the two LTCAMs and the two ILTCAMs are independent, at most one rule (in each LTCAM and ILTCAM) will match during a search. So these TCAMs do not need a priority encoder. A priority encoder assists in resolving multiple TCAM matches and is used with the ITCAM to access the ISRAM word corresponding to the highest priority matching rule in the ITCAM.

A lookup in PC-TRIO is pipelined with 6 stages marked A-F in Figure 7-3. In the first stage A, the ILTCAMs are searched. The ILSRAMs are accessed, using the address of the matching ILTCAM1 and ILTCAM2 entries in stage B. The matching wide ILSRAM words are processed in stage C to obtain the corresponding bucket index for
LTCAM1 and LTCAM2. In stage D, the bucket indexes so obtained are used to search the corresponding buckets in the LTCAMs. The ITCAM is also searched in this stage. In the next stage E, the ISRAM, and the LSRAMs are accessed using the addresses of the matching TCAM entries. In the final stage F, the contents of the wide LSRAM words are processed and the best action is chosen from the at most three actions returned by the ISRAM, LSRAM1 and LSRAM2 by comparing the priorities of the corresponding rules.

7.2.2 Storing Rules in TCAMs

There are several steps of processing a packet classifier to store the rules in the TCAMs. The first step is to create a priority graph and multi-dimensional tries for the rules in the classifier. This is further discussed in Section 7.2.2.1. In the second and third steps, the LTCAM1 and LTCAM2 subsystems are populated as discussed in Sections 7.2.2.2 and 7.2.2.3, respectively. The fourth step is to store the remaining rules in the ITCAM in priority order, which is discussed in Section 7.2.2.4.

7.2.2.1 Representing classifier rules

A priority graph is created first which contains one vertex for each rule in the classifier. Next we create a multi-dimensional trie, Trie1, where each dimension represents one field of a rule. Initially, Trie1 is three-dimensional, with the three fields, source, destination and protocol of a classifier rule used for this purpose. The fields appear in the following order in the trie: <destination, source, protocol>. We assume that the destination and source fields as well as the protocol field of the filters are specified as prefixes. So, these are represented in a trie in the standard way with the left child of a node representing a 0 and the right child a 1. A classifier rule, along with its source and destination port ranges, is stored on the protocol node that is arrived at after traversing the trie starting from its root, using first the destination, then the source and finally the protocol fields of the rule.

We identify a set of independent rules as described in Section 7.2.2.2. All the remaining rules are used to create another multi-dimensional trie, Trie2, in which fields
**Algorithm: findNode(node)**

**Inputs:**
node: a trie node, initially set to the root of a multi-dimensional trie.

**Output:**
a leaves of leaves set of protocol nodes storing classifier rules.

```plaintext
for each child i of node
    findNode(node→child[i]);
endfor
if (node is a leaf) // true if node has no left or right child.
    if (node contains root of a trie)
        findNode(node→trie→root);
    else // node belongs to trie for the last field (protocol)
        append protocol node to leaves of leaves set
    endif
endif
```

---

**Figure 7-4. Selecting protocol nodes for leaves of leaves set**

<table>
<thead>
<tr>
<th>Match start position</th>
<th>Count</th>
<th>len (S1)</th>
<th>len (S2)</th>
<th>...</th>
<th>len (Sk)</th>
<th>C1</th>
<th>CK</th>
<th>Data1</th>
<th>DataN</th>
<th>S1</th>
<th>...</th>
<th>SK</th>
<th>S. Port Range1</th>
<th>D. Port Range1</th>
<th>...</th>
<th>S. Port RangeN</th>
<th>D. Port RangeN</th>
</tr>
</thead>
</table>

![Data encoding diagram]

**Figure 7-5. Data encoding in a wide SRAM word**

in a filter rule appear in the order `<source, destination, protocol>`. Note that the source and destination tries are switched in Trie2, with respect to Trie1. So, while destination trie is the outermost trie in Trie1, in Trie2, source is the outermost trie.

### 7.2.2.2 Storing rules in the LTCAM1

The process of storing rules in the LTCAM1 subsystem is described in five subsections below. First, we discuss how independent rules are identified, next, the format of storing information in a wide LSRAM word is discussed, then we describe the creation of LTCAM1 entries using the process of carving. Next we describe partial port range expansion that may be necessary, and finally, the creation of ILTCAM1 and ILSRAM1 entries.

**Identifying Independent Rules.** To find a large set of independent rules in PC-TRIO, we first create a *leaves of leaves set* of protocol nodes in a multi-dimensional trie using the algorithm in Figure 7-4. The nodes belonging to the leaves of leaves set
in Trie1 are obtained by traversing the multi-dimensional trie from the root to the leaves of the destination trie and then from these leaves into their attached source trie and then from the leaves of the source trie into the leaves of their attached innermost trie for the protocol field.

