UNIVERSITY OF CALIFORNIA

Los Angeles

Scaling IP Database Size using Trees of Content Addressable Memories

A thesis submitted in partial satisfaction
of the requirements for the degree Master of Science
in Computer Science

by

Victor Eduardo Rios

2021

ABSTRACT OF THE THESIS

Scaling IP Database size using Trees of Content Addressable Memories

by

Victor Eduardo Rios

Master of Science in Computer Science

University of California, Los Angeles, 2021

Professor George Varghese, Chair

Due to costs, TCAMs (ternary content addressable memories) were once seen as an impractical solution for performing fast IP lookup. Thanks to modern improvements, TCAMs have been reintroduced into the market, notably by Barefoot, as a practical resource to be included on specialized chips. Given that TCAM is a premium, the included amount is limited and supports moderately large datasets but still fails to scale to larger datasets such as backbone routers or datacenters. This thesis proposes that TCAM resource allocation can be reduced, therefore allowing larger datasets to be supported, by using a tree of smaller TCAMs as opposed to a single large TCAM. Furthermore, a method for building the tree and finding an optimal fixed stride are presented. The results show that TCAM usage is reduced at the cost of additional RAM usage and that this tradeoff can be tuned to meet different needs.

The thesis of Victor Eduardo Rios is approved.

Yuval Tamir

Lixia Zhang

George Varghese, Committee Chair

University of California, Los Angeles

2021

To my father, for providing me the opportunity and loving me with everything he had,

To my mother, for always believing in me and supporting me through thick and thin,

And to George, for being a great friend and guiding me through the chaos.

TABLE OF CONTENTS

List of Figures

List of Tables

CHAPTER 1

Introduction

In this chapter, we motivate the issue of poor Ternary Content Addressable Memory allocation for Internet prefix lookups, clearly state the problem and proposed solution, and provide an overview for the structure of this thesis.

1.1 Motivation

An issue that has plagued the Internet since the dawn of Classless Internet Domain Routing (CIDR) is performing IP lookup at wire speeds. This issue has been overcome and reexamined countless times as both the demands of the network and the solutions that address those demands evolve with the progression of science and technology. During one episode of this conflict, the use of ternary content addressable memory (TCAM) was proposed because of its ability to search a table in parallel as well as its implementation of wildcard (aka *do not care*) bits.

In order to realize this behavior in hardware, additional transistors are required per bit, leading to increased area costs, and most transistors must be active during the search, leading to increased power costs [1]. These issues are so well known that most people who are familiar with TCAM can immediately note them as the disadvantages of using TCAM. The costs are not so high that TCAM is unusable, but for situations where chip real estate or power usage are critical, as in the Internet core, TCAMs may not be a viable solution. This led to the research and development of IP lookup solutions based on software and algorithmic techniques [1][2]. In the meanwhile, TCAM research and development was aimed at mitigating its two big drawbacks [3][4].

When we jump to the present, TCAMs have now been reintroduced to the market, notably by Barefoot, as configurable resources on powerful networking chips. The disadvantages of TCAM still exist, but thanks to modern improvements, the reduced costs justify the allocation of TCAM resources so that fast table lookups are an option.

Sadly, the previously mentioned issue of performing IP lookup at wire speed has begun resurfacing as the size of IP databases continues to grow to millions in the Internet core. To be clear, the amount of TCAM available on modern chips is capable of supporting moderately large data sets and for larger data sets, slower algorithmic solutions are still an option. The issue comes from enterprises and devices with very large data sets, such as a backbone router or a datacenter router, that want to utilize fast TCAM lookups. Today, they simply cannot use TCAMs because the available resources are not enough to store their entire database in a single TCAM table. This is the problem my thesis tackles.

1.2 Problem Statement

Problem Statement: The use of a single TCAM table to support all the entries in a data set does not scale well in terms of hardware resources as many ternary bits are wastefully allocated, primarily due to the redundancy of storing shared prefixes multiple times. Because of this waste, practical solutions to large data sets cannot afford to use TCAM.

Proposed Solution: Due to modern technological advances, it is possible to implement a *tree of TCAMs* on pipelined architecture with the same output behavior as a single large TCAM. We show in this thesis that a trie of TCAMs requires less TCAM in exchange for additional SRAM. This can allow larger IP prefix databases to be supported with a given amount of TCAM resources.

1.3 Thesis Road Map

I begin by covering contextual background information in Chapter 2 that facilitates the feasibility of my proposed solution. This includes information on ternary content addressable memory (TCAM), the reconfigurable match table (RMT) architecture, and existing trie techniques. The problem is described in detail in Chapter 3 and a simple example highlighting how TCAM can be inefficiently allocated is shown. In Chapter 4, the proposed solution is described and methods for building the tree and finding an optimal fixed stride are given. Chapter 5 presents a theoretical analysis of worst-case storage requirements followed by experimental results and analysis obtained from the simulation of a large BGP table in Chapter 6. Finally, Chapter 7 presents conclusions and considerations for the future.

CHAPTER 2

Background and Related Work

In this chapter, we review technologies and ideas that enable the implementation of a tree of TCAMs. The information reviewed includes ternary content addressable memory, reconfigurable match table hardware architectures, and earlier trie-based solutions to IP lookup. We also take a look at related work and highlight the differences of this thesis.

2.1 Ternary Content Addressable Memory (TCAM)

Ternary Content Addressable Memories (TCAMs) are a form of associative memory that is implemented as hardware. During use, a search key is provided and is compared against an entire table of stored keys in a single clock cycle. On a match, the address of the matching entry is returned. Due to the ternary nature of stored keys, it is possible for there to be more than one match, in which case a priority encoder is used to choose which address is returned (implemented as longest prefix match when keys are sorted by length). There is also typically a hit signal that flags the case when no match is found. Using the returned result, a decoder is used to index into RAM in order to decide what action should be taken [3]. Table 1 shows a simple routing table and Figure 1 shows an abstract TCAM-based implementation of Table 1.

TABLE I
EXAMPLE ROUTING TABLE

Entry No.	*Prefix (Ternary)*	*Output Port*
1	010**	A
2	1101*	B
3	110**	C
4	10101	D

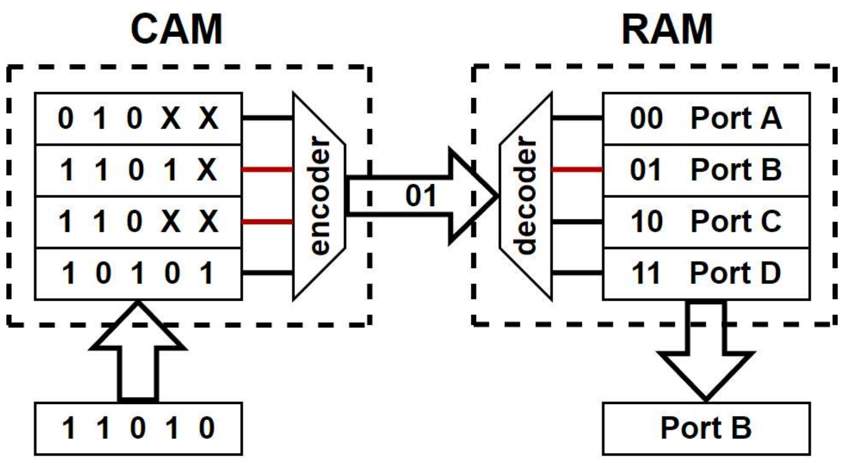


Figure 1: TCAM-base implementation of Table 1. In this case, the search key 01101 is provided, entry 2 is matched, and the resulting action is to place the packet on output Port B

The main drawbacks of TCAMs are their area and power costs. Binary content addressable memory (CAM) already suffers from inflated area costs due to the additional transistors required to perform comparison on every bit, but TCAMs increase area costs further by requiring additional transistors for the mask bits needed to implement wildcard matching [1][3]. Power costs are inflated because large areas of the TCAM must be active in order to perform parallel search within a single clock cycle. Much research and development has been targeted at mitigating these two drawbacks.

Even though TCAM typically has an area cost that is 6-7x larger than that of an equivalent amount of SRAM [1], continued development in transistor manufacturing has allowed products with up to 8.4 Mb of on-board TCAM to be manufactured. Regarding power costs, methods have been introduced to reduce the amount of power required per bit as well as reducing the portion of memory that is active during a search [3]. For these reasons, TCAM is now included on powerful chips as an optional limited resource to be utilized for fast lookups.

2.2 Reconfigurable Match Tables (RMTs) on a Pipe-lined Architecture

Given the recent rise in popularity of Software Defined Networking (SDN), there has been interest in the design and production of switching chips that can support the programmability and adaptability offered by this new paradigm. SDN champions the idea that the control plane and forwarding plane should be physically separated from each other. Under this idea, the control plane is implemented as a centralized controller that is used to program the forwarding plane through an open interface, such as the popular OpenFlow [5].

The network has always had performance demands and the addition of programmability has been a tough transition due to the innate conflict between these two features. Implementations that try to include both speed and programmability are either too slow or expensive for practical networking application or their programmability is too limited or specialized for standardized adoption. Fortunately, a RISC-inspired pipelined architecture known as the RMT (reconfigurable match table) model was proposed in 2013 as an implementation for flexible switching chips that could support OpenFlow programmability at a hardware level [6].

The OpenFlow interface is based on an approach known as "Match-Action", which can be summarized as using a subset of packet bits to match against a table and then using the match result to specify a corresponding action to take on the packet. The naïve approach of using a single match table for general packet processing presents an impractical design due to the excessive amounts of entries generated from trying to match a plethora of fields in a single iteration.

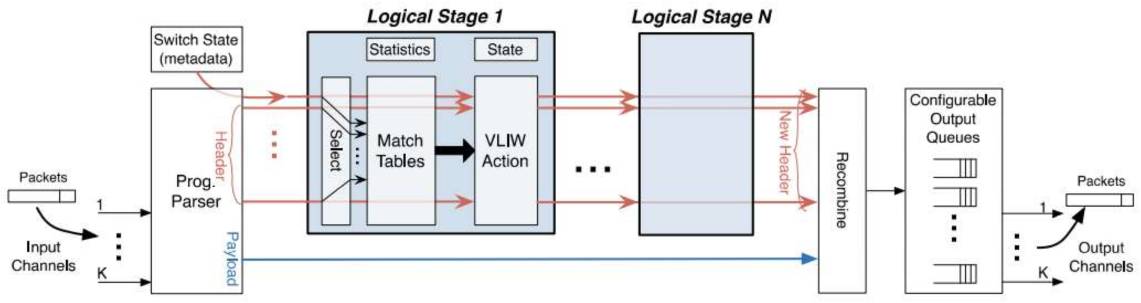
Orthogonal fields would lead to large amounts of wildcard bits and dependent fields would require many entries to cover the cartesian product of both fields. This naturally led to the use of multiple match tables that are pipelined. This approach allowed different tables to be used for different fields as well as allowing one stage of processing to be dependent on a prior stage of processing. The main issue with this model was that existing implementations had table and pipeline characteristics fixed at fabrication, as well as a limited set of specified actions that could be performed on these tables.

