Report Number: WUCSE-2004-14

2004-03-31

High Performance Packet Classification

Edward W. Spitznagel

In this proposal, we seek to develop methods for packet classification that can deliver high performance (e.g. wire speed processing at 10 Gb/s) while being reasonably cost-effective (e.g. memory-efficient and having low hardware complexity). In particular, we discuss a new approach involving extended TCAMs. This new approach eliminates the problems that preclude TCAMs from being considered viable solutions to the packet classification problem.

2004-14

High Performance Packet Classification

Authors: Spitznagel, Edward W.

March 31, 2004

Abstract: In this proposal, we seek to develop methods for packet classification that can deliver high performance (e.g. wire speed processing at 10 Gb/s) while being reasonably cost-effective (e.g. memory-efficient and having low hardware complexity). In particular, we discuss a new approach involving extended TCAMs. This new approach eliminates the problems that preclude TCAMs from being considered viable solutions to the packet classification problem.

High Performance Packet Classification

Edward W. Spitznagel

WUCSE-2004-14

March 31, 2004

Department of Computer Science and Engineering
Campus Box 1045
Washington University
One Brookings Drive
St. Louis, MO 63130-4899

Abstract

In this proposal, we seek to develop methods for packet classification that can deliver high performance (e.g. wire speed processing at 10 Gb/s) while being reasonably cost-effective (e.g. memory-efficient and having low hardware complexity). In particular, we discuss a new approach involving extended TCAMs. This new approach eliminates the problems that preclude TCAMs from being considered viable solutions to the packet classification problem.

High Performance Packet Classification

Edward W. Spitznagel
ews1@cse.wustl.edu
Washington University
St. Louis, MO 63130-4899

1. Introduction

Packet classification is important for a multitude of emerging network services. Advanced network services such as DiffServ edge routers [12], firewalls, intrusion-detection devices, and many QoS-enabled routers need to classify packets to determine what to do with them. In this proposal, we seek to develop methods for packet classification that can deliver high performance (e.g. capable of wire speed processing at 10 Gb/s) while being reasonably cost-effective (e.g. memory-efficiency, low power consumption and low hardware complexity).

Several approaches to packet classification have been proposed; in Section 3 we mention the more significant advances in terms of packet classification methods. One such method is Recursive Flow Classification; we discuss a means for compressing its data structures in Section 4. Another method is the use of Ternary Content Addressable Memories (TCAMs). TCAMs are the most popular practical approach to packet classification in high performance routers, but they require substantial amounts of power. In Section 5 we describe how to use *partitioned* TCAMs to dramatically reduce power consumption; we also discuss another extension which eliminates inefficiencies in storing filters with arbitrary port ranges in TCAMs. These ideas can produce significant savings in power consumption and storage efficiency, but the research in this area is not complete. In this proposal we discuss the development of incremental update procedures, a better understanding of the filter grouping problem's computational complexity, alternate partitioning algorithms, multi-level indexing schemes, and application of Extended TCAM ideas to DRAM-based classifiers.

2. Packet Classification Problem

The object of packet classification is to categorize packets by applying a set of rules called *filters* to the header fields of a packet. Each rule consists of a specification of header field values, and an action to perform on packets whose headers match that specification.

The information relevant for classifying a packet is contained inside the packet in K distinct header fields, denoted $H[1], H[2], ..., H[K]$. For example, the fields typically used to classify Internet Protocol (IP) packets are the destination IP address, source IP address, destination port number, source port number, protocol number and protocol flags. The number of protocol flags is limited, so they are often combined into the protocol field itself.

Using those fields for classifying IP packets, a filter $F = (128.252.*, *, TCP, 23, *)$, for example, specifies a rule matching traffic addressed to subnet 128.252 using TCP destination port 23, which is used for incoming Telnet; using a filter like this, a firewall may disallow Telnet into its network.

A filter database consists of N filters $F_1, F_2, ..., F_N$. Each filter F_j is an array of K values, where $F_j[i]$ is a specification on the i-th header field. The i-th header field is sometimes referred to as the i-th dimension or the i-th axis, when considering a packet's header as specifying a point in K-dimensional space. The value $F_j[i]$ specifies what the i-th header field of a packet must contain in order for the packet to match filter j. These specifications often have (but need not be restricted to) the following forms: exact match, for example "source address must equal 128.252.169.16"; prefix match, like "destination address must match prefix 128.252.*"; or range match, e.g. "destination port must be in the range 0 to 1023."

Each filter F_j has an associated directive $disp_j$, which specifies the action to perform for a packet that matches this filter. This directive may indicate whether to block the packet, send it out a particular interface, or perform some other action. Filter databases look like the example in Table 1, but most real-world databases have many more filters in them.

A packet P is said to *match* a filter F if each field of P matches the corresponding field of F. For instance, let $F = (128.252.*, *, TCP, 23, *)$ be a filter with $disp = block$. Then, a packet with header (128.252.169.16, 128.111.41.101, TCP, 23, 1025) matches F, and is therefore blocked. The packet (128.252.169.16, 128.111.41.101, TCP, 79, 1025), on the other hand, doesn't match F.

Since a packet may match multiple filters in the database, we associate a *cost* for each filter to resolve ambiguous matches. The packet classification problem is to find the lowest cost filter matching a given packet P.

Destination Address	*Source Address*	*Dest. Port*	*Src. Port*	*Protocol and flags*	*Comments*
host M_1	*	25	*	TCP	allow inbound mail to M_1
host M_2	*	53	*	UDP	allow DNS access to M_2
*	network N	*	*	*	allow outgoing packets
network N	*	*	*	TCP-ack	return ACKs OK
*	*	*	*	*	block everything else

Table 1: Example: simplified firewall filter database

To classify a packet, one could simply apply each rule, in increasing order of cost, until a match is found. This approach is easy to use, but is clearly not fast enough when a large number of rules are used.

Several more sophisticated algorithms have been developed that use data structures cleverly to improve the speed of packet classification. Each algorithm's performance can be measured in terms of the time required to classify a packet, the storage space required for the algorithm's data structures, and the complexity of updating the data structures when a filter is added, deleted or changed.

3. Related Work

Much research has been done in the Packet Classification area. At the time of this writing, however, no proposed classification method appears to be an ideal solution. In this section, we briefly discuss the solutions proposed thus far, and note relevant characteristics of each approach.

3.1. One-dimensional Classification

IP Routing lookups are a special case of classification, where classification consists of prefix matching on a single field, the destination address. Several algorithms have been developed for this special case that perform very well [3] [11] [21] [8] [16] [5].

A recent paper [13] describes a method for performing IP routing lookups using a partitioned TCAM. By restricting the search to only relevant partitions in the TCAM, power requirements are greatly reduced. One may think of it as carving out portions of a routing tree, and placing each portion in a partition of the TCAM. The prefix for each portion carved from the tree is then placed in a special partition called the *index partition*, along with the best matching route covering that portion. A routing lookup, then, requires a search of only two partitions: first, the index partition is searched, to determine which portion of the tree is relevant to that query; secondly, the relevant partition itself (as indicated by the result from the index) is searched. If a route is not found in that partition, the best matching cover (which was obtained from the lookup in the index) is used.

3.2. Two-dimensional Classification

Packet classification in two dimensions (e.g. classifying by Source Address and Destination Address) is more difficult than in just one dimension, but reasonable algorithms for this case exist also.

The Grid-of-Tries [17] algorithm constructs a search trie for the Destination Address, followed by several search tries for the Source Address. Clever use of switch pointers allows this algorithm to avoid backtracking; it also keeps storage requirements linear in terms of the number of filters, but this is only possible for the two dimensional case.

Area-based quadtrees (AQT) [4] are based on space decomposition techniques. In the two dimensional case, AQT requires $O(N)$ storage, takes $O(aw)$ memory accesses for query processing and $O(aN^{1/a})$ time for updates, where N is the number of filters, w is the width of the fields used for classification, and a is a tunable parameter.

Filters can also be organized using FIS-trees [6], which can support query processing with $O(\log w)$ memory accesses. This approach has reasonable storage requirements in the two dimensional case, but does not scale well beyond two dimensions.

3.3. General Multi-dimensional Classification

In the most general case, classifying packets in K dimensions is inherently hard for K greater than 2 [10] [17] [18] [7] [9]. The best algorithms at this time require $O(\log^{K-1} N)$ time and linear space, or $\log N$ time and $O(N^K)$ space, where N is the number of rules [10].