In the second step, for each protocol node in the leaves of leaves set, we identify a set of independent rules stored in that protocol node by building a small priority graph with rules only in that protocol node. Vertices in the priority graph with in-degree 0 comprise a set of independent rules. A collection of independent rules from all protocol nodes in the leaves of leaves set, gives us the rules to be entered in the LTCAM1.

**Wide SRAM Word Format.** Once the rules to be stored in LTCAM1 are identified, subtries of the multi-dimensional trie are carved and rules in the protocol nodes in a subtrie are stored in a LSRAM1 word. In particular, for each rule in a protocol node we store the rule’s source and destination port ranges, block number, and action. We also store the suffix of a protocol node, which is the path from the root of the carved subtrie to the protocol node. Figure 7-5 shows a format for encoding this information in a wide SRAM word. The fields in this format are described briefly as follows:

1. **Match start position:** This field specifies the positions of the first bit in the source, destination and protocol fields of a packet header starting from which suffixes of protocol nodes in the SRAM word must be matched.

2. **Count**: This is the number of protocol nodes in the leaves of leaves set stored in the SRAM word.

3. **len(Si)**: This field specifies the length of the suffix for protocol node $i$ in the SRAM word.

4. **Ci**: This gives the number of classifier rules stored for protocol node $i$.

5. **Data$_j$; Data$_1$, \ldots, Data$_N$** give details of the $N$ rules in the carved subtrie. The rules for protocol node 1 of this subtrie come first, followed by those of the second protocol node and so on. $Data_j$ gives the block number, action, source and destination port range types for the $j$th classifier rule.

6. **Si**: This field stores the suffix for protocol node $i$. 
7. **Port ranges**: Stores the port ranges for the \( N \) rules.

There are three types of ranges found in a classifier. These are: a whole range ([0-65535]), a range with the same start and end point, and a range with different start and end points. The port range type subfield in the Data field represents these three types of ranges using two bits. To save space in a SRAM word, a whole range is never entered and only one port number is entered for a range with the same start and end points.

**Creating LTCAM1 entries.** A trie is carved into subtries to assign rules to the wide SRAM words. The Trie1 is carved using the carving heuristic \textit{visit\_postorder} of DUO (see Chapter 3) that has been enhanced for multi-dimensional tries. This carving heuristic creates independent (disjoint) entries for the LTCAM1. The path starting from the root of

![Figure 7-6. Nodes in a source trie is being carved.](image)

from destination trie
with prefix 1101

**Figure 7-6. Nodes in a source trie is being carved.**

Trie1 to the root of the subtrie defines an LTCAM1 entry. Figure 7-6 shows a portion of a source trie that hangs off a destination trie, where carving takes place at nodes 00, 01, and 11 of the source trie. The path from the root to the node of the destination trie from which the source trie hangs off is 1101. Thus, after carving the node at 00 on the source trie, the LTCAM1 entry is 1101 00?? ????, assuming addresses and protocol fields are represented using 4 bits each. Similarly, the two other LTCAM1 entries in this example are 1101 01?? ???? and 1101 11?? ????. Figure 7-6 also shows a size assignment (in bits) on the three nodes where carving takes place. These sizes are computed for all the trie nodes even before the carving algorithm is invoked. The size assigned to a trie node represents the number of LSRAM1 bits needed to store all the classifier
rules (for LTCAM1) in a subtrie rooted at that node. For example, for a subtrie rooted at the source node 01, the number of bits needed to store the action, block number, port ranges of classifier rules and suffixes of protocol nodes present in this subtrie, is 450. If the actual width of a SRAM word is, say, 500 bits, then the rules in this subtrie will fit in an SRAM word and we may carve at the source node 01. A corresponding LSRAM1 entry is constructed for the classifier rules in the format given by Figure 7-5. The carving heuristic carves a node \( n \) on the trie when any of the following two conditions is true. Here, \( p \) is the parent of \( n \) in the trie.

**C1)** The size assigned to \( n \) is less than the width of a SRAM word, but that assigned to \( p \) is more than the width of a SRAM word.

**C2)** A descendant of \( p \) was carved.

The second condition ensures that the carving creates disjoint TCAM entries (see Chapter 3).

