TOWARDS MORE POWER EFFICIENT IP LOOKUP ENGINES

A Thesis

by

SERAJ AHMAD

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

December 2005

Major Subject: Computer Engineering

TOWARDS MORE POWER EFFICIENT IP LOOKUP ENGINES

A Thesis

by

SERAJ AHMAD

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Approved by:

Chair of Committee,	Rabi N. Mahapatra
Committee Members,	Sunil P. Khatri
	Duncan M. Hank Walker
Head of Department,	Valerie E. Taylor

December 2005

Major Subject: Computer Engineering

ABSTRACT

Towards More Power Efficient IP Lookup Engines. (December 2005)

Seraj Ahmad, Bachelor of Technology, Indian Institute of Technology, Guwahati, India

Chair of Advisory Committee: Dr. Rabi N. Mahapatra

The IP lookup in internet routers requires implementation of the longest prefix match algorithm. The software or hardware implementations of routing trie based approaches require several memory accesses in order to perform a single memory lookup, which limits the throughput considerably. On the other hand, IP lookup throughput requirements have been continuously increasing. This has led to ternary content addressable memory(TCAM) based IP lookup engines which can perform a single lookup every cycle. TCAM lookup engines are very power hungry due to the large number of entries which need to be simultaneously searched. This has led to two disparate streams of research into power reduction techniques. The first research stream focuses on the routing table compaction using logic minimization techniques. The second stream focuses on routing table partitioning. This work proposes to bridge the gap by employing strategies to combine these two leading state of the art schemes. The existing partitioning algorithms are generally employed on a binary routing trie precluding their application to a compacted routing table. The proposed scheme employs a ternary routing trie to facilitate the representation of the minimized routing table in combination with the ternary trie partitioning algorithm. The combined scheme offers up to 50% reduction in silicon area while maintaining the power economy of the partitioning scheme.

To My Loving Mother

ACKNOWLEDGMENTS

I would like to thank Dr. Rabi Mahapatra for all the valuable guidance and numerous help in understanding and carrying out academic research. I am highly grateful to him for allowing me to take up this interesting piece of work.

I would like to thank Dr. Sunil P. Khatri and Dr. Duncan M. Hank Walker for serving on my committee. I never took any course under Dr. Walker, but I enjoyed several elevator and corridor conversation with him and I wished I would have had the opportunity for classroom interaction.

In addition, I would like to thank Dr. Sunil Khatri for offering such enjoyable courses and imparting the best of his knowledge and experiences encompassing the academic as well professional arena. I am really grateful to him for numerous personal advise that in some way helped to shape my overall attitude.

I would like to thank V.C. Ravikumar, Dr. Frank Vahid and Roman Lysecky at UCR for providing the routing table traces used in the experimental results section.

I enjoyed several heated academic conversations with Vivek, Anuj, Praveen and Rupak. I thank them for all their helpful tips and troubleshooting conversations.

I would also like to thank one of my college friends, Amnaya Awashthi, who taught me all the skills I needed to survive in College Station, without whose help I could never be the same person I am.

I am extremely grateful to all my friends at Texas A&M who made my graduate years memorable and enjoyable at all times. In particular, I would like to express my gratitude to Kausik, Rudram and Anshuman. They kept me laughing under the toughest and most stressful conditions.

TABLE OF CONTENTS

CHAPTER		Page
I	INTRODUCTION	1
II	EXISTING WORK ON ROUTING TABLE COMPACTION	6
	A. Logic Minimization Problem	6
	B. Logic Minimization Approaches	7
	C. Routing Table Compaction Approaches	7
III	EXISTING WORK ON TCAM PARTITIONING	9
	A. Bit Selection	9
	1. Hashing	9
	B. Trie Based Partitioning	10
	1. Subtree Split	10
	2. PostOrder Split	11
IV	ARCHITECTURE OF TCAM BASED IP LOOKUP ENGINES	13
	A. TCAM Cell	13
	B. TCAM Memory Architecture	14
	C. IP Lookup Engine Architecture for Bit Selection	16
	D. IP Lookup Engine Architecture for Trie Based Partitioning	17
V	ROUTING TRIE FOR PARTITIONING	19
	A. Binary Routing Trie	19
	B. Ternay Routing Trie	21
	C. Routing Trie Extension to Perform Partitioning	22
	D. Path Count Properties	23
VI	PARTITIONING OF MINIMIZED ROUTING TABLE	26
	A. Bucket Carving and UnTraversal	26
	B. Ternary SubTrie Partitioning	27
	C. Ternary PostOrder Partitioning	30
	1. CarveOneBucket Algorithm	30
	2. Ternary PostOrderSplit Algorithm	32
VII	EXPERIMENTAL RESULTS	34

CHAPTER Page

VIII CONCLUSION AND FUTURE WORK 45

REFERENCES 46

APPENDIX A 49

VITA 56

LIST OF TABLES

TABLE — Page

I — Encoding of Ternary Symbols in TCAM 14

II — Sample Routing Table 22

III — Sample Minimized Routing Table 23

IV — Comparison of Performance for `paix` Routing Table 50

V — Comparison of Performance for `aads` Routing Table 51

VI — Comparison of Performance for `pacbell` Routing Table 52

VII — Comparison of Performance for `maewest` Routing Table 53

VIII — Comparison of Performance for `att` Routing Table 54

IX — Comparison of Performance for `bbn` Routing Table 55

LIST OF FIGURES

FIGURE Page

1 A NOR-based TCAM Cell 15

2 TCAM Memory Architecture for Lookup 16

3 IP Lookup Architecture to Support Bit-Selection Based Partitioning 17

4 IP Lookup Architecture to Support Trie Based Partitioning 18

5 Binary Routing Trie for Sample Routing Table 20

6 Ternary Routing Trie for Sample Routing Table 25

7 Comparison of Index TCAM Size for `SubTreeSplit` Algorithm w.r.t. Bucket Sizes 38

8 Comparison of Index TCAM Size for `PostOrderSplit` Algorithm w.r.t. Bucket Sizes 39

9 Comparison of Power for `SubTreeSplit` Algorithm w.r.t. Bucket Sizes 40

10 Comparison of Power for `PostOrderSplit` Algorithm w.r.t. Bucket Sizes 41

11 Comparison of Area for `SubTreeSplit` Algorithm w.r.t. Bucket Sizes 42

12 Comparison of Area for `PostOrderSplit` Algorithm w.r.t. Bucket Sizes 43

13 Case Study - `att` Routing Table 44

LIST OF ALGORITHMS

ALGORITHM — Page

1 Traverses the routing trie from node v up to the root 27

2 The algorithm to carve subtrie rooted at node v 28

3 The partitioning algorithm based on ternary subtrie splitting 29

4 The algorithm used by post-order split to carve exactly one subtrie from node v 31

5 The partitioning algorithm based on ternary postorder splitting . . . 32

CHAPTER I

INTRODUCTION

The Internet is a packet switched network consisting of a number of routers and hosts interconnected together by communication links. Hosts reside at the boundary of the network and host-to-host communication takes place via a number of routers using Internet Protocol (IP). The communication between two nodes is converted into a series of packets known as IP *datagrams.* Every datagram carries a 32-bit destination address to facilitate independent routing. The packet is forwarded to a node corresponding to the best-known path to the destination, which is called *nexthop.* The next hop is determined by an *IP lookup* operation performed on a *routing table.* IP lookup operation searches the most specific network hosting the specified destination in a list of variable length network identifiers known as *IP prefixes.* IP prefixes can be represented using a ternary string $P = P_l \underbrace{xx \cdots x}_{32-l\, times}$ where P denotes a 32-bit prefix, x represent don't care symbol and l the prefix length. P_l represents the l most significants bits of the prefix. Only P_l is compared against the specified destination address to decide a *match.* Since the routing table contains several overlapping prefixes, the destination address can match multiple IP prefixes. The IP lookup operation searches the routing table to find the *longest prefix* matching the destination.

The longest match semantics requires pattern matching as well length determination, which makes a practical implementation of IP lookup operation harder, especially in high-end routers. The difficulty is attributed to super-linear growth of the size of the routing table as well as increasing gap between silicon and optic fiber speed, with the latter growing at an exponential rate [1].

The journal model is *IEEE Transactions on Automatic Control.*

The most obvious way to represent a routing table is to construct a binary retrieval tree known as *Binary Routing Trie*. The binary routing trie stores all the prefixes of length n at the nth level. The IP lookup starts at the root node and proceeds left(right) for a 0(1) in the destination IP address starting from the most significant bit. The binary routing trie is not a very efficient representation as it can have long paths containing single leaf nodes. A technique of path compression in binary trie was suggested in [2] and adopted for performing IP lookup in [3]. Another similar technique proposed in [4], focuses on level compression instead of path compression. A number of other techniques focus on combination of level compression and path compression along with strategies to improve the lookup performance such as controlled prefix expansion [5].

A survey of software and hardware based methods for IP lookup can be found in [1] and [6]. The lookup algorithms designed for conventional memory to solve the *longest prefix match* problem require several memory accesses to retrieve the nexthop. This can quickly become a bottleneck for high-speed backbone routers operating at gigabit speed, and with large routing tables containing up to a million entries. Therefore hardware based solutions are needed to support such high performance lookup operation. A technique proposed in [1] use custom hardware to expand all the prefixes into 32 bit entries. The expanded entries are then used as memory location to store the nexthop corresponding to that prefix. This transforms the IP lookup operation into a single memory access. However this approach may not be scalable for IPv6 addresses. Francis et. al. investigate techniques for O(1) IP lookup using *binary content addressable memory (BCAM)* and *ternary content addressable memories(TCAM)* [7]. BCAMs allow storage of 0 and 1 in each memory cell and can perform only fixed length match. Hence multiple BCAMs are required to search variable length prefixes in a single cycle. This can lead to significant *under utilization*

of the available memory. TCAMs are similar to BCAMs but allow storage of 0, 1, and x states. The state x is treated as *don't care* and ignored during a matching operation. Thus TCAMs allow the storage of variable length prefixes in a single unit giving storage economy. Also TCAMs offer easier management and update of the routing tables. Despite its advantages, TCAM based lookup solutions remained unpopular due to its high cost, low capacity and poor performance. However, recent advances in manufacturing and interconnection technology allows fabrication of high capacity, high performance and low cost TCAM units matching the requirement of today's backbone routers. For example, the latest available TCAM in the market operates at 100 million searches per second and offer capacities up to 16MB [8].

TCAM based fast lookup seems promising but it is not without its disadvantages. TCAMs consume a lot of power in normal operating conditions which is proportional to number of TCAM entries enabled for searching. A typical high-end TCAM memory today may consume up to 15 Watts. Also, please note that the IP lookup engines may require several TCAMs in order to support future growth of the routing table. Research efforts to reduce TCAM power consumption can be divided into two categories. The first approach attempts to reduce power consumption by partitioning the entire TCAM memory into a set of TCAM pages and then finding a suitable hashing algorithm to map each entry into a set of target pages [9], [10] and [11]. During searching only target pages are enabled. This reduces power consumption by a ratio of $\frac{p}{n}$, where p and n are average number of target pages and total pages respectively. The second approach reduces the power consumption by compacting routing table entries using logic minimization techniques as discussed in [12] and [13], [14]. IP prefixes contain the symbol x only at the end while in minimized IP prefixes it can occur at any position. Since TCAM allows storage of x at any bit position, routing semantics can be guaranteed even with minimized routing table. Here, the reduction

in power consumption is dependent on the compaction ratio achieved by the logic minimization technique applied. Experimental results show that logic minimization can reduce power consumption by up to 60% [12], [14]. The logic minimization based power reduction can be applied to existing TCAMs while partitioning based architecture requires hardware modification to TCAM architecture to support paging.

The performance of a partitioning scheme is proposed to be evaluated based on three different metric. The first metric is the worst-case size of the partition which is directly responsible for the worst-case power consumed by the lookup engine. The second metric is the number of partitions needed to accommodate the routing table and may contribute to the worst-case power. These two metrics are generally considered by the partitioning scheme mentioned earlier. We introduce a third metric, the amount of silicon area required to implement the routing table. This metric is important in order to support large routing tables expected in future backbone routers in storage efficient manner. All the three metrics must be simultaneously evaluated in order to assess the performance of a partitioning scheme.

This work proposes a hybrid power reduction approach which takes advantage of routing table compaction as well as routing table partitioning scheme to achieve high performance and economy. The proposed approach which relies on partitioning of a compacted routing table attempts to maximize the third metric while maintaining the first two metrics. The experimental results obtained on a six different routing table traces suggest that proposed scheme reduces the silicon area required to implement the partitioned routing table by up to 60%. The results also suggest the reduction of number of routing table partitions as compared to partitioning approach presented in [9]

The rest of the paper is organized as follows. Chapter II discusses existing approaches on routing table compaction. Chapter III discusses related work on routing

table partitioning. Chapter IV discusses the IP lookup engine architecture based on TCAMs. Chapter V discusses a ternary routing trie with a brief introduction to 0-1 or binary trie. Chapter VI present the approach and algorithm on partitioning of the minimized routing table in detail. The experimental setup and results are presented in Chapter VII. The paper concludes in Chapter VIII, with highlights on important results and future research direction. The raw experimental data is included in the Appendix.