This lack of flexibility was not aligned with SDN's interest in flexibility and programmability, so the RMT model was proposed as a practical design that was congruent with SDN's goals. RMT provides features that facilitate flexibility and notable amongst them are 1) the ability to reconfigure the number, topology, widths, and depths of match tables, limited only by an overall resource limit on matched bits and 2) A minimal set of primitive RISC-like instructions to implement various desired actions at fast speeds in heavily pipelined hardware [6].

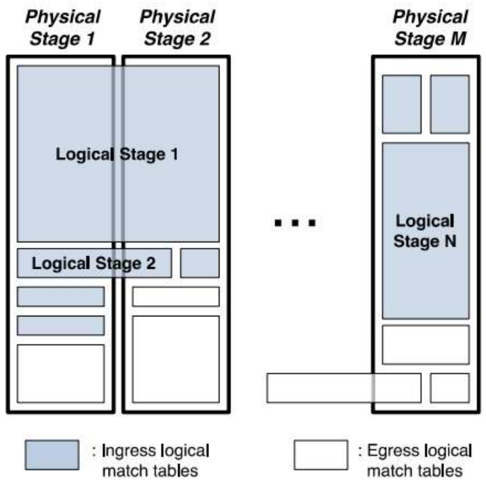
The RMT pipelined architecture proposed by *Forwarding Metamorphosis* [6] consists of; a parser that produces a packet header vector, a series of logical stages that perform Match-Action behavior based on modifiable fields in the vector, a recombination block to reattach packet headers to their packet, and output queues that can be configured with different queuing disciplines. Figure 2, taken from the *Forwarding Metamorphosis* paper, shows the proposed RMT architecture model.

As is seen in Figure 2b, the proposed real implementation consists of M pipelined physical stages that N logical stages can be mapped to. Logical stages are mapped to one, multiple, and/or fractions of physical stages and different independent logical stages can be mapped to the same physical stage(s). This separation of physical and logical is because the resources required by a logical stage can vary significantly while fabrication requires that a fixed set of resources be given to a fixed number of stages.

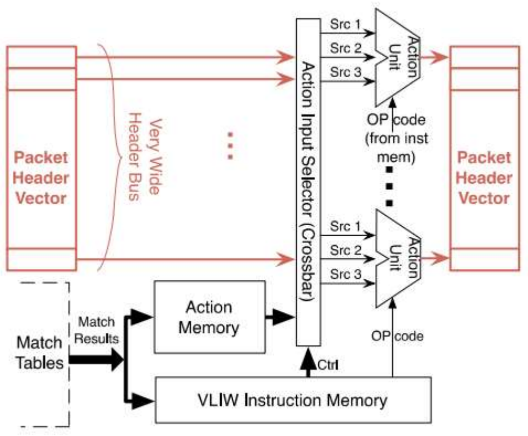
The choice of physical stages M is not arbitrary, a larger number would require more hardware overhead (wiring and powering) while a smaller number would lead to potentially larger waste of resources since a single logical stage requiring few resources could waste up 1/Mth of the required resource. The chosen number of physical stages for the chip design proposed in the paper was 32 since it provided a nice balance between both issues.



(a) RMT model as a sequence of logical Match-Action stages.



(b) Flexible match table configuration.



(c) VLIW action architecture.

Figure 2: RMT Architecture model

Entries in a table have additional memory associated with them: action memory, instruction memory, size of action input, a next table address and pointers to the first two. Action memory holds the arguments that are used for a given instruction and the size of action memory is dependent on the behavior being implemented. In the case of a single TCAM vs a Tree of TCAMs, there should be no difference in the size of action memory since both implementations possess the same set of possible next hops. The size of action input is used to determine how much memory should be read from action memory. Note that this info can be made obsolete if all action inputs are the same size, which is the case for IP lookup. Instruction memory is the memory dedicated to holding the primitive RISC-like instructions and is dependent on the number of instructions provided by the chip, not on the behavior being implemented.

The size of an action memory pointer is dependent on the size of a behavior's action memory and, similarly, the size of an instruction memory pointer is dependent on the number of instructions needed to realize that behavior (for example if only 4 of the offered actions are needed then the pointer should be of size 2 bits). Next table address is like a pointer because it provides the location of the next table that is matched against. Its size is dependent on the number of tables implemented on the chip.

Although the size of the first 2 pointers is the same for both a single TCAM and a tree of TCAMs, the number of pointers is larger in the case of a tree since a pointer is needed for every level of the tree. Similarly for next table address, the amount of these pointers is larger since one is needed at every level of the tree. All these values represent the overhead required for an entry in a table and this overhead is implemented using SRAM.

Each logical stage begins by using a selector to select which fields in the packet header vector are used as a search key for the table match. A match result returns a pointer to action memory, the size of action input, a pointer to instruction memory, and a next table address. These are used, along with the packet header vector, to provide input to action units, as is seen in Figure 2c. There is an action unit for every field in the packet header and although this may seem excessive, there are many protocols that require various fields to be modified at once (although in the case of IP lookup, that is not needed) and the cost of using so many action units is small when compared to the match tables [6]. Afterwards, the potentially modified packet header vector is taken to the next logical stage and the process repeats.

This RMT model architecture provides an ideal setting for a tree of TCAMs to be implemented on since every level of the tree can be thought of as a logical stage such that later levels of the tree are dependent on previous levels of the tree. Every logical stage either directs the lookup to continue to the next logical stage or terminates the lookup and returns the next hop.

In the presented model, a parser graph and table graph would be needed to configure the chip, but these have no significant differences between a single TCAM and a tree of TCAMs. The parser graph should not change, and a table graph would transform the single IP lookup table to a tree of IP lookup tables. The concrete chip design proposed in this paper has hardware specifics, but those are not considered since they can be changed and most likely have since the initial publishing of the paper. What is important is the RMT model and RISC-like pipeline architecture presented in the paper.

2.3 Trie-Based Solutions

The main idea presented in this thesis is motivated by the data structure known as a trie (also known as a prefix tree). A trie is a type of tree data structure where the position of a node in the tree is determined by its value's hierarchal position in the namespace. The root typically represents the empty string and all the children of a given node share the same prefix, namely, the current node's value.

This data structure is typically used to search for strings but can be used to search any hierarchical namespace such as the IP address space where it is used to search IP prefixes. Figure 3 shows the structure of a uni-bit trie for the database presented in Table 2. A uni-bit trie is a trie where at every node there are only 2 possible children, a 1 and 0 child. A uni-bit trie is a great starting point but the issue is that it requires 32 memory accesses in the worst-case for IPv4's IP lookup. This number of memory accesses takes too long to allow packet forwarding at wire speeds.

The natural extension to uni-bit tries is multi-bit tries. In a multi-bit trie, instead of looking at a single bit, a stride of bits is used to find the next node. Figure 4 shows the structure of a multi-bit

TABLE II
ANOTHER EXAMPLE ROUTING TABLE

Entry No.	*Prefix (Ternary)*	*Output Port*
1	1*****	A
2	1000**	B
3	10001*	C
4	10010*	D
5	100110	E
6	100111	F

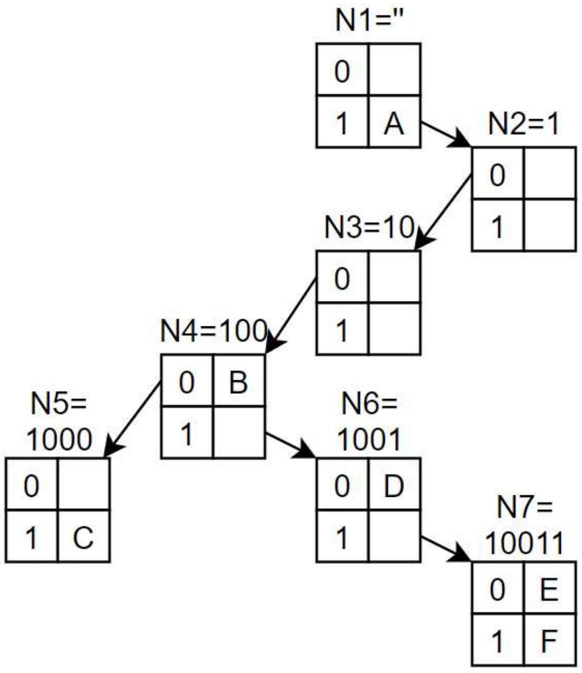


Figure 3: Uni-bit Trie for Table 3

trie for Table 2. Note that the tree uses a stride list of 3-3. By increasing the radix of the tree, we decrease the height of the tree. This reduces the number of memory access that are required to search the tree. The drawback of using multi-bit tries is that prefix expansion is required.

During prefix expansion, prefixes are expanded so that they align with the strides that are being used for search. This is required due to RAMs inability to perform wildcard matches. For example, in Figure 4, prefix 1 from Table 2 was expanded into four prefixes ranging from 100 to 111.

Additional logic is required to handle collision during expansion, such as when expanding entries 2 and 3 of Table 2. Both entries produce the expansion 100011 but entry 3 takes priority since it is the longer prefix.

The use of multi-bit tries represents trading memory for time, namely that additional prefixes are generated in order to reduce the number of memory accesses needed [1]. This paper deals with comparing a single TCAM to a tree of TCAMs that form a trie data structure. A major difference between using TCAM as opposed to SRAM for the nodes is that prefix expansion is avoided due to a TCAM's ability to match wildcards.

Furthermore, multi-bit tries are typically compared to uni-bit tries but, in this case, the multi-bit trie is compared to a single TCAM. Although the former comparison was motivated by a trade of increased memory for reduced speed, the latter comparison is motivated by a trade of decreased CAM memory for increased RAM memory. Increases in latency are not considered since IP lookup is typically not the bottleneck during packet forwarding.

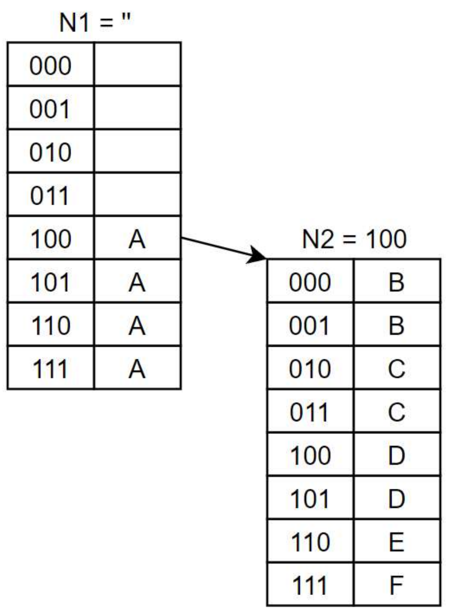


Figure 4: Multi-bit trie for Table 2

2.4 Coolcams and Related Work

The use of a tree structure to implement IP lookup using TCAM is not completely novel and has been considered before. A paper titled *Coolcams: power-efficient TCAMs for forwarding engines* [7] presented the idea of using a 2-level tree of TCAM in order to reduce the power costs associated with using the technology. With that being said, there are significant differences between the ideas presented in *Coolcams* and the ideas presented in this thesis.