**Partial port range expansion.** It is possible that the SRAM bits needed to store the classifier rules for LTCAM1 on a protocol node exceeds the capacity of a wide SRAM word. This case is shown in Figure 7-7(a) where the black node is a protocol node in the leaves of leaves set and the size assigned to it is 600 bits. Suppose the width of the SRAM word is 500 bits. Then to avoid overflowing an SRAM word, we must split the rules in the protocol node, into two or more SRAM words. Instead of replicating the LTCAM1 entry for each of the split SRAM words, we create a source port range trie as shown in Figure 7-7(b), and carve nodes on this trie to create independent LTCAM1 entries. Each node in the source port trie inherits those classifier rules (for LTCAM1) from the protocol node that have their source port range overlap with the port range represented by the trie node. Thus multiple copies of a rule may be created, one for each trie node with port range overlapping the source port range of the rule. After the source port trie is created, the carving heuristic resumes its traversal along the source port trie, and carves source port nodes if they satisfy either condition C1, or C2. In
the example of Figure 7-7(b), two LTCAM1 entries are created, one each for the two carved nodes. These LTCAM1 entries differ on the first bit on the source port field, with one entry having a 0 while the other having a 1. If the classifier rules in a leaf node of the source port trie overflows an SRAM word, then a destination port trie is created for the destination port ranges on rules of that leaf node, and the carving heuristic finds appropriate nodes to carve on the destination port trie.

The source and destination port tries are thus created in PC-TRIO only when necessary, and then, to minimize the range expansion problem we use multi-bit tries for storing the port ranges. The bits used to arrive at a node in the multi-bit trie define an LTCAM1 entry.

**Creating ILSRAM1 and ILTCAM1 entries.** After carving Trie1 to create suffixes for entering into LSRAM1, we carve Trie1 again a second time, to create subtries that contain LTCAM1 entries. All LTCAM1 entries in a subtrie are entered in a LTCAM1 bucket. Thus, at the end of this carving step, the LTCAM1 entries are partitioned into buckets. The bits from the root of the multi-dimensional trie to a carved node defines an index that points to an LTCAM1 bucket.

After partitioning the LTCAM1 into buckets, Trie1 is carved a third and final time. This time, a carved subtrie contains indexes to LTCAM1 buckets. Suffixes of these indexes, along with the corresponding LTCAM1 bucket indexes, are stored in the
ILSRAM1, and the bits on path from the root of the Trie1 to a carved node define an ILTCAM1 entry.

7.2.2.3 Storing rules in LTCAM2

This is done exactly as for LTCAM1, by processing the rules stored in Trie2. In particular, Trie2 undergoes carving in a similar manner as described for Trie1 and the LTCAM2 system is populated. The remaining rules, i.e. rules that are stored neither in the LTCAM1 nor in the LTCAM2 subsystem, are stored in the ITCAM.

7.2.2.4 Storing rules in the ITCAM

The ITCAM does not have a wide ISRAM, hence, a rule to be entered in the ITCAM, must have its port range stored in the ITCAM itself. An ISRAM word contains the action and block number of a classifier rule stored in the corresponding ITCAM entry. We use DIRPE to encode these port ranges on the ITCAM. DIRPE is suitable for incremental updates, unlike database dependent range encoding schemes. However, if fast incremental updates are not needed, then any range encoding scheme may be chosen for the ITCAM.

7.2.3 Incremental Updates

When an update request is received, the priority graph is updated as described in Section 7.2.3.1. Then Trie1 and, if necessary, Trie2 are updated as described in Section 7.2.3.2. As the tries are updated, it may be necessary to carve the tries at different trie nodes. This is discussed in Section 7.2.3.3. Updating the TCAMs is discussed in Section 7.2.3.4.

7.2.3.1 Updating the priority graph

To insert a new rule, the first step is to store the rule in the priority graph. A new vertex $v$ is created for the rule. The existing rules that overlap with $v$ are identified and new edges are formed between $v$ and the vertices of overlapping rules, with directions of the edges set from the higher to the lower priority rules. Then, a block number is assigned to $v$, which is one more than the maximum block number of the nodes from
which \( v \) has an incoming edge. If the block number of a child vertex is not more than that assigned to \( v \), the child's block number is updated so that it is at least one more than the block number of \( v \). If the rule \( r \) corresponding to this child vertex is stored in ITCAM, then, \( r \) must be moved to the ITCAM block represented by its updated block number, and the ISRAM entry for \( r \) is also updated with the changed block number. On the other hand, if \( r \) is in one of the LTCAMs, then, we simply change \( r \)'s block number in the corresponding LSRAM entry. Updates to block numbers are propagated to all vertices reachable from \( v \).

To process a delete request, the vertex corresponding to the rule along with the incident edges is removed from the priority graph.

### 7.2.3.2 Updating the tries

To insert a new rule, the rule is first added to Trie1. If the rule is an independent rule in a protocol node in the leaves of leaves set, then it is inserted in the LTCAM1. Otherwise, the rule is added to Trie2. If the rule is an independent rule in a protocol node in the leaves of leaves set for Trie2, then the rule is inserted in the LTCAM2. Otherwise, the rule is inserted in the ITCAM.