CHAPTER II

EXISTING WORK ON ROUTING TABLE COMPACTION

Logic Minimization techniques traditionally found its use in logic synthesis to reduce the number of logic gates required for a given circuit. The routing table compaction using logic minimization techniques was first proposed in [12]. The routing table compaction methodology described in [12] utilized Espresso-II to perform logic minimization. However, Espresso-II has been designed to handle logic synthesis problems and requires large computing and memory resources making the routing table compaction a little impractical in actual router implementation. For example *Espresso-II* logic minimization algorithm described in [15] takes about 109 seconds to minimize a routing table containing 11091 entries and supports 2 worst case updates per seconds on a 400 MHz ARM platform. Further, it requires 500 kilobytes data memory and 100 kilobytes of instruction memory.

The following subsection define the logic minimization problem and presents a brief survey of existing approaches for general purpose logic minimization as well as specific approaches for handing *routing table compaction*.

A. Logic Minimization Problem

A boolean function $\mathbb{F}(x_1, x_2, \ldots, x_n)$ can be specified by an *on-set* $F(x_1, x_2, \ldots, x_n)$ and a *dc-set* $D(x_1, x_2, \ldots, x_n)$. The on-set contains all the input combinations where the function $\mathbb{F}$ assumes a value of logic 1. The dc-set or don't care set contains all the input combinations for which the value of $\mathbb{F}$ is unspecified. Two level logic minimization involves finding a minimum covering set $\mathbb{G}$ for the specified function $\mathbb{F}$. The set $\mathbb{G}$ contains the sum-of-products(SOPs) representation of the input variables.

B. Logic Minimization Approaches

The first exact solution to logic minimization problem was given by Quine [16], [17] and McClusky [18]. The procedure first generates all the primes(SOPs which are not contained in another SOP). This is followed by finding a set containing the minimum set of primes which covers all the points in on-set F.

Studies have suggested that Quine-McClusky procedure and its variant are inefficient as they generate a huge number of primes(sometimes more than ten millions) in the first stage. Therefore many newer logic minimizers try to generate only those primes which are part of some minimal cover of $\mathbb{F}$ thereby pruning out a large number of primes. Since logic minimization problem is NP-Complete, these algorithm still requires a considerable amount of time to find an exact solution. A number of heuristics algorithm has been proposed which find a near optimal solution within an acceptable time-budget. The most notable of these are SPAM [19], PRESTO [20], MINI [21] and Espresso-II. Several other variants exists for these algorithm offering better performance, however Espresso-II is by far the most popular two-level logic minimizer.

C. Routing Table Compaction Approaches

To deal with resource-constrained applications, Espresso-II provides a *fast* options which uses a single-expand stage during the refinement of initial minimal cover. One study has suggested a distance-one merge (d_1-merge) heuristics to achieve acceptable level of compaction for on-chip applications in considerably lesser time [22]. The complexity of single-expand stage is $O(N^2)$ while the complexity of d_1-merge stage is $O(N \log N)$. Both of these heuristics achieve good speedup but they do not minimize the memory requirement.

Another logic minimizer ROCM studied in [13] also uses a single expand stage and effectively minimizes the required memory. Although the execution performance of ROCM is not better than Espresso-II, it offers good performance for incremental minimization needed by most on-chip logic minimization applications. *ROCM* takes 120 seconds to minimize the routing table containing 11091 entries but offers about 30 worst case updates per seconds. Still contrast this with over 100,000 routing table entries and peak rates of over 2000 worst-case updates per seconds required in current backbone routers.

[14] introduces a novel approximate minimization technique of the complexity O(N) based on a trie data structure called *m-Trie* (minimization trie). *m-Trie* is a ternary trie (3-ary tree) with several minimization constraint. Every product term describes a path in the *m-Trie*. The constraint on *m-Trie* cause the path being inserted to get merged with other paths already present in the *m-Trie*. Thus logic minimization is performed as a series of path insertion. The insertion procedure also embeds several other information such as path count and a merge map in order to allow efficient path deletion. Thus *m-Trie* provides fast minimization and efficient incremental updates using localized minimization technique. It has a very small code footprint of about 20 KB and can operate with a data memory budget as small as 16KB. Experimental results show that it can attain up to 25,000 updates per second.

CHAPTER III

EXISTING WORK ON TCAM PARTITIONING

The routing table partitioning schemes were initially proposed to distribute the load of IP lookup operations to several independent processors. Later, complex schemes based on trie partitioning [10], [9], [23], bit selection [9] and multi-level fixed-stride hashing [11] were proposed to reduce the power consumption in TCAM based IP lookup engines. The following subsection describe the various existing partitioning strategies for routing tables.

A. Bit Selection

1. Hashing

The routing table is partitioned according to a specially selected set of bits on the routing prefixes. The length of the routing prefixes are assumed to be from 8 to 24 bits. Thus a total of 16 bits are available for hashing. The number of available hash function utilizing k hashing bits is therefore nCk. The quality of partitioning for any hash function is dependent on the size of largest partition which corresponds to worst case power consumed. [9] proposes three heuristics to select the best hash function which keeps the power consumption below a specified power budget. The first heuristics involves selection of first k bits from 16 available bits starting from rightmost bit. In most practical cases, the scheme provides a satisfactory hash function. The second scheme involves a heuristics which selects the k bits, starting one at a time and minimizing the worst case partition size at each step. The partitioning strategy is similar to *Binate Select* heuristics extensively used in unate recursive pardigm based logic minimization algorithms [15]. The third techniques involves brute force

enumeration of all the possible hash function until the given power budget is satisfied.

The scheme requires a k bit multiplexor to support the partitioning scheme. Although the hardware overhead is quite small but the design suffers from significant TCAM capacity under-utilization due to uneven sized partitions produced by the hash function.

B. Trie Based Partitioning

The following subsections present the partitioning algorithms based on a trie data structure.

1. Subtree Split

The subtree split is proposed in [9] and uses 1-bit trie to achieve the routing table partitioning. The 1-bit trie also known as 0-1 trie or routing trie contains paths corresponding to each prefix in the routing table. The routing lookup scans the input IP address from left to right and traverses left edge for an '0' and right edge for a '1' until it reaches a leaf node. Each node in the trie maintains a count of the prefixes stored in the subtrie rooted at that node. The subtree split algorithm performs a post-order traversal of the trie, as soon as it encounters a node with a *count* value atleast $\lceil \frac{b}{2} \rceil$ and parent count greater than b, the entire subtrie rooted at that node is carved out and the count value of all the ancestors are decremented by the count of the carved subtrie. The prefix corresponding to root of the carved subtrie is added to the index TCAM. The entry in index TCAM points to the TCAM bucket which stores all the prefixes in the carved subtrie. Please note that the root of the carved subtrie may not be a prefix itself and can therefore lead to erroneous longest prefix match. In order to avoid erroneous matching behavior, the covering prefix of the root

of carved subtrie is also stored in the TCAM bucket. This modification preserves the longest prefix match semantics.

The subtree split algorithm generates the TCAM bucket sizes in the range $[\lceil \frac{b}{2} \rceil, b]$ excluding the last bucket. The last bucket size can be anything in the range $[1, b]$. The total number of buckets created by the algorithm is bounded by the interval $[\lceil \frac{N}{b} \rceil, \lceil \frac{2N}{b} \rceil]$. The size of index TCAM is equal to total number of TCAM buckets generated by the algorithm. The maximum number of entries searched in the partitions generated by subtree split algorithm is $K + \lceil \frac{2N}{K} \rceil + 1$, where K is the number of buckets available in the TCAM.

2. PostOrder Split

The postorder split algorithm is similar to subtree split algorithms as described in the previous subsection, except it tries to fill the bucket completely in order increase the TCAM bucket capacity utilization. The algorithms performs a post-order traversal in order to carve subtrie groups which collectively have a prefix count equal to the chosen bucket size b. The first step in the algorithm is carving out a node with a count value of less than or equal to b. Suppose the count of the caved subtrie is $b_1 (< b)$, another subtrie is searched starting from the first node in post-order traversal which is not in the carved subtrie. This time procedure tries to carve out a subtrie with a count less than or equal to $b - b_1$. The procedure continues until all the TCAM bucket is full. An entry in the index TCAM is added for every subtrie carved out by the procedure. The index TCAM entries corresponding to the subtries in the same collection, point to the same bucket. Thus a bucket in post-order split can contribute several entries in the index TCAM. In addition, a covering prefix is also added in the TCAM bucket for every carved subtrie. This is accounted for during the bucket filling so that buckets do not overflow.

The postorder split algorithm generates equal sized bucket except for the last bucket which has a size less than or equal to b. Thus the total number of buckets created by the algorithm is $\lceil \frac{N}{b} \rceil$. If the maximum prefix length encountered is W, the algorithm generates $W + 1$ index TCAM entries for each bucket in the worst-case. The maximum number of entries searched in the partitions generated by postorder split algorithm is $(W+1)K + \lceil \frac{N}{K} \rceil + W$, where K is the number of buckets available in the TCAM.

CHAPTER IV

ARCHITECTURE OF TCAM BASED IP LOOKUP ENGINES

The commercial off-the-shelf TCAM memories can be used as an IP lookup engine, however they are not power or storage efficient for the reasons described in Section I. Therefore a number of different power reduction technique have been proposed in literature, spawning a host of IP lookup engine architectures. All these architecture seems to share a common theme such as use of an intermediate stage of index TCAM, range comparator or simply a multiplexor to support partitioning. The second level TCAM is segmented to support smaller sized partition in order to reduce the overall power consumption. Some IP lookup engines employ an on-chip logic minimizer to perform routing table compaction. We describe the IP lookup engine architecture in this section needed to support the proposed partitioning scheme. The following section describe the anatomy of a TCAM cell and TCAM memory module in detail. This is followed by a description of different TCAM lookup architectures to support bit selection and trie based partitioning scheme.

A. TCAM Cell

A NOR based TCAM cell is shown in Figure 1. It uses two SRAM based storage cell to store states 0, 1 and x based on the encoding scheme given in Table I. Each TCAM cell is provided with four transistor switches to assist comparison.

These transistor switches prevent matchline from getting shorted to ground when a match occurs. For example a state 0 in TCAM cell will turn off the transistor $T3$. A search for 0 applied on search lines sl_0 and sl_1 will turn off transistor $T1$ blocking match line from getting shorted to ground. However a search for 1 will turn on the transistor $T1$ creating a path to ground through transistor $T4$. On the other hand

Table I: Encoding of Ternary Symbols in TCAM

d_1	d_0	T
0	0	x
0	1	0
1	0	1
1	1	x_{sr}

a state x in TCAM will turn off both $T3$ and $T4$ blocking all path to ground thus matching all search keys applied to TCAM cell giving a don't care match semantics needed to encode the *absence* of a literal.

B. TCAM Memory Architecture

A simplified TCAM architecture is shown in Figure 2. Here an array of TCAM cells are arranged to form a TCAM word. In order to perform word comparison, all cells belonging to a single word share a common match-line.

Since data being searched can match multiple words in TCAM due to variable length matching, all the matchlines are connected to a priority encoder. The priority encoder selects the word at the lowest address among all the matched words.

To initiate a search in TCAM, the *matchline* is pre-charged to a high level. The data to be searched is stored in search register and asserted. TCAM words which do not match the search data cause the matchline to be discharged, which is detected with the aid of a sense amplifier. Since the search data is fed to large number of cells, TCAM uses *searchline drivers* to handle the capacitive sink load contributed by each cell.