First of all, the primary goal of *Coolcams* was to reduce power costs whereas the primary goal in this thesis is to reduce storage costs. Although these 2 metrics are related, they are not identical and because of this, there are differences in the methods and techniques that are used when implementing the tree. Furthermore, the *Coolcams* paper was restricted to only 2 levels whereas this thesis makes no definitive restriction on the number of levels a tree can have. Lastly, at the time of *Coolcams* publication*,* the idea of the RMT pipelined architecture was not around. The existence of this idea, and its lack thereof, also had an influence on the methods and techniques used to implement the TCAM tree.

More recent work related to this topic also exists. A patent issued in 2019, after the creation of the RMT pipelined architecture, claimed 2 algorithmic approaches for doing longest prefix match on a programmable switch [8]. One of these 2 ideas made use of a tree that involved TCAM, but there were still significant differences. One proposed idea was similar to *Coolcams* in that it limited the tree to 2 levels, but a major difference was that the second level was completely implemented using SRAM, not TCAM. Furthermore, the ideas presented in the patent were also geared towards reducing power consumption, not storage costs. These differences influenced the methods and techniques used for implementing the tree.

CHAPTER 3

The Problem

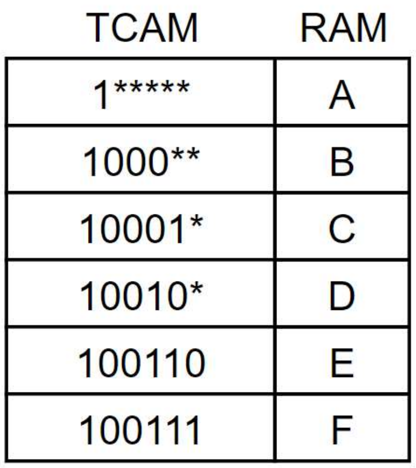
In this chapter we reintroduce the problem statement and provide an example that illustrates how using a single TCAM can lead to wasted resources.

3.1 The Problem

Problem Statement: How can we break up a single large TCAM into multiple smaller TCAMs connected by pointers in SRAM so that we can scale to a large IP lookup database that cannot be supported by the single large TCAM?

3.2 An Example

The major source of waste comes from needing to store a shared prefix multiple times, one for every entry that contains that shared prefix. Figure 5 represents how Table 2 is stored using a single TCAM (the associated return value is not stored using TCAM, but RAM). Note how the shared prefix 100 is stored 5 times, wasting 15 ternary bits for 3 bits of information.



TCAM	RAM
1*****	A
1000**	B
10001*	C
10010*	D
100110	E
100111	F

Figure 5: Single TCAM for Table 2

Although the waste appears small in the example, keep in mind that this example was for a database with only 6 entries of width 6. Real backbone routers are nearing 1 million entries as the internet continues to grow every day [9]. Using the IPv4 address space, a single initial prefix of size 8 can be written thousands of times, wasting tens of thousands of ternary bits in the process.

Another source of waste occurs due to the need to store shorter prefixes as full entries. Backbone routers typically hold prefixes of various lengths, with a large density spike occurring at 24. The need to store these prefixes as 32-bit entries leads to a large number of resources being allocated as wildcard bits. Of course this is not completely terrible, recall that the ability to do wildcard matching is one of the big advantages of using TCAM.

It begins to become a noticeable source of waste when it approaches the scale of entries scene in backbone routers and other large data sets. In these domains, hundreds of thousands of entries can be of size 24 and below and having all these entries stored as full 32-bit entries leads to hundreds of thousands of ternary bits being used as do not care bits. This thesis hopes to alleviate these two problems using a tree-like approach.

CHAPTER 4

The Tree of CAMs Solution

In this chapter we go over the proposed solution and an example illustrating how TCAM resources can be saved. We then go on to introduce algorithms for building the tree, searching the tree as well as insertion and deletion. Lastly, we discuss an algorithm for finding an optimal set of strides to construct a fixed stride tree.

4.1 The Solution

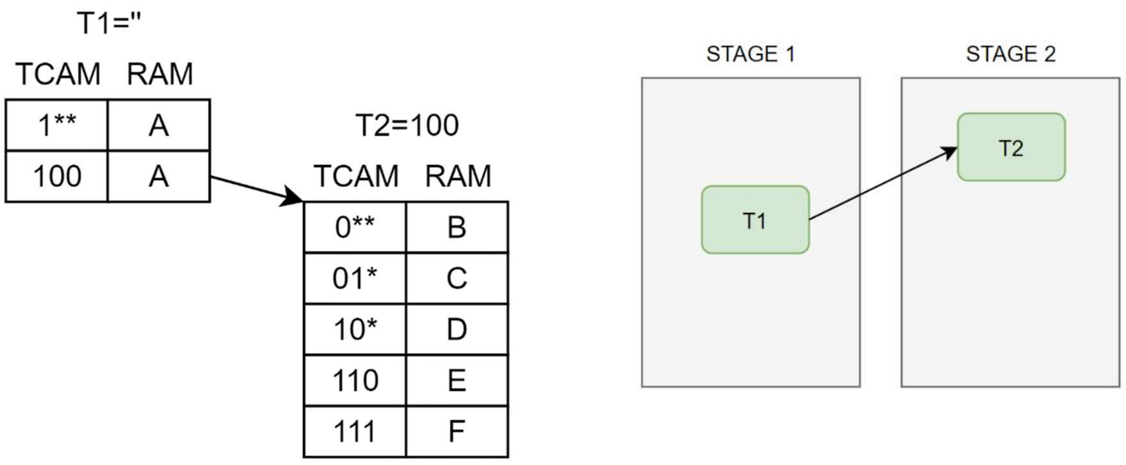
Proposed Solution: Due to modern technological advances (e.g., RMT), it is possible to implement a pipelined tree of TCAMs with the same output behavior as a single large TCAM but that uses less overall TCAM, allowing larger data sets to be supported for a given amount of TCAM.

4.2 An Example

Figure 6 depicts the use of a tree of TCAMs to store the database from Table 2, the tree is built using a 3-3 stride. By using a tree approach, we can reduce the total amount of ternary bits needed from 36 to 21. By comparing Figure 6 to Figure 5, we see that most savings come from avoiding the repetitive storage of the shared prefix 100. As mentioned previously, there was also additional savings from not needing to store entry 1 as an entire prefix, but these are an additional benefit of using the tree structure, not the primary motivation for this implementation.

4.3 Tradeoffs

Note that although this implementation benefits from a reduction of required TCAM resources, there are costs associated with using a tree. The first is additional RAM dedicated for the pointers needed to organize the tree, which corresponds to the pointer seen in Figure 6. The second cost is



a) Trie for table 2

b) Trie mapped to pipeline stages

Figure 6: Tree of TCAMs for table 2

additional RAM dedicated to store an action at every level of the tree, which corresponds to the entry for 100 (the value of A is obtained by from 1**, but it can be its own value as well since having a prefix of 100* is possible).

There is also an increase in latency since a round of TCAM-based match-action behavior is required for every level of the tree as opposed to a single round, but this is not a concern since IP lookup is not typically the bottleneck when it comes to packet forwarding. Furthermore, this structure could reduce power costs since power draw is proportional to table size, and the tree approach searches less TCAM per search then a single table does. Lastly, throughput is not a concern since this tree is pipelined on an RMT pipeline architecture.

4.4 Building the Tree

Figure 7 presents the pseudocode for building a fixed stride tree with a depth-first approach. Provided a list of strides and a database of IP prefixes, a tree is built by recursively attempting to add a prefix entry to a table. Case 1 refers to when the prefix ends in the current node, in this case a sanity duplication check is done. If it passes, the entry is added to the current node, and we move onto the next prefix. Case 2 refers to when the prefix does not end in the current node, in this case a stub key is formed. We first check if a stub entry exists and if not, we add one and give it the return value of the stub's longest prefix match in the current table. We then check if a child node exists for that stub and if not, we add one. We then shorten the prefix by removing the stub portion and try again at the child. The tree is built after all the prefixes in a database have been inserted.

In order to configure a reconfigurable RMT pipeline, a table flow graph is required to express the match table topology. The process of building a general table flow graph is beyond the scope of this paper, but we are responsible for building the portion of the table flow graph that represents the tree of TCAMs needed for IP lookup. The tree of tables replaces a single table in the table flow graph by having all input flows point to the root and have all output flows point to the existing successors of the replaced single table.

The generated tree can be abstracted to a tree of tables for the sake of the table flow graph. These tables replace the single table in the table flow graph as previously mentioned. The process of mapping the tree to memory on a chip is handled by a compiler, but such a compiler is beyond the scope of this paper, so no such compiler is provided.

4.5 Search, Deletion, Update, and Insertion

Once the tree has been built and mapped onto the RMT pipeline, the process of search is straightforward and similar to searching a multi-bit trie. The main difference is that instead of having to search an SRAM table that represents a trie node, a TCAM table that represents the same trie node is searched. This potentially leads to lower latency per node given a TCAMs single cycle parallel search.