If a new rule is stored in the LTCAM1 or the LTCAM2, then some of the existing rules in that TCAM may no longer be independent. If such a non-independent rule exists in the LTCAM1, then that rule is added to the Trie2 and if the rule can be added to the LTCAM2 it is moved from the LTCAM1 to the the LTCAM2. Otherwise, the rule is moved from the LTCAM1 to the ITCAM. Similarly, a new rule added to the LTCAM2 may cause some of the existing LTCAM2 rules to be moved to the ITCAM.

To delete a rule, the rule is deleted from Trie1 and also from Trie2 if it was stored in Trie2. The rule is then deleted from the TCAM that stores the rule.

### 7.2.3.3 Updating the trie carving

We now discuss the dynamics of creation and merging of LSRAM words when a new rule is added or an existing rule is deleted. Both Trie1 and Trie2 contain nodes
that were carved to create TCAM and SRAM entries. We describe how these entries change for Trie1. The process is similar for Trie2. When a rule is added to Trie1 at node $t$, if there is an ancestor $a$ of $t$, where carving was done to create a wide LSRAM1 word $s$, and if there is space in $s$ to place the action, block number, port ranges of the new rule, then, the new rule is placed in $s$. If there is no space in $s$, then the contents of $s$ are split, by carving descendants of $a$ to create two or more LTCAM1 entries. If, on the other hand, $t$ does not have an ancestor $a$, then one of the two things below may happen. If there is an ancestor $b$ of $t$, such that $b$ has at least one carved descendant and the subtrie rooted at $b$ needs fewer SRAM bits than the width of a SRAM word to represent the classifier rules, then $b$ is carved. As a result, the new rule is stored with some existing rules in a new SRAM word. Note that the existing rules, have additional suffix bits in the newly created SRAM word and old LTCAM1 entries for the existing rules are deleted. If no such $b$ exists, a new LTCAM1 entry is created by carving at $t$. The corresponding LSRAM1 word contains only the newly added rule.

When a rule in an LTCAM1 is deleted, then the rule is first removed from the LSRAM1 word. If the LSRAM1 word becomes empty, then the corresponding LTCAM1 word is deleted. Otherwise, if the contents of the LSRAM1 word can be merged with another LSRAM1 word then a new LTCAM1 entry is created while the LTCAM1 entries for the merged words are deleted.

The algorithms to merge and split buckets on the LTCAMs are similarly based on manipulating the carving in Trie1 and Trie2.

7.2.3.4 Updating the TCAMs

To insert or move a rule in a TCAM we need a free slot at an appropriate location in the TCAM. This slot can be obtained efficiently using memory management algorithms developed for TCAMs. In particular, the memory management schemes from DUO (see Chapter 3) may be used. For the ITCAM of PC-TRIO, we implemented the DLFS,PLO scheme, as its the most efficient scheme known to us for moving free slots to a desired
location in a TCAM. In the DLFS_PLO initial rule placement scheme, free slots are kept in the region between two blocks. Additionally, there may be free slots within a block. So a list of free slots is maintained for each block on the TCAM, with the list being empty initially. As rules are deleted from a block, the freed slots are added to the list for that block. Thus, DLFS_PLO requires no moves for most of the time to get or return a free slot.

The memory management scheme for the LTCAM of DUO is relatively simple as all the rules in the LTCAM are independent so a new rule may be inserted anywhere in the TCAM. However, we still need to locate a free slot. The LTCAM memory management algorithm of DUO creates a linked list of the free slots. When a free slot is needed, a slot is obtained from the head of the free slot list. PC-TRIO uses the memory management algorithm in DUO for its LTCAM1 and LTCAM2.

7.3 Differences among PC-DUOS, PC-DUOS+, PC-DUOS+W and PC-TRIO

PC-DUOS, PC-DUOS+ and PC-TRIO differ from each other in the way the selection of rules for the LTCAM is made. PC-DUOS filters the leaves of leaves set in a multi-dimensional trie to keep only the highest priority rules among all overlapping rules. The rules in the filtered leaves of leaves set is then entered in the LTCAM. PC-DUOS+, on the other hand, uses a priority graph to select rules for the LTCAM. PC-TRIO selects the leaf rules for an LTCAM using a two-step process. The first step is to identify the leaves of leaves set in a three dimensional trie and the second step is to identify independent rules from the rules stored on the nodes of the leaves of leaves set. PC-DUOS+ and PC-TRIO also use enhanced algorithms for ITCAM rule insertion which require fewer moves to rearrange rules for priority based adjustments.