Perhaps the most popular approach for general multi-dimensional packet classification is the use of Ternary Content-Addressable Memories (TCAMs). TCAMs are circuits that compare a packet's headers against every rule in parallel [12]. TCAMs are fast, but scale poorly due to power dissipation issues and inefficiencies in handling range fields.

3.4. Heuristic Approaches

Fortunately, the filters in real-world classifiers tend to exhibit certain properties; these properties allow clever algorithms to beat the worst-case bounds in most real-world applications. This idea has led to a number of classification algorithms which, in spite of the aforementioned bad worst-case bounds, tend to perform very well with "typical" filter databases.

Recursive Flow Classification (RFC) [7] is one of the earliest of the heuristic approaches. It uses crossproducting in stages, and groups intermediate results into equivalence classes to reduce storage requirements. It appears to be the fastest classification algorithm in current literature; this speed, however, comes at the cost of substantial memory usage. A technique for reducing the memory requirements is described in Section 4.

Tuple-space search [18] is another early heuristic approach. In this approach, filters are categorized by the lengths of each specified field; this decomposes the classification problem into a number of exact-match problems, where hashing is used to probe for a match. In many cases this works well, but the hashing can lead to unpredictable search times.

HiCuts [9] constructs a decision tree to partition the classification space into regions until the number of filters in each region falls below a threshold. Thus classification is performed by walking the decision tree and then performing a short linear search through the filters in that region. HiCuts is not as fast as RFC, but its memory usage is not quite as bad.

Extended Grid-of-Tries (EGT) and EGT with Path Compression (EGT-PC) [2] use a slightly modified two dimensional Grid-of-Tries for classification on Source Address and Destination Address, and then a linear search of the filters which match the filters at that point. For "typical" databases, the storage requirements appear to grow linearly with the number of filters; the query time, however, is not as fast as HiCuts or RFC.

HyperCuts [20] is similar to HiCuts, but uses multidimensional cuts at each step and includes several other storage-related optimizations. Nodes containing the same rulesets are merged; any rule covered by a higher priority rule in the same node is not stored in that node; the region associated with each node is compacted to the minimum cover for the rules in that region; and, filters that are stored in many leaves of the decision tree are pushed "upward" (i.e. towards the root), to reduce the number of times these filters are stored. Furthermore, the heuristics used to build the decision tree are specifically designed with minimization of storage requirements in mind. HyperCuts appears to have excellent lookup performance (very close to RFC), but thus far results have not been shown for databases with more than 25,000 filters. It has excellent storage efficiency in many cases, but does not fare quite as well with heavily wildcarded filters.

It is often difficult to evaluate these heuristic classification techniques, due to the lack of publicly-available filter databases. A recent proposal [19] seeks to develop a tool for generating synthetic databases that retain the characteristics of a seed database and provide systematic mechanisms for varying the number and composition of filters.

3.5. Packet Classification Repository

A repository [23] has been established for retaining papers and source code for some of these algorithms. This repository allows researchers to share information, and to study and compare implementations of known algorithms, without the time-consuming and potentially error-prone process of re-implementing them.

4. Improving Storage Efficiency of RFC

Recursive Flow Classification [7] is the fastest packet classification algorithm in the literature at the time of this writing; it has the disadvantage, however, of requiring large amounts of storage.

RFC divides the header fields into chunks. An example of such a division is shown in Table 2. In each subsequent step, it combines information from two or more chunks together, using equivalence classes (with respect to which filters are matched) to represent these intermediate results; one possible reduction tree is shown in Figure 1. After some number of steps, the algorithm has reduced the information from the header fields into a single equivalence class, which indicates exactly which filters match that packet.

Chunk Number	*Chunk Contents*
0	First 16 bits of IP source address
1	Last 16 bits of IP source address
2	First 16 bits of IP destination address
3	Last 16 bits of IP destination address
4	Source port number
5	Destination port number
6	Transport protocol number and flags

Table 2: An example set of chunks for a typical 5-tuple classifier.

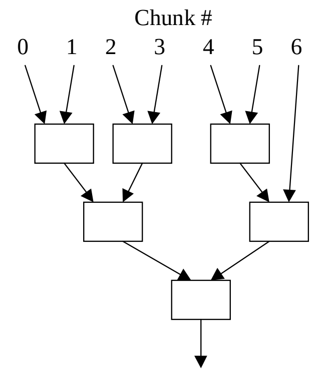


Figure 1: An example reduction tree for a typical 5-tuple classifier

When the algorithm combines information from two or more equivalence classes together at lookup time, it accomplishes this by finding the result in a *crossproducting table.* These tables are precomputed before classification is done. As the number of rules used in the classifier increases, the crossproducting tables tend to become enormous.

We can improve the storage efficiency of RFC at the cost of increased lookup time by compressing its crossproduct tables [14].

4.1. Table Compression Problem

The majority of the storage used by a typical RFC classifier is required by the crossproducting tables. To find a means of reducing RFC storage requirements, we consider compressing these crossproduct tables. The table compression problem can be defined as follows:

Given a sequence of integers (e.g. an RFC crossproducting table stored in row-major order) with several repeated values, produce an alternate representation typically requiring less memory space but allowing fast lookups of arbitrary elements in the sequence (table elements).

Compression results are measured in terms of the total number of bits required for the compressed representation, versus total number of bits needed for the uncompressed form. If the sequence (table) contains N integers (table entries), and the integers range from 0 to $M-1$ (i.e. the crossproduct table has M distinct possible equivalence class results), then $N\lceil \log_2 M \rceil$ bits are needed for the uncompressed representation.

In addition to requiring reduced storage space, the compressed representation must also allow any table lookup to be performed in a small, deterministic number of memory accesses. Otherwise, we lose all the benefits of RFC's fast lookup performance.

4.2. Simple Compression Algorithm

The crossproduct tables tend to have many repeated elements, and often these repeated elements are contiguous. Thus we look for a way to take advantage of this in the compression.

Let us consider a table stored in row-major order in an array. The original array can be represented by a compressed array and a bit vector. For each run of repeated elements, we store only one such element in the compressed array. Thus, the compressed array for $A\ A\ A\ A\ B\ B\ B\ C\ B\ B\ C\ C$ would be $A\ B\ C\ B\ C$. The bit vector has a bit corresponding to each element in the original (uncompressed) array; this bit is a 0 if that element is the first element or is the same as the previous element, and 1 otherwise; thus, the bit vector for $A\ A\ A\ A\ B\ B\ B\ C\ B\ B\ C\ C$ would be 0 0 0 0 1 0 0 1 1 0 1 0.

The bit vector is used to find the results of a lookup in the compressed array. If we want to know what the ith item was in the original array, we count the number of 1s in bit vector elements zero through i, inclusive. If there are j 1s, then we can find the result by looking at the jth element of the compressed array. Continuing the example from the previous paragraph: To look up the 7th element of the original array $A\ A\ A\ A\ B\ B\ B\ C\ B\ B\ C\ C$, we count the 1s in bit vector elements 0 through 7; there are 2, so the answer is element 2 of the compressed array $A\ B\ C\ B\ C$, i. e. the answer is C.

Counting the number of 1s becomes expensive when the bit vector is large; to avoid performing much of this work at classification time, we use precomputation as follows: If we precompute the total of 1s set in the first W bits, the first $2W$ bits, the first $3W$ bits, etc., then at lookup time we only need to count at most $W-1$ bits in the bit vector.

Thus, each crossproducting table can be represented efficiently by a compressed array, a bit vector with a bit for each item in the original array, and a set of precomputed counts of ones. Note that it is possible for the compressed form actually required more storage than the uncompressed form; this occurs when the size reduction from the original array to the compressed array is less than the additional storage needed to hold the bit vector.

To evaluate the performance of this compression method, we have applied the compression to a real firewall database with 159 rules. In the absence of a technique for determining the best reduction tree for RFC, we consider all possible reduction trees using 2D crossproducting tables for the seven header field chunks shown in Table 2. The size reduction of the crossproduct tables ranged from -7.9% in the worst case to 83.1% in the best case, with an average of 36.7% and a median of 39.5%.

4.3. TSP-based Compression Algorithm

The compression technique just described relies on the tendency for adjacent table entries in the same row to have the same value. For this reason, some tables compress well, and others do not. For example, the table shown in Table 3 does not compress particularly well.

Since the integers used for indexing the tables in RFC are assigned in an arbitrary order, it is possible to re-arrange rows and/or columns of the tables by reassigning these integers. Thus it is possible to re-arrange rows and columns in a table to improve compression. If we re-order the columns of Table 3 we can produce Table 4, which will result in improved compression since it has more runs of repeated elements.