The pseudocode for searching the tree is seen in Figure 8. Given that an RMT pipeline uses a Match-Action approach, the pseudo-code is a match followed by an action based on the returned results. The search begins at the root table and at every table along the search the best matching

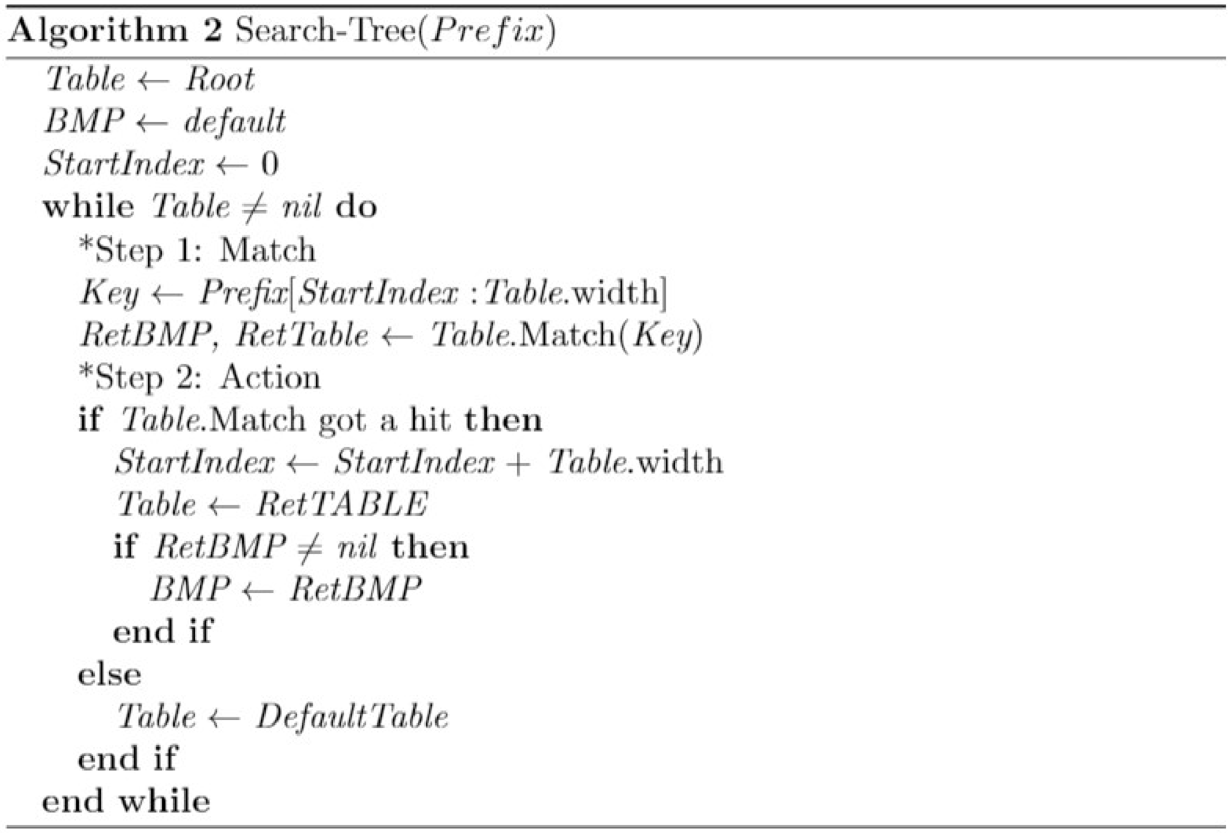
Algorithm 1 Build-Tree($Strides, SortedDatabase$)

```
*Note that an entry is of the form (Key, Next Hop, Node Pointer)
$Root \leftarrow$ Table of width $Strides[0]$
for all ($Prefix$, $NextHop$) pair in $Database$ do
  $Node \leftarrow Root$
  $Key \leftarrow Prefix$
  $Depth \leftarrow 0$
  while $true$ do
    *Case 1: When the prefix ends in the current node
    if Size of $Key \leq Strides[Depth]$ then
      if $Node$ has entry for $Key$ then
        if $Node$.Get-BMP[$Key$] $\neq nil$ then
          Report Duplicate Prefix Entry
          Continue
        else
          $Node$.Add-BMP($Key, NextHop$)
          Continue
        end if
      else
        $Node$.Add-Entry($Key, NextHop, nil$)
        Continue
      end if
      *Case 2: When the prefix continues to next node
    else
      $Stub \leftarrow Key[0{:}Strides[Depth]]$
      if $Node$ has no entry for $Stub$ then
        $Node$.Add-Entry($Stub$, $Node$.LPMatch($Stub$), $nil$)
      end if
      if $Node$.Get-Child($Stub$) $= nil$ then
        $Node$.Add-CHILD($Stub$, $Strides[Depth+1]$)
      end if
      $Node \leftarrow Node$.Get-CHILD($Stub$)
      $Key \leftarrow Key[Strides[Depth{:}]]$
      $Depth \leftarrow Depth+1$
    end if
  end while
end for
```

Figure 7: Build-Tree Pseudocode

prefix is updated on valid hits. If at any point along the path we cannot find a match, the search is redirected to a singular default table. The reason this default table exists is due to the insertion of new prefixes.

Deletion, update, and insertion are nearly identical to search except that upon termination you delete/change/insert the desired entry. However, there are extra considerations when it comes to insertion. Note that although maintaining a TCAM table sorted under insertion and deletion is not a simple task, this paper does not cover how to maintain a single table, but how to maintain the tree structure in a pipelined architecture. The maintenance of TCAM tables is an actively studied topic and methods have been developed to improve insertion times [10]. The issue specific to using a tree of TCAMs on a reconfigurable pipelined architecture is that of overflow.



```
Algorithm 2 Search-Tree($Prefix$)
  $Table \leftarrow Root$
  $BMP \leftarrow default$
  $StartIndex \leftarrow 0$
  while $Table \neq nil$ do
    *Step 1: Match
    $Key \leftarrow Prefix[StartIndex : Table.\text{width}]$
    $RetBMP, RetTable \leftarrow Table.\text{Match}(Key)$
    *Step 2: Action
    if $Table.$Match got a hit then
      $StartIndex \leftarrow StartIndex + Table.\text{width}$
      $Table \leftarrow RetTABLE$
      if $RetBMP \neq nil$ then
        $BMP \leftarrow RetBMP$
      end if
    else
      $Table \leftarrow DefaultTable$
    end if
  end while
```

Figure 8: Search-Tree Pseudocode

Overflow refers to when there is an attempt to insert a prefix into a table, but there is insufficient space to insert the prefix. The most naïve solution is to rebuild the tree with the new database but rebuilding the tree typically requires the chip to be reconfigured and reconfiguring on every update is not a practical solution.

A single TCAM allocates additional resources to ensure new prefixes can be added. This allows the single TCAM to handle various insertions before overflow occurs, needing the chip to be reconfigured to allocate more resources if possible. The issue with a tree structure is that it is not possible to predict what path new prefixes take and whether an inserted prefix requires a new table to be created.

One could try to allocate additional resources to every table on the tree as well as fill out the tree so no new tables would need to be created, but this would be severe overkill as inserted prefixes only effect a single table on the tree. The simpler proposed idea is to create an additional table, of width 32 in the case of IPv4, that acts as a reserve for insertions that cause overflow.

If overflow occurs, then the full prefix is inserted into this reserved table, referred to as the Default Table in the pseudocode. This allows the system to handle overflow and defer reconfiguration until the Default Table is full, similar to how a single TCAM functions. There are 2 drawbacks to using this solution. The first obvious drawback is that additional resources need to be allocated, but this is not a big deal since a single TCAM typically does this as well and the number of additional resources is small when compared to the number of required resources for large databases.

The second drawback is that for all tables on the tree, the default action when no match is found should be to refer search/deletion/update/insertion to the default table. Fortunately, on a pipelined architecture this translates to a single additional stage which is not a significant modification. This modification is reflected in the pseudocode by the use of the Default Table when a match misses.

With this solution the chip needs to be reconfigured less frequently. It does not guarantee that the chip will not need to be reconfigured at some point, but that is okay since the chip would need to be reconfigured routinely in order to reoptimize the tree. Upon rebuilding the tree for optimality, the chip is reconfigured, and the Default Table is emptied, ready to handle new overflow insertions.

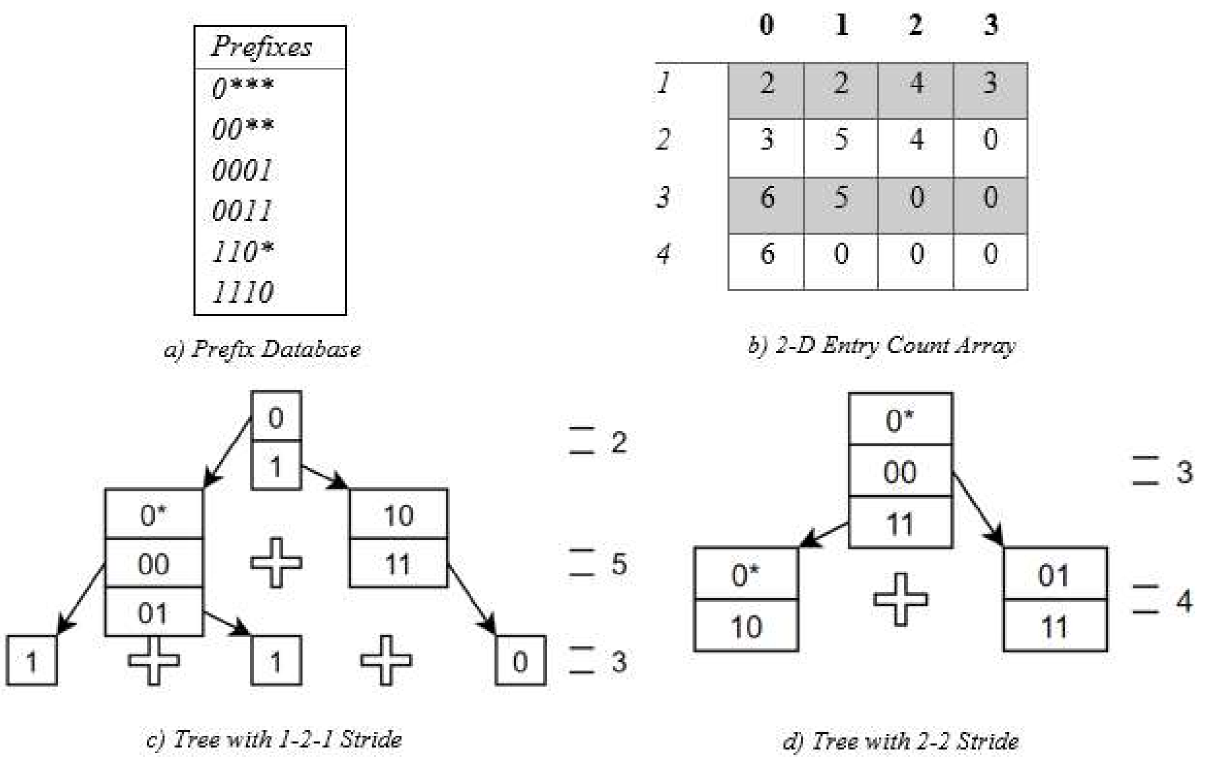
4.6 Fixed-Stride Optimization

As you may have noticed from the build-tree pseudocode, a list of strides(widths) is required to build the tree. Note that the tree that is built is a fixed-stride tree where every table on the same level has the same width, as opposed to a variable-stride tree where different tables on the same level can have different widths. With that being said, the choice of strides effects the number of resources required to store the database.

A simple example of this is seen by comparing Figure 5 and 6 again. The single TCAM can be thought of as a tree with a stride list of 6, whereas the tree in Figure 6 has a stride list of 3-3. As was previously discussed, the tree with a stride list of 3-3 is superior to the tree with a stride list of 6 in terms of storage requirements. The problem of optimization boils down to choosing a set of strides such that the number of resources required is minimal among all possible set of strides. This problem is solved in two steps.

