AFRL-RI-RS-TR-2012-055

ENERGY-EFFICIENT HIGH-PERFORMANCE ROUTERS

UNIVERSITY OF FLORIDA

FEBRUARY 2012

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the 88th ABW, Wright-Patterson AFB Public Affairs Office and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2012-055 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/s/
WILLIAM E. STANTON, 1st Lt, USAF
Work Unit Manager

/s/
PAUL ANTONIK, Technical Advisor
Computing & Communications Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
FEB 2012	Final Technical Report	AUG 2010 – AUG 2011

4. TITLE AND SUBTITLE

ENERGY-EFFICIENT HIGH-PERFORMANCE ROUTERS

5a. CONTRACT NUMBER
FA8750-10-1-0236

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
63788F

6. AUTHOR(S)

Sartaj Sahni

5d. PROJECT NUMBER
T3TE

5e. TASK NUMBER
HP

5f. WORK UNIT NUMBER
IR

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of Florida
339 Weil Hall
P.O. Box 116550
Gainsville, FL 32611-6550

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/RITD
525 Brooks Road
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RI

11. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-RI-RS-TR-2012-055

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. PA# 88ABW-2012-0311
Date Cleared: 19 JAN 2012

13. SUPPLEMENTARY NOTES

14. ABSTRACT
Under this award, the PI Dr. Sartaj Sahni, AFRL research scientist Dr. Gunasekaran Seetharaman, and University of Florida Ph.D. student Ms. Tania Banerjee-Mishra collaboratively researched TCAM (Ternary Content Addressable Memory) architectures for Internet packet classifiers. The objective was to develop low-energy high-performance TCAM architectures that supported both lookup and update. To this end, the architectures PC-DUOS, PC-DUOS+, and PC-TRIO were developed and evaluated. The first two of these use 2 TCAMs while the third uses 3 TCAMs. Significant improvements in performance and power consumption are achieved.

15. SUBJECT TERMS

TCAM (Ternary Content Addressable Memory), Internet packet classifiers

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			WILLIAM E. STANTON, 1st Lt, USAF
U	U	U	UU	66	19b. TELEPHONE NUMBER (*Include area code*) N/A

Table of Contents

List of Figures

1 INTRODUCTION

Packet classification is a key step in routers for various functions such as routing, creating firewalls, load balancing and differentiated services. Internet packets are classified into different flows based on packet header fields and 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. 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], …, 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], …, 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.

TCAMs are used widely for packet classification. The popularity of TCAMs is mainly due to their high-speed table lookup mechanism in which all the TCAM entries are searched in parallel. Each bit of a TCAM may be set to one of the three states 0, 1, and '?' (don't care). A TCAM is used in conjunction with an SRAM. Given a rule (F, A) , the filter F of a packet classifier rule is stored in a TCAM word and action A is stored in an associated SRAM word. All TCAM entries are searched in parallel and the first match is used to access the corresponding SRAM word to retrieve the action. So, when the packet classifier rules are stored in a TCAM in decreasing order of priority (increasing order of cost), we can determine the action corresponding to the matching rule of the highest priority, in one TCAM cycle. The main limitation of TCAMs is that these memories are power hungry. The more the number of entries in the TCAM, the higher the power needed to perform a search. This problem is worsened for packet classifiers since typically a classifier rule includes port range fields that need multiple TCAM entries per rule for representation in the TCAM. This is called range expansion. Given that the source and destination port numbers are represented in 16 bits, the number of TCAM entries needed to represent a port range in the worst case is 30 corresponding to the range $[1, 2^{16}-2]$. Thus, a filter having both source and destination port ranges set to $[1, 2^{16}-2]$ undergoes a worst case expansion of 30 x 30 = 900 TCAM entries.

2 METHODS, ASSUMPTIONS, AND PROCEDURES

2.1 Background and Related Work

The starting point for our research on TCAM-based packet classifiers is the packet forwarding architecture DUOS proposed earlier by us [7]. DUOS, as shown in Figure 1, has two TCAMs,

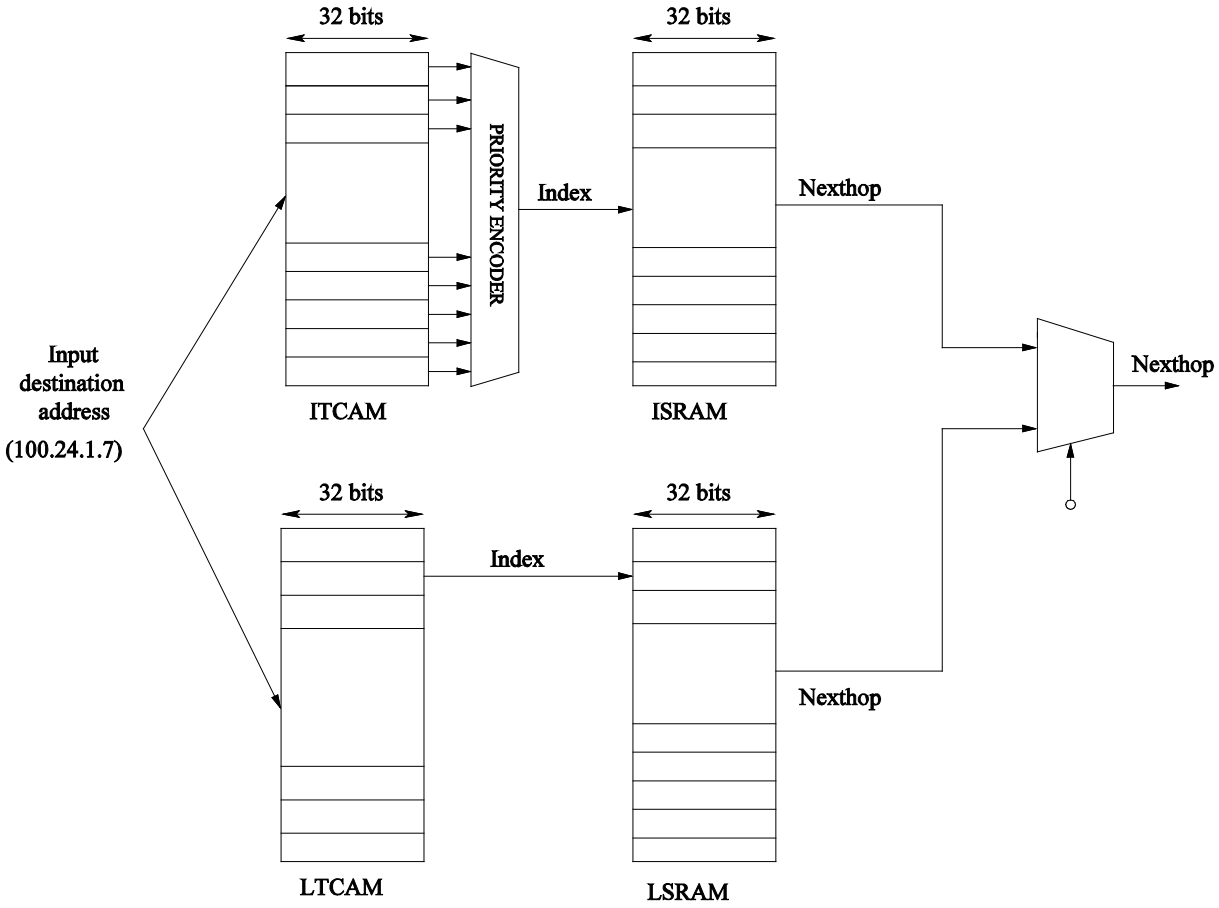


Figure 1 Dual TCAM with simple SRAM

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 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 (interior TCAM), 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% [2]. The final stage logic in Figure 1 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.

The memory management schemes used in DUOS are highly efficient. The ITCAM needs to store the prefixes in decreasing order of length, for example, so that the first matching prefix is also the longest matching prefix. DUOS [7] uses a memory management scheme (Scheme 3, also known as DLFS_PLO), which initially distributes the free space available in a TCAM between blocks of prefixes (of same length) in proportion to the number of prefixes in a block. A free slot needed to add a new

prefix is moved from a location that requires the minimum number of moves. As a prefix is deleted, the freed slot is added to a list of free spaces for that prefix block. Each prefix block has its own list of free slots. With this scheme even with 99% prefix occupancy in the TCAM and 1% free space, the total number of prefix moves using DLFS_PLO is at most 0.7% of the total number of prefix inserts and deletes.

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 [6] and [5]. The authors in [6] 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 [6] is that it performs the deletes in the update batch first. This could lead to temporary inconsistencies in lookup [8].

Given a packet classifier, a naive approach is to store it in a TCAM by entering each rule sequentially as it appears in the classifier and distribute all the empty slots between rules. As mentioned in [5], this approach could lead to high power consumption for a 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 comparing our PC-DUOS performance. 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 [5], 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.

The authors in [5] 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 (2n) 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.

2.2 PC-DUOS

PC-DUOS uses the same two TCAM architecture as used in DUOS [7] (Figure 1). Lookup also works in the same way as for DUOS. That is, the LTCAM and ITCAM (interior TCAM) 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%, because the LTCAM has no priority encoder. For this lookup strategy to yield correct results, the following requirements must hold:

1. No packet is matched by more than one rule in the LTCAM.
2. When a packet is matched by a rule in the LTCAM, the matched rule must be the highest priority matching rule.

Figure 2 shows the overall flow of our methodology of storing rules in the ITCAM and LTCAM. The first phase involves storing all the rules in a multi-dimensional trie maintained on the control plane of the classifier. The second phase in our methodology consists of traversing the multi-dimensional trie and identifying independent rules for inclusion in the LTCAM. In the third phase, rules not stored in the LTCAM are stored in the ITCAM in priority order. Further details including lookup and update algorithms are given in [1].

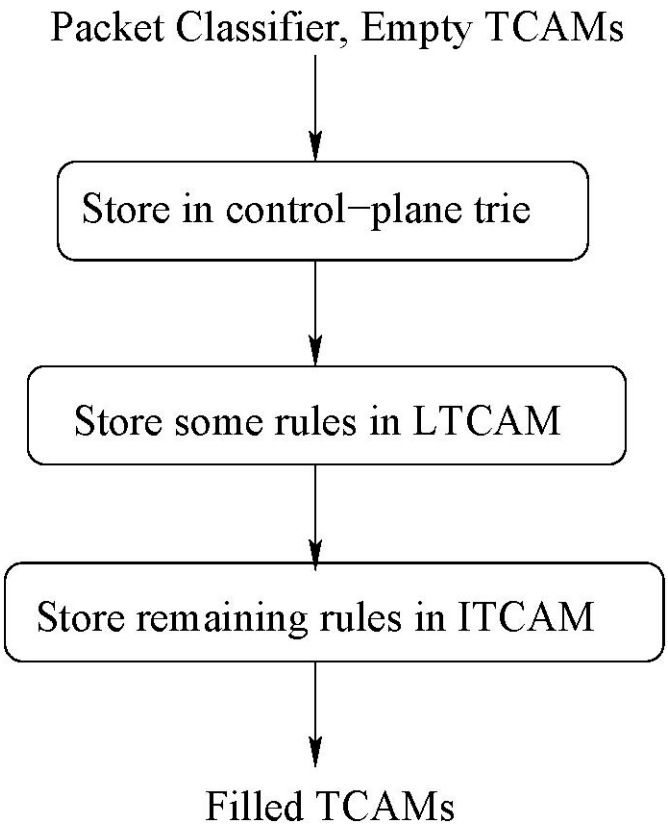


Figure 2 Flow diagram for storing packet classifiers in TCAMs

2.3 PC-DUOS+

PC-DUOS+ uses the two TCAM architecture used in PC-DUOS and DUOS (Figure 1). During lookup, the LTCAM and ITCAM are searched in parallel using the packet header information. If a match is found in the LTCAM, the ongoing search in the ITCAM is aborted.

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. PC-DUOS+ also uses enhanced algorithms for ITCAM rule insertion which require fewer moves to rearrange rules for priority based adjustments. Further details including lookup and update algorithms are given in [2].

2.4 PC-TRIO

Figure 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.

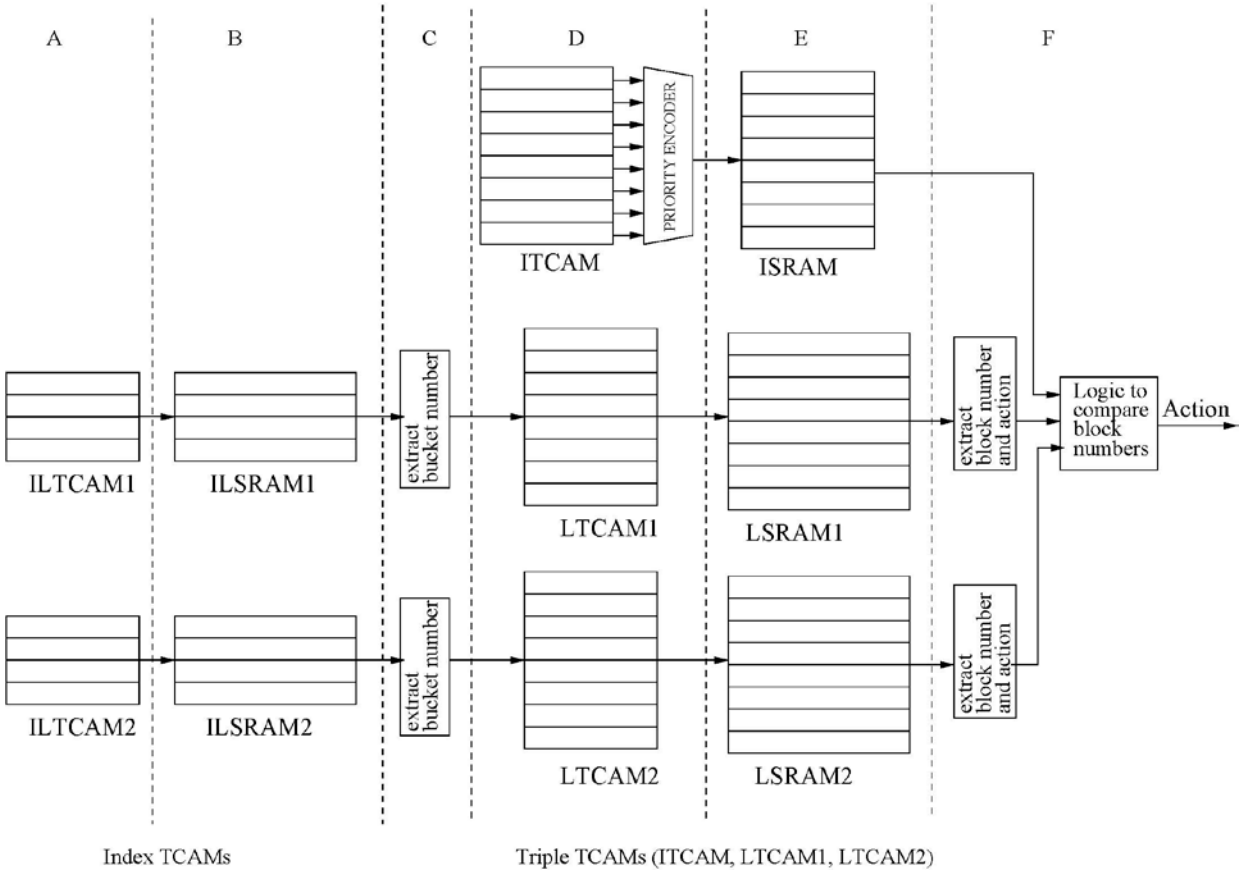


Figure 3 PC-TRIO architecture

A lookup in PC-TRIO is pipelined with 6 stages marked A-F in Figure 3. In the first stage A, the ILTCAMs (index TCAM for an LTCAM) are searched. The ILSRAMs (index SRAM for an ILTCAM) are accessed, using the address of the matching ILTCAM1 andILTCAM2 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. Further details including lookup and update algorithms are given in [3].

2.5 Differences among PC-DUOS, PC-DUOS+ and PC-TRIO

Figure 4 highlights the differences among PC-DUOS, PC-DUOS+ and PC-TRIO.

	PC−DUOS	PC−DUOS+	PC−TRIO
1.	Uses single LTCAM	Uses single LTCAM	Uses two LTCAMs
2.	No wide SRAMs or index TCAMs	No wide SRAMs or index TCAMs	Uses wide SRAMs and index TCAMs
3.	LTCAM stores highest priority independent rules	LTCAM stores highest priority independent rules	LTCAMs store independent rules
4.	Aborts ITCAM search when LTCAM search succeeds	Aborts ITCAM search when LTCAM search succeeds	Waits for ITCAM search to finish
5.	Independent rules are filtered leaves of leaves set in trie	Independent rules are vertices in priority graph with indegree=0	Independent rules are leaves of leaves set in trie

Figure 4 Classifier rules stored in an indexed TCAM

We note that the methodology used 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. Similarly, PC-DUOS may be extended to obtain PC-DUOS+W.

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.

3 RESULTS AND DISCUSSION

We compare PC-TRIO, with PC-DUOS+W and PC-DUOS+. (A detailed comparison of PC-DUOS and PC-DUOS+ appears in [2] where the superiority of PC-DUOS+ over PC-DUOS is established.) We first give the setup used by us for the experiments in Section 3.1 and then describe our datasets in Section 3.2. Finally we present our results in Section 3.3.

3.1 Setup

We programmed the rule assignment, trie carving and update processing algorithms of our packet classification architectures using C++. We designed a circuit for processing wide SRAM words using

Verilog and synthesized it using Synopsys Design Compiler to obtain power, area and gate count estimates. We used CACTI [13] and a TCAM power and timing model [14] to estimate the power consumption and search time for the SRAMs and the TCAMs respectively. The process technology used in the experiments is 70 nm and the voltage is 1.12 V. It is assumed that the TCAMs are being operated at 360 MHz [14].