We note that the methodology used in this chapter for PC-TRIO may be used to add index TCAMs and wide SRAMs to PC-DUOS+ to arrive at a new architecture PC-DUOS+W. Although PC-DUOS may be similarly extended to obtain PC-DUOSW, we do not consider this extension here as PC-DUOS+ is superior to PC-DUOS as was
 Unlike the other architectures, PC-TRIO does not guarantee that the rules in the LTCAMs are of the highest priority among all overlapping rules. Thus, PC-TRIO must wait for an ITCAM lookup to complete even if there are matching rules in the LTCAMs. Although the rule assignment algorithms for PC-TRIO may be modified so that the LTCAM rules are the highest priority among all overlapping rules (and thus avoid having to wait for an ITCAM lookup to complete in cases when a match is found in an LTCAM), doing so retards the performance of PC-TRIO to the point where it offers little or no power and lookup time benefit over PC-DUOS+W.

## 7.4 Experimental Results

In this section, we compare PC-TRIO, with PC-DUOS+W and PC-DUOS+ [29]. We first give the setup used by us for the experiments in Section 7.4.1 and then describe our datasets in Section 7.4.2. Finally we present our results in Section 7.4.3.

### 7.4.1 Setup

We programmed the rule assignment, trie carving and update processing algorithms of PC-TRIO using C++. We designed a circuit for processing wide SRAM

---

### Table: Differences among the architectures

<table>
<thead>
<tr>
<th></th>
<th>PC-DUOS</th>
<th>PC-DUOS+</th>
<th>PC-TRIO</th>
<th>PC-DUOS+W</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Uses single LTCAM</td>
<td>Uses single LTCAM</td>
<td>Uses two LTCAMs</td>
<td>Uses two LTCAMs</td>
</tr>
<tr>
<td>2.</td>
<td>No wide SRAMs or index TCAMs</td>
<td>No wide SRAMs or index TCAMs</td>
<td>Uses wide SRAMs and index TCAMs</td>
<td>Uses wide SRAM and index TCAM</td>
</tr>
<tr>
<td>3.</td>
<td>LTCAM stores highest priority independent rules</td>
<td>LTCAM stores highest priority independent rules</td>
<td>LTCAMs store independent rules</td>
<td>LTCAM stores highest priority independent rules</td>
</tr>
<tr>
<td>4.</td>
<td>Aborts ITCAM search when LTCAM search succeeds</td>
<td>Aborts ITCAM search when LTCAM search succeeds</td>
<td>Waits for ITCAM search to finish</td>
<td>Aborts ITCAM search when LTCAM search succeeds</td>
</tr>
<tr>
<td>5.</td>
<td>Independent rules are filtered leaves of leaves set in trie</td>
<td>Independent rules are vertices in priority graph with indegree=0</td>
<td>Independent rules are leaves of leaves set in trie</td>
<td>Independent rules are vertices in priority graph with indegree=0</td>
</tr>
</tbody>
</table>

---

Figure 7-8. Differences among the architectures

found from our experiments, the results for which are presented in Chapter 6. Figure 7-8 highlights the differences among PC-DUOS, PC-DUOS+, PC-DUOS+W and PC-TRIO.
words using Verilog and synthesized it using Synopsys Design Compiler to obtain power, area and gate count estimates. We used CACTI [30] and a TCAM power and timing model [1] to estimate the power consumption and search time for the SRAMs and the TCAMs respectively. The process technology used in the experiments is 70nm and the voltage is 1.12V. It is assumed that the TCAMs are being operated at 360MHz [36].

The TCAM and SRAM word sizes used are consistent for all the architectures used in the comparison. The word size is 144 bits for the TCAMs. For SRAMs we have different word sizes depending upon the TCAMs they are used with. The ISRAM words of all the architectures, as well as the LSRAM words of PC-DUOS+, are 32 bits wide. The LSRAM1 and LSRAM2 words of PC-TRIO and the LSRAM words of PC-DUOS+W are 512 bits, while the ILSRAMs are 144 bits wide. The bucket size for LTCAMs in PC-TRIO and PC-DUOS+W is set to 65 TCAM entries. PC-DUOS+ uses DIRPE [18] to encode port ranges. The classifier rules stored in the ITCAMs of PC-TRIO and PC-DUOS+W also use DIRPE to encode port ranges. Since the TCAM word size is set to 144 bits, we assume that 36 bits are available for encoding each port range in a rule. With this assumption, we use the strides 223333 as these give us minimum expansion of the rules [18, 28].

7.4.2 Datasets

We used two sets of benchmarks derived from ClassBench [50]. The first set of benchmarks consists of 12 datasets each containing about 100,000 classifier rules and is generated from seed files in ClassBench. This dataset is used to compare the number of TCAM entries, power, lookup performance and space requirements of PC-TRIO, PC-DUOS+W and PC-DUOS+ [29].