The first step is to use the database to build a 2-D array with information regarding the number of entries present in a level given a starting index and a width. The starting index refers to the position of the bit, relative to a full address, that is used as the first bit for the key. The width of the key simply refers to the size of the key, and therefore the width of a table.

For example, if the 2-D array was indexed with (3, 8), then the returned value would be the number of entries whose search-key is formed by looking at 8 bits, starting from bit 3 of the address. A small example of this 2-D array is shown in Figure 9, where a small 4-bit address space is used. Figure 9c depicts the number of entries required using a 1-2-1 tree. The 3 levels of the tree correspond to the entries (0, 1), (1, 2), and (3, 1) in the 2-D array.



Prefixes
0***
00**
0001
0011
110*
1110

a) Prefix Database

	0	1	2	3
1	2	2	4	3
2	3	5	4	0
3	6	5	0	0
4	6	0	0	0

b) 2-D Entry Count Array

c) Tree with 1-2-1 Stride

d) Tree with 2-2 Stride

Figure 9: Example of 2-D optimization array using 4-bit address space

Building this array involves looking at every prefix in the database and incrementing every (starting index, width) pair that is possible for that given prefix, making sure that no duplicate entries are recorded. The process of ensuring that no duplicate entries are recorded can be expensive and complicated, but if the database is sorted by IP value beforehand, the process becomes as simple as checking the position of the bit where a prefix and its predecessor differ.

This difference tells us that all keys that end before this bit have already been recorded since all the bits before that position are identical in both the current and previous prefix. Figure 10 is the pseudocode that builds the 2-D array for a given sorted database. This 2-D array is then used to find the optimal set of strides given constraints.

Algorithm 3 Build-Optimization-Array(*SortedDatabase*)

```
$PrevPrefix \leftarrow$ empty string
$OptArray \leftarrow$ new int Array[32][32]
for all $Prefix$ in $SortedDatabase$ do
    $DiffIndex \leftarrow$ Position of the first different bit between $Prefix$ and $PrevPrefix$
    for all $StartIndex$ from 0 to length of $Prefix$ do
        for all $Stride$ from ($DiffIndex$ - $StartIndex$) to (32 - $StartIndex$) do
            OptArray[$StartIndex$][$Stride$]++
        end for
    end for
    $PrevPrefix \leftarrow Prefix$
end for
```

Figure 10: Build-Optimization-Array Pseudocode

The second step involves finding the optimal set of strides using the 2-D array and a set of constraints. The constraints include action pointer size, instruction pointer size, next table pointer size, max height, min width, and alpha. The sum of the first 3 constraints is used to determine the binary overhead per entry. These values are determined by hardware characteristics and the memory required to store outputs and instructions.

The max height constraint limits the depth of the tree. This constraint is needed because pipelines have a finite number of stages, or if an implementor would like to limit how many stages IP lookup should take. The min width constraint limits how narrow a table can be. This constraint is dependent on the hardware characteristics of a chip and can be used to avoid having small tables since narrower tables support less entries (i.e., a table of width 1 can only have 2 entries and at that point RAM is a better resource to use).

The last constraint, alpha, is a non-negative decimal that represents the relative worth of binary RAM to ternary CAM (for example a value of 0.5 means that every 2 binary bits are worth 1 ternary bit). This value is dependent on various factors such as the amount of each memory, performance, the power costs etc. Using a tree always leads to more binary overhead, and the choice of strides as well as the depth of the tree effects the amount of overhead needed to implement the tree. Alpha allows us to take this overhead into account by specifying how much additional RAM we are willing to pay in order to reduce the amount of CAM required.

The process of optimization involves using the 2-D array to determine the cost of a given stride list. Using the min width constraint, we iterate through all possible stride lists whose cardinality is less than or equal to max height. Then, using the remaining constraints and the 2-D array, we calculate the cost of each list and find the list with the smallest cost. This stride list represents the optimal list that, when used to generate a tree for a given database, has the smallest required storage cost of all possible trees. Figure 10 represents the pseudocode for finding the optimal stride list given the previous constraints.

4.7 Scope of Solution/Limitations

There are 2 important things to consider. The first is that the solution, both the tree builder and the optimizer, only consider the required storage costs. The actual storage costs are likely to be larger given that reconfigurable chips are typically manufactured using memory units of fixed width and depth. If either the width or depth of a table in the tree is not a multiple of these fixed values, then resources beyond the required amount will be used to implement that node. While the solutions given in this thesis may not lead to improvements if the TCAMs cannot be reconfigured in widths of 8 bits or less for IPv4, similar ideas may be helpful for IPv6 and for packet classification where widths can be larger.

The second is that this solution aims to reduce wasted resources primarily by reducing the number of times a shared prefix is recorded in a table. The larger a database, the larger the likelihood a shared prefix is recorded multiple times. Therefore, this solution is not meant for small databases, but that is not a notable detriment since small databases do not suffer from resource exhaustion.

Algorithm 4 Find-Optimal($APSize, IPSize, NTPSize, MaxH, MinW, \alpha$)

```
$OptCost \leftarrow \infty$
$OptStride \leftarrow [\,]$
$BOverhead \leftarrow APSize + IPSize + NTPsize$
for all $StrideList$ s.t. sum of strides is 32 and all strides $\geq MinW$ do
  if $|StrideList| \leq$ MaxH then
    $BitIndex \leftarrow 0$
    $Cost \leftarrow 0$
    for every $stride$ in $StrideList$ do
      $TernaryCost \leftarrow$ OptArray$[BitIndex][stride]*stride$
      $BinaryCost \leftarrow$ OptArray$[BitIndex][stride]*BOverhead$
      $Cost \leftarrow Cost + TernaryCost + \alpha*BinaryCost$
      $BitIndex \leftarrow BitIndex + stride$
    end for
    if $Cost \leq OptCost$ then
      $OptStride \leftarrow StrideList$
      $OptCost \leftarrow Cost$
    end if
  end if
end for
```

Figure 11: Find-Optimal Pseudocode

CHAPTER 5

Theoretical Worst-Case Analysis

In this chapter we look at the theoretical worst-case gains from using a tree of TCAMs as opposed to a single TCAM. We first build an argument for the worst-case storage cost for a given database and stride list. Then we turn this worst-case cost into a worst-case gain by comparing it against the cost of a single TCAM. This equation is then modified to account for the size of a database, and lastly, we make another modification to account for binary overhead.

5.1 Terminology

Given a database of size *N*, with prefixes for an address space of length *l*, we construct a TCAM tree with fixed strides given by the list *S*. The length of list *S* is *n* ($|S| = n$), which means the resulting tree is of height *n*. We also construct a second list *B* such that

$$b_x = \sum_{i=1}^{x} s_i \quad and \quad b_0 = 0$$

This second list provides the total number of bits that have been searched by the time we have traversed level *x* of the tree (assuming the root is considered level 1). For example, given an address space of length 8 and *S* = (3,3,2), we get *B* = (0,3,6,8) (i.e., at the root we have used 3 bits, at the 2nd level we have used 6 bits, etc.). The initial 0 within *B* is for the purpose of producing a general formula. These variables are used to state what the worst case required storage is for a given database and list *S*. In order to find the worst-case ternary storage required for the tree, we look at each level of the tree, beginning with the root.

5.2 Starting at the Root

The width of the root is given by s_1. In the worst case, the root would be completely full, which entails there is an entry for every possible ternary prefix of size s_1. This is equal to

$$\#ofEntries_1 = 2^1 + 2^2 + \cdots + 2^{s_1} = \sum_{i=1}^{s_1} 2^i = \sum_{i=b_0+1}^{b_1} 2^i$$

$$= 2^{b_1+1} - 2^{b_0+1} - 1 = 2^{b_1+1} - 2^{b_0+1} - 2^{b_0}$$

The first term in the series (2^1) represents the number of entries where all bits but the first are wildcard bits (*'s), the next term (2^2) is for all entries where the first 2 bits are not wildcards, and so and so forth. The last term (2^{s_1}) is for all full prefixes where no terms are wildcards. This series is rewritten as a geometric sum, which we then change from being in terms of S to terms of B. Furthermore, we turn the geometric sum into a closed-form formula. Lastly, we rewrite the constant 1 into terms of B. These transformations may seem unnecessary, but they are done for the purpose of making equations consistent. No values are altered, simply relabeled.

Once we have the worst-case number of entries in the node, all that is left to do is to multiply them by the width of the root, s_1. This gives us a worst-case ternary storage for the root of

$$Worst\ Case\ Cost_1 = \left(2^{b_1+1} - 2^{b_0+1} - 2^{b_0}\right) s_1$$

5.3 Moving onto the Next Level

The next level on the tree has a width of s_2. Similar to before, a table would be completely full in the worst-case, but unlike the root, there is more than one table on this level. The largest number of tables possible is equal to 2^{b_1}, the number of full prefixes possible using b_1 bits (note that entries on a previous level that contain wildcard bits cannot have a child on the next level). In the worst-case every one of these tables would be completely full, which leads to the following derivation of the number of entries on the next level:

$$\#ofEntries_2 = 2^{b_1}(2^1 + 2^2 + \cdots + 2^{s_2}) = 2^{b_1}\sum_{i=1}^{s_2} 2^i$$

$$= 2^{b_1}(2^{s_2+1} - 2^1 - 1) = 2^{b_1+s_2+1} - 2^{b_1+1} - 2^{b_1}$$

$$= 2^{b_2+1} - 2^{b_1+1} - 2^{b_1}$$

This value is once again multiplied by the width of the level, which is equal to s_2. This yields the worst-case required ternary cost for the next level

$$Worst\ Case\ Cost_2 = \left(2^{b_2+1} - 2^{b_1+1} - 2^{b_1}\right)s_2$$

Comparing the cost of the root to the cost of the next level, both equations have the same structure. As opposed to going through every level one by one, it would be simpler to talk about any arbitrary level *x*. Note that the following argument is identical to the previous argument except that the number 2 is replaced with *x* and the number 1 is replaced with *x*-1. The argument is written out for the sake of clarity.

5.4 Arbitrary Level *x*

An arbitrary level on the tree has a width of s_x. This level may contain up to $2^{b_{x-1}}$ tables and in the worst-case every one of these tables would be completely full. This gives a total number of entries, for an arbitrary level *x*, of

$$\#ofEntries_x = 2^{b_{x-1}}(2^1 + 2^2 + \cdots + 2^{s_x}) = 2^{b_{x-1}}\sum_{i=1}^{s_x} 2^i$$

$$= 2^{b_{x-1}}(2^{s_x+1} - 2^1 - 1) = 2^{b_{x-1}+s_x+1} - 2^{b_{x-1}+1} - 2^{b_{x-1}}$$

$$= 2^{b_x+1} - 2^{b_{x-1}+1} - 2^{b_{x-1}}$$

We multiply this value by the width s_x in order to obtain a required ternary cost of

$$Worst\ Case\ Cost_x = \left(2^{b_x+1} - 2^{b_{x-1}+1} - 2^{b_{x-1}}\right) s_x$$

This formula holds for every level of the tree, including the root (this was the reason that b_0 was defined as 0). We now calculate the total worst-case cost for the tree.