	0	1	2
0	a	b	a
1	b	a	b
2	a	b	a

Table 3: Two-dimensional crossproducting table with poor compression

	0	1	2
0	a	a	b
1	b	b	a
2	a	a	b

Table 4: Two-dimensional crossproducting table with better compression

In general, there are too many ways re-arrange a large table to conduct an exhaustive search. However, there are some heuristics that usually produce good results. For example, if tables are stored in row-major order, then re-arranging rows will have little effect compared to re-arranging columns.

The question of how to re-arrange the columns for best compression can be transformed into a variation of the Traveling Salesman Problem, as follows: Let each column in the table be represented as a node in the TSP problem. The goal is to select an ordering (tour) of the columns (nodes) such that the number of elements in the compressed array (cost of the tour) is minimized.

The total cost of an ordering of columns is the number of elements in the compressed array. An element in column $i+1$ is only added to the compressed array if it differs from the element in column i of the same row; thus, the cost contribution of placing column $i+1$ immediately after column i is equal to the number of rows in which the two columns have differing entries. This way, the cost of a particular tour reflects the cost of using that ordering for columns, except for the cost of the first column itself (due to wrap-around from last column of a row to first column of the next row.)

With this definition of cost, note that $cost(A, C) \leq cost(A, B) + cost(B, C)$ (i.e. the triangle inequality applies.) Thus, TSP approximation algorithms based on a minimum spanning tree will work and can be used to find an ordering of columns that produces good results.

This generally produces better results than the naive compression scheme described earlier, but computing the TSP cost matrix can be expensive. For a 2-dimensional table with R rows and C columns, this requires $O(C^2R)$ time and $O(C^2)$ space.

To evaluate the performance of this compression method, we have applied the compression to the same 159 rule firewall database as before, again studying all possible reduction trees with 2D crossproduct tables. With the TSP-guided compression, the size reduction of the crossproduct tables ranged from -7.9% in the worst case to 87.3% in the best case, with an average of 54.4% and a median of 61.6%.

Furthermore, it appears that compression may tend to improve as the number of rules increases. The TSP-based compression has been applied to subsets of a larger filter database, varying the number of rules each time. This time we use only one reduction tree, selected arbitrarily from those having average results with the other database. The results are shown in Figure 2. Note that, as the number of filters increases, the size reduction of the crossproduct tables tends to increase (exceeding 60% at around 250 filters.)

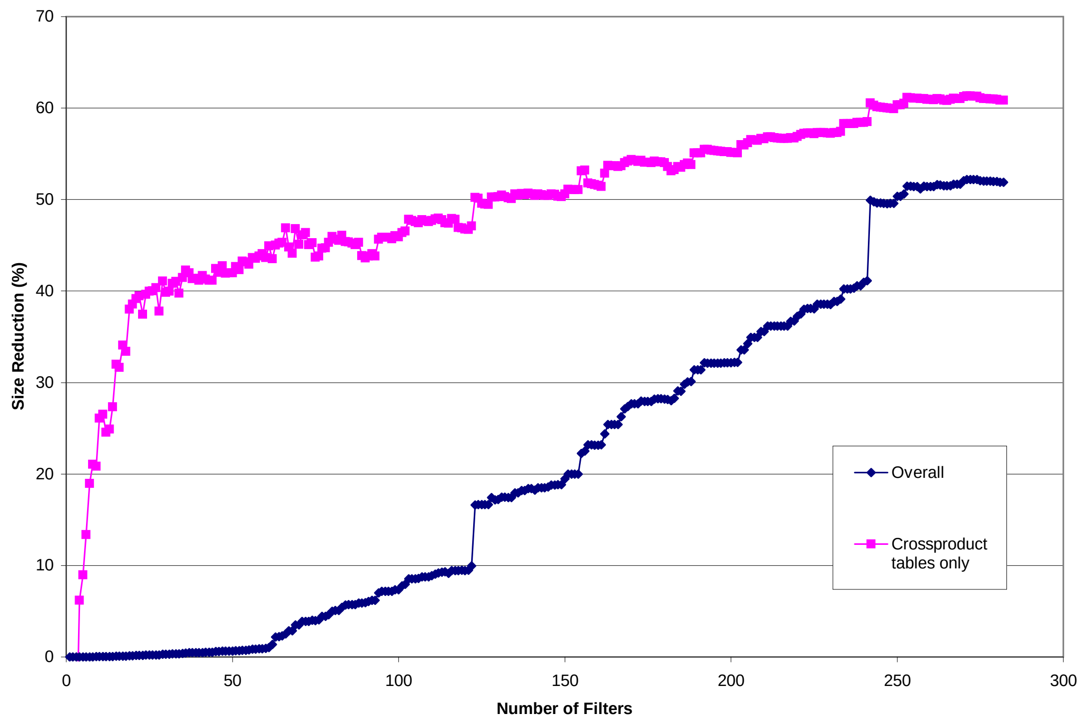


Figure 2: Compression Efficiency vs. Number of Filters.

5. Extended TCAMs

Ternary Content-Addressable Memories (TCAMs) are the most popular practical approach to general multidimensional packet classification in high performance routers. Ordinary TCAMs perform matching in a bitwise fashion; a query word q matches a stored value, mask pair (v, m) if $q \& m = v \& m$, where the ampersand denotes the bitwise logical "and" operation. Another way to think about this is to consider the stored words to be sequences over not only the traditional bits 0 and 1, but also a "don't care" value (often represented as an X.) The "don't care" bits correspond to bits with mask 0 in the bitmask representation.

TCAMs are used for classifying packets by concatenating all header fields of interest, and using that concatenation as a word for TCAM lookup. This works well for prefix matching (often used on the IP address fields) but is not well-suited for range matching (which is used on the port fields.) The usual way to handle a port range is to replace each filter with several filters, each using a prefix match that covers a portion of the desired port range. For example, the range 2-10 can be expanded into the bit patterns 001*, 01*, 100* and 1010, which exactly cover that range.

In this manner, any sub-range of a k bit field can be represented as a set of prefixes, requiring up to $2(k-1)$ prefixes per sub-range. So, a 16-bit port field can require as many as 30 distinct TCAM entries. But if ranges are present in both the source and destination port fields, then we need a filter for *all combinations* of the sub-ranges for the two fields. Thus a single packet filter may require 900 TCAM entries.

Figure 4 shows the circuitry for a single bit in a typical TCAM design, requiring 16 transistors. A ternary bit T stored in the cell has three possible values: 0, 1, and X, where X is the "don't care" value that matches on both 0 and 1. To store a ternary bit, the TCAM cell uses two SRAM cells

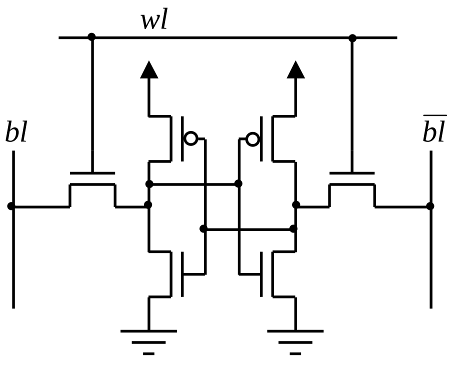
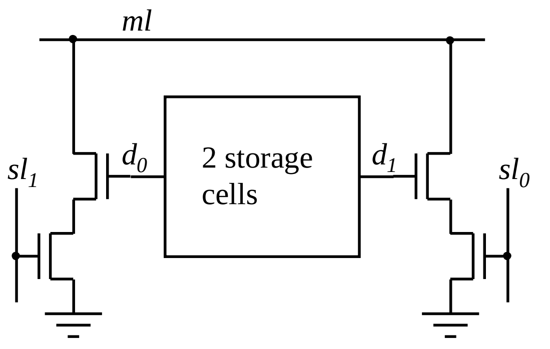


Figure 3: Conventional 6-transistor SRAM storage cell



Ternary bit *T*	*Storage cell values* d_0	d_1
0	0	1
1	1	0
X	0	0

Figure 4: TCAM cell (details of SRAM storage cells omitted)

with values d_0 and d_1 as shown in Figure 4. The value d_0 is 0 when a zero bit in the query word should match, and the value d_1 is 0 when a one bit in the query word should match. Details of an SRAM cell are shown in Figure 3. In the SRAM, data is read from and written to the storage cell via the bitlines bl and $\overline{bl}$, under control of the wordline wl.