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 [9] 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 [9].

3.2 Datasets

We used two sets of benchmarks derived from ClassBench [10]. 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+.

The second set of benchmarks has 13 datasets, which are used to compare incremental update performance of PC-TRIO, with PC-DUOS+ and PC-DUOS+W.

3.3 Results

Number of TCAM entries

Using wide SRAM words to store portions of classifier rules reduces the number of TCAM entries. Figures 5-7 give the results of storing our datasets 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, Figure 6 gives the corresponding numbers for PC-DUOS+W and Figure 7 gives those statistics for PC-TRIO.

Index	Dataset	#Rules	Entries	#ITCAM	Watts	Time(ns)
1	acl1	99309	117033	379	36	2624.39
2	acl2	74298	101857	19421	31	1122.39
3	acl3	99468	131243	30859	40	1640.47
4	acl4	99334	127320	25189	39	1730.46
5	acl5	98117	105375	1535	32	2072.16
6	fw1	89356	142085	91473	43	2466.72
7	fw2	96055	129249	27084	39	1543.76
8	fw3	80885	117731	39199	36	1007.04
9	fw4	84056	211403	116149	64	3182.03
10	fw5	84013	111989	55650	34	930.94
11	ipc1	99198	112154	22165	34	1288.02
12	ipc2	100000	100000	30133	30	784.69

Figure 5 Statistics for PC-DUOS+

Index	Dataset	#Rules	Entries	#ITCAM	Watts	Time(ns)
1	acl1	99309	21146	379	0.23	0.50
2	acl2	74298	37491	19421	6.35	30.36
3	acl3	99468	52632	30859	9.47	80.49
4	acl4	99334	49912	25189	7.98	45.95
5	acl5	98117	32932	1535	0.53	0.41
6	fw1	89356	98425	91473	27.92	2318.82
7	fw2	96055	43146	27084	8.30	86.77
8	fw3	80885	51228	39199	11.99	215.21
9	fw4	84056	131505	116149	35.46	2139.21
10	fw5	84013	65598	55650	17.00	615.49
11	ipc1	99198	41920	22165	6.82	45.11
12	ipc2	100000	47247	30133	9.23	113.77

Figure 6 Statistics for PC-DUOS+W

Index	Dataset	#Rules	Entries	#ITCAM	Watts	Time(ns)
1	acl1	99309	21085	182	0.19	1.00
2	acl2	74298	36593	18439	6.04	149.43
3	acl3	99468	26823	1017	0.40	2.19
4	acl4	99334	34034	6547	2.32	24.12
5	acl5	98117	34993	2209	0.77	4.98
6	fw1	89356	26610	4864	1.60	15.01
7	fw2	96055	22196	1494	0.53	3.18
8	fw3	80885	26269	7479	2.38	30.09
9	fw4	84056	27617	4894	1.60	15.16
10	fw5	84013	22361	3454	1.15	9.02
11	ipc1	99198	23894	567	0.26	1.40
12	ipc2	100000	20195	0	0.09	0.75

Figure 7 Statistics for PC-TRIO

Figure 8(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. So, all LTCAM entries in PC-DUOS+W and PC-TRIO were at most 72 bits.

Power

Figures 5-7 give the TCAM power consumption during a lookup, while Figure 8 (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 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.

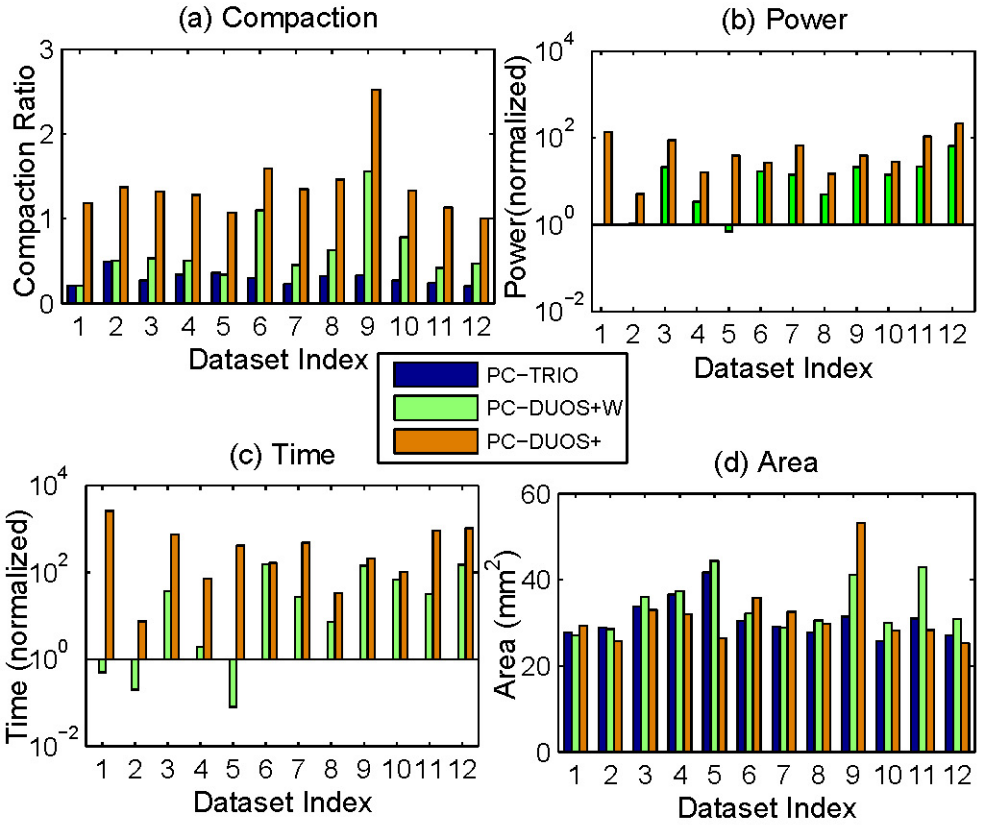


Figure 8 Comparison of compaction ratio, total power, lookup time and area

Lookup Performance
Figure 8 (c) gives the average lookup time, normalized with respect to that of PC-TRIO. The average lookup time was computed from TCAM search times obtained using the timing models of [11]. 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 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. 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%). For PC-TRIO, the average look up time was the maximum time a TCAM took. For PC-DUOS+ and PC-DUOS+W, the average lookup time is an weighted average of the ITCAM and LTCAM search times where the weight for the ITCAM is the number of classifier rules in the ITCAM divided by the total number of rules and the weight for the LTCAM is similarly determined. Recall that during lookup in PC-DUOS+ (and also for PC-DUOS+W) ITCAM lookup is aborted upon finding an LTCAM match. It was found that the ratio of the number of hits in the ITCAM to that in the LTCAM depends on the ratio of the number of entries in these TCAM. Thus, the weighted average closely models the actual time.

Space requirements
We obtained SRAM area from CACTI results and estimated TCAM area using the same technique as used in PETCAM [12], where area of a single cell is multiplied by the number of cells and then adjusted for wiring overhead. Figure 8(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+.

Update Performance
Figure 9 shows the average number of TCAM writes used per update.

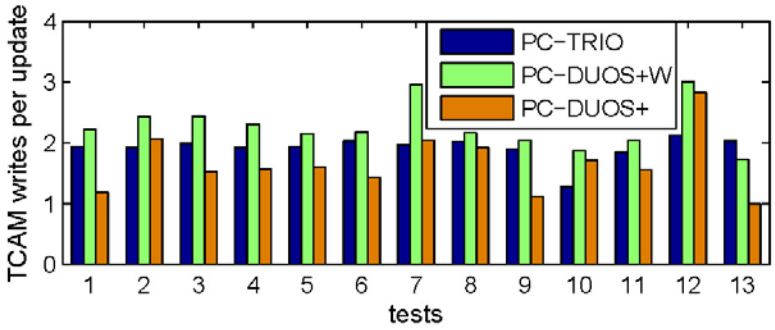


Figure 9 TCAM writes

PC-TRIO needs comparable number of 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.

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 [15, 16] and it was found that the design successfully met the timing constraints with a 500 MHz clock. The results are presented in the Figure 10. The throughput is represented in terms of million searches per second (Msps). An example of a TCAM with a speed of 143 MHz (effectively, 143 Msps) is found in [17], using 0.13 μm technology. It is expected that the delay overhead and throughput of our design will improve on using a 0.13 μm library. Thus, our design can operate at the same speed as that of a TCAM.

Process	Time (ns)	Throughput (Msps)	Voltage (V)	Power (mW)	Gate Count
0.18 μm	2	500	1.8	61.13	59724

Figure 10 Timing and power results for additional hardware

4 CONCLUSIONS

Under this award, the PI Dr. Sartaj Sahni, AFRL research scientist Dr. Gunasekaran Seetharaman, and University of Florida Ph.D. student Ms. Tania Banerjee-Mishra collaboratively researched TCAM (Ternary Content Addressable Memory) architectures for Internet packet classifiers. The objective was to develop low-energy high-performance TCAM architectures that supported both lookup and update. To this end, the architectures PC-DUOS, PC-DUOS+, and PC-TRIO were developed and evaluated. The first two of these use 2 TCAMs while the third uses 3 TCAMs. Three technical papers [1, 2, 3], one for each of the three developed architectures, were written. The paper on PC-DUOS was published in the 2011 IEEE International Symposium on Computers and Communications and the other two are in the review process (one at a journal and the other at a conference); all three can be referenced in Appendices A, B, and C respectively. PC-DUOS+, which is an enhancement of PC-DUOS and an extension of DUOS uses 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 (simple TCAM). The average improvement in lookup time is found to be between 19% and 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.

PC-TRIO and PC-DUOS+W (which as an extension of PC-DUOS+ to indexed TCAMs and wide SRAMs) may be updated incrementally. The average improvements 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+.

Therefore, we recommend PC-TRIO for packet classifiers.

5 REFERENCES

1. T. Mishra, S.Sahni, and G. Seetharaman, PC-DUOS: Fast TCAM Lookup and Update for Packet Classifiers, ISCC, 2011.

2. T. Mishra, S. Sahni and G. Seetharaman, PC-DUOS+: A TCAM Architecture for Packet Classifiers, http://www.cise.ufl.edu/~sahni/papers/pcduosplus.pdf, 2010.

3. T. Mishra, S. Sahni and G. Seetharaman, PC-TRIO: An Indexed TCAM Architecture for Packet Classifiers, http://www.cise.ufl.edu/~sahni/papers/trio.pdf, 2011.

4. M. Akhbarizadeh and M. Nourani, Efficient Prefix Cache For Network Processors, IEEE Symp. on High Performance Interconnects, 41-46, 2004.

5. H. Song and J. Turner, Fast Filter Updates for Packet Classification using TCAM, Routing Table Compaction in Ternary-CAM, GLOBECOM, 2006

6. Z. Wang, H. Che, M. Kumar, and S.K. Das, CoPTUA: Consistent Policy Table Update Algorithm for TCAM without Locking, IEEE Transactions on Computers, 53, 12, December 2004, 1602-1614.

7. T. Mishra and S.Sahni, DUOS -- Simple Dual TCAM architecture for routing tables with incremental update, IEEE Symposium on Computers and Communications, 2010.

8. T. Mishra and S. Sahni, CONSIST - Consistent Internet Route Updates IEEE Symposium on Computers and Communications, 2010.

9. K. Lakshminarayan, A. Rangarajan and S. Venkatachary, Algorithms for Advanced Packet Classification with Ternary CAMs, SIGCOMM, 2005.

10. D. E. Taylor and J. S. Turner, ClassBench: A Packet Classification Benchmark, TON, 15, 3, Jun 2007, 499-511.

11. B. Agrawal and T. Sherwood, Ternary CAM Power and Delay Model: Extensions and Uses, TVLSI, 16, 5, May 2008, 554-564.

12. T. Mishra and S.Sahni, PETCAM -- A Power Efficient TCAM for Forwarding Tables, http://www.cise.ufl.edu/~sahni/papers/petcam.pdf, 2010.

13. N. Muralimanohar, R. Balasubramonian and N. P. Jouppi, Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0, ISM December 2007, 3-14

14. Renesas R8A20410BG 20Mb Quad Search Full Ternary CAM. http://am.renesas.com/products/memory/TCAM/tcam_root.jsp. Jan 2010.

15. J. B. Sulistyo, J. Perry and D. S. Ha, Developing Standard Cells for TSMC 0.25um Technology under MOSIS DEEP Rules, Virginia Tech, Technical Report VISC-2003-01 Nov 2003.

16. J. B. Sulistyo and D. S. Ha, A New Characterization Method for Delay and Power Dissipation of Standard Library Cells, VLSI Design 15, 3, Jan 2002, 667-678.

17. H. Noda, K. Inoue, M. Kuroiwa, F. Igaue and K. Yamamoto, A Cost-Efficient High-Performance Dynamic TCAM With Pipelined Hierarchical Searching and Shift Redundancy Architecture, IJSSC, 40, 1, Jan 2005, 245-253.

APPENDIX A:

PC-DUOS+: A TCAM Architecture for Packet Classifiers

Tania Banerjee-Mishra and Sartaj Sahni
Department of Computer and Information Science and Engineering,
University of Florida, Gainesville, FL 32611
{tmishra, sahni}@cise.ufl.edu

Gunasekaran Seetharaman, AFRL, Rome, NY
Gunasekaran.Seetharaman@rl.af.mil

***Abstract*—We propose algorithms for distributing the classifier rules to two TCAMs (ternary content addressable memories) and for incrementally updating the TCAMs. The performance of our scheme is compared against the prevalent scheme of storing classifier rules in a single TCAM in priority order. Our scheme results in an improvement in average lookup speed by up to 49% and an improvement in update performance by up to 3.72 times in terms of the number of TCAM writes.**

***Index Terms*—Packet classifiers, TCAM, updates.**

I. Introduction

Internet packets are classified into different flows based on the packet header fields. This classification of packets is done 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. When an incoming packet matches a filter in the classifier, the corresponding 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 $(F[1], F[2], \cdots, F[d])$, 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 $F[1], \cdots, F[d]$. 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 corresponding to 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 costs.

[4], [5] survey the many solutions that have been proposed for packet classifiers. Among these, TCAMs have widely been used for packet classification as they support high speed lookups and are simple to use. Each bit of a TCAM may be set to one of the three states 0, 1, and x (don't care). A TCAM is used in conjunction with an SRAM. Given a rule (F, A), the filter F of a packet classifier rule is stored in a TCAM word whereas and action A is stored in an associated SRAM word. All TCAM entries are searched in parallel and the first match is used to access the corresponding SRAM word to retrieve the action. So, when the packet classifier rules are stored in a TCAM in decreasing order of priority (increasing order of cost), we can determine the action in one TCAM cycle.

We present a TCAM architecture, update algorithms and a TCAM lookup mechanism in this paper for packet classifiers. We begin in Section II by reviewing the background and related work. In Section III we describe our scheme of storing packet classifiers in TCAMs. An experimental evaluation of our scheme is done in Section IV and we conclude in Section V.

II. Background and Related Work

PC-DUOS+ is an extension of PC-DUOS (Packet Classifier - DUOS) proposed by us in [14]. PC-DUOS+ and PC-DUOS use an architecture as shown in Figure 1, which was first proposed for DUOS [9] for packet forwarding. There are 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% [1]. The logic on the final stage in Figure 1 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.

The memory management schemes used in DUOS are highly efficient. The ITCAM needs to store the prefixes in

This material is based upon work funded by AFRL, under AFRL Contract No. FA8750-10-1-0236.

decreasing order of length, for example, so that the first matching prefix is also the longest matching prefix. DUOS [9] uses a memory management scheme (Scheme 3, also known as DLFS_PLO), which initially distributes the free

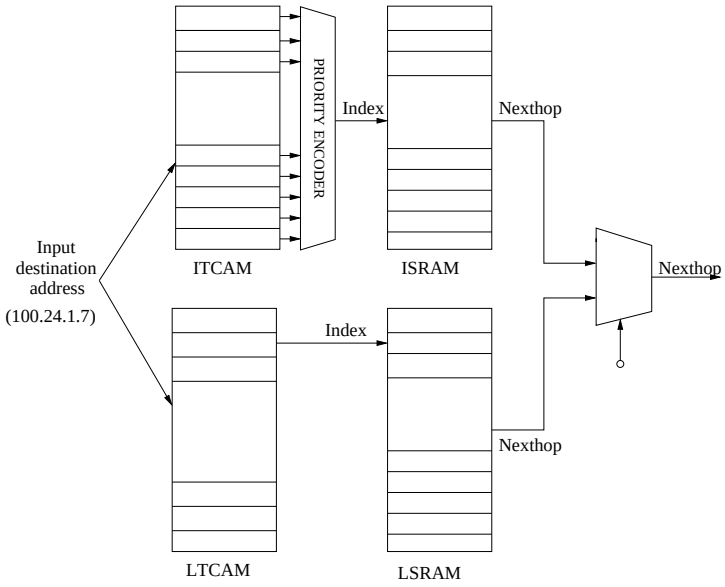


Fig. 1. Dual TCAM Architecture

space available in a TCAM between blocks of prefixes (of same length) in proportion to the number of prefixes in a block. A free slot needed to add a new prefix is moved from a location that requires the minimum number of moves. As a prefix is deleted, the freed slot is added to a list of free spaces for that prefix block. Each prefix block has its own list of free slots. With this scheme even with 99% prefix occupancy in the TCAM and 1% free space, the total number of prefix moves using DLFS_PLO is at most 0.7% of the total number of prefix inserts and deletes.