5.5 Worst-Case Ternary Cost of the Entire Tree

By adding the cost of every level in the tree, we calculate the cost of the entire tree. This equates to the summation from level 1 to *n* (recall *n* = |*S*|)

$$Ternary\ Worst\ Case\ Tree\ Cost(TWCTC) = \sum_{i=1}^{n} Worst\ Case\ Cost_i$$

Plugging in the formula for level cost we come up with the following equation

$$TWCTC = \sum_{i=1}^{n} \left(2^{b_i+1} - 2^{b_{i-1}+1} - 2^{b_{i-1}}\right) s_i$$

This estimation requires a small modification in order to account for the number of prefixes in a database.

5.6 Considering the Size of a Database, *N*

In the previous construction, it was assumed without mentioning that the size of *N* was very large such that $N > Entries_x$ for all *x*. In reality this may not be the case, which would mean that the number of entries is limited by the value of *N*. More accurately the number of entries at an arbitrary level *x* is limited by the value $N_{b_{x-1}}$, the number of entries whose length is larger than b_{x-1}. This inaccuracy is rectified by the simple addition of a minimum function to the previous equation, so that we now consider the size and prefix length density of a database

$$TWCTC = \sum_{i=1}^{n} \left(\min\left(N_{b_{i-1}}, 2^{b_i+1} - 2^{b_{i-1}+1} - 2^{b_{i-1}}\right)\right) s_i$$

5.7 From Ternary Costs to Gains

We look at the minimum gains by comparing the cost of a tree to the cost of a single TCAM.

$$Ternary\ Gains = Single\ Cost - Tree\ Cost$$

The cost of a single table for a database of size *l* and an address space of length 32 (like the IPv4 address space), is 32*N*. This can also be written as the following series with respect to *S*

$$Single\ Table\ Ternary\ Cost = 32N = \sum_{i=1}^{n} Ns_i$$

This equation has a form similar to our previous equation for the cost of a tree. Comparing the two equations provides us the value for the minimum gains in required ternary storage

$$Ternary\ Gains = \sum_{i=1}^{n} \left(N - \min\left(N_{b_{i-1}}, 2^{b_i+1} - 2^{b_{i-1}+1} - 2^{b_{i-1}}\right)\right) s_i$$

Since the second term is clamped by $N_{b_{i-1}}$, gains are always non-negative when it comes to the required ternary storage. Although that is great for CAM resources, the last modification required is to account for the total cost that includes the binary overhead of implementing a tree.

5.8 Total Cost

The binary overhead to managing a tree includes the pointers that connect parent to child as well as the memory required to store actions and their respective inputs. This modification is reflected in the following formula, which is a simple adjustment of our previous formula

$$TotalWorstCaseTreeCost = \sum_{i=1}^{n} \min\left(N_{b_{i-1}}, 2^{b_i+1} - 2^{b_{i-1}+1} - 2^{b_{i-1}}\right)(s_i + \alpha k)$$

The adjustment is the inclusion of the additional term α*k*, which is the cost of binary overhead added to every entry. The value *k* represents the number of binary bits required for every entry whereas α represents the relative worth of a binary bit to a ternary bit, as mentioned previously when presenting optimization in stride choice. This modification changes the gains formula to:

$$Total\ Gains = \sum_{i=1}^{n} \left(N - \min\left(N_{b_{i-1}}, 2^{b_i+1} - 2^{b_{i-1}+1} - 2^{b_{i-1}}\right)\right) s_i$$

$$- \sum_{i=1}^{n} \min\left(N_{b_{i-1}}, 2^{b_{i+1}} - 2^{b_{i-1}+1} - 2^{b_{i-1}}\right) \alpha k$$

The second term, which represents the costs of implementing a tree, could potentially offset the gains from the first term, leading to net negative gains. This all depends on the choices of *k* and α. The choice of *k* is dictated by the size of pointers and techniques that could be used to reduce binary overhead. Meanwhile, the choice of α is dictated by the relative worth of a binary bit to a ternary bit. This may include several factors such as the amount of each resource, performance requirements, power budgets, etc. The addition of these two terms allows the cost evaluation to be more tailored for specific specifications and designs.

This equation still shows that, even in the worst case, the amount of required ternary storage for a tree is at least as bad as a single table, but now it also includes the costs of implementing such a structure. Namely that additional RAM is needed, and if this overhead is sufficiently high, then it may not be wise to construct a tree. Of course, all of this is for a worst-case database of size *N*. Such a database would have the worst possible sharing of prefixes for the entries and, given that the internet is structured in a hierarchical fashion, this is most likely not the case. We now see what the storage requirements of an actual database looks like to see how practical a tree approach is.

CHAPTER 6

Experimental Results

In this chapter, we present the results of building a tree for a real-world database with a significantly large size. This includes building a tree for naïve strides as well as optimized strides. We then review and discuss the results.

6.1 Experimental Setup

Results were gathered using the BGP routing table database for AS65000. Only active entries (those in the FIB) were used. This database is found at https://bgp.potaroo.net/as2.0/bgp-active.html and is updated periodically. The version that was used to gather results was pulled from the database on November 9th, 2021 and had around 900,000 active entries. All experiments were run on a machine with a Windows 10 OS, 32 GB of RAM, and an AMD Ryzen 3900x CPU. Both the tree builder and the fixed-stride optimizer were implemented using C++ (20 standard). All code and the used database can be found at (https://github.com/Mr-Verios/TTree.git).

6.2 Unoptimized Intuitive Fixed-Strides

Although optimization is a great tool to have, we begin by using the builder without first finding optimal fixed strides. There are two primary reasons for doing this. The first reason is to build a reference point to compare against optimized trees. This allows us to not only observe the savings from using simple intuition for stride choice, but also allows us to see the benefits of using optimization.

The second reason is that optimization is always done with regards to certain goals and conditions. In this thesis, our goal was to optimize for storage requirements given constraints that were described in section 4.6 and although the choice of constraints was satisfactory, they are by no means the objectively "best" constraints. If others wish to modify the constraints of optimization or even the goal itself, they are free to do so.

The results presented in this section provide the storage requirements with no specific constraints on tree height and min width. A value for binary overhead is still specified since it is required to calculate storage requirements. A value for α is arbitrarily chosen for presentation purposes.

The intuitive strides that were used are the following: 16-16, 16-8-8, 8-8-8-8, 16-4-4-4-4, 4-4-4-4-4-4-4-4. The intuition for these strides comes from using powers of 2. This was done because it is the most straightforward approach to dividing an address space into smaller components. Note that the second and fourth stride lists were chosen due to the intuition that most of the prefix sharing occurs during the initial portion of entries and the use of a larger initial stride would allow us to shorten the height of the tree.

A binary overhead of 30 bits was used for each entry: 18 for next table pointers, 4 for instruction pointers, and 8 for action pointers. A value of 18 was sufficient to cover the number of tables generated by the all-4s tree, a value of 4 was chosen because IP forwarding does not require a diverse set of actions, and a value of 8 was chosen because modern switches typically do not exceed 256 output ports.

An arbitrary α value of 0.25 was used, meaning 4 binary bits are valued the same as a single ternary bit. Figure 12 shows the storage requirements needed for these strides as well as a single table, which is marked as a 32 stride. Note that the redline that marks the weighted storage cost moves across the orange segment as alpha ranges from 0 to 1. Table 3 provides a summary of the results seen in Figure 12.

The results illuminate an interesting trend. They appear to show that having a large initial stride followed by smaller strides is a great choice for minimizing storage requirements while also

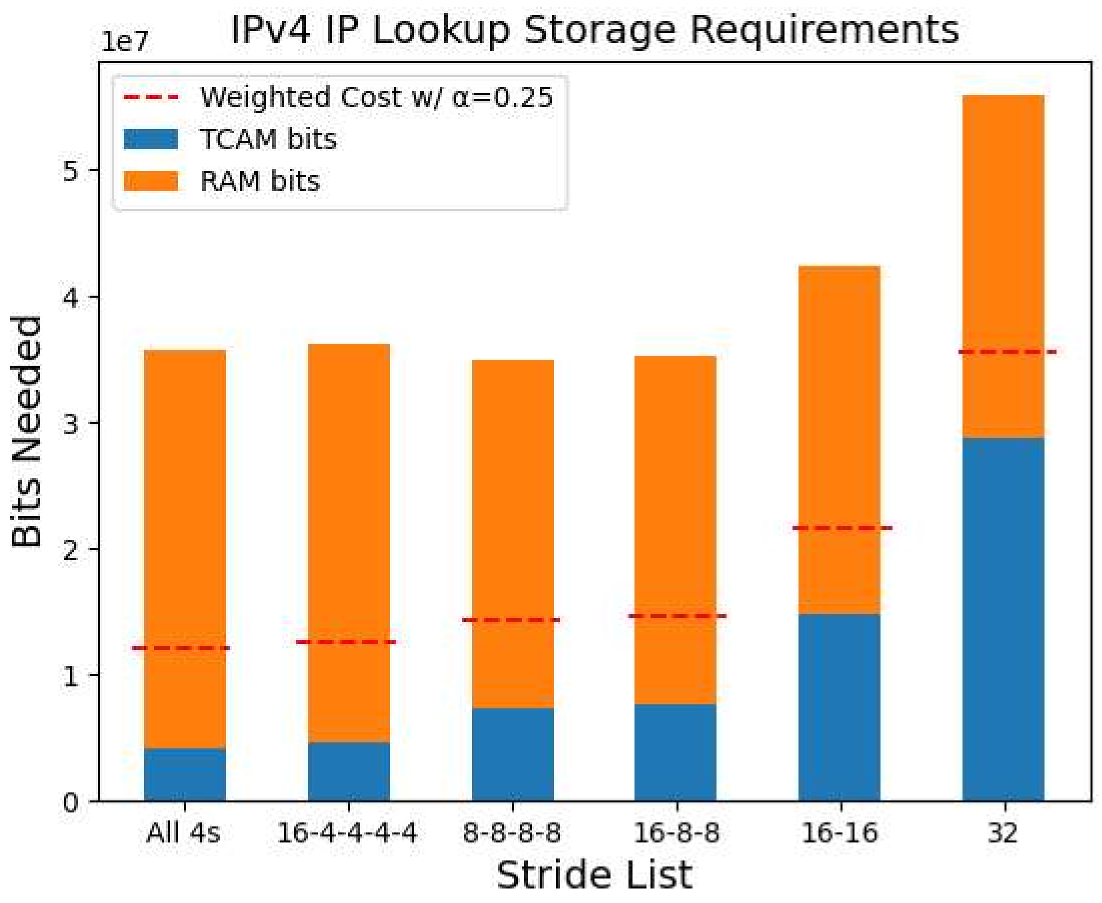


Figure 12: Storage Requirements for Various Intuitive Strides, with α=0.25

TABLE III
SUMMARY OF INTUITIVE STRIDE RESULTS

Stride List	CAM requirements	RAM requirements
32	28,856,192 bits	27,052,680 bits
16-16	14,737,408 bits	27,632,640 bits
16-8-8	7,675,208 bits	27,661,680 bits
8-8-8-8	7,378,080 bits	27,667,800 bits
16-4-4-4-4	4,652,260 bits	31,530,900 bits
4-4-4-4-4-4-4-4	4,215,916 bits	31,619,370 bits

TABLE IV
PREFIX LENGTH DENSITIES OF DATABASE

Prefix Lengths	Percentage Of Prefixes
$<= 8$	$\sim 0.00\%$
$<= 16$	$\sim 2.00\%$
$<= 20$	$\sim 12.20\%$
$<= 22$	$\sim 30.22\%$
$<= 23$	$\sim 41.26\%$
$<= 24$	$\sim 99.84\%$

limiting the height of the tree. A plausible explanation for this may be found by looking at the prefix length density of the database used, which is partially seen in Table 4.