In the TCAM circuit, the query value and its complement are input via search lines sl_0 and sl_1. A mismatch in this circuit creates a path to ground from the matchline ml. Comparison for all TCAM words is done in parallel, which allows a lookup to be very fast, but it also means that power consumption is quite high [13].

A recent paper [13] describes CoolCAMs, a technique for using *partitioned TCAMs* to perform IP routing lookups with greatly reduced power dissipation. A partitioned TCAM is a TCAM divided into blocks of words, where each block can be enabled or disabled during a query; power consumption is approximately proportional to the number of blocks searched. [13] With the CoolCAMs method, IP routing lookups require searching just two TCAM blocks. This technique described relies on certain properties of the one-dimensional case, and thus does not generalize to multi-dimensional classification. But, we will take the basic idea of using partitioned TCAMs to reduce power usage, and find ways to apply it in the case of multi-dimensional classification.

A recent trend among packet classification methods [22] [9] [2] [20] is a slight coarsening of data structures. In these methods, a lookup is performed by traversing a sophisticated data structure which leads not to just one matching filter, but a short list over which a linear search is performed. If the lists are kept short, this typically results in a large savings in terms of data structure space, at only a small cost in terms of lookup performance. This idea has some distant parallels in the Extended TCAMs classification scheme we describe later in this proposal.

The Extended TCAM concepts are based on two main ideas. First is the idea that the range-match storage problem can be addressed by incorporating range-matching logic into the hardware itself. This solves that problem, but not the issue of power dissipation; this brings us to the second idea, that power consumption can be reduced by the use of a partitioned TCAM. The technique for using a partitioned TCAM is not precisely the same as in [13], but much inspiration came from that paper.

The incorporation of range-matching functionality into the hardware involves the storage of a pair of bit values (lo, hi) for each range match field, and circuitry to compare a query word q against these stored values. Figure 5 shows the iterative structure of the *range check* circuit, with a separate stage for each bit. The comparison proceeds from the most significant bits to the least significant bits. The four inter-stage signals geh_i, leh_i, gel_i, and lel_i are used to represent the following, where W is the width of a word:

geh_i: The quantity represented by the first $W - i$ bits of the query (i.e. q_{W-1} through q_i) is greater than or equal to the quantity represented by the first $W - i$ bits of the stored upper bound hi.

leh_i: The quantity represented by the first $W - i$ bits of the query is less than or equal to the quantity represented by the first $W - i$ bits of the stored upper bound hi.

gel_i: The quantity represented by the first $W - i$ bits of the query is greater than or equal to the quantity represented by the first $W - i$ bits of the stored lower bound lo.

lel_i: The quantity represented by the first $W - i$ bits of the query is less than or equal to the quantity represented by the first $W - i$ bits of the stored lower bound lo.

The names are abbreviations indicating that, up to bit i, the query is **g**reater (**l**ess) than or **e**qual to **h***i* (**l***o*).

The assertion of both leh_0 and gel_0 at the same time happens if and only if the query value q is within the range defined by hi and lo; therefore, a "query is in range" signal can be formed by taking the logical AND of signals leh_0 and gel_0.

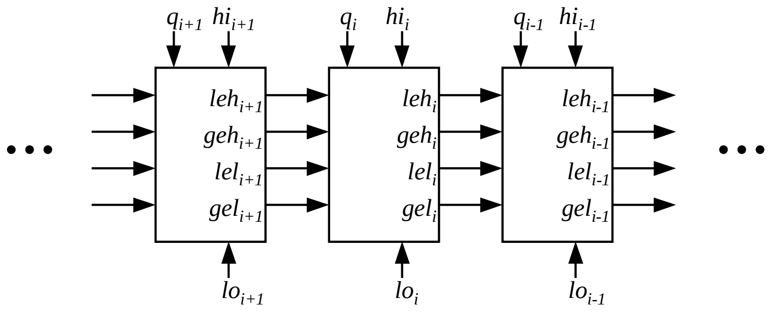


Figure 5: Iterative structure of range check circuit.

The logic for the upper bound check of one stage is shown in Figure 6. If the query bits before this stage were greater than hi (i.e. leh_{i+1} is not asserted), then the query value will still be greater than hi once this stage is included as well; therefore we ensure in this case that leh_i is not asserted. If, on the other hand, the query bits before this stage were not greater than hi, then there is only one condition under which the query value including this bit will be greater than hi. That condition

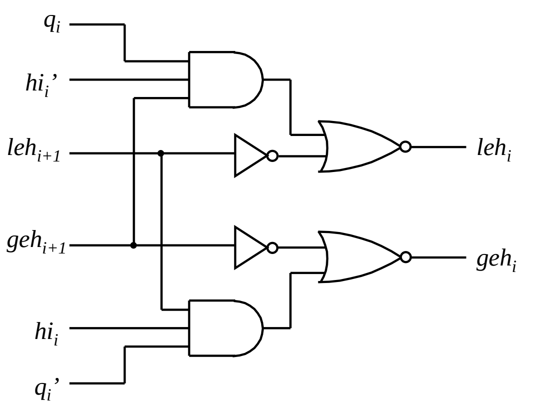


Figure 6: Range-check sub-circuit for upper bound comparison

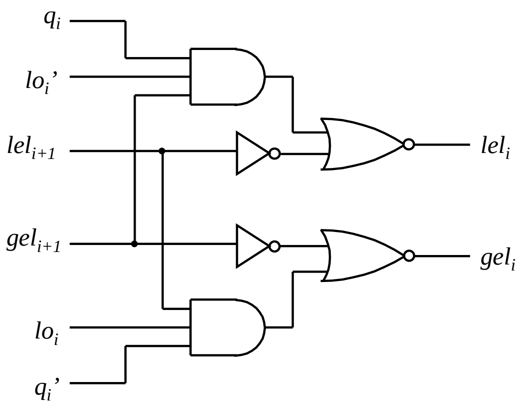


Figure 7: Range-check sub-circuit for lower bound comparison

is that the previous query bits equal hi (implied by geh_{i+1} and leh'_{i+1}, but we can omit leh'_{i+1} since that case already causes us to assert leh'_i), q_i is 1, and hi_i is 0; therefore in this other case we also ensure leh_i is not asserted. In all other cases we assert leh_i.

The logic for lower bound check is shown in Figure 7; its operation is the same as the upper bound check, except that it uses lo_i (bit ith of the lower bound) instead of hi_i (bit i of the upper bound.)

Each stage requires an upper bound check (20 transistors), a lower bound check (20 transistors), and two SRAM storage cells (6 transistors each) for storing hi_i and lo_i. Thus 52 transistors are required for each bit in the query, using a standard CMOS implementation of this design. This is slightly more than three times as many as a standard TCAM storage cell, but the impact of this on the cost of the entire TCAM is much smaller. The two port fields represent 22% of a 144 bit TCAM word suitable for IPv4 (allowing 16 bits for the protocol field including flags, plus 32 bits for an action field and priority). Using the range-check logic for this portion of the TCAM would increase the total number of transistors per word by just 50%, and this could be improved by a more efficient range check circuit.

After resolving the port range inefficiency problem, TCAMs still suffer from the problem of high power consumption. The main component of power consumption in TCAMs is proportional to the number of entries searched; thus, to reduce a TCAM's power consumption, we can partition it into blocks of words, and only search a small number of block(s) when performing a lookup. For this to work, of course, an index mechanism is needed to determine which block(s) of the TCAM to search for a given query.

In the IP lookup case [13], a lookup in the index identifies a single bucket that needs to be searched; thus the index can actually be implemented simply by using one of the blocks inside the TCAM. Multi-dimensional packet classification, on the other hand, may require searching several blocks for filters; thus we use an indexing mechanism that can support this. Each Extended TCAM block has an associated *index filter*, which consists of the same match circuitry as one word in the Extended TCAM device. When this filter is matched, it enables its corresponding TCAM block for search.

So, a search in an Extended TCAM device works as follows: first, the index filters are searched (in parallel) to determine which blocks to enable; then, the enabled blocks are searched (in parallel) for matching filters. Each block then returns its highest priority matching filter, and a final priority resolution step returns the highest priority filter of those.

An example is shown in Figure 8; for simplicity, we perform two-dimensional classification in the example, with a four bit range field and a four bit bitmask field. To perform a lookup for a packet with header field values (2, 10), we first check the index filters and determine that the second and fourth index filters match the packet. The search then progresses to the second filter block and the fourth filter block. In the second block we find the matching filter (1-2, 1x1x), and in the fourth block we find the matching filter (0-14, 1010).