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 and PC-DUOS+ architectures.

The problem of incorporating updates to packet classifiers stored in TCAMs has been studied in [6] and [2]. The authors in [6] 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 [6] is that it performs the deletes in the update batch first. This could lead to temporary inconsistencies in lookup [10].

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 [2], this approach could lead to high power consumption during look 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 [2], 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 [2] 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.

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. PC-DUOS+ also uses enhanced algorithms for ITCAM rule insertion which require fewer moves to rearrange rules for priority based adjustments.

III. PC-DUOS+: Methodology

PC-DUOS+ uses the two TCAM architecture used in PC-DUOS[14] and DUOS[9] (Figure 1). During lookup, the LTCAM and ITCAM are searched in parallel using the packet header information. If 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%[1], because the LTCAM has no priority encoder. For this lookup strategy to yield correct results, the following requirements must hold:

- R1) No packet is matched by more than one rule in the LTCAM.
- R2) 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 below.

A. Storing Rules in TCAMs

Figure 2 shows the overall flow of storing rules in the ITCAM and the LTCAM. The first phase involves creating a priority graph and a multi-dimensional trie for the rules in the classifier. This is further discussed in Section III-A1. 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 III-A2. In the third phase, the remaining rules are stored in the ITCAM in priority order. This is discussed in Section III-A3.

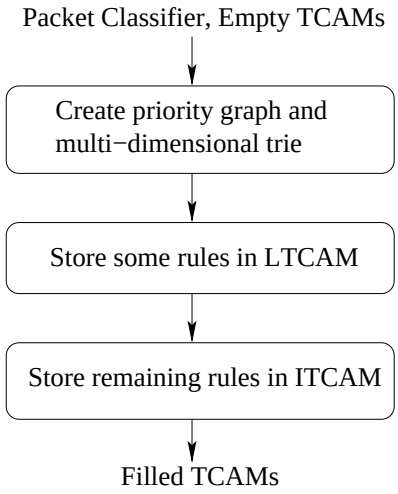


Fig. 2. Flow diagram for storing packet classifiers in TCAMs

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.

Each dimension in a multi-dimensional trie 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 paper, we use the DIRPE scheme of [3] that requires the use of a multi-bit trie. Our methodology may also be applied to other range encoding schemes, such as those in [12] and [13].

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 (R1) 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 vertices of the priority graph with in-degree 0 are independent rules. Further, these rules are also the highest priority rules among all rules that overlap with them. This satisfies the second requirement (R2) for a lookup to work correctly. Hence, we choose to enter these rules into the LTCAM. All remaining rules are entered in the ITCAM.

3) Storing rules in the ITCAM: The rules to be stored in the ITCAM, are assigned block numbers. The priority graph is used to assign block numbers as follows [2]. All vertices, to which there are no incoming edges, are assigned a block number of 1. All children of the vertices with block number 1 are assigned a block number of 2 and so on. A $parent$ of a

vertex v in the priority graph, is a vertex from which there is an incoming edge to v. Similarly, a $child$ of v is a vertex to which there is an out-going edge from v. Thus a child of any vertex is assigned a block number that is at least one more than that of this vertex. A $path$ in a graph is a sequence of vertices such that from each vertex there is an edge to the next vertex in the sequence. A non-trivial path is a path with at least two vertices. An $ancestor$ of a vertex v is a node that has a non-trivial path to v. A $descendant$ of v is a vertex to which there is a non-trivial path from v. In other words, a descendant of v has v as one of its ancestors.

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, 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 3. 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 are assigned using an iterative process.

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+ 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.

Algorithm: findOverlappingRules($ruleInstance$**)**
Inputs:
$ruleInstance$: a trie node representing a rule and storing its action.
Output:
list: a list of rules overlapping with the input rule

```
    get destination prefix Dest, source prefix Src from ruleInstance
  nodeD = root of destination trie;
  for (i=0; i<length of destination prefix; ++i)
    if (root of a source trie is stored at nodeD)
      nodeS = root of source trie
      for (j=0; j<length of source prefix and nodeS; ++j)
        if nodeS stores a rule R
          append R to list.
        branchBitS = Src[j];
        nodeS = nodeS→child[branchBitS];
      endfor
      if (nodeS != NULL) then
        visit all nodes in subtrie rooted at nodeS
        if (any node stores a rule R)
          append R to list.
        endif
      endif
    endif
    branchBitD = Dest[i];
    nodeD = nodeD→child[branchBitD];
  endfor
  visit all nodes in subtrie rooted at nodeD
  if (any node stores a rule R)
    append R to list.
  endif
```

Fig. 3. Find overlapping rules by trie traversal

B. Update algorithms

When an update request is received, the priority graph and the multi-dimensional trie are updated. Section III-B1 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 III-B2.

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 4.

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.

Fig. 4. 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

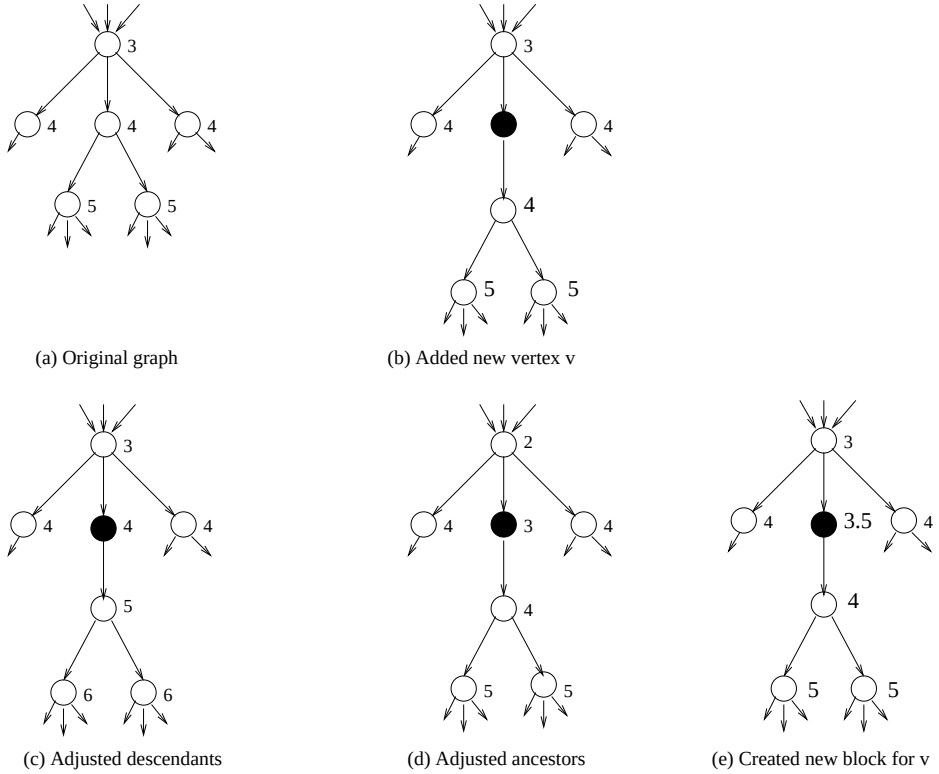


Fig. 5. Setting $hpri$ on a new vertex in a priority graph

be so moved is first inserted into the LTCAM and then deleted from the ITCAM using insert/delete procedures described in Sections III-B2b and III-B2d. 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 3. After adding a new vertex v to the priority graph, $v{\rightarrow}hpri$ is calculated. If v has no incoming edges $v{\rightarrow}hpri$ 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 $v{\rightarrow}hpri$ 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 5(c), (d) and (e). Figure 5(a) depicts a portion of the original graph. The number next to each vertex shows the $hpri$ value on that vertex. The newly added vertex v is colored black in Figure 5(b). In Figure 5(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 5(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 5(e) shows a case where a new block is inserted between the parent block and the child block, and the hpri associated with the new block is 3.5. Thus $v{\rightarrow}hpri$ is set to 3.5, and neither the descendants nor the ancestors of the new block are moved.

Figure 6 shows the algorithm to set $v{\rightarrow}hpri$. Figure 7 shows how the descendants are moved downwards. In Figure 6, we first calculate the number of moves to set $v{\rightarrow}hpri$ when descendants are moved downwards (childMoves) and when the ancestors are moved upwards (parentMoves). These calculations are based on the flow diagram in Figure 8(b). Suppose $v{\rightarrow}hpri$ 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 7. The algorithm to set $v{\rightarrow}hpri$ 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 and 7) 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.

Algorithm: insertRule(v)
Input: Rule stored in vertex v in the priority graph.

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

Fig. 6. Insert a rule in the ITCAM

Algorithm: moveChildDown(child)
Input: Rule stored in vertex 'child' in the priority graph.

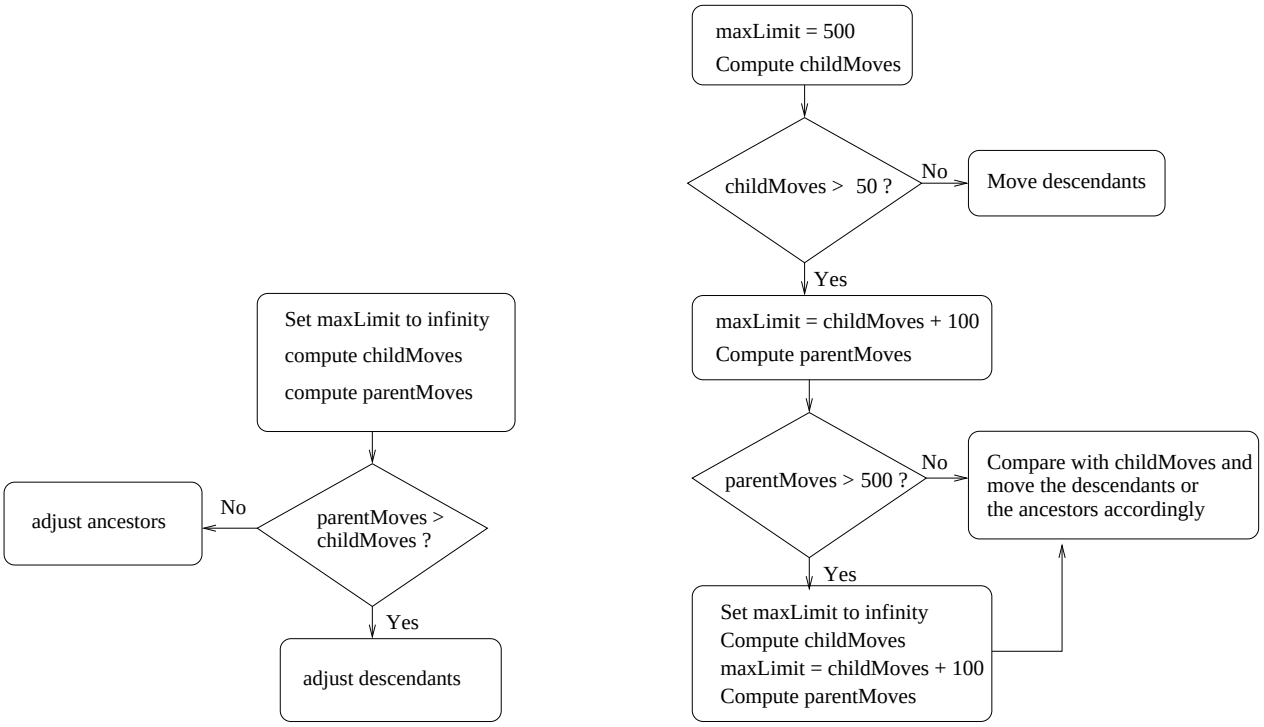
```
mP = find max(parent→hpri) from all parents of child
mC = find min(child→hpri) from all children of child
if (mP < child→hpri and child→hpri < mC) return;
block = priorityMap(maxP) + 1;
child→hpri = reversePriorityMap(block);
assign a slot in block for child; append child to nodeList;
if (!(child→hpri < 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
```

Fig. 7. Moving descendants downward in the ITCAM

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 8(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 8(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 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 5(e) and on lines 18 and 43 of Figure 6. 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 $v\rightarrow hpri$ is set to the average of the hpri-s of the two blocks (i.e. (hpri_of(B) + hpri_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



(a) Inefficient flow (b) Efficient flow

Fig. 8. Decision diagrams for priority adjustment of descendants vs. ancestors

top of the topmost ITCAM block as on line 38 of Figure 6, then $v\rightarrow$hpri is set to (1 + hpri_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 5(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 [10], [11], 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 55-59 of Figure 6 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 60).

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+ as well as 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. 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. 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+, as well as PC-DUOS, uses the memory management algorithm for DUOS for its LTCAM [9].

Since the blocks grow both ways, up as well as down, PC-DUOS+ has a modified initial rule placement policy as shown in Figure 9 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.

a) 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

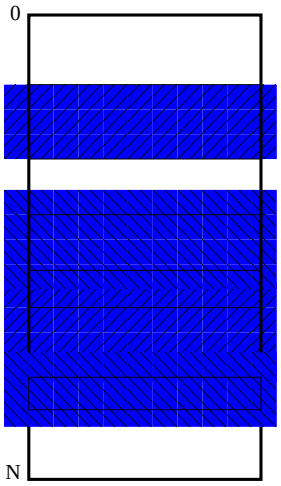


Fig. 9. Initial ITCAM layout

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.

b) 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.

c) 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.

d) 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.

IV. Experimental Results

The experimental setup is first described in Section IV-A. The results obtained for lookup and update performance are described in Sections IV-B and IV-C.

A. Setup

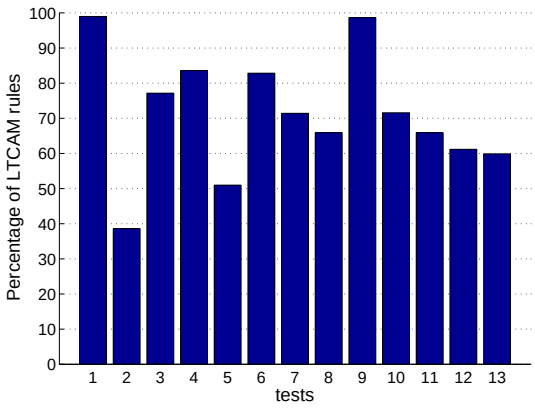
We programmed the lookup and update algorithms for STCAM, PC-DUOS and PC-DUOS+ in C++ and compared their performance on an x86 Linux box with a 64-bit, 1.2GHz CPU. We generated test data using ClassBench [7]. Each dataset was generated using one of the seeds provided in ClassBench. We randomly marked some of the rules in a dataset for insertion and some others for deletion. The rules marked for insertion were removed from the dataset to arrive at the initial configuration for the classifier. A random permutation of the removed rules (i.e., those marked for insertion) together with those marked for deletion define the update sequence. Figure 10 describes the data sets generated in this way using ClassBench. The first and second columns in this figure give the indexes and names of the classifiers, the third column shows the seed files in ClassBench from which these tests were derived, the fourth column shows the number of rules in the initial configuration of a classifier, and columns five to seven give the number of insert and delete operations in the update sequence. We used 12 seed files based on access control lists (acl), firewalls (fw) and IP chains (ipc) to generate the 13 classifiers. Out of these 13 tests, the first seven were used in [14]. Each rule in a dataset consists of the fields: source address, destination address, source port range, destination port range, and protocol.

Index	Dataset	seed_file	#Rules	#Inserts	#Deletes
1	acl1	acl1_seed	30075	69300	29700
2	fw1	fw1_seed	7989	28800	7200
3	ipc1	ipc1_seed	15338	34300	14700
4	acl2	acl2_seed	53970	45000	45000
5	fw5	fw5_seed	5571	45900	5100
6	acl4	acl4_seed	34254	5000	5000
7	ipc2	ipc1_seed	5165	94050	4950
8	acl3	acl3_seed	19745	2976	3124
9	acl5	acl5_seed	19492	12500	12500
10	fw2	fw2_seed	16668	15000	15000
11	fw3	fw3_seed	16841	33400	16600
12	fw4	fw4_seed	12882	10000	10000
13	ipc3	ipc2_seed	20000	15000	15000

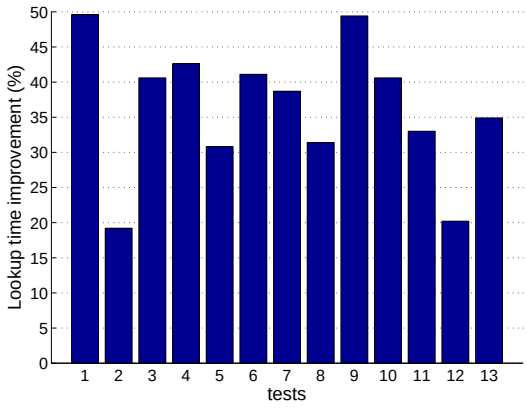
Fig. 10. Synthetic classifiers and update traces used in the experiments

We use DIRPE [3] to store the port ranges in the TCAM. DIRPE was implemented by using multi-bit tries for source

and destination port ranges. We assume that 36 bits are available for encoding each port range in a rule. With this assumption, we use strides 223333 for 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.



(a) Percentage of rules in the LTCAM



(b) Percentage of improvement in lookup time

Fig. 11. Number of rules in the LTCAM and improvement in lookup time relative to STCAM

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 [2], since PC-DUOS+' lookup performance is far superior to the worst case of [2], which is at least 4 times slower in the worst case, on our datasets (obtained as logarithm of the number of blocks).

We analyze the results based on two perspectives – improvement in lookup performance and improvement in update performance.