Using this information, it is deduced that the initial drop in storage requirements seen from 32 to 16-16 cannot be attributed to wildcard reduction because the vast majority of prefixes are longer than 16 bits. This means that the initial drop is due to the removal of numerous redundant entries. The next reduction from 16-16 to 16-8-8 is most likely due to the removal of wildcard bits because the vast majority of entries are less than or equal to 24.

The remaining drops are not as clear to interpret, but speculation for their justification is still made. The drop to 8-8-8-8 is very minimal and this may have two justifications: the first is that the initial levels of a tree do not constitute a significant portion of the overall tree cost and the second is that if most prefix sharing occurs in the first 16 bits, then further division(s) of initial shared prefixes provides little to no benefit.

The remaining drops continue this trend, although the use of 4 strides provides visible drops when compared to the use of 8 strides. The justification for using a smaller stride is not 100% clear, but it is likely due to the aggressive exploitation of shared prefixes (the probability of sharing 4 bits is larger than sharing 8 bits) and the forceful removal of wildcard bits (prefixes of length less than or equal to 20 only require 20 bits as opposed to 24 with an 8 stride). The first reason is likely the primary culprit because RAM requirements increased significantly when changing from an 8 stride to a 4 stride, which entails that entries were most likely split across 2 levels as opposed to being shortened to 1 level.

At a glance, it may appear that 16-4-4-4-4 offers the best choice given that it provides the smallest weighted storage requirements using only 5 levels, but there is an issue with using a smaller stride. Both stride lists that make use of the 4-stride possess a significantly larger number of barren tables. For the sake of being precise, we define a *barren table* as a table with less than 6 entries. Both of the trees produced using a 4-stride component have over 50% of their tables be barren tables, whereas the remaining multi-stride trees have around a 33% composition of barren tables.

The reason this is significant is because a barren table represents an inefficient use of resources. They still require a whole clock cycle to search, as well as adding table overhead that cannot be amortized over numerous entries, and this situation is made even worst by the fact that multiple barren tables can form a chain of barren tables (an example of this would be a unique entry that shares no common prefix with other entries).

A simple fix for barren tables would be to create an additional 32-bit TCAM table that is responsible for covering entries that would produce barren tables. This idea is similar to the default table mentioned in chapter 4 section 5 and can in fact be the same table used for both purposes. A method for identifying such entries is not provided but is an interesting addition that should be considered in future work.

The presence of barren tables is not easily fixable, and may not even be addressable, but that does not contradict our goal of exploiting redundancy in shared prefixes. Barren tables are typically produced by entries that do not share prefixes with other entries, and entries such as this are not the target of this thesis. Furthermore, it is intuitive that unique information like this does not lend itself to being easily compressed by most schemes.

With that in mind, it appears that 16-8-8 provides the best intuitive choice for strides. Not only does it reduce TCAM requirements by a factor of 4, but it also does this using only a tree of height 3 and an increased RAM requirement of only ~2.25%. Although these numbers may not be realizable in a real product due to limitations described previously, they present a great starting point for a tree implementation.

6.3 Comparing Real Requirements to Worst Predicted Requirements

Before moving on to optimization, it may be fruitful to compare the real requirements seen from simulation to the worst requirements predicted by theoretical analysis from the prior chapter. Table 5 shows a comparison between real and worst requirements for the strides provided in 6.2. Note that RAM requirements are ignored (α=0) because we would like to focus on CAM requirements.

TABLE V
REAL VS WORST REQUIREMENTS

Stride Lists	**Real CAM Requirements**	**Worst CAM Requirements**
32	28,856,192 bits	28,856,192 bits
16-16	14,737,408 bits	16,236,992 bits
16-8-8	7,675,208 bits	9,178,768 bits
8-8-8-8	7,378,080 bits	8,128,168 bits
16-4-4-4-4	4,652,260 bits	8,804,828 bits
4-4-4-4-4-4-4-4	4,215,916 bits	7,214,528 bits

In order to compare these numbers, it is best to recall the assumptions that were made during theoretical analysis. The first major assumption was that minimal sharing was assumed, meaning that entries would be as distinct as possible from each other. The second assumption was that the number of entries at a given level was limited to the number of entries whose length would permit them to have an entry at that level. This second assumption entails that all unnecessary wildcard bits were removed, which means that any discrepancies between the 2 numbers is likely due to the extent of sharing present in the database.

The third major assumption, which inflated worst requirements, was that all possible prefix combinations for a table were present in the tree. This assumption leads to the counting of redundant entries, which are entries that never match because they are dominated by other entries. On the other hand, this is offset by the fact that trees naturally produce redundant entries. A simple example that can be drawn on paper comes from having three entries (1*,1111,100*) and a simple 2-2 stride. At the root, entry 1 always loses to either entry 2 or entry 3 and will never be used unless those entries are removed. Thankfully, since wildcard entries cannot have children, the effect of this inflation is limited to the current table.

Looking at these details, a conjecture that seems valid is that an initial large stride is a useful intuition for stride choice. This is seen from the discrepancies between rows 3 and 4 as well as rows 5 and 6. For both of these pairs, the difference in their real requirements does not match the difference in their theoretical requirements. Because the difference between real and theoretical requirements is most likely due to the extent of sharing, it is implied that 16 was a good choice to exploit shared prefixes and further divisions are limited in the exploitation they provide.

A second plausible conjecture is that when smaller strides are used, more aggressive exploitation occurs. This is reflected in the table by comparing the differences between columns. As smaller strides are used, the percent difference between real and worst requirements grows. This means that the extent of sharing in the database increased but recall from the previous section that this exploitation of shared prefixes is most likely too aggressive because it causes RAM requirements to increase significantly.

There are 3 other interesting insights that can be discerned. The first is that even in the inflated worst case, there are significant reductions in CAM requirements for large databases. The second is that real databases exhibit a level of sharing that goes beyond the minimum. Lastly, storage requirements may likely be significantly hindered by the production of redundant entries, which is an area that has the potential to be improved.

6.4 Optimized Fixed-Stride

Optimization was performed under 4 sets of constraints, each representing a different scenario an individual may come across. All scenarios had a binary overhead of 30 bits per entry. Table 6 describes the 4 sets of constraints as well as the results of optimization presented in Figure 13.

Case 1 represents a scenario where the height of the tree is the defining constraint on the optimizer. Case 2 is when min width is the defining constraint. Case 3's defining constraint is α and Case 4 represents a situation where there is no defining constraint.

TABLE VI
OPTIMZATION CONSTRAINTS AND RESULTS

Case	Height	Width	α	Stride	CAM requirements	RAM requirements
1	2	4	0.15	18-14	13,866,976 bits	28,853,880 bits
2	32	6	0.15	6-6-6-6-8	5,796,236 bits	28,966,530 bits
3	32	4	0.75	5-4-4-5-6-4-4	5,694,113 bits	29,047,650 bits
4	32	4	0.15	All 4's	4,215,916 bits	31,619,370 bits

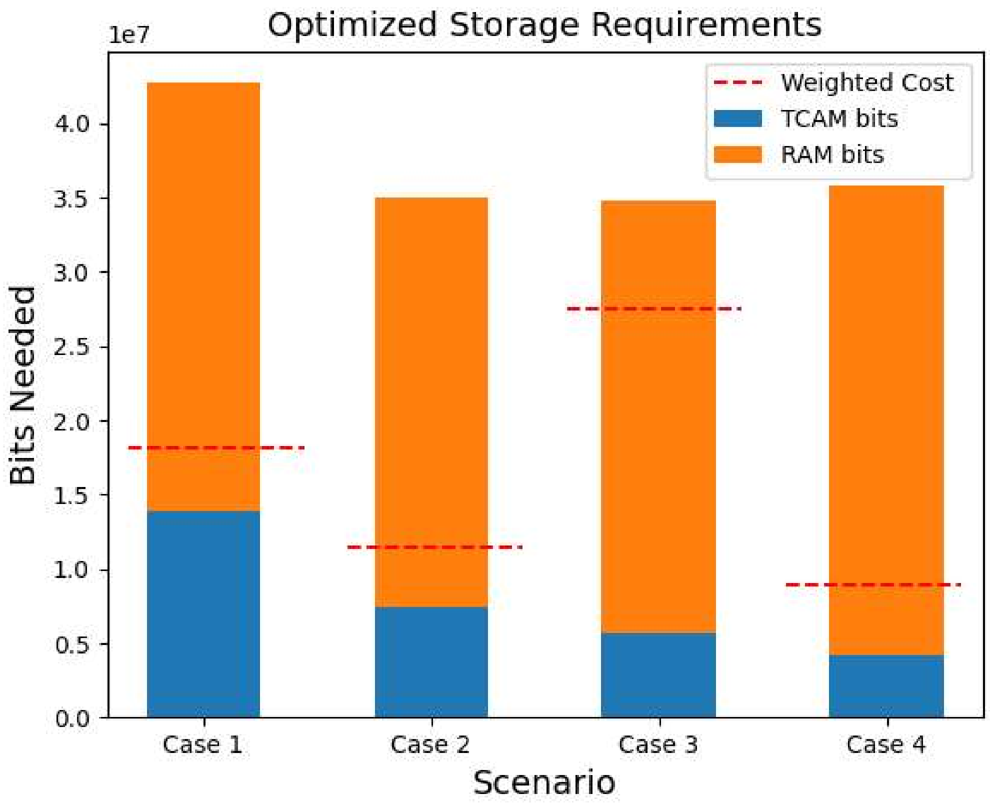


Figure 13: Optimized Storage Requirements for Various Constraints

An interesting insight is that there appears to be a greedy strategy that prefers smaller strides when possible. This is most likely a weakness of the optimizer and not a commentary on the nature of optimal strides. The reason for this is that the optimizer only focuses on total storage requirements and does not consider other factors such as the ratio of barren tables or the cost of using an additional level. This leads the optimizer to choose smaller strides even when they only offer minor improvements at the cost of additional levels and many barren tables. When height is the defining constraint, this is not the case, but that is due to the limited nature of only using 2 levels. Furthermore, even when α is the constraint there are only minor deviations from using the smallest possible stride.