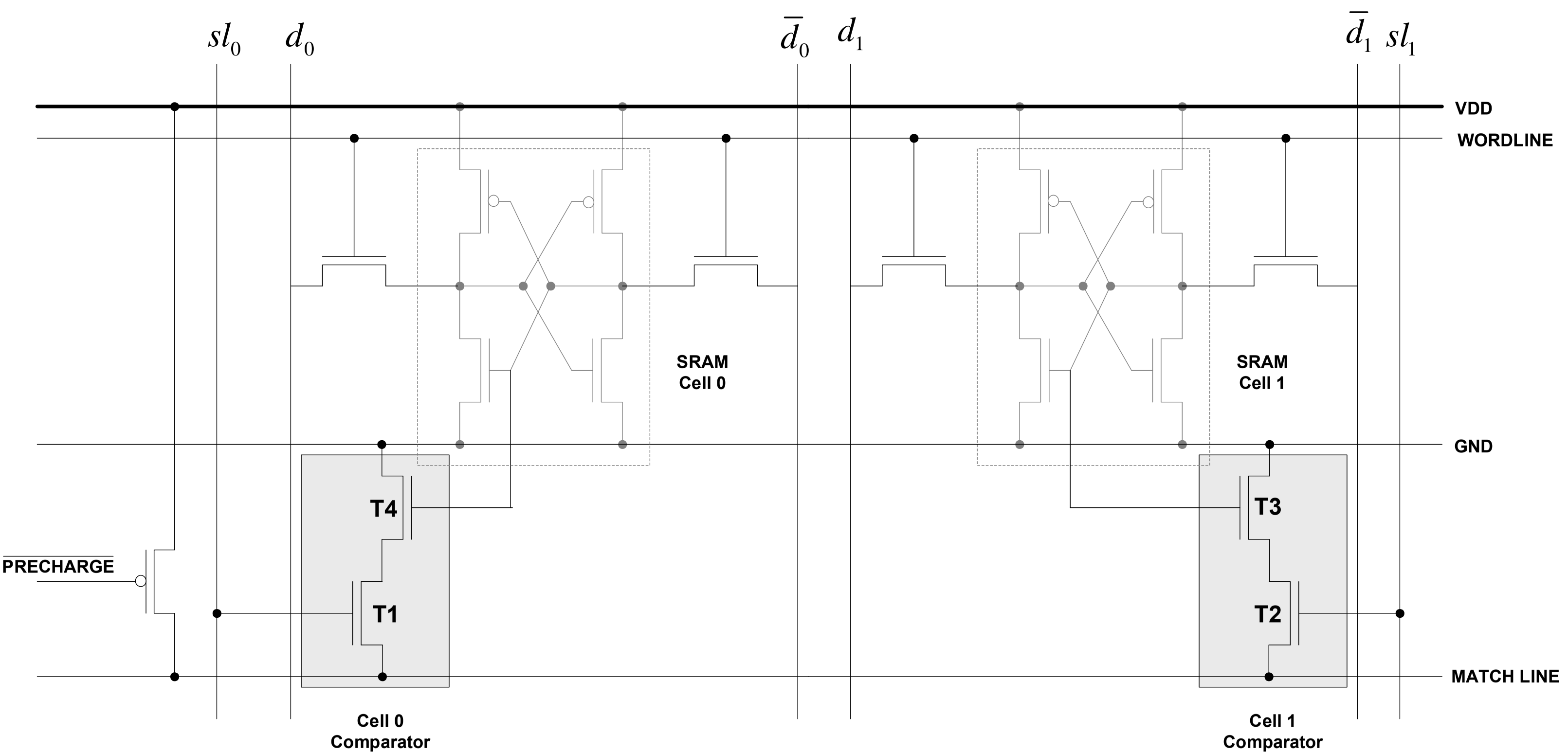

Figure 1: A NOR-based TCAM Cell

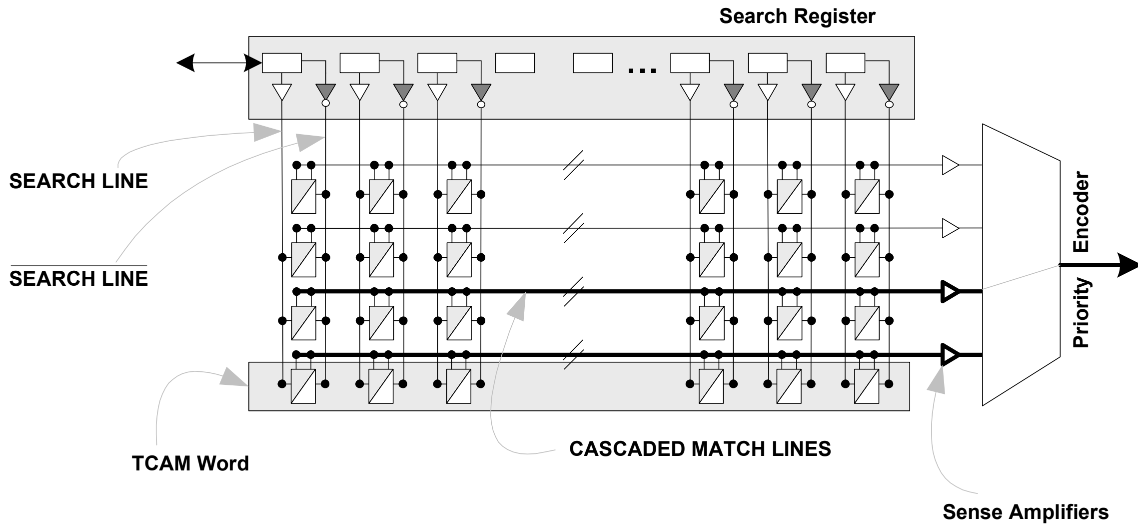

Figure 2: TCAM Memory Architecture for Lookup

C. IP Lookup Engine Architecture for Bit Selection

The low power TCAM based IP lookup engine employs a number of smaller TCAM segments known as buckets. The buckets holds the routing table prefixes sharing a common k bits hash signature. The buckets can be selectively enabled based on the hash signature extracted from incoming destination IP address.

The bit-selection logic extracts a set of k bits out of 32-bit destination IP address in a way that minimizes the worst-case partition size. Since the choice of k bits are highly dependent on the routing table snapshot, the architecture requires use of programmable multiplexors to extract the signature and selectively enable the corresponding buckets. The architecture is shown in Figure 3.

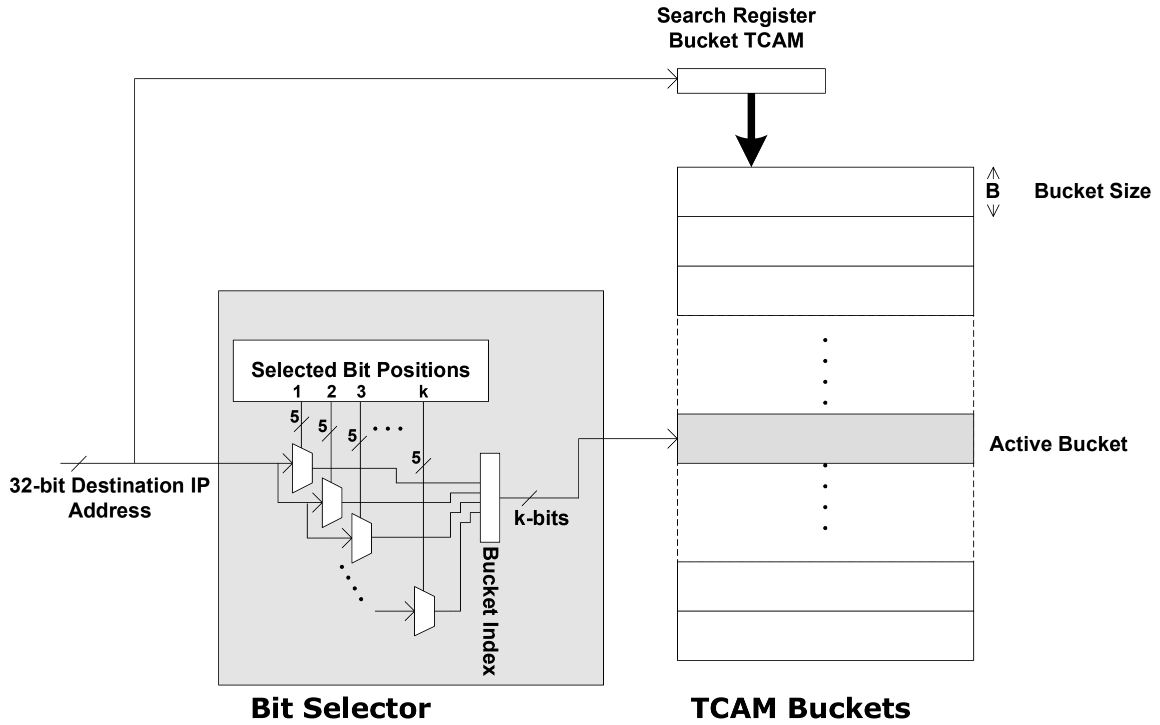

Figure 3: IP Lookup Architecture to Support Bit-Selection Based Partitioning

D. IP Lookup Engine Architecture for Trie Based Partitioning

The trie-based partitioning algorithms also employ the TCAM buckets in order to break the overall routing table into smaller chunks. However the architecture employs another intermediate TCAM known as $indexTCAM$. The trie-based partitioning algorithms carve a portion of the routing trie, which can be stored in TCAM buckets. The buckets are indexed by the prefix corresponding to root of subtrie carved as a bucket.The indexing prefixes are of variable length and can be stored in the indexTCAM. The incoming destination IP address is first matched with entries in the indexTCAM which enables the corresponding bucket in the second level TCAM. The incoming IP address is again searched in the active bucket in the second level yielding the desired match. The whole operation can be pipelined to yield one lookup opera-

tion per cycle. The architecture is shown in Figure 4.

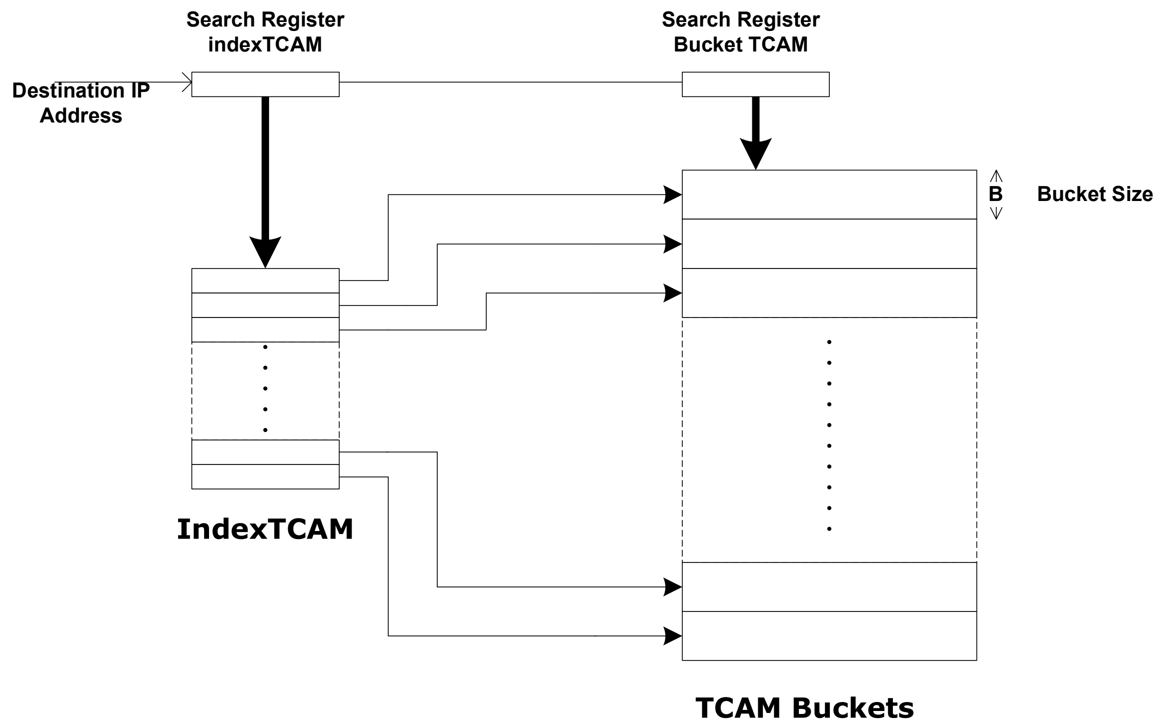

Figure 4: IP Lookup Architecture to Support Trie Based Partitioning

CHAPTER V

ROUTING TRIE FOR PARTITIONING

Most of the partitioning algorithms employ a routing trie which can be treated as a two dimensional map of the routing prefixes. The partitioning scheme presented in [9], [24] utilizes a binary routing trie. The partitioning scheme presented in [10] uses PATRICIA [2] based routing trie. The intermediate trie nodes act as clustering points for the prefixes stored in the descendent nodes. Thus the partitioning algorithm needs to mark the nodes in routing trie which will lead to good partitioning. The following sections present an overview of routing tries and extensions necessary to support partitioning algorithms. The chapter concludes with a discussion on a few useful properties of routing trie to perform efficient partitioning.

A. Binary Routing Trie

The binary routing trie can be defined in terms of a binary retrieval tree known as *binary trie*. Each non-leaf node v in the binary trie can have up to two children labeled v_0 and v_1. The edges connecting the node its children v_0 and v_1 can be labeled as 0 and 1 respectively to represent its direction. The subtrie routed at a node v is denoted as $T(v)$.

The basic unit of insertion and deletion in the binary trie is a path. A path containing all the edges between node v_i and v_j is denoted as $P_{v_i \sim v_j}$ and can be uniquely mapped to a string $\mathfrak{s}(P_{v_i \sim v_j})$ in $\{0,1\}^*$ also known as route. The route is formed by concatenating directions of all the edges between nodes v_i and v_j. Therefore a path can be specified as $P(u, \mathfrak{s})$, where u is the starting node and $\mathfrak{s}$ specifies the route traversed by the path. If the starting node is root, the path can be simply represented as $\mathfrak{s}$ omitting the root node.