B. Lookup Performance

Recall that during a lookup, if a match is found in LTCAM of PC-DUOS+ then the corresponding action is returned faster. Figure 11(a) shows the percentage of rules that are entered in the LTCAM of PC-DUOS+. The graph shows that for two tests 1 (acl1) and 9 (acl5), over 99% of the rules are in the LTCAM. On the other hand, for test 2 (fw1), about 39% of the rules are in LTCAM. Having a large number of rules in the LTCAM makes the probability of finding a match in the LTCAM, higher. We computed the overall improvement in lookup time using the lookup traces generated using ClassBench. Each lookup trace had about 100,000 packet headers. Figure 11(b) shows the improvement in 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 12 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.

Index	Dataset	#Rules	#ITCAM	#LTCAM	#Lookups	%Improve
1	acl1	30075	305	29731	120301	49.6
2	fw1	7989	4885	3068	103857	19.2
3	ipc1	15338	3504	11834	107618	40.6
4	acl2	53970	8875	45095	107940	42.6
5	fw5	5571	2689	2796	105430	30.8
6	acl4	34254	5882	28372	103104	41.1
7	ipc2	5165	1476	3689	98136	38.7
8	acl3	19745	6737	13007	102851	31.4
9	acl5	19492	260	19209	97460	49.4
10	fw2	16668	4739	11929	100008	40.6
11	fw3	16841	5688	10986	103794	33
12	fw4	12882	5004	7878	103266	20.2
13	ipc3	20000	8027	11973	100163	34.9

Fig. 12. Number of rules in ITCAM and LTCAM of PC-DUOS+ and improvement in lookup time relative to STCAM

C. Update Performance

Figure 13 shows the number of TCAM writes needed to process the test update sequence by PC-DUOS+, PC-DUOS [14] and STCAM, normalized with respect to that of PC-DUOS+. A noticeable improvement in the number of writes

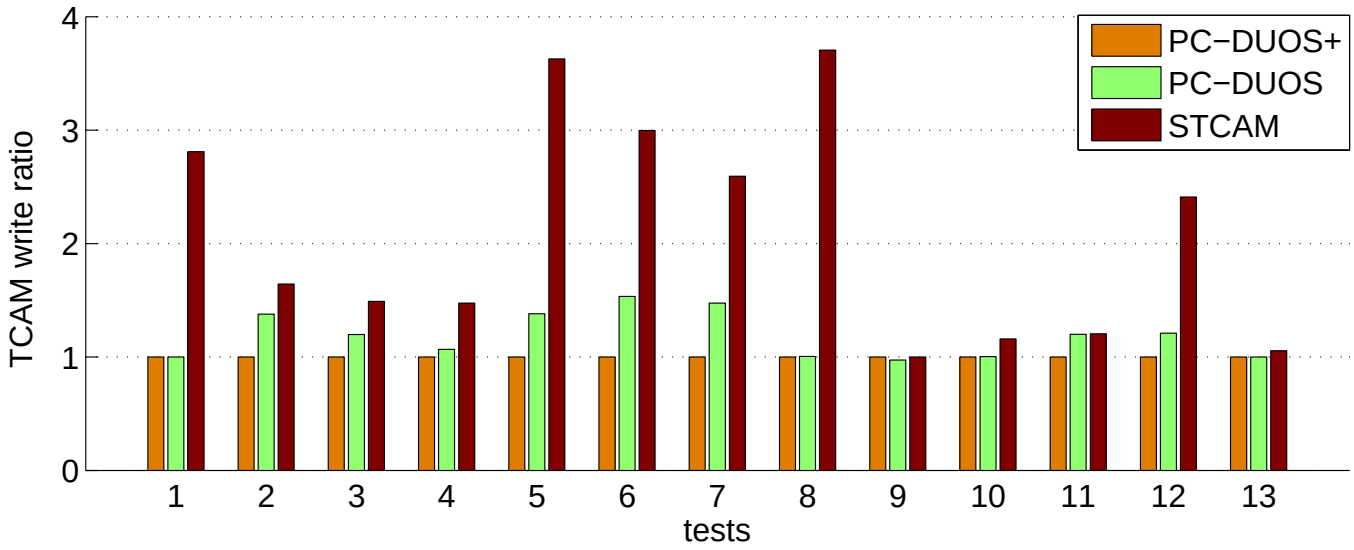


Fig. 13. Number of TCAM writes with respect to PC-DUOS+

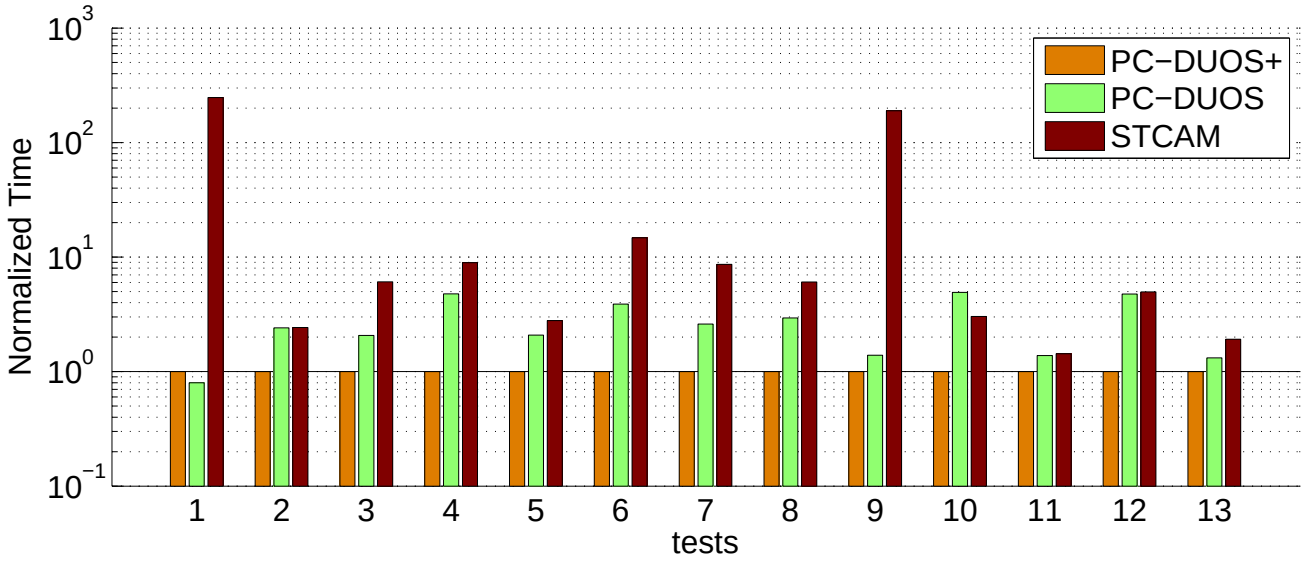


Fig. 14. Run time normalized with respect to PC-DUOS+

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 14 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 14 and 11(a), we see that the improvement in runtime is closely related to the number of rules that are in the LTCAM. Figure 14 shows that compared to PC-DUOS, there is an improvement in the runtime too, for all tests except test 1 (acl1).

From Figure 13 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 15), and the length of the maximum chain is just 3 (Figure 16). 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 graph with three distinct types of vertices. The first type is for rules with very specific

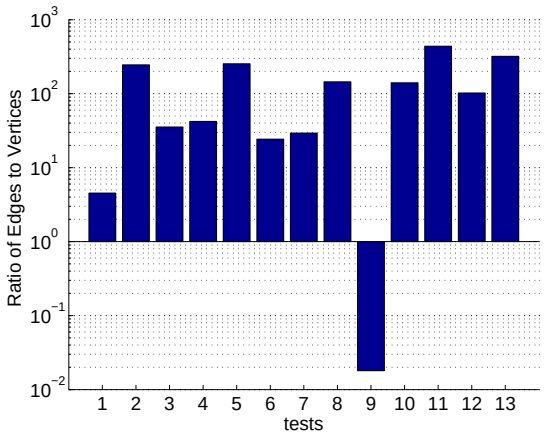


Fig. 15. Ratio of edges to vertices of graph

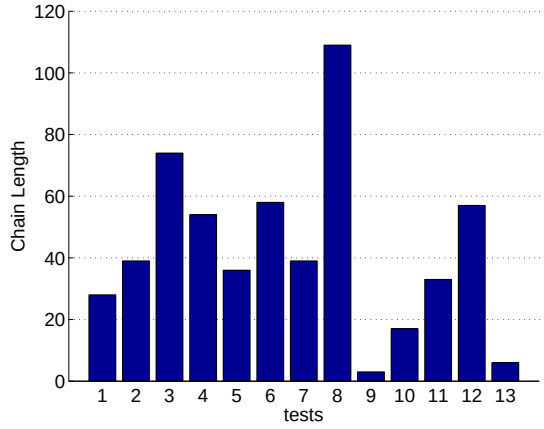
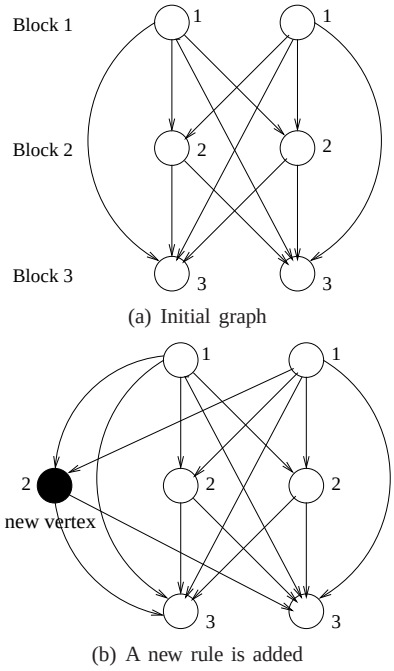


Fig. 16. Maximum chain length in graph before processing updates



(a) Initial graph

(b) A new rule is added

Fig. 17. A small graph representing test ipc3

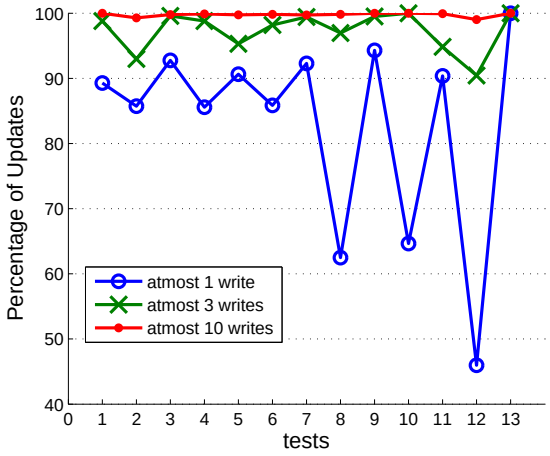


Fig. 18. Percentage of updates that require 1 write, ≤ 3 and ≤ 10 writes

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 17(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 17(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 18, 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.

Figure 19 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.

Figure 20 shows the actual number of TCAM writes for inserting or deleting rules in the different datasets and the time

Index	DataSet	PC-DUOS+		STCAM	
		#Average TCAM writes	#Worst case TCAM writes	#Average TCAM writes	#Worst case TCAM writes
1	acl1	1.18	31	3.31	53
2	fw1	2.07	3971	3.5	12014
3	ipc1	1.52	1900	2.25	3945
4	acl2	1.56	4274	2.28	6194
5	fw5	1.6	5167	5.84	26020
6	acl4	1.43	418	4.3	9821
7	ipc2	2.04	5152	5.27	17285
8	acl3	1.92	651	7.13	10946
9	acl5	1.12	10	1.124	11
10	fw2	1.71	9	1.98	9
11	fw3	1.55	2603	1.87	14693
12	fw4	2.83	1962	6.97	6996
13	ipc3	1	1	1.05	6

Fig. 19. Average and worst case TCAM writes for PC-DUOS+

Index	Data-Sets	PC-DUOS+		PC-DUOS		STCAM	
		#TCAM writes	Time(s)	#TCAM writes	Time(s)	#TCAM writes	Time(s)
1	acl1	116418	10	116393	8	327675	2469
2	fw1	76792	385	105866	928	126225	935
3	ipc1	74059	128	88736	265	110346	780
4	acl2	139397	193	148727	921	205568	1725
5	fw5	82044	969	113358	2012	297624	2702
6	acl4	14357	8	22030	31	43017	118
7	ipc2	201082	557	296663	1449	521653	4808
8	acl3	11736	16	11798	47	43494	97
9	acl5	28104	0.88	27366	1.2	28090	167
10	fw2	51250	161	51402	792	59406	488
11	fw3	77516	767	92955	1057	93426	1098
12	fw4	57822	76	69958	362	139434	378
13	ipc3	30000	217	30000	287	31647	416

Fig. 20. Total TCAM writes in PC-DUOS+, PC-DUOS and STCAM

taken to perform these updates in PC-DUOS+, PC-DUOS and STCAM.

V. Conclusion

PC-DUOS+, which is an enhancement of PC-DUOS [14] and an extension of DUOS[9], is proposed 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.

References

[1] M. Akhbarizadeh and M. Nourani, Efficient Prefix Cache For Network Processors, *IEEE Symp. on High Performance Interconnects*, 41-46, 2004.

[2] H. Song and J. Turner, Fast Filter Updates for Packet Classification using TCAM, Routing Table Compaction in Ternary-CAM, *GLOBECOM*, 2006

[3] K. Lakshminarayan, A. Rangarajan and S. Venkatachary, Algorithms for Advanced Packet Classification with Ternary CAMs, *SIGCOMM*, 2005.

[4] D. E. Taylor, Survey and taxonomy of packet classification techniques *ACM Computing Surveys (CSUR)* Volume 37 Issue 3, September 2005, 238-275 .

[5] S. Sahni, K. Kim, and H. Lu, Data structures for one-dimensional packet classification using most-specific-rule matching, *International Journal on Foundations of Computer Science*, 14, 3, 2003, 337-358.

[6] Z. Wang, H. Che, M. Kumar, and S.K. Das, CoPTUA: Consistent Policy Table Update Algorithm for TCAM without Locking, *IEEE Transactions on Computers*, 53, 12, December 2004, 1602-1614.

[7] D. E. Taylor and J. S. Turner, ClassBench: A Packet Classification Benchmark, *IEEE/ACM Transactions on Networking*, Volume 15, No. 3, June 2007, 499-511

[8] T. Mishra and S.Sahni, PETCAM – A Power Efficient TCAM for Forwarding Tables, http://www.cise.ufl.edu/˜sahni/papers/petcam.pdf

[9] T. Mishra and S.Sahni, DUOS – Simple Dual TCAM architecture for routing tables with incremental update, *IEEE Symposium on Computers and Communications*, 2010.

[10] T. Mishra and S. Sahni, CONSIST - Consistent Internet Route Updates *IEEE Symposium on Computers and Communications*, 2010.

[11] Z. Wang, H. Che, M. Kumar, and S.K. Das, CoPTUA: Consistent Policy Table Update Algorithm for TCAM without Locking, *IEEE Transactions on Computers*, 53, 12, December 2004, 1602-1614.

[12] H. Che, Z. Wang, K. Zheng and B. Liu, DRES: Dynamic Range Encoding Scheme for TCAM Coprocessors, *IEEE Transactions on Computers*, 57, 7, July 2008, 902-915.

[13] A. Bremler-Barr, D. Hay and D. Hendler, Layered Interval Codes for TCAM-based Classification, *INFOCOM* 2009.

[14] T. Mishra, S. Sahni and G. Seetharaman, PC-DUOS: Fast TCAM Lookup and Update for Packet Classifiers, *ISCC*, 2011.

APPENDIX B:

PC-TRIO: An Indexed TCAM Architecture for Packet Classifiers

Tania Banerjee-Mishra and Sartaj Sahni, University of Florida, Gainesville, FL, USA
{tmishra, sahni}@cise.ufl.edu
Gunasekaran Seetharaman, AFRL, Rome, NY, USA
Gunasekaran.Seetharaman@rl.af.mil

***Abstract*—We propose an indexed TCAM architecture, PC-TRIO, for packet classifiers. PC-TRIO uses wide SRAMs and index TCAMs. On our classifier datasets, PC-TRIO on an average reduced TCAM power by 96% and lookup time by 98%, compared to PC-DUOS+ [24] that does not use indexing or wide SRAMs. We extend PC-DUOS+ by augmenting it with wide SRAMs and index TCAMs using the same methodology as used in PC-TRIO, to obtain PC-DUOS+W. On ACL datasets, PC-DUOS+W reduced TCAM power by 86% and lookup time by 98%, compared to PC-DUOS+.**

I. Introduction

Packet classification is a key step in routers for various functions such as routing, creating firewalls, load balancing and differentiated services. Internet packets are classified into different flows based on packet header fields and 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. 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 $(F[1], F[2], \cdots, F[d])$, 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 $F[1], \cdots, F[d]$. 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.

TCAMs are used widely for packet classification. The popularity of TCAMs is mainly due to their high-speed table lookup mechanism in which all the TCAM entries are searched in parallel. Each bit of a TCAM may be set to one of the three states 0, 1, and '?' (don't care). A TCAM is used in conjunction with an SRAM. Given a rule (F, A), the filter F of a packet classifier rule is stored in a TCAM word and action A is stored in an associated SRAM word. All TCAM entries are searched in parallel and the first match is used to access the corresponding SRAM word to retrieve the action. So, when the packet classifier rules are stored in a TCAM in decreasing order of priority (increasing order of cost), we can determine the action corresponding to the matching rule of the highest priority, in one TCAM cycle. The main limitation of TCAMs is that these memories are power hungry. In fact at the same access rate, a TCAM may consume 30 times more power than an SRAM used for a software based classification [18]. The more the number of entries in the TCAM, the higher the power needed to perform a search. This problem is worsened for packet classifiers since typically a classifier rule includes port range fields that need multiple TCAM entries per rule for representation in the TCAM. This is called range expansion. Given that the source and destination port numbers are represented in 16 bits, the number of TCAM entries needed to represent a port range in the worst case is 30 corresponding to the range $[1, 2^{16} - 2]$. Thus, a filter having both source and destination port ranges set to $[1, 2^{16} - 2]$ undergoes a worst case expansion of $30 \times 30 = 900$ TCAM entries.