The second set of benchmarks was reused from [29]. There are 13 datasets here which are used to compare incremental update performance of PC-TRIO, with PC-DUOS+ [29] and PC-DUOS+W.
<table>
<thead>
<tr>
<th>Index</th>
<th>Dataset</th>
<th>#Rules</th>
<th>Entries TCAM</th>
<th>ITCAM Entries</th>
<th>TCAM Watts</th>
<th>lookup Time(ns)</th>
<th>Entries TCAM</th>
<th>ITCAM Entries</th>
<th>TCAM Watts</th>
<th>lookup Time(ns)</th>
<th>Entries TCAM</th>
<th>ITCAM Entries</th>
<th>TCAM Watts</th>
<th>lookup Time(ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>acl1</td>
<td></td>
<td>99309</td>
<td>117033</td>
<td>379</td>
<td>2624.39</td>
<td>21146</td>
<td>379</td>
<td>0.23</td>
<td>0.50</td>
<td>21085</td>
<td>182</td>
<td>0.19</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>acl2</td>
<td></td>
<td>74298</td>
<td>101857</td>
<td>19421</td>
<td>6.35</td>
<td>37491</td>
<td>19421</td>
<td>30.36</td>
<td>56.41</td>
<td>36593</td>
<td>135620</td>
<td>6.04</td>
<td>149.43</td>
</tr>
<tr>
<td>3</td>
<td>acl3</td>
<td></td>
<td>99468</td>
<td>131243</td>
<td>30859</td>
<td>40</td>
<td>1640.47</td>
<td>52632</td>
<td>9.47</td>
<td>26.96</td>
<td>26823</td>
<td>1017</td>
<td>0.40</td>
<td>2.19</td>
</tr>
<tr>
<td>4</td>
<td>acl4</td>
<td></td>
<td>99334</td>
<td>127320</td>
<td>25189</td>
<td>40</td>
<td>1640.47</td>
<td>52632</td>
<td>9.47</td>
<td>26.96</td>
<td>26823</td>
<td>1017</td>
<td>0.40</td>
<td>2.19</td>
</tr>
<tr>
<td>5</td>
<td>acl5</td>
<td></td>
<td>98117</td>
<td>105375</td>
<td>1535</td>
<td>32</td>
<td>2072.16</td>
<td>32932</td>
<td>0.53</td>
<td>0.41</td>
<td>34993</td>
<td>2209</td>
<td>0.77</td>
<td>4.98</td>
</tr>
<tr>
<td>6</td>
<td>fw1</td>
<td></td>
<td>89356</td>
<td>142085</td>
<td>91473</td>
<td>43</td>
<td>2466.72</td>
<td>98425</td>
<td>27.92</td>
<td>2318.82</td>
<td>26610</td>
<td>4864</td>
<td>1.60</td>
<td>15.01</td>
</tr>
<tr>
<td>7</td>
<td>fw2</td>
<td></td>
<td>96055</td>
<td>129249</td>
<td>27084</td>
<td>39</td>
<td>1543.76</td>
<td>43146</td>
<td>8.30</td>
<td>22196</td>
<td>1494</td>
<td>0.53</td>
<td>3.18</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>fw3</td>
<td></td>
<td>80885</td>
<td>117731</td>
<td>39199</td>
<td>36</td>
<td>1007.04</td>
<td>51228</td>
<td>11.99</td>
<td>215.21</td>
<td>26269</td>
<td>7479</td>
<td>2.38</td>
<td>30.09</td>
</tr>
<tr>
<td>9</td>
<td>fw4</td>
<td></td>
<td>84056</td>
<td>211403</td>
<td>116149</td>
<td>64</td>
<td>3182.03</td>
<td>131505</td>
<td>35.46</td>
<td>2139.21</td>
<td>27617</td>
<td>4894</td>
<td>1.60</td>
<td>15.16</td>
</tr>
<tr>
<td>10</td>
<td>fw5</td>
<td></td>
<td>84013</td>
<td>111989</td>
<td>55650</td>
<td>34</td>
<td>930.94</td>
<td>65598</td>
<td>17.00</td>
<td>615.49</td>
<td>22361</td>
<td>3454</td>
<td>1.15</td>
<td>9.02</td>
</tr>
<tr>
<td>11</td>
<td>ipc1</td>
<td></td>
<td>99198</td>
<td>112154</td>
<td>22165</td>
<td>34</td>
<td>1288.02</td>
<td>41920</td>
<td>6.82</td>
<td>45.11</td>
<td>23894</td>
<td>567</td>
<td>0.26</td>
<td>1.40</td>
</tr>
<tr>
<td>12</td>
<td>ipc2</td>
<td></td>
<td>100000</td>
<td>100000</td>
<td>30133</td>
<td>30</td>
<td>784.69</td>
<td>47247</td>
<td>9.23</td>
<td>113.77</td>
<td>20195</td>
<td>0</td>
<td>0.09</td>
<td>0.75</td>
</tr>
</tbody>
</table>
7.4.3 Results

7.4.3.1 Number of TCAM entries

Using wide SRAM words to store portions of classifier rules, reduces the number of TCAM entries. Table 7-1 gives the results of storing our datasets from the second benchmark, in the three architectures. The first, second and third columns show the index, name, and the number of classifier rules, respectively, of a dataset. The fourth, fifth and sixth and seventh columns give for PC-DUOS+, the total number of TCAM entries, the number of ITCAM entries, the TCAM power and lookup time, respectively. Similarly, the eighth, ninth, tenth and eleventh columns give the corresponding numbers for PC-DUOS+W and the remaining four columns give those for PC-TRIO.