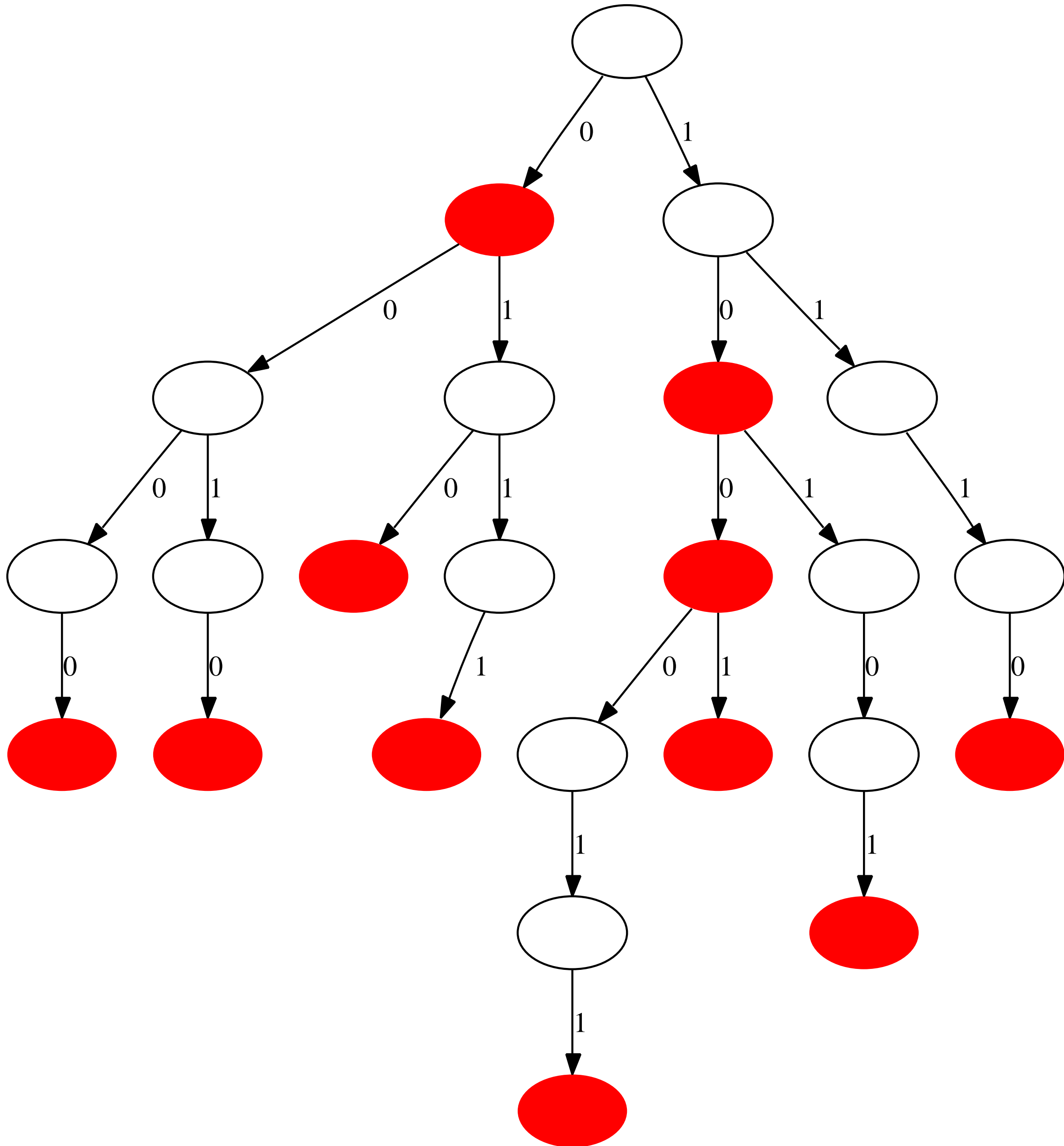

Figure 5: Binary Routing Trie for Sample Routing Table

A binary routing trie can be formed by treating the prefixes P_ℓ from the specified routing table as paths in the binary trie with root as starting node and route $\mathfrak{s} = P_\ell$. The end node of the path corresponding to prefixes are marked to indicate the presence of a prefix. The binary routing trie for a sample routing table given in Table II is shown in Figure 5. As we can see, all the prefixes of length n are stored at n^{th} level in the binary trie. The covering prefix of a node v in the binary routing trie is the lowest ancestor of p_v such that $\mathfrak{s}(P_{root \sim p_v})$ is a prefix in the routing table or in other words p_v is marked in binary routing trie.

IP lookup procedure scans the input destination IP address starting at the most significant bit and traverses the binary trie in order to find the longest matching prefix. The traversal proceeds towards the child $v_0(v_1)$ for an input 0(1) in the destination IP address until it has found the deepest node containing a prefix.

B. Ternay Routing Trie

Ternary trie are similar to binary trie except that they allow the possibility of a third child. Thus, each non-leaf node in the ternary trie can have up to three childer labeled v_0, v_1 and v_x. The set of children of a node v is denoted by $\Sigma(v)$. The edges connecting the node to its children are labeled as 0, 1 and x to specify its direction. The parent of a node v is denoted by $\pi(v)$.

The direction 0 and 1 are treated disjoint direction as in binary trie however the direction x is defined as a union of both the directions 0 and 1. Thus, the route $\mathfrak{s}(P_{u \sim v})$ between the nodes u and v can be treated as a string in $\{0, 1, x\}^*$. This important extension allows ternary trie to represent minimized routing table.

The ternary routing trie can be formed by treating the prefixes P'_ℓ in minimized routing table as paths in ternary trie with starting node as root and route $\mathfrak{s} = P'_\ell$.

Table II: Sample Routing Table

No	Prefixes	No	Prefixes
1	0—–	7	10—-
2	0000–	8	100—
3	0010–	9	1001–
4	010—	10	1110–
5	0111–	11	10101-
6	100011		

Please note that P'_ℓ may contain intermediate don't cares (x symbols) as a result of routing table compaction. Therefore P'_ℓ can not be represented in the binary routing trie. The ternary trie for a sample minimized routing table given in Table III is shown in Figure 6.

We propose a partitioning scheme for minimized routing table based on the ternary trie. Authors in [14] presents a variation of ternary trie to perform routing table compaction by introducing additional constraint during prefix insertion. These constraint acts as minimization constraint and keeps the routing table minimized at all time. Thus ternary trie may be advantageous for a hybrid approach employing routing table compaction as well as partitioning.

C. Routing Trie Extension to Perform Partitioning

The trie-based partitioning algorithms analyzes the two dimensional trie map of the routing table to cluster the routing entries present in the subtrie according to a given power constraint. In order to be able to perform the clustering, the trie-based

partitioning algorithm requires each node v in the trie to maintain a count of prefixes present in the subtrie rooted at v. Since each routing prefix is visualized as a path in the trie, the count will be referred as a *path count* and denoted by $|v|$.

Table III: Sample Minimized Routing Table

No	Prefixes	No	Prefixes
1	0110-	11	00100
2	1-01-	12	101-0
3	1000-	13	111-1
4	1011-	14	1010-
5	0-110	15	10000
6	1100-	16	11-01
7	110–	17	11110
8	101–	18	01-00
9	01111	19	1-101
10	11–1	20	11001

D. Path Count Properties

The path count for n-ary trie can be recursively defined as :

$$|v| = \begin{cases} 1 & \text{if v is leaf node} \\ 1 + |v_0| + |v_1| + \cdots |v_n| & \text{if v is a prefix} \\ |v_0| + |v_1| + \cdots |v_n| & \textit{otherwise.} \end{cases}$$

The following useful properties of the path count are utilized by trie-based partitioning algorithm in order to generate efficient partitions.

Property 1: $|v| > 0$

Proof: This property is trivially true as leaf nodes have the least path count and are initialized to 1.

Property 2: $|v| >= |v_i|$

Proof: The equality holds if all the other nodes except i^{th} children are absent. The inequality holds if more than one children are present.

Property 3: $|v| > B \Rightarrow \exists i,\ |v_i| \in [\lceil \frac{B}{n} \rceil, B]$

Proof: This property can be proved by pigeon-hole principle. In an n-ary trie, the path count of a node v has to be distributed to at most n children. This is equivalent to $|v|$ pigeons and n holes, hence each whole must contain at-least $\lceil \frac{|v|}{n} \rceil$ pigeons. Therefore there is at least a child of node v which has a path count more than $\lceil \frac{|v|}{n} \rceil$. Also the path count of a child node can not exceed that of its parent which proves the property.

Figure 6: Ternary Routing Trie for Sample Routing Table

CHAPTER VI

PARTITIONING OF MINIMIZED ROUTING TABLE

The partition of minimized routing table can be performed by suitably modifying the trie based partitioning algorithms described earlier. In this chapter, we first present the bucket carving procedure which is common to the proposed partitioning algorithm. This is followed by a description and analysis of proposed modified algorithm.

A. Bucket Carving and UnTraversal

The actual bucket carving at an intermediate node v in the routing trie involves two main steps. The first step involves un-traversal of all the paths corresponding to prefixes stored in the subtrie $T(v)$ rooted at v. The second step involves outputting all the prefixes stored in $T(v)$ into the specified bucket.

The algorithm for un-traversal is shown in Algorithm 1. The algorithm starts with the specified node v and ascend by backtracing $\pi(v)$ successively until it encounters the root node. Since the path count of starting node v is also the count of prefixes stored in $T(v)$, it decrements the path count of all the nodes encountered from v to $root$ (including) by $|v|$. This ensures that the routing trie remains in consistent state after bucket carving.

The algorithm for bucket carving is shown in Algorithm 2 and requires a $BucketId$ and start node v. The algorithm visits all the nodes $u \in T(v)$ in the post-order manner to find all the nodes containing prefixes. These prefixes are stored in the bucket specified by the given $BucketId$. This behavior is shown in lines 2-3 in Algorithm 2. The index TCAM entry corresponding to covering prefix $\rho(v)$ of the node v is made to point to the specified bucket as shown in line 4. The algorithm then deletes the subtrie $T(v)$ rooted at v and suggest the partitioning algorithm to continue at parent

of node v. The bucket carving consumes at least $|v|$ nodes in the routing trie.

UnTraverse(v)

```
1   Δ = |v|
2   while(v ≠ ∅)
3       |v| = |v| − Δ
4       v = π(v)
5   endwhile
end
```

Algorithm 1: Traverses the routing trie from node v up to the root

B. Ternary SubTrie Partitioning

The ternary subtrie split algorithm is adapted from [9] to operate on ternary trie. The algorithm splits the entire minimized routing table into a number of partitions subjected to an upper bound B on the size of the partition. The ternary subtrie split algorithm guarantees that size of generated partitions will be at least $\lceil \frac{B}{3} \rceil$. This may sound a little inferior as compared to a lower bound $\lceil \frac{B}{2} \rceil$ of the approach described in [9]. However we will demonstrate in the result section that benefits obtained from partitioning the minimized routing table greatly offset the drawback of this slightly inferior lower bound.

The pseudo code for subtrie splitting is shown in Algorithm 3. The algorithm requires the maximum bucket size B and a ternary routing trie T as input parameter. The algorithm recursively visits its children starting from the root node $r(T)$ until it encounters a node v whose path count is greater than B. This behavior is shown in

```
CarveBucket(v, BucketId)
---------------------------------------------
1   if(|v| > 0)
2       ∀u ∈ T(v)
3           OutputPrefix(u, BucketId)
4       indexTCAM[ρ(v)] ← BucketId
5   endif
6   p = π(v)
7   delete T(v)
8   ↩ return p
end
```

Algorithm 2: The algorithm to carve subtrie rooted at node v

lines 4-6 in Algorithm 3. The function AdvanceToUnvisitedChild creates a post-order traversal effect with the children being visited in the order v_x, v_0, and v_1 respectively.