In this paper we evaluate a triple TCAM architecture, PC-TRIO for packet classifiers. In PC-TRIO, the TCAMs are augmented with indexing and wide SRAMs. The technique of indexing directly reduces the power consumption during lookup by selectively searching only a specific TCAM partition on the second stage of the lookup. In this architecture, port ranges are stored in wide SRAM words, rather than in the TCAM for most of the rules, and hence do not need multiple TCAM entries to represent them. The content of the wide SRAM word may be processed by a specialized and fast hardware. Finally, we present efficient incremental update algorithms. To the best of our knowledge, this is the first work that attempts to use an indexed TCAM architecture for packet classifiers.

Our paper is organized as follows. Section II presents background and related work in this area. Section III describes the PC-TRIO architecture and associated algorithms and Section IV presents experimental results. We conclude in Section V.

II. Background and Related Work

We describe the research on TCAM based packet classifiers in Section II-A, and describe existing indexed TCAM architectures for packet forwarding tables in Section II-B. We discuss the main problems in having an indexed TCAM architecture for packet classifiers in Section II-C and then in Section II-D show how to overcome these problems.

A. 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

This material is based upon work funded by AFRL, under AFRL Contract No. FA8750-10-1-0236.

in [1], [7], [6], [9], [13], [16] 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. [11] 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 [4], [15], [10], [12], [14] 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 [8] 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 [21] and PC-DUOS+ [24] use dual TCAMs for representation and incremental update of classifiers.

B. Forwarding tables with indexed TCAMs

The concept of using an index TCAM for a forwarding table was proposed by Zane et al. [2] and further refined by Lu and Sahni in [3]. A forwarding table can be viewed as a one dimensional packet classifier, containing only destination prefixes. Zane et al. [2] proposed a 2-level TCAM architecture in which the first level TCAM is an index to the partitions in the second level TCAM. We refer to a partition in a TCAM as a $bucket$. The partitions and indexes are constructed by $carving$ the binary trie representing the prefixes in the forwarding table.

Lu and Sahni in [3], further augment the traditional 1-level TCAM lookup structure as well as the 2-level TCAM structure of Zane et al. [2] 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. Mishra and Sahni, in PETCAM [19] and DUO [20] obtained further reduction in power and TCAM space for packet forwarding, using the indexing and wide SRAM schemes. In particular, DUO [20] is a dual TCAM architecture used for packet forwarding that uses efficient memory management algorithms for the two TCAMs. These algorithms help DUO in executing consistent incremental updates [22], [23].

C. Problems in storing a classifier in an indexed TCAM

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 1, where each rule has two fields - a destination, and a source. The classifier is mapped to the indexed TCAM in Figure 2. The data TCAM has two buckets

Filter		Action	Priority
Destination	Source		
00*	1000	A1	1
0*	0101	A2	2
000*	01*	A3	3
*	*	A4	4

Fig. 1. An example classifier

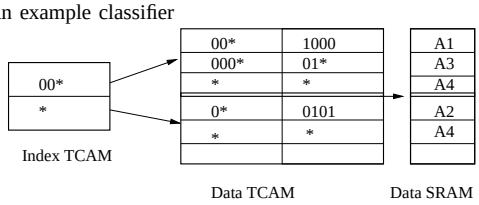


Fig. 2. Classifier rules stored in a indexed TCAM

and the index TCAM uses bits from the destination prefix of each rule, to index into the buckets of the data TCAM. In this 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 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 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 $covering\ prefix$ [2], [3], 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 1 is a covering rule and hence entered in both the TCAM buckets in Figure 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.

D. Overcoming these problems

The dual TCAM architecture presented for PC-DUOS [21] and PC-DUOS+ [24], as well as the PC-TRIO architecture presented in this paper, makes it possible to get around both

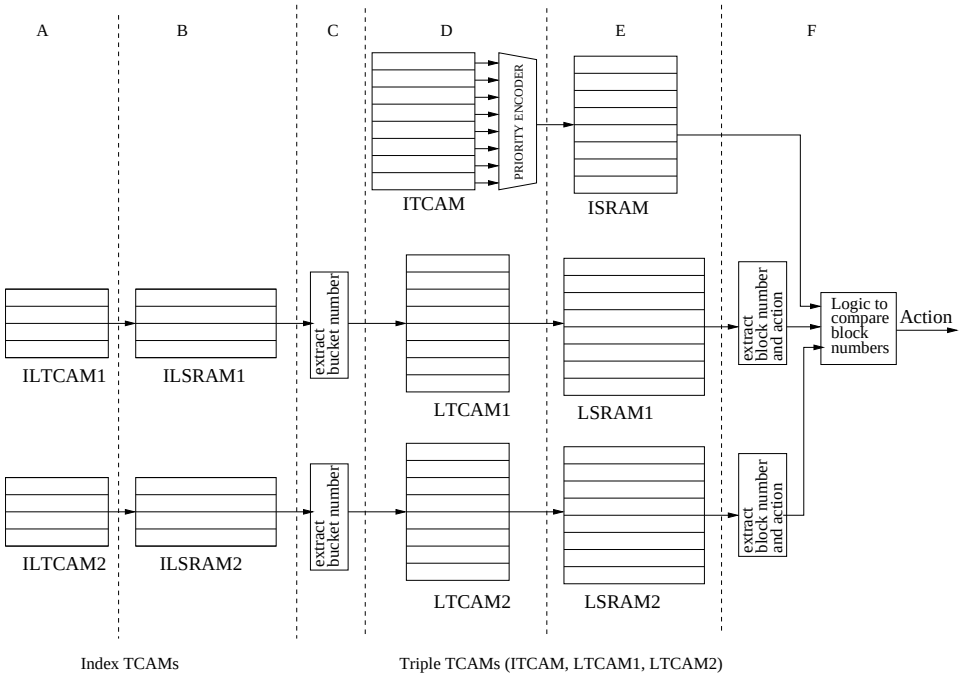


Fig. 3. PC-TRIO Architecture

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.

III. PC-TRIO

The PC-TRIO architecture is presented in Section III-A. The algorithms for storing and updating the TCAMs are discussed in Sections III-B and III-C. The differences with related architectures are presented in Section III-D.

A. The Architecture

Figure 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 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.

B. 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 III-B1. In the second and third steps, the LTCAM1 and LTCAM2 subsystems are populated as discussed in Sections III-B2 and III-B3, respectively. The fourth step is to store the remaining rules in the ITCAM in priority order, which is discussed in Section III-B4.

1) Representing Classifier Rules: The classifier rules are represented in a priority graph, which contains one vertex for

each rule in the classifier. A priority graph contains one vertex for each rule in the classifier. 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 edge (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 [8]. All vertices with in-degree 0 are assigned the block number 1. Each remaining vertex v is assigned a block number equal to

$$1 + \max_{(u,v) \in E} \{block\ number\ of\ u\}$$

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.

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 III-B2. All the remaining rules are used to create another multi-dimensional trie, Trie2, in which fields 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.

2) Storing rules in the LTCAM1: The process of storing rules in the LTCAM1 subsystem is described in five subsections below. First, independent rules are identified (Section III-B2a), next, the format of storing information in a wide LSRAM word is discussed (Section III-B2b), then we describe the creation of LTCAM1 entries using the process of carving (Section III-B2c). Next we describe partial port range expansion (Section III-B2d) that may be necessary, and finally, the creation of ILTCAM1 and ILSRAM1 entries (Section III-B2e).

a) Identifying Independent Rules: Recall that two rules are $independent$ iff no packet is matched by both rules. For the LTCAM1, we are interested in identifying the largest set of rules that are pairwise independent. 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 using a two step process. In the first step, we create a *leaves of leaves set* [21] of protocol nodes in a multi-dimensional trie using the algorithm in Figure 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.

```
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.
    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
```

Fig. 4. Selecting protocol nodes for leaves of leaves set

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.

b) 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 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) $Data_j$: $Data_1, \cdots, Data_N$ 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 jth 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

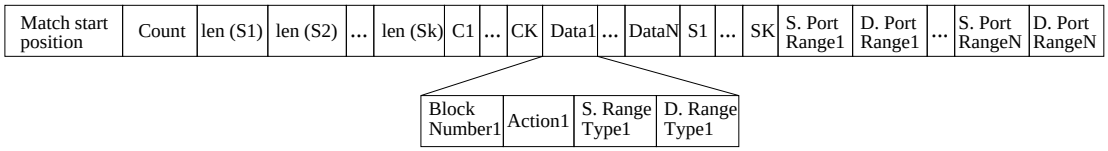


Fig. 5. Data encoding in a wide SRAM word

word, a whole range is never entered and only one port number is entered for a range with the same start and end points.

c) 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 $visit\_postorder$ of DUO [20] that has been enhanced for multi-dimensional tries. This carving heuristic creates independent (disjoint) entries for the

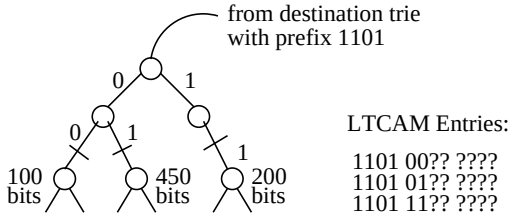


Fig. 6. Nodes in a source trie is being carved.

LTCAM1. The path starting from the root of Trie1 to the root of the subtrie defines an LTCAM1 entry. Figure 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 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 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 the width of a SRAM word.
- C2) A descendant of p was carved.

The second condition ensures that the carving creates disjoint TCAM entries [20].

d) 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(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(b), and carve nodes on this trie to

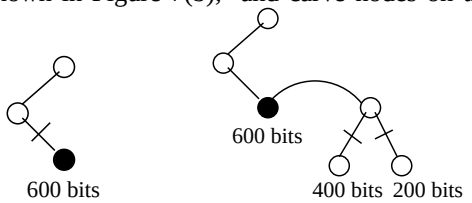


Fig. 7. Prefixes in forwarding table before and after applying updates

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(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.

e) 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.

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.

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.

C. Incremental Updates

When an update request is received, the priority graph is updated as described in Section III-C1. Then Trie1 and, if necessary, Trie2 are updated as described in Section III-C2. As the tries are updated, it may be necessary to carve the tries at different trie nodes. This is discussed in Section III-C3. Updating the TCAMs is discussed in Section III-C4.

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.

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.

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.

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 [20] 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

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

Fig. 8. Differences among the architectures

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.

D. Differences among PC-DUOS, PC-DUOS+, PC-DUOS+W and PC-TRIO

We note that the methodology used in this paper 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 [21] may be similarly extended to obtain PC-DUOSW, we do not consider this extension here as PC-DUOS+ was shown to be superior to PC-DUOS [24]. Figure 8 highlights the differences among PC-DUOS, PC-DUOS+, PC-DUOS+W and PC-TRIO.

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.

IV. Experimental Results

We compare PC-TRIO, with PC-DUOS+W and PC-DUOS+ [24]. We first give the setup used by us for the experiments in Section IV-A and then describe our datasets in Section IV-B. Finally we present our results in Section IV-C.

A. 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 words using Verilog and synthesized it using Synopsys Design Compiler to obtain power, area and gate count estimates. We used CACTI [25] and a TCAM power and timing model [17] 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 [29].

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 [1] 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 [1], [21].

B. Datasets

We used two sets of benchmarks derived from ClassBench [5]. 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+ [24].

The second set of benchmarks was reused from [24]. There are 13 datasets here which are used to compare incremental update performance of PC-TRIO, with PC-DUOS+ [24] and PC-DUOS+W.

C. Results

1) Number of TCAM entries: Using wide SRAM words to store portions of classifier rules, reduces the number of TCAM entries. Figure 9 gives the results of storing our datasets 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 10(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

Index	Dataset	#Rules	PC-DUOS+				PC-DUOS+W				PC-TRIO			
			Entries	#ITCAM	Watts	Time(ns)	Entries	#ITCAM	Watts	Time(ns)	Entries	#ITCAM	Watts	Time(ns)
1	acl1	99309	117033	379	36	2624.39	21146	379	0.23	0.50	21085	182	0.19	1.00
2	acl2	74298	101857	19421	31	1122.39	37491	19421	6.35	30.36	36593	18439	6.04	149.43
3	acl3	99468	131243	30859	40	1640.47	52632	30859	9.47	80.49	26823	1017	0.40	2.19
4	acl4	99334	127320	25189	39	1730.46	49912	25189	7.98	45.95	34034	6547	2.32	24.12
5	acl5	98117	105375	1535	32	2072.16	32932	1535	0.53	0.41	34993	2209	0.77	4.98
6	fw1	89356	142085	91473	43	2466.72	98425	91473	27.92	2318.82	26610	4864	1.60	15.01
7	fw2	96055	129249	27084	39	1543.76	43146	27084	8.30	86.77	22196	1494	0.53	3.18
8	fw3	80885	117731	39199	36	1007.04	51228	39199	11.99	215.21	26269	7479	2.38	30.09
9	fw4	84056	211403	116149	64	3182.03	131505	116149	35.46	2139.21	27617	4894	1.60	15.16
10	fw5	84013	111989	55650	34	930.94	65598	55650	17.00	615.49	22361	3454	1.15	9.02
11	ipc1	99198	112154	22165	34	1288.02	41920	22165	6.82	45.11	23894	567	0.26	1.40
12	ipc2	100000	100000	30133	30	784.69	47247	30133	9.23	113.77	20195	0	0.09	0.75

Fig. 9. Number of TCAM entries, ITCAM entries and TCAM power and lookup time in PC-DUOS+, PC-DUOS+W, PC-TRIO

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 III-B2d). So all LTCAM entries in PC-DUOS+W and PC-TRIO were at most 72 bits.

2) Power: Figure 9 gives the TCAM power consumption during a lookup, while Figure 10(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 axis is scaled logarithmically

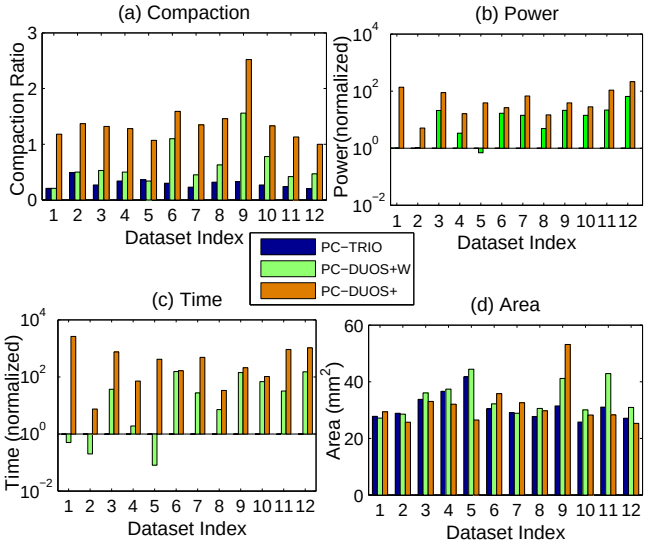


Fig. 10. Comparison of compaction ratio, total power, lookup time and area

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.

3) Lookup Performance: Figure 10(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 Figure 9) 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 Figure 9). 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 then 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%).

4) Space requirements: We obtained SRAM area from CACTI results and estimated TCAM area using the same technique as used in PETCAM [19], where area of a single cell is multiplied by the number of cells and then adjusted for wiring overhead. Figure 10(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+.

5) Update Performance: Figure 11 shows the average number of TCAM writes used per update on the datasets from [24]. PC-TRIO needs comparable number of 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

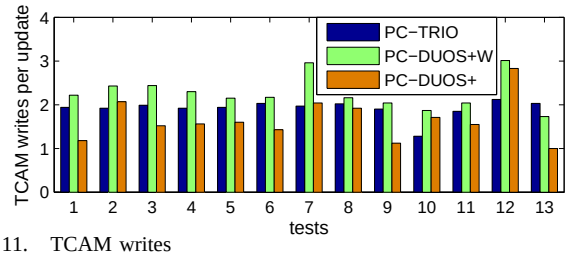


Fig. 11. TCAM writes

LTCAM have the highest priority compared to overlapping rules.

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 [26], [27] and it was found found that the design successfully met the timing constraints with a 500MHz clock. The results are

Process	Time (ns)	Throughput (Msps)	Voltage (V)	Power (mW)	Gate Count
0.18μm	2	500	1.8	61.13	59724

Fig. 12. Timing and power results for additional hardware

presented in the Figure 12. 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 [28], 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.

V. Conclusion

We presented an indexed TCAM architecture, PC-TRIO, for packet classifiers. The methods to add indexing and wide SRAMs were applied on PC-DUOS+ [24] 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.

References

[1] K. Lakshminarayan, A. Rangarajan and S. Venkatachary, Algorithms for Advanced Packet Classification with Ternary CAMs, *SIGCOMM*, 2005.

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

[3] W. Lu and S. Sahni, Low Power TCAMs For Very Large Forwarding Tables, *INFOCOM*, 2008.

[4] R. Draves, C. King, S. Venkatachary, and B.Zill, Constructing Optimal IP Routing Tables, *INFOCOM*, 1999.

[5] D. E. Taylor and J. S. Turner, ClassBench: A Packet Classification Benchmark, *TON*, 15, 3, Jun 2007, 499-511.