Figure 7-9(a) gives the TCAM compaction ratio of the three architectures, obtained by dividing the number of TCAM entries for each dataset by the number of rules in the classifier. PC-DUOS+ does not use wide SRAMs, hence there is no compaction, instead, there is expansion to handle port ranges. Thus, the compaction ratio for PC-DUOS+ is at least 1 for every dataset. The compaction achieved by PC-TRIO is more than that of PC-DUOS+W for almost all the datasets. This is because, PC-TRIO has fewer ITCAM entries and therefore stores more rules in wide SRAM words. For acl5, PC-DUOS+W identified more independent rules compared to PC-TRIO. The algorithm to identify independent rules is the same for PC-DUOS+W and PC-DUOS+ which results in identical ITCAM entries for these two architectures.

No classifier rules in the LTCAMs of PC-DUOS+W and PC-TRIO needed partial port range expansion (Section 7.2.2.2). So all LTCAM entries in PC-DUOS+W and PC-TRIO were at most 72 bits.

7.4.3.2 Power

Table 7-1 gives the TCAM power consumption during a lookup, while Figure 7-9(b) gives the normalized total power obtained for each dataset by dividing the total TCAM and SRAM power in an architecture by that of PC-TRIO during a lookup. The vertical
Figure 7-9. Comparison of compaction ratio, total power, lookup time and area axis is scaled logarithmically and based at 1. PC-TRIO uses less power for all datasets except acl5. The average improvement in power with PC-TRIO is 96% relative to PC-DUOS+, and 65% relative to PC-DUOS+W. The average improvement in power with PC-DUOS+W is 71%, relative to PC-DUOS+. The maximum improvement with PC-TRIO is observed for ipc2 (99%) and the minimum for acl2 (80%), compared to PC-DUOS+. The maximum improvement with PC-DUOS+W is observed for acl1 (99%) and the minimum for fw1 (35%), compared to PC-DUOS+. The maximum improvement with PC-TRIO is observed for ipc2 (98%) and the minimum for acl1 (2%), compared to PC-DUOS+W.

### 7.4.3.3 Lookup performance

Figure 7-9(c) gives the average lookup time, normalized with respect to that of PC-TRIO. TCAM search time is proportional to the number of TCAM entries. Hence, PC-DUOS+ requires the maximum time.
PC-DUOS+W is faster than PC-TRIO for the ACL tests acl1, acl2 and acl5. For these datasets, the number of ITCAM entries in PC-DUOS+W and PC-TRIO (columns 9 and 13 of Table 7-1) are comparable. Thus, the ITCAM search times are comparable, as are the number of lookups served by the ITCAMs. This, coupled with the fact that ITCAM searches are slower, give PC-DUOS+W an immediate advantage since it, unlike PC-TRIO, aborts an ITCAM search after finding a match in the LTCAM. However, for these three tests, the lookup times using PC-TRIO are quite reasonable (column 15 of Table 7-1). For the other datasets PC-TRIO has fewer rules in the ITCAM, which makes PC-TRIO lookups faster even though it has to wait for ITCAM search to finish.

The average improvement in lookup time with PC-TRIO and PC-DUOS+W (relative to PC-DUOS+) are 98% and 76%, respectively. The average improvement in lookup time with PC-TRIO (relative to PC-DUOS+W) is 68%. The maximum improvement using PC-TRIO rather than PC-DUOS+ is observed for acl1 (99.96%) and the minimum for acl2 (86.6%). The maximum improvement using PC-DUOS+W rather than PC-DUOS+ is observed for acl1 (99.98%) and the minimum for fw1 (5%). The maximum improvement with PC-TRIO rather than PC-DUOS+W is observed for tests fw1, fw4 and ipc2 (99%) and the minimum for acl4 (47%).

7.4.3.4 Space requirements

We obtained SRAM area from CACTI results and estimated TCAM area using the same technique as used in PETCAM [27], where area of a single cell is multiplied by the number of cells and then adjusted for wiring overhead. Figure 7-9(d) gives the total area needed for the TCAMs and associated SRAMs. The total area is comparable for the three architectures. PC-TRIO and PC-DUOS+W have lower TCAM area (due to fewer TCAM entries) and higher SRAM area (due to wider SRAM words) than PC-DUOS+.