Once a node $v = u$ with $|u| \leq B$ has been found, it further checks if the path count of the node satisfies the lower bound $\lceil \frac{B}{3} \rceil$ on the bucket size. If the bounds are satisfied a bucket is carved out at the node u. Otherwise, algorithm expects to find a bucket satisfying the lower bound rooted at the siblings of u. Thus algorithm stops recursively visiting the trie at node u and proceeds towards its first siblings. This is achieved by setting the node u into *visited* state and back-tracing to its parent $v = \pi(u)$, when AdvanceToUnvisitedChild cuases the algorithm to visit siblings of u. After carving one or more bucket at the subtrie rooted at its first sibling, the algorithm backtracks to $v = \pi(u)$. However at this time, due bucket carving $v = |\pi(u)| > B$ may no longer be valid. In such case, the traversal of second sibling of u is avoided as we can always carve out a equal at $\pi(u)$ or even larger bucket further up in the routing

SubTrieSplit$(T,\ B)$

1. $BucketId = 0,\ v = \underline{\text{root}}(T)$
2. **while**$(v \neq \emptyset)$
3. $\quad Carving = $ TRUE
4. $\quad$ **if**$(|v| > B)$
5. $\quad\quad v = \underline{\text{AdvanceToUnvisitedChild}}(v)$
6. $\quad\quad Carving = $ FALSE
7. $\quad$ **elseif**$(\underbrace{(\pi(v) = \emptyset)}_{1} \vee \underbrace{(|\pi(v)| \leq B)}_{2} \vee \underbrace{(|v| < \lceil \frac{B}{3} \rceil)}_{3})$
8. $\quad\quad$ **if**$(\pi(v) \neq \emptyset)$
9. $\quad\quad\quad v \leftarrow visited$
10. $\quad\quad\quad v = \pi(v)$
11. $\quad\quad\quad Carving = $ FALSE
12. $\quad\quad$ **endif**
13. $\quad$ **endif**
14. $\quad$ **if**$(Carving = $ TRUE$)$
15. $\quad\quad \underline{\text{UnTraverse}}(v)$
16. $\quad\quad v = \underline{\text{CarveBucket}}(v, BucketId{+}{+})$
17. $\quad$ **endif**
18. **endwhile**

end

Algorithm 3: The partitioning algorithm based on ternary subtrie splitting

trie. This behavior is implemented in the lines 7-13 of the algorithm. The bucket carving is done in the lines 14-17 if $Carving$ flag is set to $TRUE$. The algorithm continues until all the nodes in routing trie have been consumed.

C. Ternary PostOrder Partitioning

The ternary post-order split algorithm is similarly adapted from [9] to operate on ternary trie. Although, the number of $indexTCAM$ entries generated in subtree split algorithm is less, the bucket utilization is very poor and can be just 33.3% in the worst-case. The modified post-order split algorithm can generate equal sized buckets except for the last bucket. The algorithm is described in the following subsections.

1. CarveOneBucket Algorithm

The heart of modified post-order split algorithm is another algorithm called CarveOne-Bucket. The post-order split relies upon this algorithm to carve equal sized buckets. The pseudo code for *CarveOneBucket* is provided in Algorithm 4. The algorithm requires a start node v, the remaining bucket capacity parameter β and $BucketId$ input parameters.

The algorithm checks if $|v| \leq \beta$ and $|v| > \beta$ are simultaneously satisfied, which means the node v is a right candidate for carving out a bucket. If this condition is satisfied, the algorithm carves a bucket at node v and update the remaining bucket capacity β to $\beta - |v|$. This is performed in lines 11-15 in Algorithm 4.

Another condition for carving a bucket is when a leaf node v has a count greater than β. This condition is never encountered in binary routing trie partitioning however it can occur very easily in ternary routing trie. In such cases, the algorithm carves a bucket at node v and update β to 0. This is achieved in lines 3-6 in Algorithm 4.

```
CarveOneBucket(v, β, BucketId)
1   if(|v| > β)
2       v = AdvanceToUnvisitedChild(v)
3       if(Σ(v) ≠ ∅)
4           UnTraverse(v)
5           v = CarveBucket(v, BucketId)
6           β = 0
7       endif
8   elseif((|v| < β) ∧ (π(v) ≠ ∅) ∧ (|π(v)| ≤ β))
              \__1__/   \___2___/   \____3____/
9       v ← visited
10      v = π(v)
11  else
12      β = β − |v|
13      UnTraverse(v)
14      v = CarveBucket(v, BucketId)
15  endif
16  ↩ return v
end
```

Algorithm 4: The algorithm used by post-order split to carve exactly one subtrie from node v

If the previous two conditions are not satisfied, the algorithm suggest traversing the children of node v as given in line 2. However if any one of the children of node v is already visited, the path count $|v|$ may become $\leq \beta$. In such cases, traversal to one of the unvisited children might not be advantageous and the traversal should proceed to $\pi(v)$, the parent of v, in order to carve a larger bucket under the specified constraint. This is achieved in lines 8-10 in Algorithm 4. The algorithm returns after successfully carving a bucket.

```
PostOrderSplit(T, B)
─────────────────────────────────────
1    β = B, v = root(T)
2    BucketId = 0
3    while(v ≠ ∅)
4        while((v ≠ ∅) ∧ (β ≠ 0))
5            v = CarveOneBucket(v, β)
6        endwhile
7        β = B
8        BucketId++
9    endwhile
end
```

Algorithm 5: The partitioning algorithm based on ternary postorder splitting

2. Ternary PostOrderSplit Algorithm

The modified algorithm for post-order splitting is shown in Algorithm 5. The algorithm requires a bucket size B and a ternary routing trie T as input parameter. In order to carve exact sized buckets, the algorithm relies upon the algo-

rithm CarveOneBucket. The post-order split algorithm makes repeated calls to CarveOneBucket with remaining bucket capacity until the current bucket is full or all the nodes in the routing trie have been consumed. This is shown in lines 4-6 in Algorithm 5. The procedure is repeated until all the $\lceil \frac{|r(T)|}{B} \rceil$ buckets are full as given by lines 3, 7-9 in Algorithm 5.

CHAPTER VII

EXPERIMENTAL RESULTS

In this section we present the performance of proposed partition scheme for the minimized routing table. To establish the suitability of the proposed scheme, we evaluated its performance on standard routing table traces as used in [12], [13], [14], [11]. The number of routing table entry in the traces considered ranges from 13000-125000, in order to establish consistency of the approach for smaller as well as larger routing table. All the results were obtained on a linux platform containing Intel Pentium Processor running at 600 MHz. The raw experimental data for various routing table can found in Appendix A. However, the main results have been summarized and presented in Figures7-13. The results for partitioning approach described in [9] are marked with keyword `CoolCAM`.

The index TCAM size comparison for `SubTreeSplit` algorithm is shown in Figure 7 for various routing tables. We can infer from the figure, that the index TCAM size for the proposed approach is consistently smaller than the `CoolCAM` approach. This behavior runs counter-intuitive due to fact that modified `SubTreeSplit` uses a worst-case lower bound of $\lceil \frac{B}{3} \rceil$, instead of $\lceil \frac{B}{2} \rceil$ used in `CoolCAM`. However, the benefit of compaction outweighs the disadvantage of a smaller bound as evident in the results.

The index TCAM size comparison for `PostOrderSplit` algorithm is shown in Figure 8. The index TCAM for `PostOrderSplit` also follows the same trend exhibited by `SubtreeSplit` algorithm except that the difference in sizes between proposed approach and `CoolCAM` approach is larger across all bucket sizes. Thus, we can state that the compaction is more beneficial to `PostOrderSplit` algorithm as compared to `SubtreeSplit`.

In order to calculate the power consumption, we considered the size of the bucket

as well as size of the index TCAM. Since in a pipelined operation, the index TCAM and any one of the bucket is active at all the time, the total power includes the power consumed by the index TCAM as well as TCAM bucket. The power overhead for architectural enhancement such as path between first level (index) and second level (bucket) TCAM is omitted for comparison and assumed to remain fixed for both the methods. In order to calculate the area, we similarly considered the area required for the TCAM buckets as well as index TCAM. The total silicon area includes the area dedicated to index TCAM and all the second level buckets. We excluded the area of the inactive buckets in the second level TCAM for the purpose of comparison. The unit power and area figures were obtained from [25], which discusses a ternary cam design using 0.13 μm logic process. The design consumes 0.5W/MBit and requires an area of 13mm^2/MBit. These figures are reported for an operational TCAM module and averaged over the entire design employing several other auxiliary components such as sense amplifier, priority encoder etc.

The power profile of `SubTreeSplit` algorithm is shown in Figure 9 for various routing tables. As we can infer from these figures, total power consumed increases dramatically as we go towards very fine-grained(128 and smaller) partitions or very coarse-grained(2048 and higher) partitions. The growth in size of index TCAM causes the power increase towards fine-grained partitions. However, towards coarse-grained partition, the size of bucket dominates over the size of index TCAM (containing less than 100 entries) and explains the growth in power at this end. Moreover, the power consumption for proposed methodology is below the `CoolCAMs` approach for smaller sized partitions while it coincides for large sized partitions. The difference in power consumption between proposed approach and `CoolCAMs` approach grows towards smaller sized partitions. For a given bucket size, the power difference is larger for large routing tables as compared to small routing tables. Thus we expect

the power difference to widen further for fairly large routing table in future backbone routers, and proposed methodology can offer tremendous benefits in terms of power. The power profile of `PostOrderSplit` algorithm is shown in Figure 10 and follows the same trend as in the case of `SubTreeSplit` algorithm. The only difference is that the power difference metric here is larger due to larger index TCAM as compared to `SubTreeSplit` algorithm. This behavior is consistent across all routing table traces.

The silicon area profile of `SubTreeSplit` and `PostOrderSplit` algorithms are shown in Figure 11 and Figure 12 respectively. Figure 11 shows that total silicon area required for `SubTreeSplit` algorithm follows a zig-zag pattern. This occurs due to internal fragmentation resulting from a large number of incompletely-filled buckets. The fragmentation is more prominent towards coarse-grained partitions which makes incompletely-filled buckets more probable. This trend is not observed for `PostOrderSplit` algorithm as all the buckets are fully utilized. Further we expect the area to continuously decrease for increasing bucket-size, with occasional anomaly towards large sized partition which cause a large incompletely-filled bucket. This is evident from Figure 12.

The total silicon area required for `SubTreeSplit` algorithm for the proposed approach is less than the corresponding `CoolCAM` approach. The similar trend is also observed for `PostOrderSplit` algorithm. These figures demonstrate that the proposed approach offers about 40-60% silicon area savings across all the routing tables. The behavior is consistent across both `SubTreeSplit` and `PostOrderSplit` partitioning algorithms.

A case study for `att` routing table for `SubTreeSplit` algorithm is presented in Figure 13. The figure combines the area and power profile of the proposed approach and `CoolCAM` approach on the same graph. The figure shows that the power profile for the proposed approach fares similar or better than the `CoolCAM` approach. Also,

where the power profile for these two approaches coincide, we can derive benefits from silicon area savings which is consistently less for the proposed approach. Thus the partitioning of minimized routing table results in better TCAM utilization while saving the precious silicon area.

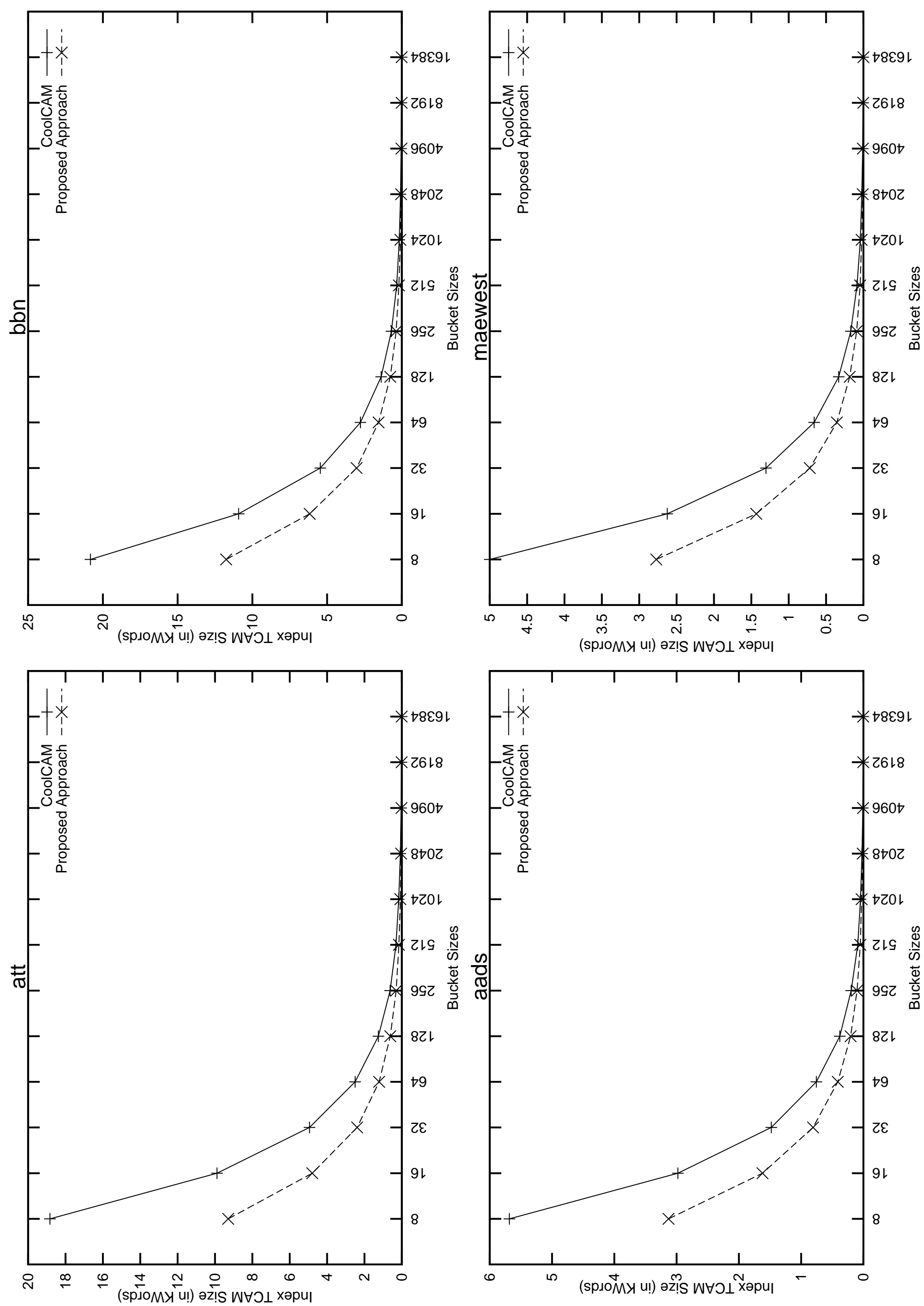

Figure 7: Comparison of Index TCAM Size for `SubTreeSplit` Algorithm w.r.t. Bucket Sizes