[6] H. Che, Z. Wang, K. Zheng and B. Liu, DRES: Dynamic Range Encoding Scheme for TCAM Coprocessors, *TOC* 57, 7, Jul 2008, 902-915.

[7] A. Bremler-Barr, D. Hay and D. Hendler, Layered Interval Codes for TCAM-based Classification, *INFOCOM* 2009.

[8] H. Song and J. Turner, Fast Filter Updates for Packet Classification using TCAM, Routing Table Compaction in Ternary-CAM, *GLOBECOM*, 2006

[9] D. Pao, P. Zhou, B. Liu, and X. Zhang, Enhanced Prefix Inclusion Coding Filter-Encoding Algorithm for Packet Classification with Ternary Content Addressable Memory, *Computers & Digital Techniques, IET*, 1, 5, Sep 2007, 572-580.

[10] S. Suri, T. Sandholm and P. Warkhede, Compressing Two-Dimensional Routing Tables, *Algorithmica*, 35, 4, 2003, 287-300.

[11] E. Spitznagel, D. Taylor, and J. Turner, Packet Classification Using Extended TCAMs, *ICNP*, 2003, 120-131.

[12] C. R. Meiners, A. X. Liu, and E. Torng, TCAM Razor: A Systematic Approach Towards Minimizing Packet Classifiers in TCAMs, *ICNP*, 2007, 266-275.

[13] H. Liu, Efficient Mapping of Range Classifier into Ternary-CAM, *Hot Interconnects*, 2002, 95-100.

[14] A. X. Liu, C. R. Meiners, and Y. Zhou, All-Match Based Complete Redundancy Removal for Packet Classifiers in TCAMs, *INFOCOM*, 2008, 574-582.

[15] Q. Dong, S. Banerjee, J. Wang, D. Agrawal, and A. Shukla, Packet Classifiers in Ternary CAMs can be Smaller, *SIGMETRICS*, 2006, 311-322.

[16] J. van Lunteren and T. Engbersen, Fast and Scalable Packet Classification, *IJSAC*, 21, 4, May 2003, 560-571.

[17] B. Agrawal and T. Sherwood, Ternary CAM Power and Delay Model: Extensions and Uses, *TVLSI*, 16, 5, May 2008, 554-564.

[18] O. Rottenstreich and I. Keslassy, Worst-Case TCAM Rule Expansion, *INFOCOM*, 2010.

[19] T. Mishra and S.Sahni, PETCAM – A Power Efficient TCAM For Forwarding Tables, *http://www.cise.ufl.edu/~sahni/papers/petcam.pdf*,

[20] T. Mishra and S.Sahni, DUO – Dual TCAM Architecture for Routing Tables with Incremental Update, *http://www.cise.ufl.edu/~sahni/papers/duo.pdf*

[21] T. Mishra, S.Sahni, and G. Seetharaman, PC-DUOS: Fast TCAM Lookup and Update for Packet Classifiers, *ISCC*, 2011.

[22] Z. Wang, H. Che, M. Kumar, and S.K. Das, CoPTUA: Consistent Policy Table Update Algorithm for TCAM without Locking, *TOC*, 53, 12, Dec 2004, 1602-1614.

[23] T. Mishra and S. Sahni, CONSIST - Consistent Internet Route Updates *ISCC*, 2010.

[24] T. Mishra, S. Sahni and G. Seetharaman, PC-DUOS+: A TCAM Architecture for Packet Classifiers *http://www.cise.ufl.edu/~sahni/papers/pcduos+.pdf*

[25] N. Muralimanohar, R. Balasubramonian and N. P. Jouppi, Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0, *ISM* December 2007, 3-14

[26] J. B. Sulistyo, J. Perry and D. S. Ha, Developing Standard Cells for TSMC 0.25um Technology under MOSIS DEEP Rules, *Virginia Tech, Technical Report VISC-2003-01* Nov 2003.

[27] J. B. Sulistyo and D. S. Ha, A New Characterization Method for Delay and Power Dissipation of Standard Library Cells, *VLSI Design* 15, 3, Jan 2002, 667-678.

[28] H. Noda, K. Inoue, M. Kuroiwa, F. Igaue and K. Yamamoto, A Cost-Efficient High-Performance Dynamic TCAM With Pipelined Hierarchical Searching and Shift Redundancy Architecture, *IJSSC*, 40, 1, Jan 2005, 245-253.

[29] Renesas R8A20410BG 20Mb Quad Search Full Ternary CAM. http://am.renesas.com/products/memory/TCAM/tcam_root.jsp.

APPENDIX C:

PETCAM–A Power Efficient TCAM For Forwarding Tables *

Tania Mishra and Sartaj Sahni
Department of Computer and Information Science and Engineering,
University of Florida, Gainesville, FL 32611
{tmishra, sahni}@cise.ufl.edu

November 10, 2008

Abstract

We investigate various TCAM architectures recently proposed for TCAM power and memory reduction and show that far better power and memory performance is possible when we use an optimal prefix set for the given router table than when the original prefix set or the reduced prefix set as proposed in other work is used. For EaseCam [8, 9], our experiments show a power and TCAM memory reduction of 96% to 98% and 62% to 69% respectively. For the suffix node architecture of [3], we get a power and TCAM memory reduction of 16% to 25% and 45% to 78% respectively.

Keywords
Packet forwarding, TCAM, power.

1 Introduction

Internet packets get from source to destination via a number of hops. At each hop, a forwarding engine uses the destination address of the packet and a set of rules to determine the next hop for the packet. A packet forwarding rule (P, H) comprises a prefix P and a next hop H. 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 that matches d (we assume, throughout this paper, that no two rules have the same prefix). We refer to the set of rules as the rule table or router table. Figure 1 shows a small router table with 6 prefixes. The prefix associated with rule R4 is 01 (the * at the end indicates a sequence of don't care bits) and the associated next hop is H4. Rule R4 matches all destination addresses that begin with 01. The length of the prefix 01 associated with R4 is 2. A destination address that begins with 010 is matched by rules R1, R2, R4, and R5. Of 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 010.

[11, 12] survey the many solutions that have been proposed for longest prefix matching in the context of packet forwarding. Our focus, in this paper, is longest prefix matching using a TCAM (ternary content addressable memory). Each bit of a TCAM may be set to one of the 3 states 0, 1, and x (don't care). A simple and fast solution to longest prefix matching results from the use of a TCAM in conjunction with an SRAM. The prefix of a rule is stored in a word of TCAM and the next hop is stored in the corresponding SRAM word. Figure 2 shows a TCAM in which each word is 4 bits long; the prefixes of our 6-rule example of Figure 1 have been stored in the

*This research was supported, in part, by the National Science Foundation under grant ITR-0326155

	Prefixes	Next Hop
R1	*	H1
R2	0*	H2
R3	1*	H3
R4	01*	H4
R5	010*	H5
R6	111*	H6

Figure 1: An example 6-prefix forwarding table

TCAM in decreasing order of length along with an SRAM in which the next hop information has been stored. A TCAM searches all its words, in parallel, for the first word that matches the content of its search register. By loading a destination address into the search register of a TCAM we can determine the index of the first TCAM word that matches this destination address. Using this index, we then access the corresponding SRAM word to determine the next hop. So, when router-table prefixes are stored in a TCAM in decreasing order of length, we can determine the next hop in 1 TCAM cycle! We note that, in practice, using the described strategy, a TCAM word will be 32 bits for IPv4 applications.

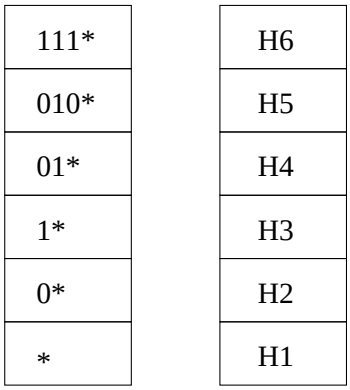


Figure 2: TCAM for the 6 rules of Figure 1

Although TCAMs lead to a very simple and fast solution to the packet forwarding problem of finding the next hop associated with the longest matching prefix, there are several pitfalls associated with their use. These pitfalls include high power consumption, limited capacity, and high cost. Several researchers have recently proposed methods to alleviate the power consumption and capacity limitations. Central to the proposed methods [8, 9, 3, 19] to reduce power consumption is the observation that the power consumed by a TCAM search is proportional to the size of the portion of the TCAM that needs to be searched rather than to the TCAM's overall size. Zane et al. [19] propose a two-level architecture in which the first level extracts some number of bits from the destination address and these extracted bits are used to index into a segment of the TCAM that is to be searched for the longest matching prefix. Ravikumar et al. [8, 9] propose a similar two-level architecture. However, the extracted bits are restricted to be a prefix of the destination address (first 8 bits) and the TCAM segments are of variable

size. The use of variable size segments requires the use of a table of segment start addresses but reduces wasted TCAM space. The two-level schemes of [8, 9, 19] also increase the effective capacity of a TCAM as the word size of the TCAM is reduced by the number of extracted bits. So, in an IPv4 application, for example, if 8 bits are used for the first-level indexing, a TCAM word need only be 24 bits rather than 32 bits (as in the scheme of Figure 3). Liu [7] proposes the use of pruning and mask extension to compact a TCAM table and hence reduce the number of rules that has to be stored. This compaction reduces power consumption and also increases the effective capacity of the TCAM. Lu and Sahni [3] propose table segmenting methods and the use of wide SRAMs to reduce power consumption and increase effective table capacity.

In this paper, we propose the use of a minimum set of rules equivalent to those in the given router table coupled with the wide SRAM strategy of [3]. We perform batch updates to the set of rules to accomodate the incoming route advertisements. We begin in Section 2 by reviewing related work. In this section, we clarify the proposal of [7] and point out deficiencies in the scheme of [8, 9]. In Section 5 we describe our proposed PETCAM method. An experimental evaluation of the various methods proposed for low-power TCAMs is done in Section 6.

2 Background and Related Work

Much research has been done to improve the power efficiency of TCAM-based router tables [7, 3, 8, 9, 19, 13, 14, 15, 16, 17]. Pure hardware approaches for power reduction are presented in [13, 14, 15, 16]. Z. Wang et al in [17] present an algorithm for consistent and incremental updates to TCAMs. We describe the results reported in [7, 3, 8, 9, 19] in this section as these are most relevant to the work we report in this paper.

Definition 1 *$P1 \subset P2$ iff $addr(P1) \subset addr(P2)$, where $addr(P)$ is the set of addresses matched by prefix P. Note that $P1 \subset P2$ iff $P2$ is a proper prefix of $P1$.*

Definition 2 *A rule $(P1, H1)$ is* Type I redundant *iff (a) there exists a rule $(P2, H2)$ such that $P1 \subset P2$ and $H1 = H2$ and (b) there is no rule $(P3, H3)$ such that $P1 \subset P3 \subset P2$.*

Definition 3 *A* generalized prefix *is a sequence comprised of the symbols 0, 1, and ? and possibly terminated by the symbol *. A* simple prefix *(or simply, prefix) is a generalized prefix that has no occurrence of the symbol ?. (Alternatively, we may limit the occurrence of the symbol ? to the right end of the sequence. Note that ?s at the right end of a sequence may be replaced by a * so that the sequence 10??? may be regarded as a simple prefix by rewriting it is 10*.)*

For example, 0??1??0* and ??100?11* are generalized prefixes. In router table applications, a generalized prefix may be stored in a word of TCAM by replacing * with a suitable number of ?s.

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

Liu [7] proposes two schemes–pruning and mask extension–to compact the rules of a router 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 elimination of type I redundant prefixes, 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 an equivalent set of generalized prefixes T such that $|T| \leq |S|$. Liu [7] proposes the use of a logic minimization heuristic–Espresso II–to compute a nearly minimal equivalent set T. Liu [7], however, does not address the issue of how to assign lengths to the generalized prefixes of T (or, equivalently, how to place the generalized prefixes of T into the TCAM) so that a TCAM search reports the same next hop as reported using longest prefix matching on the original set of simple prefixes. We address this issue in Section 3. Liu [7] reports that pruning and mask extension result in a reduction of 42% to 48% in the number of generalized prefixes that need to be stored in the TCAM. Note that without pruning and mask extension, we store simple prefixes in the TCAM. However, the TCAM word size is the same regardless of whether simple or generalized prefixes are stored. So, a 42% reduction (say) in the number of generalized prefixes translates to a 42% reduction in TCAM memory.

Ravikumar et al. [8, 9] extend the work of Liu [7] and propose the 2-level EaseCAM architecture for router tables (Figure 3). For an IPv4 router 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 (note that to store the terminating * of a generalized prefix, we must replace it with a sufficient number of ?s so that the total number of symbols in the generalized prefix is 32). Consequently, second-level TCAM words are 25% smaller than the TCAM words in the design of [7]. Prefixes shorter than 8 bits are stored in a separate bucket. The pruning process of Ravikumar et al. [8, 9] is identical to that of Liu [7]–type I redundant prefixes are eliminated. The compaction process of Ravikumar et al. [8, 9] differs from that of Liu [7] in how generalized prefixes are created from a set of same-hop prefixes that is free of type I redundancies. In an effort to reduce the time required by Espresso to process same length same hop prefixes, Ravikumar et al. [8, 9] propose aggregating prefixes (in a cluster) that have the same hop into sets in which the prefixes have a common longest sub-prefix of size a multiple of 8. Then, prefixes in each such aggregated set are expanded using prefix expansion [8, 9] so that the length of each prefix is a multiple of 8. For example, following aggregation the prefixes in an aggregated set may have length between 16 and 23 with all prefixes in this set having the same first 16 bits. Using prefix expansion, the lengths of all prefixes in the set becomes 24. Since all prefixes in this prefix-expanded aggregated set have the same first 16 bits, Espresso may be used to find a minimum number of generalized prefixes equivalent to the 7-bit suffixes in this set. Working with 7-bit suffixes rather than full 23-bit prefixes reduces the run time of Espresso [9]. The fact that the aggregated prefix-expanded sets are relatively small (compared to sets of same hop same length prefixes) is another (and significant) contributing factor to the observed reduction in time spent on Espresso optimization. Although prefix aggregation and expansion reduce Espresso time with little loss in

compaction effectiveness [8, 9], there are correctness issues that we address in Section 4. Since the power consumed by a TCAM lookup is proportional to the size of the TCAM segment that is searched rather than to the overall TCAM size, the scheme of [9] achieves power reduction from using a compacted set of prefixes, storing only the last 24 bits of each prefix in TCAM, and from searching only the prefixes in a cluster.

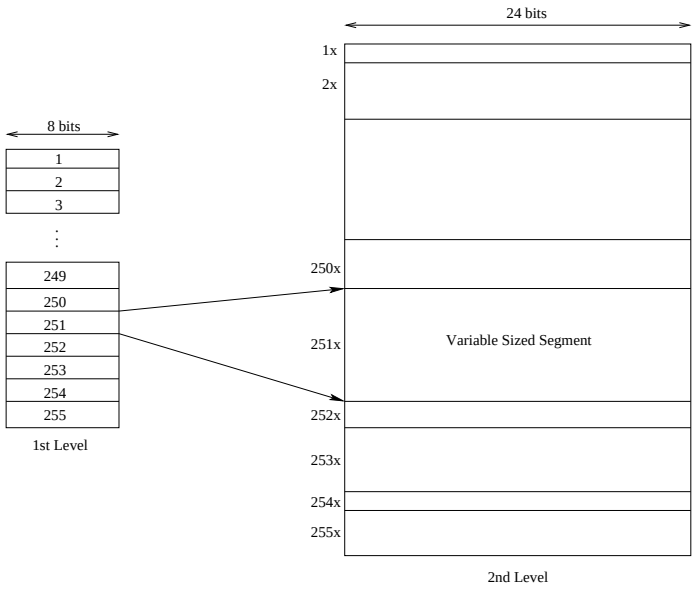


Figure 3: EaseCam architecture of [9]

Zane et al. [19] propose two schemes to achieve power reduction. In the first, bit selection, a few bits (not necessarily the first few) 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 router table are stored in the TCAM, power reduction results from having to search only one partition [1]. Additional power reduction is possible if the partition selector bits are extracted from the prefixes before storage in the TCAM as this results in a reduction in the total number of bits in a partition. Note that bit selection, which predates the work of [9], is similar to the 2-level strategy employed in [9], where the first 8 bits are used to determine the partition to search.

The second strategy proposed by Zane et al. [19] 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 router-table prefixes. Although both Zane et al. [19] and Ravikumar et al. [9] propose 2-level TCAM architectures, Zane et al. [19] do not compact the router table (except when bit selection is used and the partition selector bits are not stored in the second level TCAM) while Ravikumar et al. [9] do. As a result, the total TCAM memory required by the schemes of Zane et al. [19] is more than that required by the

[1] The power required by a TCAM lookup is proportional to the total number of bits in the TCAM partition that is searched.

scheme of Ravikumar et al. [9].

The most recent work on TCAM power reduction in the context of router tables appears to be that of Lu and Sahni [3]. They augment the traditional 1-level TCAM lookup structure as well as the 2-level TCAM structure of Zane et al. [19] 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.

3 Issues Related to [7]

As noted in Section 2, when logic minimization is applied to a set of same-hop same-length prefixes, we get a set of equivalent generalized prefixes. So, 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 router table has another prefix $00*$ whose next hop is $H2$ and $H1 \neq H2$. When using the original prefix set $C = \{000*, 001*, 010*, 011*, 00*\}$, packets with destination address beginning with 000 are sent to $H1$. Consider what happens when we apply the compaction scheme of Liu [7]. 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 $H2$. Using the prefix set D, packets with destination addresses that begin with 000 are sent to $H2$! We can overcome this difficulty in one of two ways. The first and simplest is to declare the length of each generalized prefix in the optimized set D to be the same as that of the prefixes in the set A. This ensures that, when prefixes are loaded to the TCAM in length order, the outcome is the same (in terms of next hop) as when the original prefix set is loaded in length order. For example, using this definition of length for a generalized prefix, 0* in set D has length 3, and prefix 00* has length 2. Thus, 0* is loaded first in the TCAM followed by 00*.