7.4.3.5 Update performance

Figure 7-10 shows the average number of TCAM writes used per update on the datasets from the first benchmark. PC-TRIO needs comparable number of writes
Figure 7-10. TCAM writes

as PC-DUOS+ and hence supports efficient and consistent incremental updates.

PC-DUOS+W needs more writes than PC-TRIO to preserve the property that all rules
stored in the LTCAM have the highest priority compared to overlapping rules.

7.4.3.6 Characteristics of the logic that processes wide SRAM words

A circuit designed to process the contents of a wide LSRAM word was synthesized
using a 0.18 µm library [47, 48] and it was found that the design successfully met
the timing constraints with a 500MHz clock. The results are presented in the Table 7-2.

Table 7-2. Timing and power results for additional hardware

<table>
<thead>
<tr>
<th>Process Time (ns)</th>
<th>Throughput (Msps)</th>
<th>Voltage (V)</th>
<th>Power (mW)</th>
<th>Gate Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.18µm 2</td>
<td>500</td>
<td>1.8</td>
<td>61.13</td>
<td>59724</td>
</tr>
</tbody>
</table>

The throughput is represented in terms of million searches per second (Msps). An
example of a TCAM with a speed of 143MHz (effectively, 143 Msps) is found in [31],
using 0.13µm technology. It is expected that the delay overhead and throughput of our
design will improve on using a 0.13µm library. Thus, our design can operate at the same
speed as that of a TCAM.

We presented in this chapter an indexed TCAM architecture, PC-TRIO, for packet
classifiers. The methods to add indexing and wide SRAMs were applied on PC-DUOS+
to obtain another indexed TCAM architecture PC-DUOS+W. These two architectures
were then compared with PC-DUOS+. Both PC-TRIO and PC-DUOS+W may be
updated incrementally. The average improvement in TCAM power and lookup time using PC-TRIO were 96% and 98%, respectively, while that using PC-DUOS+W were 71% and 76%, respectively, relative to PC-DUOS+.

PC-DUOS+W performed better on the ACL datasets compared to the other types of classifiers. There was 86% reduction in TCAM power, and 98% reduction in lookup time with PC-DUOS+W on the ACL datasets on an average compared to PC-DUOS+. Even though PC-DUOS+W lookup performance was better than that of PC-TRIO on three ACL tests, PC-TRIO lookup performance was quite reasonable and in fact, using PC-TRIO, there was a reduction in TCAM power by 94% and lookup time by 97% on an average for the ACL tests, compared to PC-DUOS+.

So, we recommend PC-TRIO for packet classifiers.
The focus of this dissertation is enabling low power TCAMs for packet forwarding and classification, with efficient updating mechanism. The contributions are summarized as follows:

PETCAM, a power efficient TCAM architecture that can be used to store packet forwarding tables. It was observed that PETCAM significantly reduce both TCAM power and memory requirements, over existing TCAM schemes for storing forwarding tables. PETCAM is updated using batch update mechanism.

DUO, a dual TCAM architecture for forwarding tables. The advantages of DUO include low power lookups, in addition to, efficient incremental updates that can be applied without locking the TCAMs from lookups.

CONSIST, a consistent TCAM updating scheme useful when the updates to be applied incrementally, arrive in a cluster. The lookups happening while the updates are being applied will return appropriate next hop (for forwarding tables lookup) or action (for classifier lookup) when CONSIST is used. Further, the sequence of updates, built using our strategy, is free from redundancies in update operations and produces a near minimal increase in table size.

PC-DUOS, a dual TCAM architecture for packet classifiers that help to implement faster updates and lookup mechanisms.

PC-DUOS+, an enhancement over PC-DUOS, and leads to even faster updates to packet classifiers.

PC-TRIO, an indexed triple TCAM architecture for packet classifiers that lead to ultra low power lookups and is easy to update incrementally.
REFERENCES


Proceedings of the 40th International Symposium on Microarchitecture, Dec. 2007, pp. 3-14


[40] S. Sekiguchi, “Grid around Asia,” Grid Asia, 2009


BIOGRAPHICAL SKETCH

Tania Banerjee Mishra received her Ph.D. from Computer and Information Science and Engineering Department at the University of Florida in 2012, under the supervision of Prof. Sartaj Sahni. Her research interests are data structures and algorithms, networking algorithms and VLSI CAD.

Tania received Integrated M.Sc. in Mathematics and M.Tech in Computer Science from the Indian Institute of Technology, Kharagpur, in 1996 and 1998 respectively. She was with Interra Systems between 1998 and 2003 and with Sequence Design (now acquired by Apache Design Solutions) between 2003 and 2006 working on power estimation and optimization techniques for RTL designs.