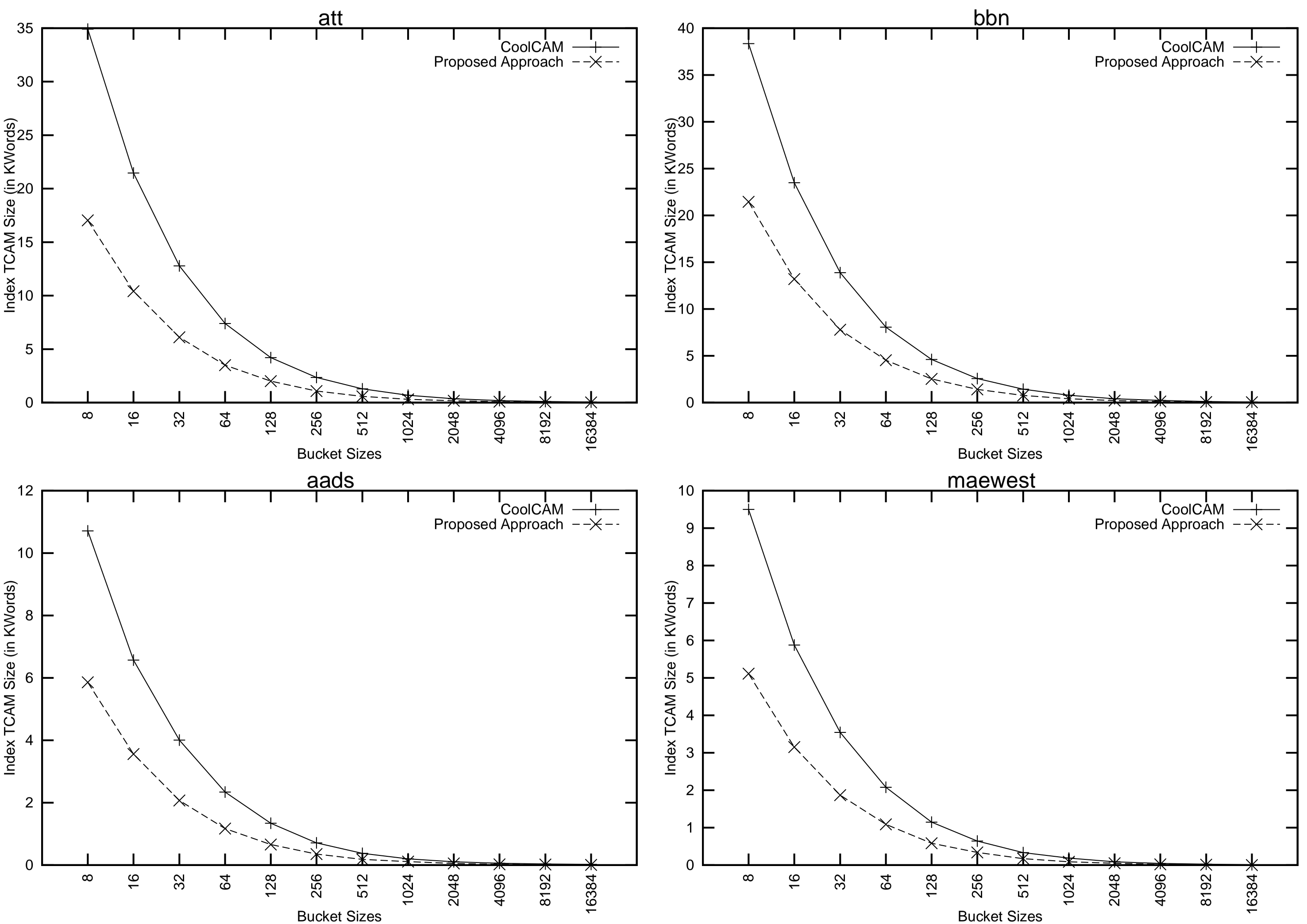

Figure 8: Comparison of Index TCAM Size for `PostOrderSplit` Algorithm w.r.t. Bucket Sizes

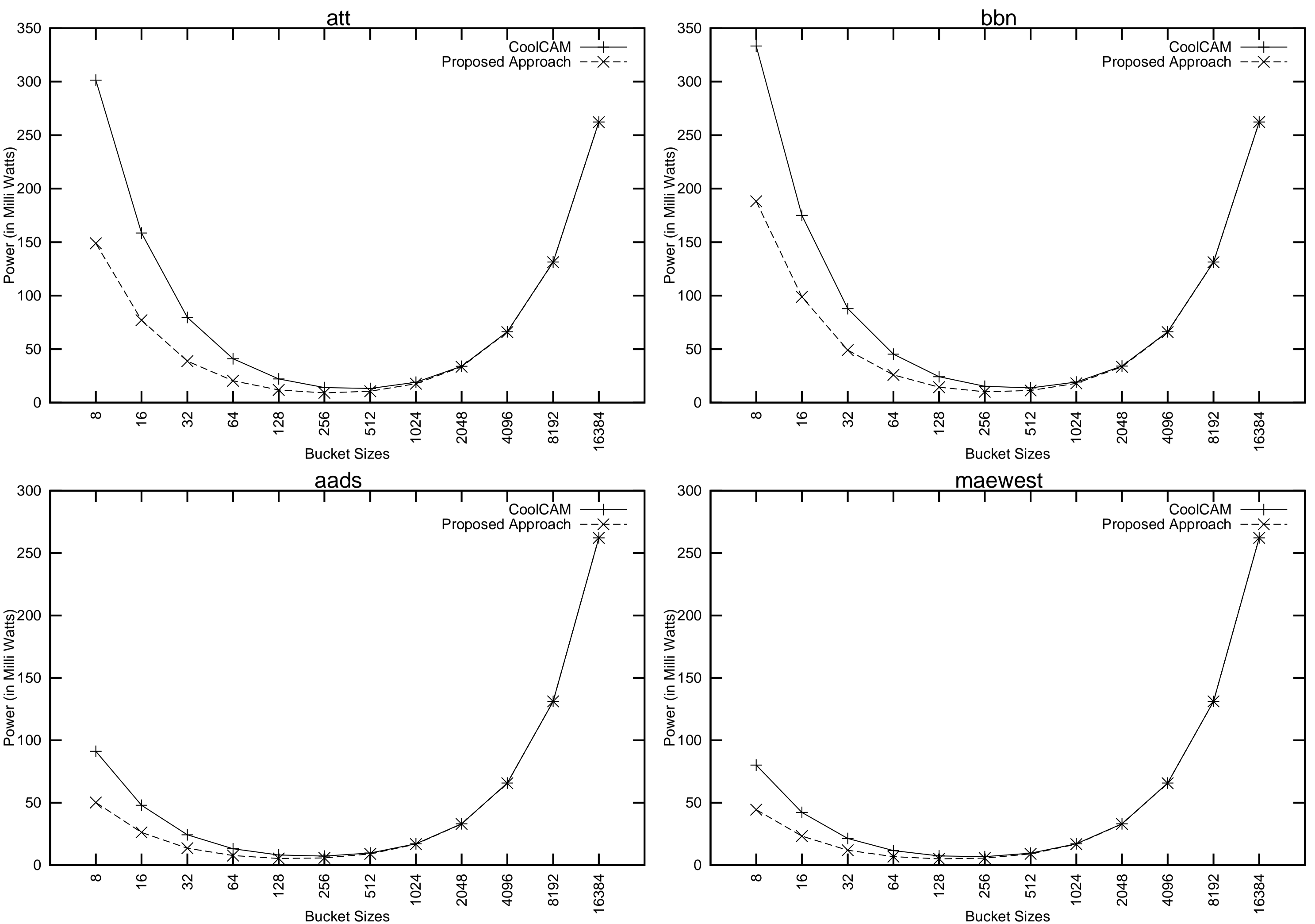

Figure 9: Comparison of Power for `SubTreeSplit` Algorithm w.r.t. Bucket Sizes

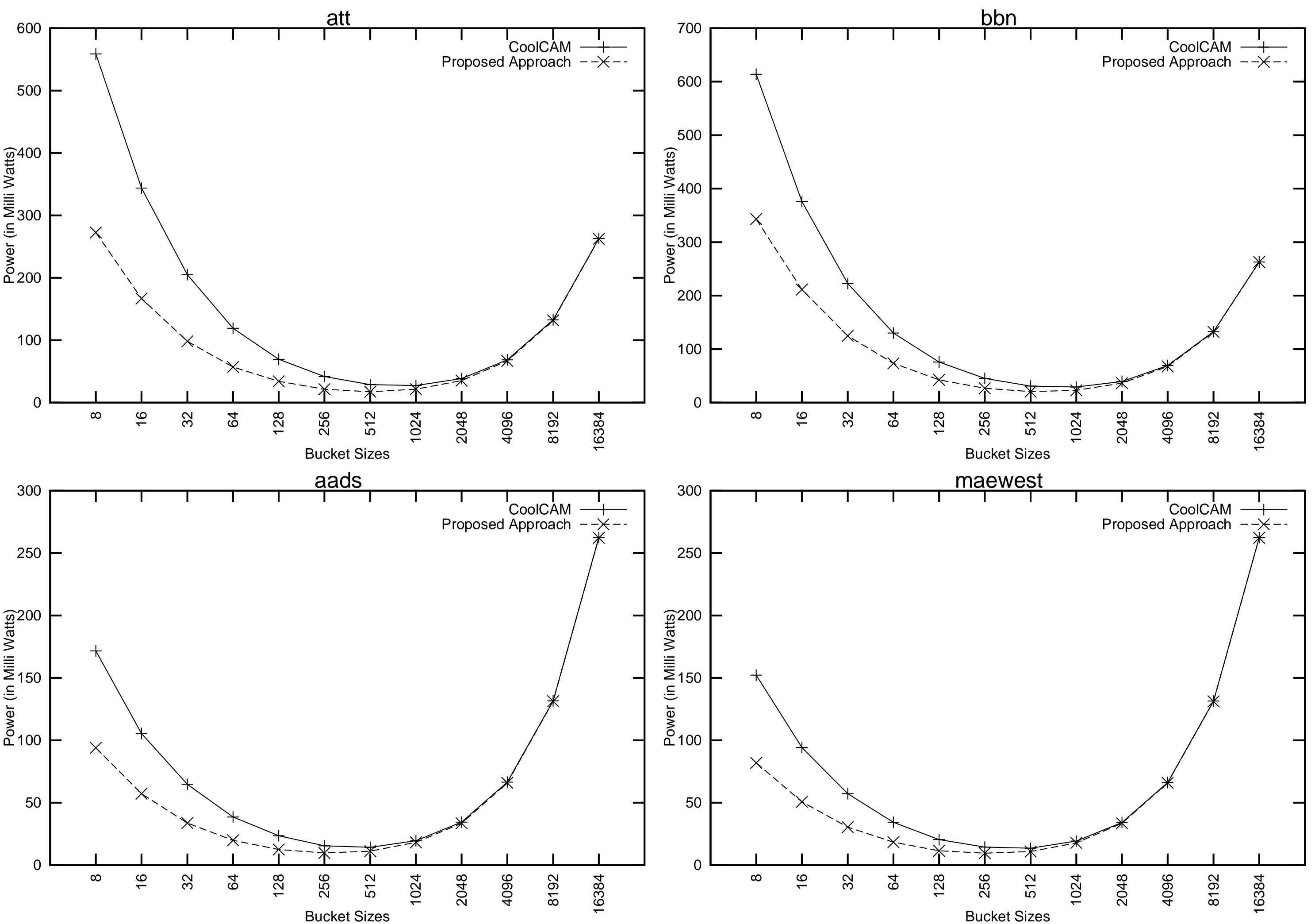

Figure 10: Comparison of Power for `PostOrderSplit` Algorithm w.r.t. Bucket Sizes