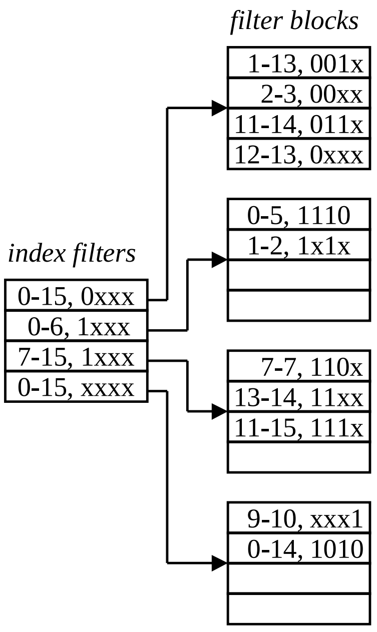


Figure 8: An example of filters and index filters in an Extended TCAM

When using a partitioned TCAM for packet classification, the key to making the search power efficient is to group the filters in such a way that only a few TCAM blocks must be searched for any given query. The current approach is to use a heuristic filter grouping algorithm to organize the filters.

The algorithm runs through a series of *phases*; each *phase* recursively divides the multi-dimensional classification space into ever smaller *regions*; this process is guided by a heuristic described in [15]. At the end of a phase, a TCAM block is allocated for each region; filters contained entirely within that region are placed in that TCAM block and removed from consideration in later phases. This runs until all filters have been placed into TCAM blocks.

When filters are grouped this way, any classification query will need to search one block for each phase, because each phase will have one region containing the point corresponding to the header fields of the query. Thus, the main component of power usage is proportional to the number of phases needed by the algorithm.

To improve power efficiency at the cost of some storage, the last phase also allows filters to span multiple regions. Thus, in the final phase, the TCAM block contains any remaining filters intersecting its region, not just those contained entirely within. This means some filters may be stored in more than one TCAM block; but, it also can reduce the number of phases required, which means fewer TCAM blocks will need to be searched in parallel at query time (i.e. less power is used.)

We evaluate the algorithm's performance by running it on filter databases produced by a synthetic database generator [19].

As a measure of power efficiency, we use the quantity $(b + sk)/N$, where b is the number of storage blocks used by the partitioning algorithm, s is the maximum number of storage blocks that must be searched in a query, k is the block size and N is the number of filters in the database. The numerator represents the TCAM words searched to perform a lookup, including index entries, and the denominator represents the TCAM words searched in a normal TCAM (assuming no replication for the range matching issue). We refer to this quantity as the *power fraction*.

As a measure of storage efficiency, we use the quantity $N/(b + bk)$. Here, the denominator is the sum of the number of words used in the index plus the number of words used in the storage blocks required by the algorithm.

The best choice for TCAM block size appears to depend on the number of filters in the database. Figures 9 and 10 respectively show power fraction and storage efficiency for various block sizes as we increase the database size. Based on this data, it looks like a good choice for the block size is the largest power of two smaller than $\frac{1}{2}\sqrt{N}$.

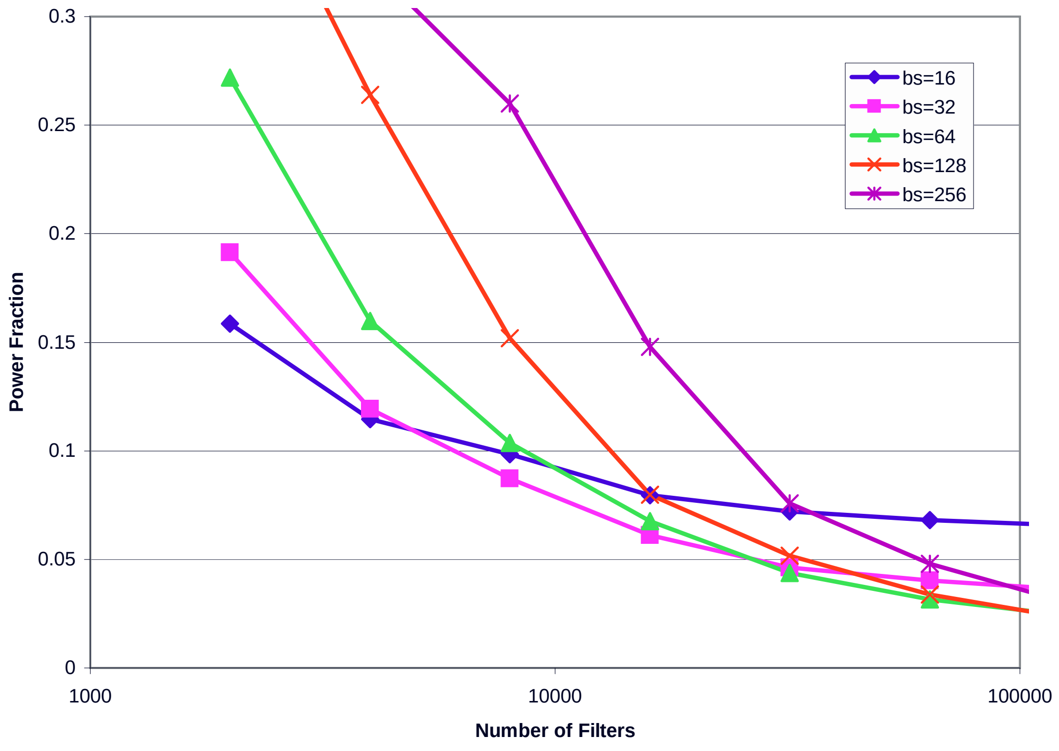


Figure 9: Power fraction vs. number of filters, for various block sizes.

Using that to determine block size, we study performance with filter databases ranging from 2,000 filters to 128,000 filters, using two different seed databases; figures 11 and 12 show these results. Note that, for larger filter sets (where power reduction is most important), the power fractions are generally below 5%.

Figure 13 shows how the performance varies as a function of the smoothness adjustment of the database generator [19]. The data reflects filter sets with 64,000 filters each, and a TCAM block size of 128. Increases in the filter smoothness adjustment appear to cause significant deterioration in the power fraction. While the results still represent a 10:1 improvement over the standard TCAM, the degraded performance is disturbing and potentially worth further investigation.

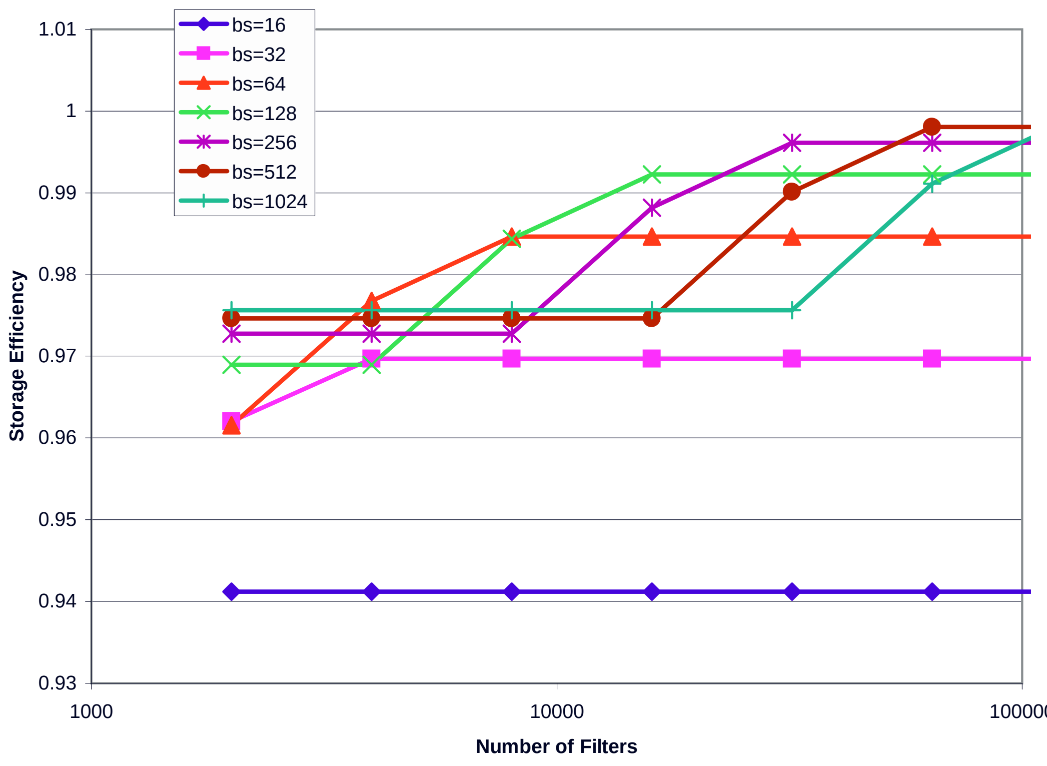


Figure 10: Storage efficiency vs. number of filters, for various block sizes.