The second strategy is to use a more intuitive definition of length such as the index of the rightmost symbol that is not a ? or a *. So, the length of 1??01* is 5 and the length of ??00??1* 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 (using the just stated intuitive definition) of the generalized prefix G. Using such a definition works provided we remove also type II redundant rules as is shown below.

Definition 5 *A rule* (P_1, H_1) *is* Type II redundant *iff the router 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 a* P_i, $2 \leq i \leq k$.

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 [7] nor Ravikumar et al [8, 9] remove type II redundant rules. 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, 1?00?1* = 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_qs.*

Theorem 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 $f(i,j)$, $1 \le f(i,j) \le s$, such that R_{ij} and $S_{f(i,j)}$ are not disjoint, $1 \le i \le r$, $1 \le j \le q(i)$. 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_qs.

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_is in R. We may use this replacement transformation as often as need to replace all R_is 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_qs.

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_qs. So, each R_{ij} and hence R_i is the sum of some of the S_qs.

■

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 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 *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 [7]. 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. ■

From Theorem 2, it follows that if we start with a set of prefixes that contains no type II redundancy, apply the reductions of [7] to obtain generalized prefixes, and enter these generalized prefixes into a TCAM in decreasing order of length, then lookups yield the same next hops as when we load the TCAM with the non-reduced prefix set in length order.

4 Issues Related to [8, 9]

The issues with the mask extension method of Liu [7] may be resolved by either using an unnatural definition for the length of a generalized prefix (i.e., length equals that of the equal-length simple prefixes that were input to the logic minimizer) or by eliminating type II redundancies prior to logic minimization and defining length as in the definition of $|G|$ provided in Section 3. These resolution methods do not, however, extend to the aggregation and prefix expansion techniques proposed in [8, 9] to reduce the number of rules to be stored in the TCAM.

4.1 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 3 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. The rule set is devoid of type I and type II redundancies and 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 and with a length assignment of 3, packets to 10* get sent to A rather than to B!

4.2 Prefix Expansion

[8, 9] propose using prefix expansion within an aggregated cluster to improve the 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 [8, 9]. In the example cluster {1*,101*} of Section 4.1, 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 4.1 cannot be made to work the same as the original rule set.

5 PETCAM

Our power-efficient TCAM, PETCAM, employs the following construction steps:

Step 1: Transform the given routing table to an equivalent optimal routing table using the dynamic programming algorithm of [10].

Step 2: Use mask extension as in [7] to reduce the number of prefixes in the optimal routing table obtained in Step 1 even further. This is possible as the optimal routing table is limited to be comprised of simple prefixes alone whereas mask extension results in generalized prefixes.

Step 3: Map the reduced set of generalized prefixes constructed in Step 2 to a 2-level TCAM augmented with a wide SRAM by extending the suffix node method developed in [3].

Since the dynamic programming algorithm of [10] transforms a set of prefix rules into a provably optimal equivalent set of prefix rules, the transformed set is guaranteed to be free of type I and type II redundancies. Hence, the generalized prefixes that result from the mask extension done in step 2 correctly classify packets when these prefixes are entered into a TCAM in decreasing order of length ($|G|$). For step 3, we need to adapt the suffix-node method of [3] so as to accommodate generalized prefixes rather than simple prefixes. For this adaptation, 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 suffix-node method of [3].

5.1 Suffix-Node Method of [3]

Lu and Sahni [3] propose the use of wide SRAMs in conjunction with TCAMs so as to reduce power consumption and increase effective TCAM capacity. Although Lu and Sahni [3] propose methods for both 1- and 2-level TCAMs, we review only the 1-level method here and adapt this to generalized prefixes. A similar adaptation may be done for the 2-level methods of [3].

Lu and Sahni [3] make the observation that the simple TCAM organization of Figure 2 does not make effective use of modern wide access SRAMs. 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. A larger number of bits may be fetched per cycle by employing multiple SRAMs that may be simultaneously accessed. 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, Lu and Sahni [3] propose packing the suffixes of several router-table prefixes that are in the same subtree of the binary trie for the router-table prefixes 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 4 gives the structure of the suffix node of [3]. 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 in [3], 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 ancestor with a prefix in case the root of the subtree does not store a prefix. To optimize SRAM 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.

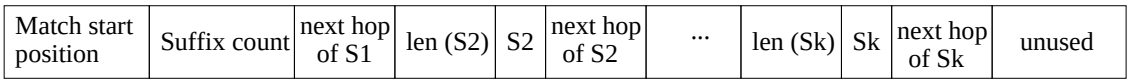


Figure 4: Suffix node of [3] with a 5-bit match start position field and an optimized representation of the first suffix.

Figure 5 shows the binary trie for the prefixes of Figure 1 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 upto 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 5, 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 5. When carving is done at a node R whose subtree doesn't contain a matching prefix for every destination address that begins with $Q(R)$, we add a *covering prefix* into the suffix node for this carving. The covering prefix for node R, which is stored as a suffix whose length is 0 is the prefix in the nearest binary trie ancestor of R. Assuming that each router table contains the default prefix *, each node of the binary trie has a well defined covering prefix. The covering prefix for node B of the binary trie of Figure 5 is P3 with a next hop of 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.

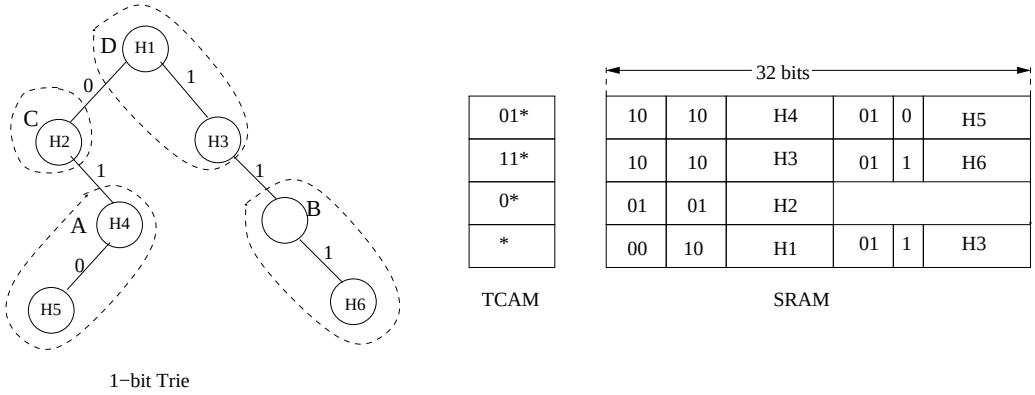


Figure 5: Suffix node example

The $Q(R)$s and associated suffix nodes are assigned, respectively, to TCAM and SRAM words in descending order of length [3].

5.2 Our Suffix-Node Structure

The suffix node method of [3] cannot be used as is for PETCAMs because, in a PETCAM, we store generalized prefixes rather than simple prefixes. Specifically, 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 [3] except for the addition of a type field (Figure 6).

A type II suffix node (Figure 7) 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 6. These

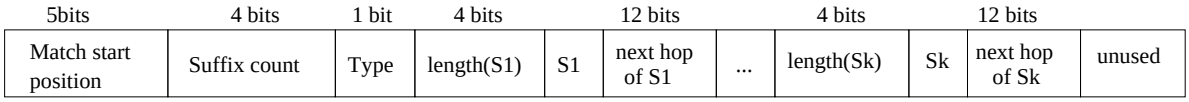


Figure 6: Type I suffix node

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.

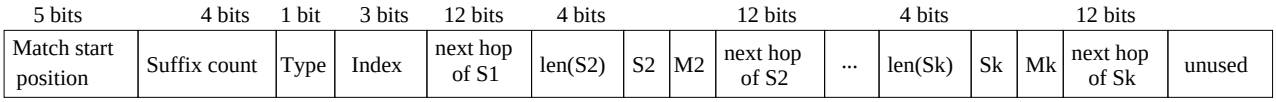


Figure 7: Type II suffix node

5.3 Normalized 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]. Figure 8 shows an example router table following steps 1 and 2 of our PETCAM construction algorithm. Figure 9 shows the corresponding ternary trie.

	Address	Next Hop
1	x0	H1
2	00x0	H2
3	00x1	H3
4	1100	H4
5	11x1	H5

Figure 8: Router table with generalized prefixes

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 9, 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 10. One may verify that the ternary tries of Figures 9 and 10 are equivalent in that both route all packets to the same next hop.

[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 an x.

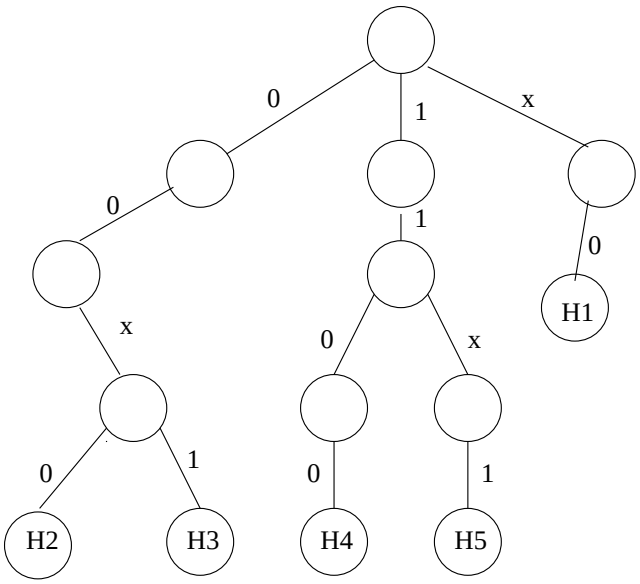


Figure 9: Ternary trie for the router-table of Figure 8

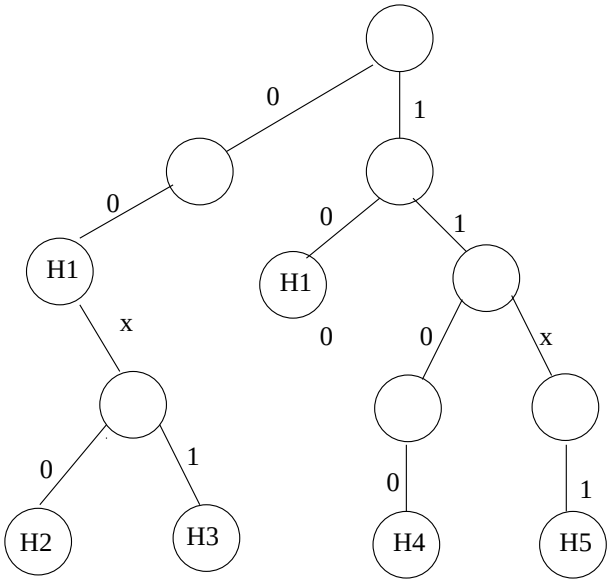


Figure 10: Ternary trie following the merging of the x-child of the root of Figure 9

The trie of Figure 10 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 11.

Figure 12 gives 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 y. 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

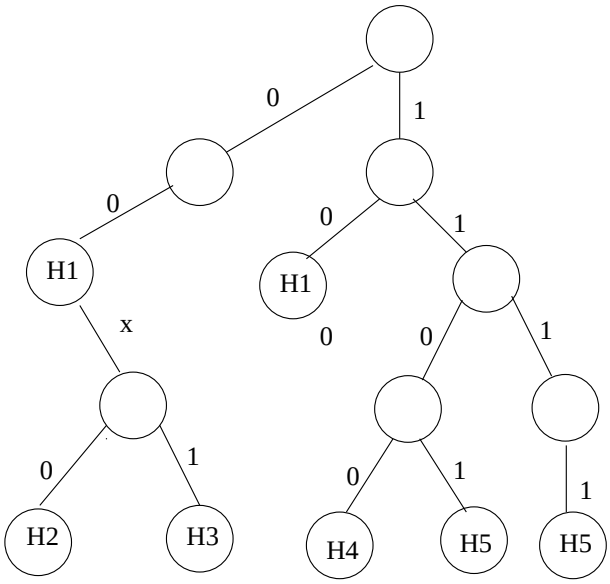


Figure 11: Normalized ternary trie following merging of the $Q(R) = 11x$ subtree of Figure 10

postorder traversal. Algorithm *merge* is specified in Figure 13.

```
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]);
          root.child[2] = NULL;
       }
    }
    normalize(root.child[0]);
    normalize(root.child[1]);
    normalize(root.child[2]);
}
```

Figure 12: Algorithm to normalize a ternary trie

In algorithm *merge*, *oChild* and *xChild* are children of *parent*. *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.

5.4 Our Carving Heuristic

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. To carve the normalized ternary trie into suffix nodes, we first compute

```
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 13: Algorithm to merge an x-subtree

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 subtree rooted at y. Each such suffix is obtained by removing $Q(p)$ from the $y.numP$ prefixes in the subtree 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 14 gives the $numP$, $xNumP$, and $size$ values for each of the nodes of the normalized ternary trie of Figure 11, where $size$ here includes only a nexthop and suffix bits for simplicity.

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 15, which differs from that of [3] in the manner in which $size$ is computed. Although the visit function, as stated in Figure 15, does not make explicit use of $numP$ and $xNumP$, these values are useful in the computation of $size$. Note that in Figure 15, 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.

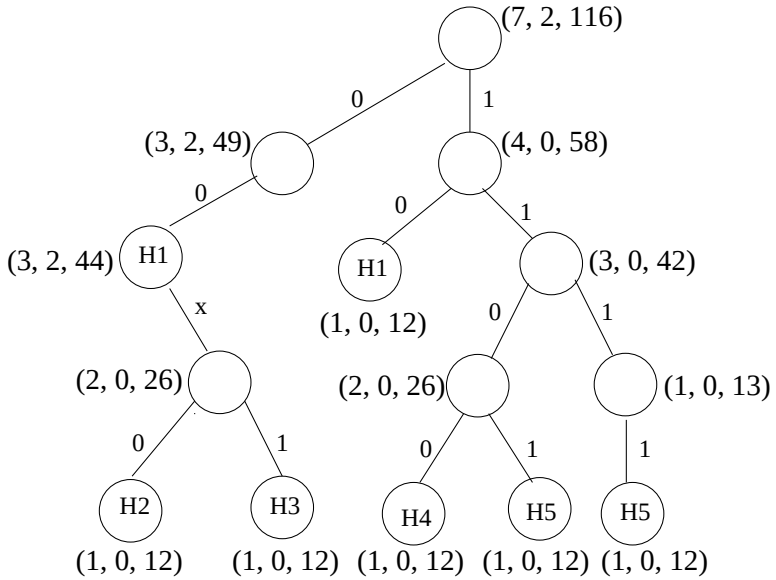


Figure 14: ($numP$, $xNumP$, $size$) for nodes of normalized ternary of Figure 11. Nodes that require a covering prefix are labeled *

```
Algorithm visit(y)

if (y.size < w) return;
if (y.size == w) {carve(y); return;}
// y.size > w
if (y has a single child z) {carve(z); return;}
       // z could be 0-, 1- or x-child