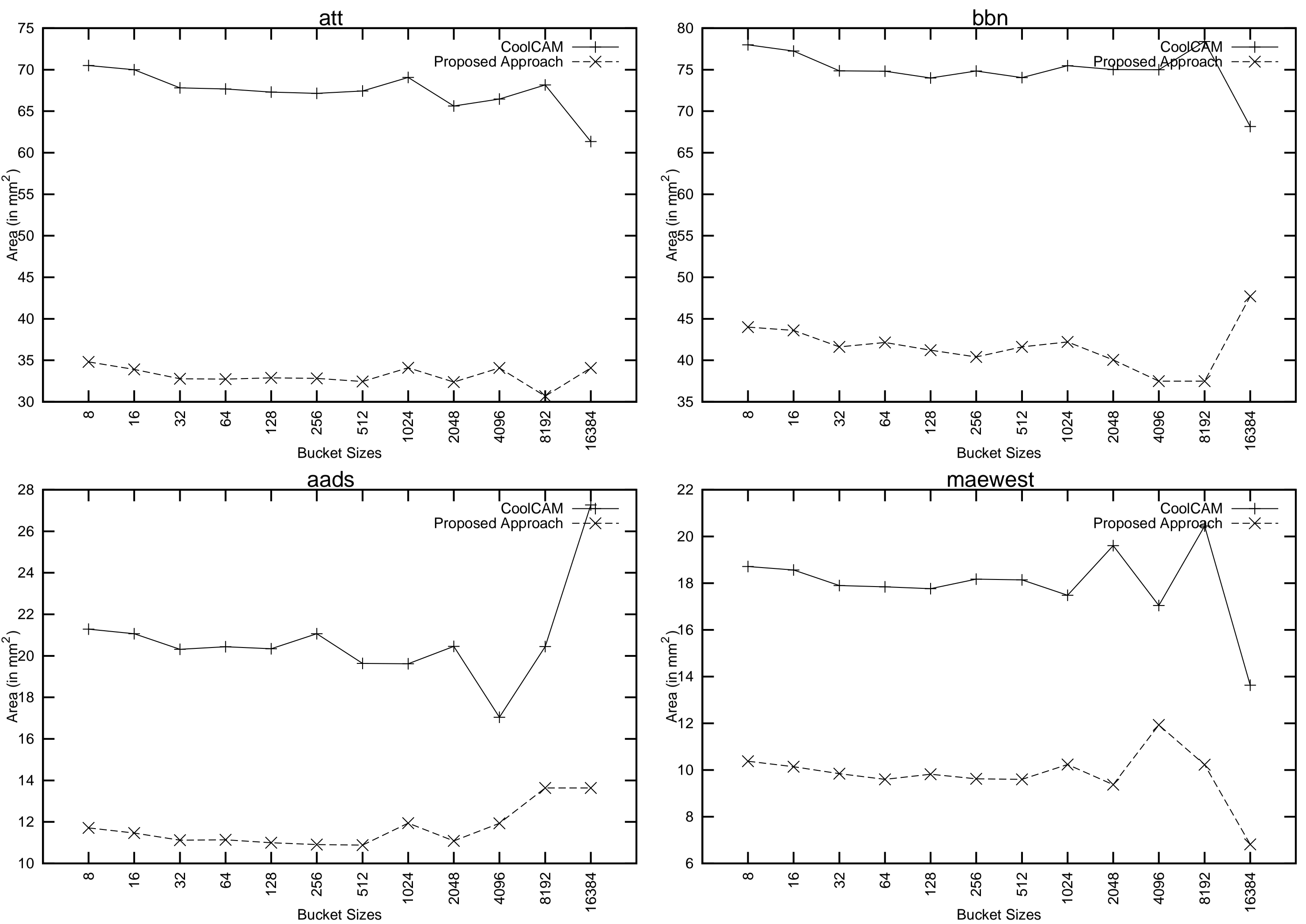

Figure 11: Comparison of Area for `SubTreeSplit` Algorithm w.r.t. Bucket Sizes

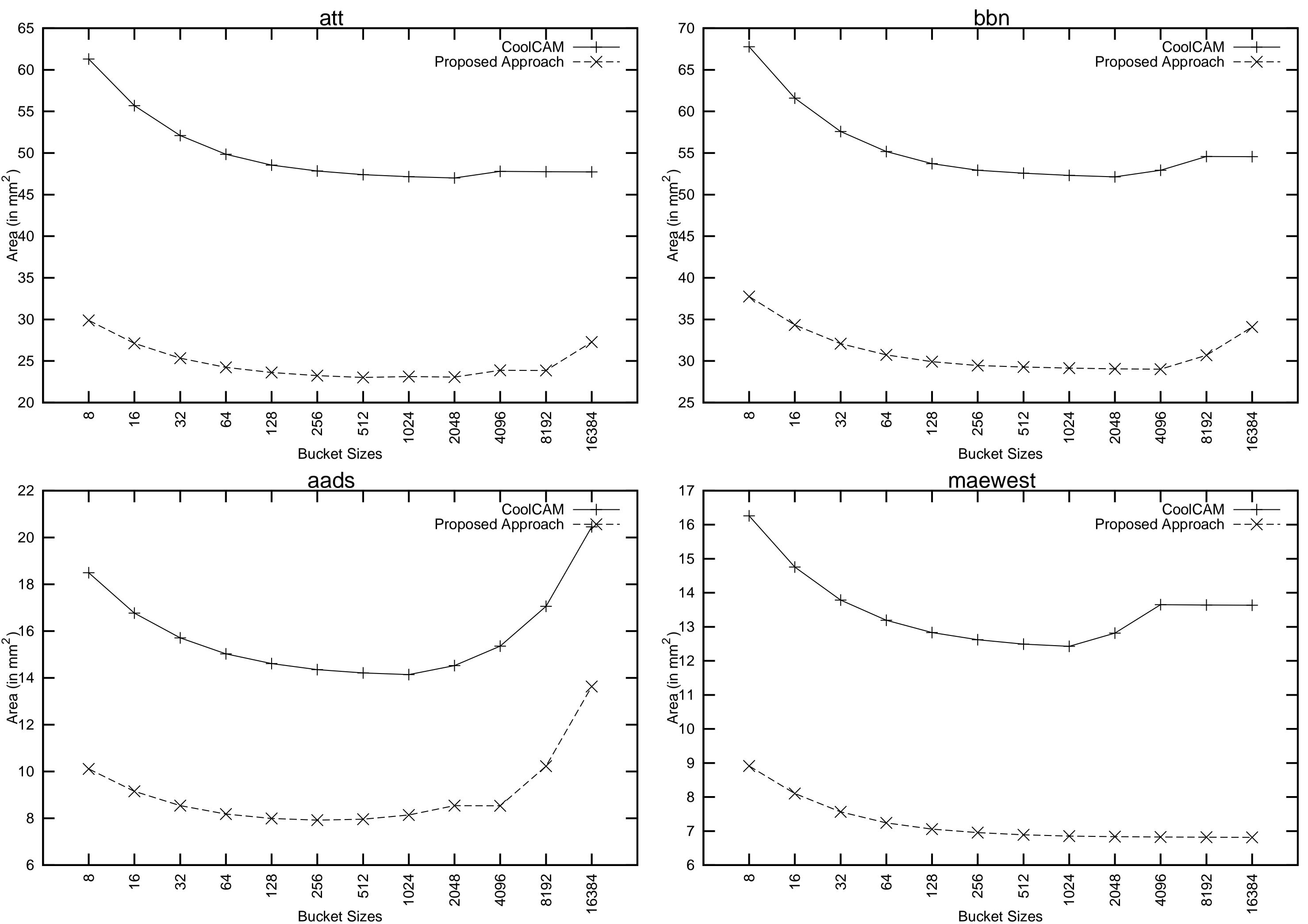

Figure 12: Comparison of Area for `PostOrderSplit` Algorithm w.r.t. Bucket Sizes

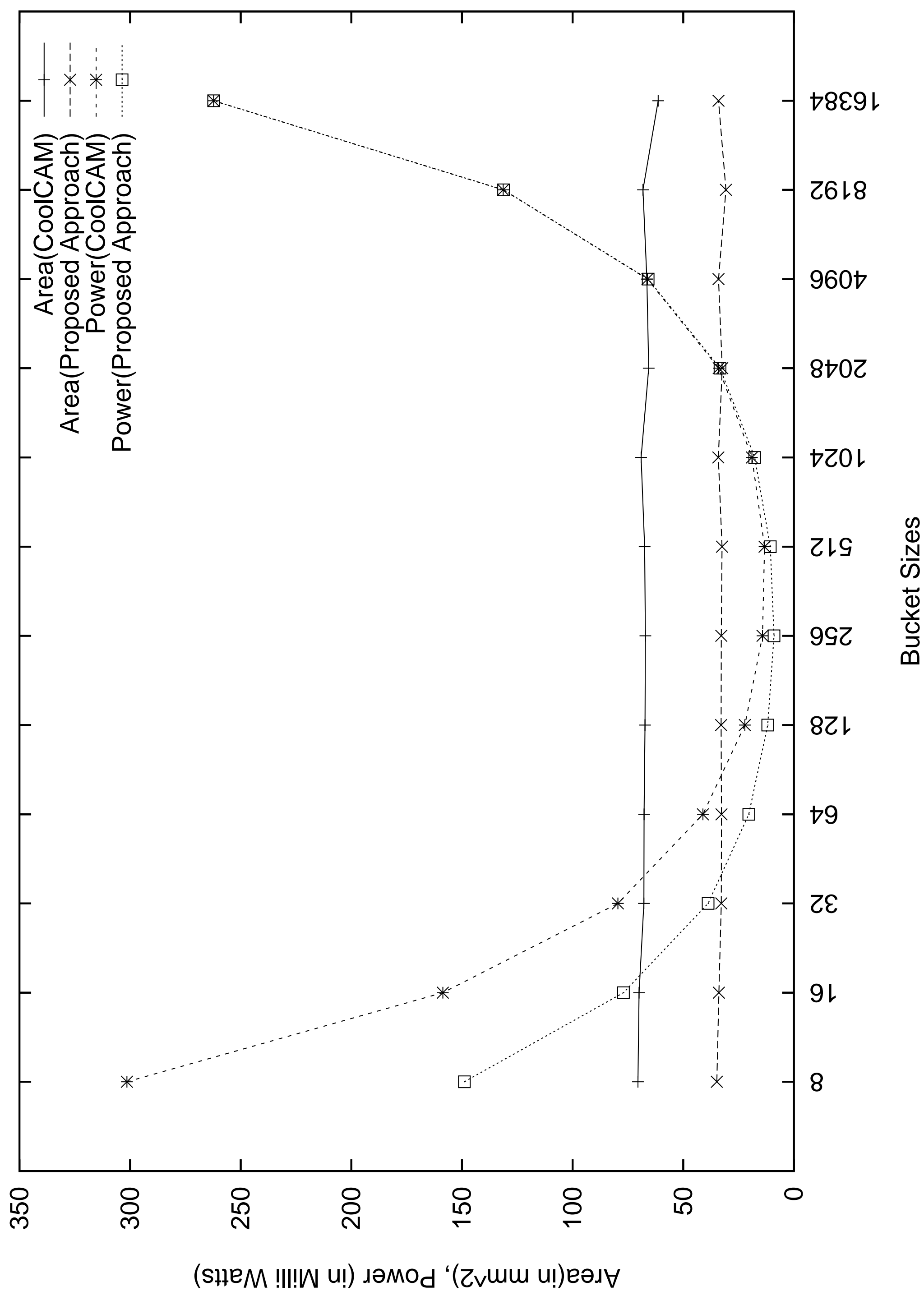

Figure 13: Case Study - `att` Routing Table

CHAPTER VIII

CONCLUSION AND FUTURE WORK

The requirement for power reduction in TCAM based IP lookup engines has led to two disparate streams of research. The first streams involves routing table minimization while the second stream focuses on routing table partitioning. We proposed a hybrid approach to unify these two schemes to achieve power economy and reduce overall silicon area. We presented a modification of sub-trie split and post-order split algorithm which can operate on ternary trie. We found that the hybrid approach can reduce the overall silicon area by up to 60% and reduce the power by up to an additional 50% for fine-grained partitions.

As a future work, we would like to study the effect of the partitioning on larger routing table traces containing over a million entries. Also, we would like to investigate the possible strategies for extending the algorithms for IPv6 routing tables. We also intend to study and investigate other partitioning scheme which may result in more power-efficiency and throughput.