Figure 14 shows the effect of the scope adjustment of the database generator. These data reflects filter sets with 64,000 filters each with smoothness set at 4, and a TCAM block size of 128. A negative scope adjustment corresponds to more specific filters, and a positive scope adjustment to less specific filters. We expect that, as filter databases grow in size, there will be a natural tendency for the filters to become more specific. Hence, it is worth understanding how algorithms perform under these circumstances. We see a marked improvement in power fraction at scope adjustments of -16 or less. This seems to make sense intuitively: as filters become more specific, we expect the filter grouping algorithm to separate them more easily. We expect that most other packet classification algorithms are also likely to perform better as filter sets become more specific.

The Extended TCAMs ideas show much promise, and the initial results are very encouraging, but open questions remain. In the following sections of this proposal, we discuss those questions and some ideas for investigating them. The proposal scope of the dissertation is a subset of those areas, as indicated in Section 5.6.

5.1. Computational Complexity of Filter Grouping

When using a partitioned TCAM for packet classification, the key to making the search power-efficient is to group the filters in such a way that only a few TCAM blocks need to be searched to find the best matching filter for a given packet. While this appears to be a difficult problem, there are no proven bounds on its complexity at this time. Thus, we currently have no proof whether the current filter grouping technique is a good idea or not. Establishing the computational complexity of the filter grouping problem will resolve that issue.

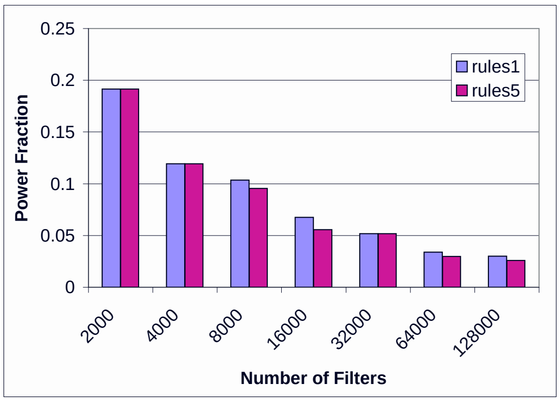


Figure 11: Power fraction vs. database size, using two different seed databases.

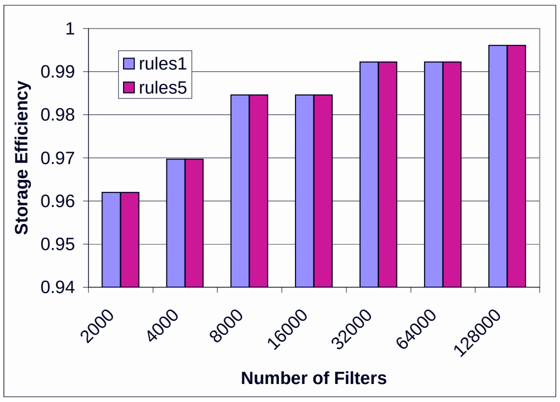


Figure 12: Storage efficiency vs. database size, using two different seed databases.

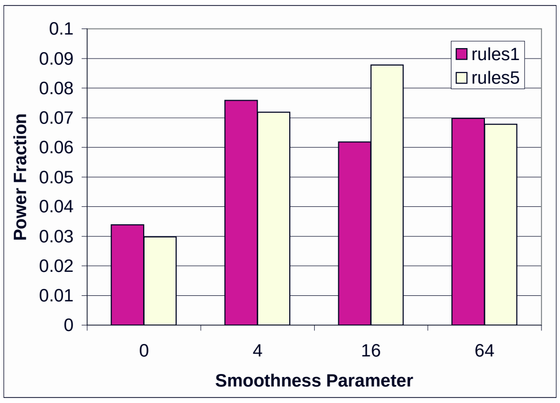


Figure 13: Power fraction vs. smoothness adjustment, using two different seed databases.

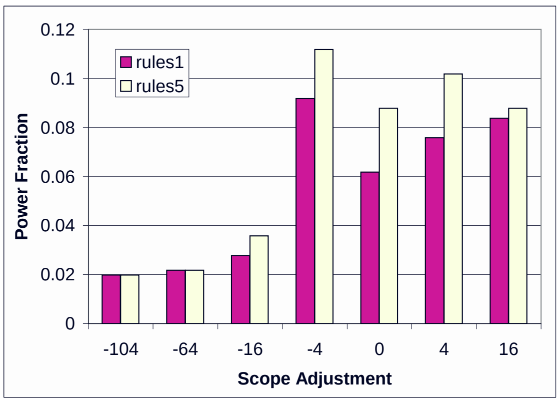


Figure 14: Power fraction vs. scope adjustment, using two different seed databases.

The current means for filter grouping in Extended TCAMs is a heuristic algorithm; results are typically good, but not they are by no means guaranteed to be optimal. The use of a heuristic is justifiable in cases where the problem is too difficult for the computation of an optimal solution to be practical. As a first step in determining whether that is the case with Extended TCAM filter grouping, we define the filter grouping problem as follows:

Definition. Let f_1 and f_2 be filters defined on the same multidimensional space. We say that f_1 *covers* f_2 if the region of space defined by f_1 completely contains the region defined by f_2. Similarly, we say a set F of filters covers a filter f if the region defined by the union of filters in F completely contains the region defined by f.

Filter Grouping Problem. Given a set F of filters and integers k, m, and r, find a set S of at most m filters and a bipartite graph $G = (V, E)$ with $V = F \cup S$ and $E \subseteq F \times S$, that satisfy the following conditions:

- for every f in F, the neighbors (in G) of f cover f,
- for every s in S, the degree (in G) of s is at most k,

- no point in the multi-dimensional space on which the filters are defined is covered by more than r members of S.

S defines the set of index filters; the graph G specifies the assignment of original filters to index filters and their associated storage blocks. The degree of a vertex for an index filter corresponds to the number of filters in the storage block associated with that index filter. The bound k, on the degree, limits the number of filters per block; the bound m, on the size of S, limits the number of index filters and hence the number of TCAM blocks in the device; and the bound r, on the number of index filters covering any point in the classification space, limits the number of TCAM storage blocks that must be searched in parallel for a query.

As an example, an instance of the filter grouping problem is shown in Figure 15, along with one possible solution to that instance. The solution in this example corresponds to the filter grouping shown in Figure 8. Each s in S corresponds to an index filter (and its corresponding block in the TCAM), and the bipartite graph indicates which filters are stored in each block.

Given the following:

F = { (0-5, 1110),
(7-7, 110x),
(13-14, 11xx),
(1-13, 001x),
(9-10, xxx1),
(2-3, 00xx),
(11-14, 011x),
(11-15, 111x),
(0-14, 1010),
(12-13, 0xxx),
(1-2, 1x1x) }

k = 4, *m* = 4, *r* = 2

One solution to the filter grouping problem is:

S = { (0-15, 0xxx), (0-6, 1xxx),
(7-15, 1xxx), (0-15, xxxx) }

Bipartite graph:

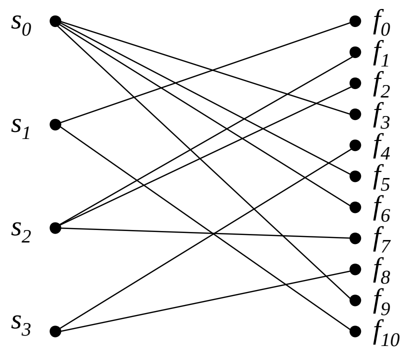


Figure 15: An example instance of the filter grouping problem

The filter grouping problem can be converted into an optimization problem, by minimizing any one of the three parameters, while leaving the other two as bounds. In a typical application, k and m are fixed numbers based on the specific Extended TCAM hardware used. In that case, the goal is to find a solution minimizing r, in order to achieve the lowest power consumption for the hardware used.

It would be interesting to prove whether finding the optimal filter grouping is NP-hard or not. If it is NP-hard, then the use of a heuristic grouping algorithm is justified; if it is not, then that would motivate a search for an efficient algorithm for finding an optimal solution.