// y has both a 0-child u and a 1-child v
if (u.size <= v.size) {
  carve (v);
  recompute y.size;
  if (y.size < w) return;
  if (y.size == w) carve(y);
  else carve(u);
}
else // u.size > v.size
this is symmetric to the case u.size <= v.size
```

Figure 15: Visit function for subtree carving heuristic [3]

5.5 PETCAM Updates

PETCAM supports batch updates rather than incremental updates. To support batch updates, we use two copies of the TCAM-SRAM lookup subsystems as shown in Figure 16. At any given time, one copy of the TCAM-SRAM subsystem is used for lookup and the other is used to prepare an updated version of the lookup table. In a batch update, the control plane processes all updates received. This is done using a control plane representation (e.g., a trie) of the routing table. With some frequency, the PETCAM construction algorithm is run, creating an updated

TCAM-SRAM representation is the subsystem not currently used for lookup. When construction is complete, lookup is switched to the new subsystem. So, lookups have minimal interruption; the interruption time being that to switch from one subsystem to another. For this strategy to work, the delay between successive rebuilds must at least equal the time to run the PETCAM construction algorithm. Hence, it is important to have an efficient PETCAM construction algorithm.

The same strategy may be used for batch updates using the TCAM schemes of [3, 7, 8, 9]. We note that none of these schemes provide efficient support for incremental updates.

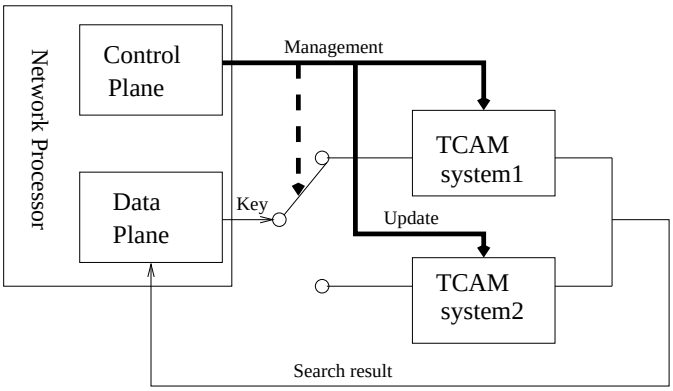


Figure 16: Router architecture using PETCAM.

6 Experimental Results

We programmed our PETCAM strategy in C++ and compared its performance with the power reduction schemes of [7, 3, 8, 9]. The comparison was done using the IPv4 router tables AS1221, AS4637, AS6447, and AS65000, which were obtained from [5] and rrc00, which was obtained from [6]. Data sets AS65000 and rrc00 are from May 2008, AS6447 is from July 2008, and the remaining data sets are earlier than 2008 and were used in [3], for example. Our experiments aim to measure the relative effectiveness of the scheme of Liu [7] (type I redundancy removal followed by mask extension) and the PETCAM scheme (dynamic programming optimization followed by mask extension) to compact the router table as well the overall relative performance of PETCAM, EaseCam, and the method of Lu and Sahni [3] 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.

6.1 Compaction Efficiency

The compaction efficiency is measured by the number of prefixes following the compaction steps. Figure 17 gives the number of prefixes in each of our data sets as well as the number of prefixes following each of steps 1 and 2 of the PETCAM strategy. The dynamic programming algorithm of [10] reduces the number of prefixes in the data sets by between 45% and 79%. Another approximately 5% reduction is achieved when mask extension is employed

on the optimal prefixes produced by the algorithm of [10]. So, PETCAM reduces the number of prefixes by about 50% to 84%.

DataSet	initial # of prefixes	# after Step 1	Reduction (%)	# after Steps 1 and 2	Total reduction (%)
AS1221	281516	153885	45.34	135879	51.73
AS4637	210119	43368	79.36	32562	84.50
AS6447	275509	149117	45.88	137151	50.22
AS65000	259026	81341	68.60	66808	74.21
rrc00	266185	92239	65.35	83146	68.76

Figure 17: Number of router table prefixes in PETCAM

Figure 18 gives the number of prefixes following the removal of type I redundant prefixes as well as following mask extension as proposed in [7] and Figure 19 gives these numbers after the removal of type I and type II redundancies followed by mask extension. We do not report the results of compaction using the enhancements to Liu's [7] compaction methods proposed in [8, 9], because, as noted in Section 4, these enhancements do not guarantee compacted prefix sets equivalent to the input prefix set. For each of our data sets, the method of [7] achieves less compaction than what is proposed for PETCAM. The bar chart in Figure 20 shows the relative efficiency of the three schemes with respect to reducing the number of prefixes in a router table.

DataSet	initial # of prefixes	# after type I	Reduction (%)	# after mask extension	Total reduction (%)
AS1221	281516	210582	25.20	146101	48.10
AS4637	210119	92099	56.17	40374	80.79
AS6447	275509	231193	16.09	162575	40.99
AS65000	259026	152267	41.22	80441	68.94
rrc00	266185	173030	35.0	105534	60.35

Figure 18: Number of router table prefixes in [7]

DataSet	initial # of prefixes	# after type I and II	Reduction (%)	# after mask extension	Total reduction (%)
AS1221	281516	209553	25.56	145467	48.33
AS4637	210119	92066	56.18	40386	80.78
AS6447	275509	227989	17.25	159909	41.96
AS65000	259026	151590	41.48	80076	69.09
rrc00	266185	171754	35.48	104827	60.62

Figure 19: Number of router table prefixes when type II redundancies are eliminated

The input for mask extension is comprised of sets of same hop prefixes that have the same length. Logic minimization is performed on each of these sets. The time for logic minimization is substantial (see Section 6.4) and critically dependent on the size of the input set. Figure 21 gives the maximum size of a set input to Espresso.

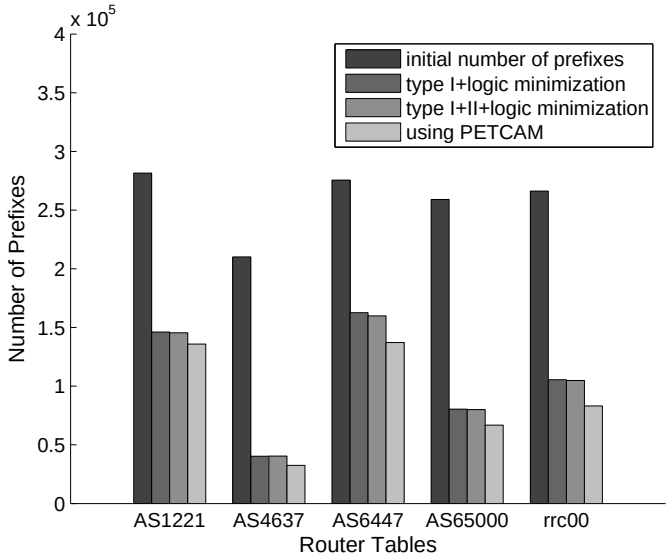


Figure 20: Relative efficiency for table compaction

Table	Original	PETCAM	Liu [7]
as1221	45285	13771	25953
as4637	111921	13188	40004
as6447	18808	7818	15110
as65000	94297	14693	45602
rrc00	62744	11669	34129

Figure 21: Maximum number of prefixes, having the same length and next hop prior to logic minimization/mask extension

6.2 Comparison With EaseCam

Although the modifications to Liu's [7] compaction strategy suggested in [8, 9] are faulty, the two-level EaseCam architecture, which is a specialization of the bit-selection architecture proposed by Zane et al. [19], 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. Figure 22 gives the number of TCAM bits required by EaseCam to store each of our sample router tables following compaction using the strategies of [7], [7] together with type II redundancy removal, and PETCAM.

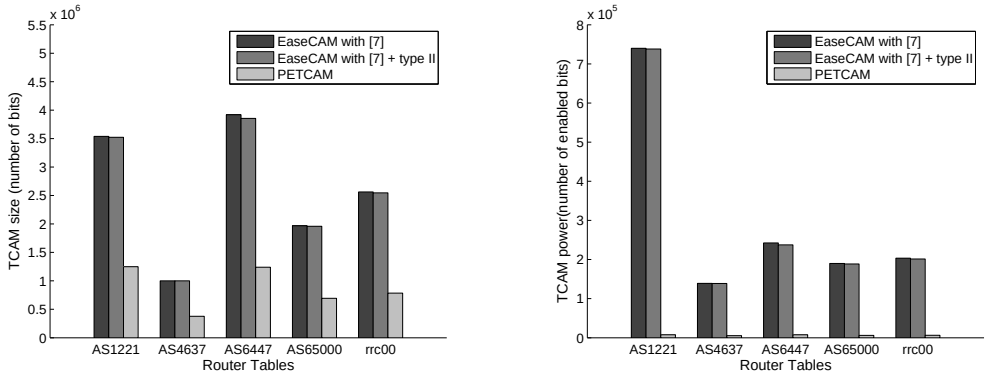
To generate the numbers for PETCAM, we employ steps 1 and 2 of the PETCAM scheme and store the resulting generalized prefixes in a 2-level TCAM system using the M-12Wb layout of [3]. We set the word-size of the second level TCAM (also known as data-DTCAM or DTCAM) to 32 bits for IPv4 prefixes. For the first level TCAM (index-TCAM or ITCAM), we need a word size of 24 bits based on (1) the bit allocation scheme to suffix nodes in Figure 7, (2) suffix node size of 144 bits and (3) our choice of DTCAM bucket size of 128 prefixes for the experiments. For example, with the given bit allocation and the suffix node size, the minimum height of

a carved subtree (carving done using the algorithm in Figure 15) is 2, corresponding to the case when all the 7 nodes of the subtree store a prefix. Thus prefixes stored in a DTCAM are of length 29 or less. Similarly, with a DTCAM bucket of size 128 prefixes, the carved subtree is of height at least 6 ($\log_2(128) - 1$, assuming a full binary tree with each node storing a prefix). So, the length of a prefix in ITCAM is at most 29-7 = 22, and in fact some further reduction is possible if we consider the wide SRAM being used with the ITCAM. However, for ease of configuration, we choose 32 bits for DTCAM and 24 bits for ITCAM word size. It is easy to see that this configuration can support DTCAM buckets of size >= 32 entries when the SRAM word size is 144 bits. When the DTCAM bucket size is < 32, a larger word size for the ITCAM is required.

Figure 22 also gives the size, $maxP$, of the largest partition that is activated in a lookup. Recall that the power needed for a lookup is proportional to the size of the activated partition. On our data sets, PETCAM requires less than half the TCAM memory required by EaseCam and the power requirement of EaseCam is between 26 and 97 times that of PETCAM. Figure 23 presents a bar chart for the comparitive space and power consumption by EaseCAM and PETCAM.

DataSet	EaseCam with [7]		EaseCam with [7]+typeII		PETCAM	
	#bits	$maxP$	#bits	$maxP$	#bits	$maxP$
AS1221	3538608	739968	3523656	738120	1248640	7552
AS4637	1000080	138936	1000080	138912	378056	5320
AS6447	3920112	242400	3854808	237360	1239608	7624
AS65000	1968408	189888	1959000	188736	694240	6112
rrc00	2563176	203448	2545920	201384	784592	6352

Figure 22: Total number of bits and maximum partition size using EaseCam



(a) TCAM size

(b) TCAM power

Figure 23: Comparison of TCAM space and power requirement between EaseCAM and PETCAM

6.3 Comparison With [3]

Figure 24 gives the $maxP$ values for the power-reduction architecture of Lu and Sahni [3] when applied to the original prefix set as is done in [3] and when applied to a compacted prefix set. In Figure 24, we use the 1-12Wc scheme of [3], which is recommended in [3] for power optimization. The size of a DTCAM bucket is set to 128 prefixes. In Figures 24 and 25, all columns use the architecture of [3]. The column labeled No Compaction [3] uses the original prefix set, that labeled [7] uses the compacted prefix set resulting from type I redundancy removal followed by logic minimization as is done in [7], the next column applies types I and II redundancy removal before logic minimization, and the column labeled PETCAM uses the 1-12Wc scheme to store the set of generalized prefixes obtained after applying steps 1 and 2 of the PETCAM scheme to the initial prefix set. As can be seen, PETCAM provides power reduction relative to the scheme of [3]. This reduction ranges from 18% to 25%.

DataSet	No Compaction [3]	[7]	[7]+type II	PETCAM
AS1221	200	180	180	164
AS4637	183	152	152	139
AS6447	196	185	185	164
AS65000	198	171	170	148
rrc00	198	172	172	152

Figure 24: Maximum partition size using 1-12Wc of [3]

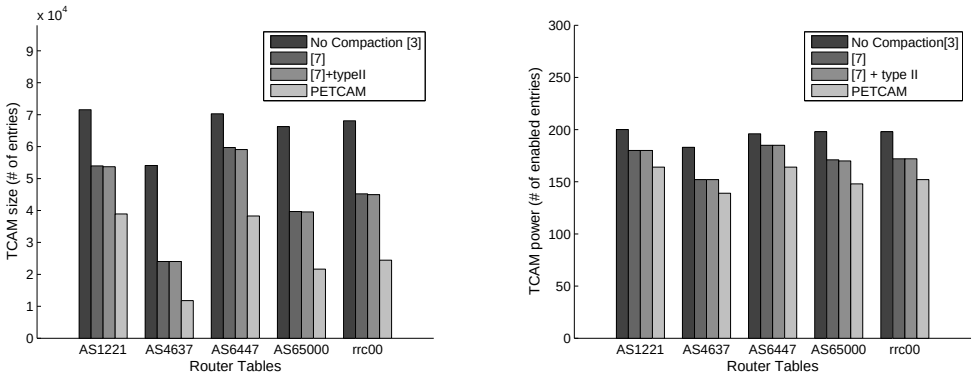
Figure 25 gives the total TCAM memory needed by the M-12Wb scheme of [3], which is the scheme recommended in [3] for TCAM memory optimization. The numbers in the column labeled PETCAM are obtained by applying the steps 1 and 2 of the PETCAM scheme to reduce the prefix set and then using the step 3 to map the resulting generalized prefixes to a 2-level TCAM system. We use the carving heuristic in Figure 15 to create the suffix nodes and then use the M1-2Wb layout of [3] to fill the first and second level TCAMs. The size of a DTCAM bucket is set to 128 prefixes. PETCAM requires between 22% and 54% as much TCAM memory as required by the architecture of [3] beginning with the original prefix set. Figure 26 shows the data of Figures 24 and 25 as bar charts.

DataSet	No Compaction [3]	[7]	[7]+type II	PETCAM
AS1221	71564	53964	53705	38913
AS4637	54076	24027	24027	11782
AS6447	70271	59728	59089	38273
AS65000	66285	39691	39560	21636
rrc00	68084	45216	44964	24449

Figure 25: Total TCAM memory using M1-2Wb of [3]

6.4 PETCAMLite

Since Step 2 (mask extension) of the compaction process for PETCAM is quite time consuming, we investigate a light version, PETCAMLite, of PETCAM in which Step 2 is omitted. Our experiments indicate that PETCAMLite



(a) TCAM size

(b) TCAM power

Figure 26: Comparison of TCAM space and power requirement

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. Figure 27 gives the CPU time on a Sun4u Sparc SunOS 5.8 machine for executing steps 1 and 2. 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 6 seconds for PETCAMLite.

DataSet	Time for Step 1 (seconds)	Time for Step 2 (seconds)
AS1221	5.38	642.83
AS4637	3.7	296.62
AS6447	5.14	347.25
AS65000	4.57	600.55
rrc00	4.78	407.05

Figure 27: Execution time

7 Conclusion

We have pointed out some of the shortcomings of the power reduction methods for TCAM lookup tables proposed in [7, 8, 9]. By starting with an optimal prefix set for the given router table prefix set, we can achieve much better power reduction and TCAM memory requirement than when we use the compaction schemes suggested in [7, 8, 9]. This is true regardless of whether we use the EaseCam [8, 9] architecture or the architecture of [3]. For EaseCam, worst case power is reduced between 96% and 98% while TCAM memory is reduced between 62% and 69%. The power and memory reduction relative to the architecture of [3] is 16% to 25% and 45% to 78%. 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 2

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.

References

[1] M. Akhbarizadeh, M. Nourani, R. Panigrahy and S. Sharma, A TCAM-based parallel architecture for high-speed packet forwarding, *IEEE Trans. on Computers*, 56, 1, 2007, 58-2007.

[2] H. Lu, Improved Trie Partitioning for Cooler TCAMs, *ACST*, 2004.

[3] W. Lu and S. Sahni, Low Power TCAMs For Very Large Forwarding Tables, *Proceedings of INFOCOM*, 2008.

[4] W. Lu and S. Sahni, Succinct representation of static packet classifiers, *International Conference on Computer Networking*, 2007.

[5] *http://bgp.potaroo.net*, 2007.

[6] *http://www.ripe.net/projects/ris/rawdata.html*, 2008.

[7] H. Liu, Routing Table Compaction in Ternary-CAM, *IEEE Micro*, 22, 3, 2002.

[8] V.C. Ravikumar, R. N. Mahapatra, and L. N. Bhuyan, EaseCAM: An Energy And Storage Efficient TCAM-Based Router Architecture for IP Lookup, *IEEE Transactions on Computers*, 54, 5, May 2005, 521-533.

[9] V.C. Ravikumar, R. N. Mahapatra, and L. N. Bhuyan, TCAM architecture for IP lookup using prefix properties, *IEEE Micro*, 24, 2, March 2004, 60-69.

[10] R. Daves, C. King, S. Venkatachary, and B.Zill, Constructing Optimal IP Routing Tables, *Proceedings of INFOCOM*, 1999.

[11] M. Ruiz-Sanchez, E. Biersack, and W. Dabbous, Survey and taxonomy of IP address lookup algorithms, *IEEE Network*, 2001, 8-23.

[12] S. Sahni, K. Kim, and H. Lu, Data structures for one-dimensional packet classification using most-specific-rule matching, *International Journal on Foundations of Computer Science*, 14, 3, 2003, 337-358.

[13] C. A. Zukowski, and S. Wang, Use of Selective Precharge for Low-Power Content-Addressable Memories, *IEEE International Symposium on Circuits and Systems*, 1997.

[14] N. Mohan, and M. Sachdev, Low Power Dual Matchline Ternary Content Addressable Memory, *IEEE International Symposium on Circuits and Systems*, 2004.

[15] H. Miyatake, M. Tanaka, and Y.Mori, A design for high-speed low-power CMOS fully parallel content addressable memory macros, *IEEE Journal of Solid State Circuits*, 36, 6, June 2001, 956-968.

[16] C.-S. Lin, J.-C. Chang, and B.-D Liu, A low-power pre-computation based fully parallel content addressable memory, *IEEE Journal of Solid State Circuits*, 38, 4, April 2003, 654-662.

[17] Z. Wang, H. Che, M. Kumar, and S.K. Das, CoPTUA: Consistent Policy Table Update Algorithm for TCAM without Locking, *IEEE Transactions on Computers*, 53, 12, December 2004, 1602-1614.

[18] M. Wang, S. Deering, T. Hain, and L. Dunn, Non-random Generator for IPv6 Tables, *12th Annual IEEE Symposium on High Performance Interconnects*, 2004.

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

LIST OF ABBREVIATIONS

Acronym	Meaning
DLFS_PLO	A memory management scheme
DUO	Dual TCAM architecture for packet forwarding
DUOS	A simple version of DUO
ILSRAM	Index SRAM for an ILTCAM
ILTCAM	Index TCAM for an LTCAM
ISRAM	Interion SRAM
ITCAM	Interior TCAM
LSRAM	Leaf SRAM
LTACM	Leaf TCAM
PC-DUO	Dual TCAM architecture for packet classification
PC-DUOS	A simple version of PC-DUO
PC-DUO+	An enhancement of PC-DUO
PC-DUO+W	A wide SRAM version of PC-DUO+
PC-TRIO	Triple TCAM architecture for packet classification
SRAM	Static random access memory
STCAM	A simple TCAM architecture for packet classification
TCAM	Ternary content addressable memory
TCP	Transmission control protocol