It is interesting to note that using intuitive strides produces similar results in terms of storage requirements and even produces better results in terms of other factors. For example, 16-16 could replace Case 1, 16-8-8 could replace Case 2 and 3, and all 4's is already an intuitive stride but can be replaced by 16-4-4-4-4. All the proposed optimal strides have over a 40% composition of barren tables whereas most of the intuitive strides only have a ~33% composition. Another improvement would be that the intuition of using a larger stride provides similar requirements while also reducing the height of tree.

These results illuminate that when constructing the tree, one should consider all potential factors that affect an implementation, not just constraints and the primary goal. On a different note, it is pleasing to see that intuitive strides perform very well and that using auxiliary information such as worst-case predictions and prefix length densities can help explain and possibly produce intuitive strides.

CHAPTER 7

Conclusions

In this chapter we summarize results and lessons learned as well as discuss limitations on the solutions presented in this thesis and possible extensions for future work.

7.1 Summary of Results

The use of algorithmic techniques to implement IP lookup in RAM based designs has been an established and well-studied solution for a long time now, but this was typically considered as an alternative option to the use of TCAMs. In this thesis we proposed the use of algorithmic techniques as a supplement for TCAMs. This was facilitated by the design of a reconfigurable match action pipeline as well as the corresponding popularity of the SDN movement. The necessity for this approach was driven by the observation that single tables tended to lead to resource exhaustion, and that this was partly due to the inefficient allocation of resources.

Results show that significant reductions in TCAM requirements can be achieved with moderately small increases in RAM requirements. Significant gains can be attributed to both the removal of redundant entries, as well as the removal of unnecessary wildcard bits. The former was primarily seen when partitioning the beginning of an address space while the latter was the primary gain when partitioning the end of an address space. Furthermore, the results show that good intuitive choices can be made for the list of strides used to traverse the tree, and that auxiliary information on the database, such as prefix densities, can help with the choice of strides.

Finally, although optimization was misguided in our approach, it revealed that there are several refinements that are possible to implement. Such refinements include considerations for tree height and the percentage of barren tables created. On a more positive note, the optimization revealed that intuitive choices for strides can provide comparable gains to greedy optimal decisions.

7.2 Lessons Learned

One interesting concept that was seen was that smaller strides lead to more aggressive exploitation of sharing, and this could easily be a bad thing. Although it is true that further partitions reduce TCAM requirements, there comes a point where these splits are not worth doing. Aside from the obvious increase in tree height and RAM overhead, there could be scenarios where a given partition has significantly reduced the number of entries present and further partitions are left with few opportunities for significant reductions to be made.

This phenomenon can be thought as a “rate of reduction”. Partitions near the beginning of an address space have a limited effect on the overall cost because the number of entries present in the first levels of the tree is small compared to later levels. This means that further partitions near the beginning do not provide significant reductions in overall cost.

Partitions near the end of an address space deal with a separate issue. Although there are numerous entries at later levels, these entries are isolated from one another due to having different initial prefixes. Further partitions near the end may lead to significant reductions in some tables but may also lead to the creation of barren tables due to the aggressive partitioning of other tables on the same level. This highlights a difficulty of choosing a stride list when implementing a fixed-stride tree. The use of variable-length strides could lead to partitions that lead to superior reductions from well-placed boundaries as opposed to aggressive partitioning.

Another interesting idea was the issue of overflow and barren tables. The issue of overflow is not unique to a tree since it can occur in a single table as well. The difference is that a single table need only reserve additional entries in order to account for overflow, whereas this approach is not practical in a tree. This is because there are several tables in a tree, and it is not possible to predict which tables will experience overflow.

On the other hand, the issue of barren tables is unique to a tree. Using a single table, all prefixes are represented as a single entry in one table. When using a tree, all prefixes are represented as entries across several tables and some of these entries may inhabit tables with few other entries. Entries that produce barren tables should not be included in the tree because barren tables represent an ineffective allocation of resources.

It was surprising to realize that both issues could be alleviated with the inclusion of an additional table that would hold overflow entries and entries that would produce barren tables. This allows overflow to be handled in a simple manner and the production of barren tables to be avoided. This additional table, previously referred to as a Default Table, acts as a supplement to the tree of TCAMs and can be easily used alongside the tree.

A final lesson was in regard to optimization. The misguided optimization criterion used in this thesis was to find a stride list such that the minimum amount of TCAM resources were used under given constraints. It is clear that other factors must be considered such as the height of the tree and the composition of table sizes.

Under the simple criteria of minimum storage requirements, impractical "optimizations" on stride list were made. An example of this would be adding an additional level to the tree only to save a small amount of TCAM. These impractical "optimizations" may even make allocation worse, such as when a stride list leads to a significant number of barren tables that cannot be implemented effectively.

7.3 Solution Limitations

The most prevalent limitation on the presented solution is that it does not make an attempt to calculate storage costs based on specific hardware characteristics. The provided solution states the number of required resources, but this is not equivalent to the number of used resources. The reason for this is that this thesis treats TCAM as a resource that can be carved up in completely arbitrary units. In reality, TCAMs are typically manufactured in fixed units of a given width and depth (often called a "grain size"), and arbitrary tables are constructed by stitching these units together. Similar to paging, this leads to internal fragmentation that our solution does not consider. Nevertheless, the results and solutions provide a great starting point for this refinement.

The second limitation is that the analysis only works for hierarchical address spaces such as IPv4 and IPv6. The reason for this is that the two major sources of savings, redundant prefixes, and unnecessary wildcard bits, are only prominent in hierarchical address spaces. Furthermore, this solution is tailored for large databases where the extent of shared prefixes is larger. This is not a major issue though because smaller databases do not typically suffer from resource exhaustion.

7.4 Potential Extensions

There are various possible extensions that exist for this work. The first two have already been mentioned and they are: a more comprehensive specification for performing optimization and a refinement on cost calculations that considers hardware specific details. The first could include considerations for performance, table size characteristics, power costs, and ad hoc modifications. The second would tailor this solution for specific existing hardware, providing a more detailed and accurate depiction of the tradeoffs associated with using a tree as opposed to a single table.

Further extensions are motivated by the RAM based alternative that inspired this idea. This includes performing variable stride optimization as opposed to fixed stride optimization. Recall that this means that different tables at the same level of the tree can be of different widths as opposed to the common widths used in this solution. The use of variable stride trees could help deal with the issue of barren tables seen in a fixed stride tree.

Another interesting extension is that some of the tables on this tree could easily be switched out with RAM-based tables, effectively producing a heterogenous tree composed of both CAM and RAM tables. The decision for which tables should be RAM vs CAM could be based on a number of factors, such as the characteristics of the table, statistics gathered on the usage of a given table, or even policy associated with the entries of a given table. Intuitively, if a node has very little prefix expansion (most prefixes "fit" into the stride), it may be worth replacing the CAM node by a cheaper (in terms of power and transistor count) RAM node.

Finally, although ACL databases make use of a multidimensional 5-tuple key, solutions have been developed to produce tree structures that help with packet classification, such as the brilliant HyperCuts algorithm [11]. Such a tree structure could also be implemented on a reconfigurable match action pipeline using TCAMs, which could present potential TCAM savings when compared to using a single table for ACL lists. Although this idea may not be as important to backbone routers, it could prove very beneficial to enterprises with large ACL databases.

REFERENCES/BIBLIOGRAPHY

[1] George Varghese. 2005. 11 Prefix-Match Lookups. In *Network algorithmics: An Interdisciplinary Approach to designing fast networked devices*. San Francisco, CA: Elsevier/Morgan Kaufmann, 233–269.

[2] V. Srinivasan and G. Varghese. 1999. Fast address lookups using controlled prefix expansion. ACM Trans. Comput. Syst. 17, 1 (Feb. 1999), 1–40. DOI:https://doi.org/10.1145/296502.296503

[3] K. Pagiamtzis and A. Sheikholeslami, "Content-addressable memory (CAM) circuits and architectures: a tutorial and survey," in IEEE Journal of Solid-State Circuits, vol. 41, no. 3, pp. 712-727, March 2006, doi: 10.1109/JSSC.2005.864128.

[4] K. Yuvaraj and V. Prabakar, "Low power TCAM using pre-charge match line technique," 2015 International Conference on Advanced Computing and Communication Systems, 2015, pp. 1-4, doi: 10.1109/ICACCS.2015.7324138

[5] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. 2008. OpenFlow: enabling innovation in campus networks. SIGCOMM Comput. Commun. Rev. 38, 2 (April 2008), 69–74. DOI:https://doi.org/10.1145/1355734.1355746

[6] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding metamorphosis: fast programmable match-action processing in hardware for SDN. SIGCOMM Comput. Commun. Rev. 43, 4 (October 2013), 99–110. DOI:https://doi.org/10.1145/2534169.2486011

[7] F. Zane, Girija Narlikar and A. Basu, "Coolcams: power-efficient TCAMs for forwarding engines," IEEE INFOCOM 2003. Twenty-second Annual Joint Conference of the IEEE Computer and Communications Societies (IEEE Cat. No.03CH37428), 2003, pp. 42-52 vol.1, doi: 10.1109/INFCOM.2003.1208657.

[8] Henry Wang. 2019. Algorithmic Longest Prefix Matching In Programmable Switch. Patent No. US 10,511,532 B2, Filed Mar. 21st, 2018, Issued Dec. 17th., 2019.

[9] CIDR Report. CIDR report. (2021, November 12). Retrieved November 12, 2021, from https://www.cidr-report.org/as2.0/.

[10] Shah, D. and Gupta, P., 2000. Fast incremental updates on Ternary-CAMs for routing lookups and packet classification. In: Hot Interconnects.

[11] Sumeet Singh, Florin Baboescu, George Varghese, and Jia Wang. 2003. Packet classification using multidimensional cutting. In Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications (SIGCOMM '03). Association for Computing Machinery, New York, NY, USA, 213–224. DOI:https://doi.org/10.1145/863955.863980