One possible candidate for reduction to the Filter Grouping problem is the Clustering problem. The Clustering problem is defined as follows: Given a finite set X, a distance $d(x, y) \in Z_0^+$ for each pair $x, y \in X$, and two positive integers K and B, is there a partition of X into disjoint sets X_1, X_2, $\cdots$, X_K such that, for all $1 \leq i \leq k$ and all pairs $x, y \in X_i$, $d(x, y) \leq B$. Clustering and filter grouping both involve partitioning a set into at most K disjoint subsets. Also, both problems include some form of restriction on what can go into a partition. Even if Clustering cannot be reduced to

Filter Grouping, the proof of Clustering's NP-Completeness (which can be done by reducing Graph 3-Colorability) may provide some insight.

If the filter grouping problem is in fact NP-hard, a reasonably good approximation algorithm may exist. Or, it may be provable that bounded approximation is hard. It would be interesting to see whether anything can be proven about the current filter grouping heuristic's performance, relative to the optimal solutions.

5.2. Handling Filter Updates

Perhaps the biggest piece missing from the Extended TCAM solution is an efficient update procedure. In many packet classification applications, it is necessary to add or delete filters from the classifier, while maintaining high throughput in terms of packets classified per second; and we expect dynamic updates to be of particular importance to classifiers with truly large numbers of rules, since those are quite unlikely to be purely static filter databases. Thus it is highly desirable to devise methods for adding and removing filters from the Extended TCAM classifier efficiently.

Handling incremental updates is not a trivial thing in many of the modern high-performance classification schemes. HyperCuts, for example, does not at this time include an incremental update procedure. Woo's modular approach to packet classification [22] mentions that incremental updates are possible, but does not go into detail on the subject; furthermore, it turns out that it requires rebuilding the entire data structure in the worst case anyway.

The region-splitting operation performed in each phase of the filter grouping algorithm creates a tree of sorts, with each node corresponding to the splitting of a region into subregions. It would be convenient to be able to alter this tree in a manner similar to incremental update operations in a red-black tree, B-tree, or similar structure. But in this case, when a node in the tree is changed, usually all children of that node must be computed. Thus, updates near the leaves of the tree may not be expensive, but updates that change the region splitting near the root of the tree are expensive; in the worst case, it would be equivalent to rebuilding that entire phase in the data structure.

If updates occur infrequently, and they need not be effected immediately, then it may be acceptable to recompute the entire data structure (TCAM entries, index filters, and the relationships created by the filter grouping algorithm.) But, since the computation can take several seconds, this will certainly not meet the performance needs of the more demanding packet classification scenarios. Also, it either precludes classification while the new data structure is loaded into the TCAM, or it requires a larger TCAM (to hold both new and old data) and a means of switching on the fly which data structure is used for classification. The latter approach could be accomplished by adding logic to the index entries, such that each index filter is also associated with either the new or the old data structure, and is enabled for search only when appropriate.

A slight improvement over the above is to reserve a few blocks of the TCAM for adding filters between recomputation of the data structure. Filter updates can then be effected immediately, either by removing the filter from the TCAM, or by adding it to the reserved blocks. Periodic rebuilding of the data structure can then clean out the reserved blocks for use later.

Another possible area for improvement would be rebuilding individual phases one at a time. To rebuild phase i, we would leave the original phase i in the TCAM, and create all the entries for the new phase i'. Lookups are still performed, using the old phase i, until the new phase is ready. At that time, phase i' is enabled; then phase i is disabled and deallocated. This may allow the data structure to be rebuilt without blocking any lookups, while only requiring an extra phase's worth of TCAM storage.

But, a more efficient approach may be possible. Deletions are easy, and can be accomplished by simply removing the relevant filter's TCAM entry or entries. Insertions are more difficult, because there may not be an available TCAM entry in any of the blocks whose indices cover the filter. Sometimes this is addressable by evicting a filter from one of those blocks to a block in a different phase that also covers that filter, but this may not always be possible.

One idea for reducing the likelihood of that is to actively attempt to keep the blocks optimally loaded. Blocks may be initially filled to, say, 50% during data structure creation. Blocks that become too heavily loaded (e.g. after many updates) can be split into separate blocks, if a good split can be found. Blocks too lightly loaded can be merged, if they are siblings in terms of the region-splitting operation. As a last resort, when an insert is requested and there are no TCAM entries available in blocks covering that filter (and nothing can be done to make an entry available), a filter can be added to multiple blocks. This corresponds to the filters added in the last phase of the grouping algorithm. And, in the worst case, another phase can be added, to make room for a new filter; but, if this happens often, recomputing the entire data structure is probably a good idea, to keep power usage low.

An evaluation should include an analysis of the worst-case performance, but should also include a study of performance under more typical conditions. Furthermore, since it is quite likely that there will be an enormous difference between those two cases, we propose that the analysis should also include various levels of adversarial behavior in the input test vectors.

Characteristics of typical filter updates are completely unknown; therefore, performance under "typical" conditions is difficult to measure. As a first step, however, it seems reasonable to randomly add and delete filters generated by the synthetic generator, with some specified rate or probability for addition and deletion.

Intermixed insertions and deletions are important input test vectors, but it is also important to study update performance when given insertion operations only. Without the deletion operations, the TCAM blocks will inevitably become full; thus it will exercise the update procedure's ability to deal with full TCAM blocks to a much greater extent, which may expose a weakness in the update procedure that did not show with the intermixed insertion/deletion operations.

Adversarial behavior testing at various levels between the typical case and worst case inputs can also provide insight regarding which properties of the updates present the greatest challenges for the algorithm. At one extreme is an omniscient, omnipotent adversary, knowing everything about the algorithm and the current state of the data structures. A less omniscient adversary knows the algorithm but not the current data structure contents or any random/arbitrary choices made by the algorithm. A less powerful adversary can control only some parameters of a random update generator, rather than having complete control over the updates used as input to the algorithm. Also, the less omniscient adversary can be combined with the less powerful one, as an additional case for study. Results from these analyses will provide more insight into the update algorithm's performance limits than just a worst-case analysis.

5.3. Different Partitioning Algorithms

The partitioning algorithm described in [15] performs very well in terms of storage efficiency, but does not achieve the best power fraction possible. Different partitioning algorithms may be able to obtain better results in terms of the power fraction, although in some cases this may come at the cost of decreased storage efficiency. It is also possible that different algorithms would perform better on certain worst-case types of inputs.

Also, there may be other filter grouping algorithms that result in structures that are more amenable to incremental updates. Even if such an algorithm had poorer performance in terms of power fraction or storage efficiency, it would still be of interest due to the importance of efficient incremental updates.

One idea for an algorithm involves creating a trie for each dimension. To add a filter to the database, it is placed in one of the tries; buckets are then carved out of the tries, as in [13]. But, in a search, *all* buckets on the search path in each trie must be searched. This is because we cannot store a single "covering filter" for each bucket, since the best filter covering the bucket depends on matching in the other fields. This scheme's power efficiency may not be spectacular, since many buckets may need searching in parallel, but it is perhaps more friendly towards incremental updates or multi-level indices.

Also, variations on the algorithm in [15] are of interest. Some of the splitting criteria used in an earlier version of that algorithm resulting in lower power fractions, at some cost in terms of storage efficiency. This represents a trade-off worth studying, and it suggests the investigation of different splitting criteria.

Other ideas are possible. The algorithm in [15] was partly inspired by HyperCuts [20]. Other packet classification algorithms may contain ideas which can lead to partitioning techniques, which may be be of interest.

Since the lookup hardware itself is not changed when using a different filter grouping algorithm, the lookup performance is still the same. Thus we need not worry about classification speed when comparing these algorithms. In other respects, though, the evaluation of any proposed new algorithm ought to be similar to the analysis in [15]. Power fraction and storage efficiency are the primary metrics of interest here. But, in addition, the evaluation of incremental update performance discussed in Section 5.2 is also relevant.

5.4. Multi-level Indices

The idea of using multi-level indices can be extended from CoolCAMs and applied to Extended TCAMs. Due to the differences between CoolCAMs indexing and Extended TCAMs indexing, there is some room for different designs to be explored.

In any case, the idea is to reduce typical power usage, by reducing the number of TCAM entries searched. Adding a "meta-index" level before the index entries would allow the device to activate far fewer of the index entries, resulting in a slight decrease in power usage.

In some cases, however, power usage may actually increase. If the additional indices do not reduce the number of TCAM blocks searched, then the net effect is a slight increase in power usage due to the additional index entries being searched.