REFERENCES

[1] P. Gupta, "Algorithms for routing lookups and packet classification," PhD Dissertation, Stanford University, CA, 2000.

[2] D. R. Morrison, "PATRICIA - Practical Algorithm To Retrieve Information Coded in Alphanumeric," *J. ACM*, vol. 15, no. 4, pp. 514–534, 1968.

[3] K. Sklower, "A Tree-Based Routing Table for Berkeley Unix," in *Proc. Winter USENIX Conference*, 1991, pp. 93–99.

[4] S. Nilsson and M. Tikkanen, "Implementing a dynamic compressed trie," in *Workshop on Algorithm Engineering*, K. Mehlhorn, Ed., Saarbrucken, Germany, 1998, pp. 1–3.

[5] V. Srinivasan and G. Varghese, "Faster IP lookups using controlled prefix expansion," in *SIGMETRICS '98/PERFORMANCE '98: in Proc. 1998 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, New York: ACM Press, 1998, pp. 1–10.

[6] M. A. Ruiz-Snchez, E. W. Biersack, and W. Dabbous, "Survey and Taxonomy of IP Address Lookup Algorithms," *IEEE, Network*, vol. 15, no. 2, pp. 8–23, Mar/Apr 2001.

[7] A. J. McAuley and P. Francis, "Fast Routing Table Lookup Using CAMs," in *Proc. IEEE INFOCOM*, Piscataway, NJ : IEEE Press, 1993, pp. 1382–1391.

[8] *SCT2000CB3, SiberCAM Ultra 2M Family*, SiberCore Technologies, [online] www.sibercore.com, March, 2004.

[9] F. Zane, G. Narlikar, and A. Basu, "CoolCAMs: Power-Efficient TCAMs for Forwarding Engines," in *Proc. IEEE INFOCOM*, Piscataway, NJ : IEEE Press, April 2003, pp. 42–52.

[10] R. Panigrahi and S. Sharma, "Reducing TCAM Power Comsumption and Increasing Throughput," in *Proc. HOT Interconnects*, Piscataway, NJ : IEEE Press, August 2002, p. 107.

[11] 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*, vol. 54, no. 5, pp. 521–533, May 2005.

[12] H. Liu, "Routing Table Compaction in Ternary-CAM," *IEEE, Micro*, vol. 15, no. 5, pp. 58–64, Jan/Feb 2002.

[13] R. Lysecky and F. Vahid, "On-Chip Logic Minimization," in *Proc. 40th Conference on Design Automation*, New York: ACM Press, 2003, pp. 334–337.

[14] S. Ahmad and R. Mahapatra, "m-Trie - A Fast and Efficient Approach to On-Chip Logic Minimization," in *Proc. Intl. Conf. on Computer Aided Design*, Piscataway, NJ : IEEE Press, November 2004, pp. 428–435.

[15] R. Brayton, G. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli, *Logic Minimization Algorithm for VLSI Synthesis*, Boston, MA: Kluwer Academic Publishers, 1984.

[16] W. V. Quine, "The Problem of Simplifying Truth Functions," *American Mathematical Monthly*, vol. 59, no. 8, pp. 521–531, 1952.

[17] W. V. Quine, "A Way to Simplify Truth Functions," *American Mathematical Monthly*, vol. 62, no. 9, pp. 627–631, 1955.

[18] E. J. McClusky, "Minimization of Boolean Functions," *Bell System Tech. Journal*, vol. 35, no. 6, pp. 1417–1444, Nov 1956.

[19] S. Kang and W. M. vanCleemput, "Automatic PLA Synthesis From a DDL-P Description," in *Proc. 18th Conference on Design Automation*, Piscataway, NJ: IEEE Press, 1981, pp. 391–397.

[20] D. W. Brown, "A State-Machine Synthesizer SMS," in *Proc. 18th Conference on Design Automation*, Piscataway, NJ: IEEE Press, 1981, pp. 301–305.

[21] S.J.Hong, R.G.Cain, and D.L.Ostapko, "MINI: A Heuristic Approach For Logic Minimization," *IBM Journal of Research and Development*, vol. 59, no. 18, pp. 443–458, Sept. 1974.

[22] D. Alessandri, "Access Control List Processing in Hardware," Diploma thesis, Eidgenossische Technische Hochschule, Zurich, Switzerland, October 1997.

[23] H. Lu, "Improved Trie Partitioning for Cooler TCAMs," in *Proc. IASTED International Conference on Advances in Computer Science and Technology*, Calgary, Canada : ACTA Press, 2004, pp. 201–212.

[24] D. Pao, C. Liu, A. Wu, L. Yeung, and K. Chan, "Efficient Hardware Architecture for Fast IP Address Lookup," in *Proc. IEE Computers and Digital Techniques*, January 2003, vol. 150, pp. 43–50.

[25] A. Roth, D. Foss, R. McKenzie, and D. Perry, "Advanced Ternary CAM Circuits on 0.13m Logic Process Technology," in *Proc. IEEE Custom Integrated Circuits Conference*, Piscataway, NJ : IEEE Press, October 2004, pp. 465–468.

APPENDIX A

EXPERIMENTAL RESULTS DATA

The main results have been summarized in Tables IV - IX for each of the routing table. The first column list the various bucket sizes considered for partitioning. The next two columns list the indexTCAM sizes for SubTrieSplit algorithm for original algorithm mentioned in [9] and the modified algorithm. The columns are correspondingly marked $Orig$ and Mod to reflect the original and modified trie partitioning algorithm. The next two columns similarly gives the power bound for original and modified algorithms based on SubTrie splitting. The same dataset is collected and reported for PostOrderSplit algorithm in the next four columns. However instead of reporting silicon area overhead in the last column, we report the number of buckets in order to aid comparison. Please note that all the buckets will be fully utilized in PostOrderSplit, hence the number of bucket gives a direct means of comparing the silicon overhead in this case.

Table IV: Comparison of Performance for `paix` Routing Table

	SubTrieSplit				**PostOrderSplit**			
Bucket	**PowerBound**		**SiliconArea**		**PowerBound**		**TotalBuckets**	
Size	**Orig.**	**Mod.**	**Orig.**	**Mod.**	**Orig.**	**Mod.**	**Orig.**	**Mod.**
8	2372	1509	18976	12072	4463	2836	1740	1108
16	1223	791	19568	12656	2782	1751	870	554
32	614	389	19648	12448	1674	1010	435	277
64	311	200	19904	12800	982	602	218	139
128	156	128	19968	13184	576	349	109	70
256	256	256	20480	13056	314	256	55	35
512	512	512	20480	14336	512	512	28	18
1024	1024	1024	19456	12288	1024	1024	14	9
2048	2048	2048	20480	14336	2048	2048	7	5
4096	4096	4096	24576	16384	4096	4096	4	3
8192	8192	8192	16384	16384	8192	8192	2	2
16384	16384	16384	16384	16384	16384	16384	1	1

Table V: Comparison of Performance for `aads` Routing Table

	SubTrieSplit				**PostOrderSplit**			
Bucket	**PowerBound**		**SiliconArea**		**PowerBound**		**TotalBuckets**	
Size	**Orig.**	**Mod.**	**Orig.**	**Mod.**	**Orig.**	**Mod.**	**Orig.**	**Mod.**
8	5686	3128	45488	25024	10714	5861	4218	2305
16	2978	1621	47648	25936	6571	3561	2109	1153
32	1480	810	47360	25920	4008	2070	1055	577
64	756	412	48384	26368	2341	1173	528	289
128	379	205	48512	26240	1340	657	264	145
256	256	256	50432	26112	709	351	132	73
512	512	512	47104	26112	512	512	66	37
1024	1024	1024	47104	28672	1024	1024	33	19
2048	2048	2048	49152	26624	2048	2048	17	10
4096	4096	4096	40960	28672	4096	4096	9	5
8192	8192	8192	49152	32768	8192	8192	5	3
16384	16384	16384	65536	32768	16384	16384	3	2

Table VI: Comparison of Performance for `pacbell` Routing Table

	SubTrieSplit				**PostOrderSplit**			
Bucket	**PowerBound**		**SiliconArea**		**PowerBound**		**TotalBuckets**	
Size	**Orig.**	**Mod.**	**Orig.**	**Mod.**	**Orig.**	**Mod.**	**Orig.**	**Mod.**
8	3723	1905	29784	15240	7068	3584	2771	1403
16	1957	998	31312	15968	4397	2187	1386	702
32	968	490	30976	15680	2634	1272	693	351
64	495	248	31680	15872	1560	715	347	176
128	250	128	32000	16000	884	419	174	88
256	256	256	31488	15872	496	256	87	44
512	512	512	31744	16384	512	512	44	22
1024	1024	1024	30720	16384	1024	1024	22	11
2048	2048	2048	30720	16384	2048	2048	11	6
4096	4096	4096	32768	16384	4096	4096	6	3
8192	8192	8192	32768	16384	8192	8192	3	2
16384	16384	16384	32768	16384	16384	16384	2	1

Table VII: Comparison of Performance for `maewest` Routing Table

	SubTrieSplit				**PostOrderSplit**			
Bucket	**PowerBound**		**SiliconArea**		**PowerBound**		**TotalBuckets**	
Size	**Orig.**	**Mod.**	**Orig.**	**Mod.**	**Orig.**	**Mod.**	**Orig.**	**Mod.**
8	4999	2771	39992	22168	9501	5113	3699	2039
16	2625	1434	42000	22944	5878	3153	1850	1020
32	1304	717	41728	22944	3543	1868	925	510
64	660	355	42240	22720	2076	1088	463	255
128	331	183	42368	23424	1149	585	232	128
256	256	256	43520	23040	642	337	116	64
512	512	512	43520	23040	512	512	58	32
1024	1024	1024	41984	24576	1024	1024	29	16
2048	2048	2048	47104	22528	2048	2048	15	8
4096	4096	4096	40960	28672	4096	4096	8	4
8192	8192	8192	49152	24576	8192	8192	4	2
16384	16384	16384	32768	16384	16384	16384	2	1

Table VIII: Comparison of Performance for `att` Routing Table

	SubTrieSplit				**PostOrderSplit**			
Bucket	**PowerBound**		**SiliconArea**		**PowerBound**		**TotalBuckets**	
Size	**Orig.**	**Mod.**	**Orig.**	**Mod.**	**Orig.**	**Mod.**	**Orig.**	**Mod.**
8	18829	9299	150632	74392	34911	17032	14052	6847
16	9897	4795	158352	76720	21465	10407	7026	3424
32	4939	2388	158048	76416	12789	6108	3513	1712
64	2503	1211	160192	77504	7390	3499	1757	856
128	1254	613	160512	78464	4202	2001	879	428
256	628	307	160768	78592	2358	1077	440	214
512	512	512	161792	77824	1282	578	220	107
1024	1024	1024	165888	81920	1024	1024	110	54
2048	2048	2048	157696	77824	2048	2048	55	27
4096	4096	4096	159744	81920	4096	4096	28	14
8192	8192	8192	163840	73728	8192	8192	14	7
16384	16384	16384	147456	81920	16384	16384	7	4

Table IX: Comparison of Performance for `bbn` Routing Table

	SubTrieSplit				**PostOrderSplit**			
Bucket	**PowerBound**		**SiliconArea**		**PowerBound**		**TotalBuckets**	
Size	**Orig.**	**Mod.**	**Orig.**	**Mod.**	**Orig.**	**Mod.**	**Orig.**	**Mod.**
8	20832	11751	166656	94008	38356	21446	15568	8665
16	10921	6165	174736	98640	23497	13190	7784	4333
32	5453	3031	174496	96992	13884	7780	3892	2167
64	2767	1559	177088	99776	8068	4522	1946	1084
128	1379	768	176512	98304	4612	2533	973	542
256	700	378	179200	96768	2572	1413	487	271
512	512	512	177664	99840	1427	767	244	136
1024	1024	1024	181248	101376	1024	1024	122	68
2048	2048	2048	180224	96256	2048	2048	61	34
4096	4096	4096	180224	90112	4096	4096	31	17
8192	8192	8192	188416	90112	8192	8192	16	9
16384	16384	16384	163840	114688	16384	16384	8	5

VITA

Seraj Ahmad was born in Jaunpur, India in July, 1977. He completed his Bachelor of Technology degree in computer science and engineering from the Indian Institute of Technology, Guwahati, India in May, 2000. He subsequently worked for three years as a software engineer before beginning his graduate studies as a computer engineering major at Texas A&M University in the Fall of 2003 and receiving his M.S. degree in December 2005. His research interests are in the field of embedded architecture, VLSI logic synthesis and physical VLSI algorithms. He can be reached at the following email address: seraj@tamu.edu.

Permanent Address:

C/O Saeedi Clinic,

Mal Godam Road,

Jaunpur - 222001,

Uttar Pradesh,

India.

The typist for this thesis was Seraj Ahmad.