But, the largest power saving may come from the ability to use smaller blocks with a multi-level index. Recall the expression $(b + sk)/N$, used as the power fraction measure for the single-level index system, where b is the number of storage blocks used by the partitioning algorithm, s is the maximum number of storage blocks that must be searched in a query, k is the block size, and N is the number of filters in the database. Smaller blocks generally mean fewer TCAM entries are searched in the final (non-index) stage of lookup (thus a smaller contribution from sk), but beyond a certain point the power used by the index entries themselves becomes significant (i.e. the b term dominates). When using multi-level indices, however, we only activate the index entries specified by the meta-index level(s); thus, we can use smaller blocks without paying as much of a power penalty for the index entries.

It is worth noting that the additional index entries do not significantly affect the throughput of the classifier, when properly pipelined; the classifier will still produce one lookup per clock cycle. It is the power fraction and the storage efficiency that are expected to change.

It is also worth noting that alternate filter grouping algorithms may be more amenable to multi-level indexing than the algorithm described in [15]; thus there is additional motivation for examining new grouping algorithms.

Multi-level indexing should be evaluated similarly to the single-level index case. The idea behind the power fraction term is still relevant, although in the multi-level index case it cannot be expressed as simply $(b + sk)/N$. Storage efficiency is also meaningful, although it may need to be normalized due to additional silicon real estate occupied by additional index entries. Incremental update performance is also important; evaluation of this is discussed in Section 5.2.

5.5. Applications for DRAM-based Solutions

Some of the ideas from Extended TCAMs can be applied in DRAM-based packet classification techniques. These techniques will not be as fast as Extended TCAMs, but they may be fast enough for many applications. And, since the cost per bit of DRAM is about 1% of the cost of SRAM, these techniques look attractive for low-cost applications.

While DRAMs are slow for independent accesses to different rows, they are reasonably fast for accesses within the same row of a memory array. A typical 256 Mbit memory array has 2 kB in a row, which is large enough to store 16 filters and can be retrieved in about 1.28 usec (assuming RDRAM at 1.6GB/s).

A straightforward DRAM-based implementation of the Extended TCAMs algorithm can be achieved by storing the filter blocks in DRAM, with one block in each row. The index entries are stored together, in the minimum number of blocks required. A lookup, then, first reads all the index blocks in sequence and determines which filter storage blocks need to be searched; then, those filter blocks are read, in sequence, and If there are $\lceil b/k \rceil$ index blocks and s phases, then $\lceil b/k \rceil + s$ DRAM rows must be read per lookup.

To speed this up, one can store the filters for each phase of the grouping algorithm in a separate DRAM bank. In this case, a lookup still starts by sequentially reading the index blocks; but after that step, the searching of the filter blocks is done in parallel. Thus, each lookup requires time for $\lceil b/k \rceil + 1$ DRAM row accesses.

The time to perform a query is now clearly dominated by the $\lceil b/k \rceil$ term, which is the time required to search the index. This quantity may be quite large; e.g. if $k = 16$, then a 100,000 filter database would require 6,250 DRAM row accesses for the index part of the lookup. One possible solution to this is the use of a multi-level index, which we expect to be particularly helpful due to the small block size. The idea is that the first level(s) of the index will allow the classification engine to eliminate the majority of the index blocks in the remaining index levels.

Another interesting approach for handling the index is to use a TCAM (including range-match hardware) to store the index entries. This allows the index lookup to occur in parallel, using a fairly small TCAM. In this approach, if $k = 16$, then a 100,000 filter database would require a TCAM with a mere 6,250 entries. This hybrid approach allows much greater density than a pure TCAM approach, by storing the bulk of the data structure in DRAM, and not duplicating the comparison logic across every single filter entry.

While a DRAM-based implementation of Extended TCAMs with multi-level index has some themes in common with HyperCuts [20], they are clearly not the same. A key difference is that the Extended TCAMs technique is more parallelizable, allowing multiple filter blocks to be searched during the lookup. It may also be interesting to design and evaluate a parallelized version of HyperCuts, where multiple HyperCuts lookup engines operate in parallel.

5.6. Scope of Dissertation

Extended TCAMs provide many areas for research; it is important to note which are within the scope of this dissertation proposal, and which are not. This proposal includes the study of incremental update processing, multi-level indexing, and different filter grouping algorithms. This proposal does not include the study of computational complexity of the filter grouping problem or the application of Extended TCAM ideas to DRAM-based classifiers.

6. Conclusion

High-performance packet classification is crucial to the deployment of many advanced network services. The most popular approach in use is the TCAM, but the usefulness of TCAMs is limited by their high power consumption and inefficient representation of range match fields.

Extended TCAMs are a promising solution to provide the performance needed while scaling up to hundreds of thousands of filters. Extended TCAMs allow classification at the same speed as TCAMs, usually with less than 5% of the power usage of a standard TCAM; they also avoid the storage efficiency problem that TCAMs have with range-match fields such as transport layer port numbers.

But, there is still work to be done. We propose research into the development of an incremental update procedure, the use of multi-level indexing, and different filter grouping algorithms.

References

[1] F. Baboescu and G. Varghese, “Scalable packet classification,” Proc. of ACM Sigcomm ’01, September 2001.

[2] F. Baboescu, S. Singh, and G. Varghese, “Packet Classification for Core Routers: Is there an alternative to CAMs?” Proceedings of Infocom, 2003.

[3] A. Brodnik, S. Carlsson, M. Degemark, S. Pink. “Small Forwarding Tables for Fast Routing Lookups,” Proc. ACM SIGCOMM 1997, pp. 3-14, Cannes, France.

[4] M. M. Buddhikot, S. Suri, and M. Waldvogel, “Space decomposition techniques for fast layer-4 switching,” Proc. of PHSN, August 1999.

[5] W. Eatherton, “Hardware-based internet protocol prefix lookups,” MS Thesis, Washington University in St. Louis, Electrical Engineering Department, May 1999.

[6] A. Feldman and S. Muthukrishnan, “Tradeoffs for packet classification,” Proc. of Infocom, March 2000.

[7] P. Gupta and N. McKeown “Packet Classification on Multiple Fields,” in *Proc. ACM Sigcomm ’99,* Sept. 1999.

[8] P. Gupta, S. Lin, and N. McKeown, “Routing lookups in hardware at memory access speeds,” Proceedings of the Conference on Computer Communications (IEEE INFOCOM), (San Francisco, California), vol. 3, pp. 1241-1248, March/April 1998.

[9] P. Gupta and N. McKeown, “Packet classification using hierarchical intelligent cuttings,” Proc. of Hot Interconnects VII, August 1999.

[10] T. V. Lakshman and D. Stidialis, "High speed policy-based packet forwarding using efficient multi-dimensional range matching," Proc. of ACM Sigcomm '98, September 1998.

[11] B. Lampson, V. Srinivasan, and G. Varghese, "IP lookups using multiway and multicolumn search," Proceedings of the Conference on Computer Communications (IEEE INFOCOM), (San Francisco, California), vol. 3, pp. 1248-1256, March/April 1998.

[12] C. Matsumoto, "Cam vendors consider algorithmic alternatives," EE Times, May 2002.

[13] G. Narlikar, A. Basu, F. Zane, "CoolCAMs: Power-Efficient TCAMs for Forwarding Engines," Proc. of Infocom, 5/2003.

[14] E. Spitznagel, "Compressed Data Structures for Recursive Flow Classification," Washington University Department of Computer Science and Engineering, Technical Report WUCSE-2003-65 May 2003.

[15] E. Spitznagel, D. Taylor, and J. Turner, "Packet Classification Using Extended TCAMs," Proc. ICNP, November 2003.

[16] V. Srinivasan and G. Varghese, "Fast IP Lookups using Controlled Prefix Expansion," Proc. ACM Sigmetrics, June 1998.

[17] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel, "Fast and scalable layer 4 switching," Proc. of ACM Sigcomm '98, September 1998.

[18] V. Srinivasan, S. Suri, and G. Varghese, "Packet classification using tuple space search," Proc. of ACM Sigcomm '99, September 1999.

[19] D. Taylor, J. Turner, "Towards a Packet Classification Benchmark," Washington University Department of Computer Science and Engineering, Technical Report WUCSE-2003-42, May 2003.

[20] G. Varghese, et. al., "Packet Classification Using Multidimensional Cutting," Proc. of ACM SIGCOMM, August 2003.

[21] M. Waldvogel, G. Varghese, J. Turner, B. Plattner, "Scalable High-Speed IP Routing Lookups," Proc. ACM SIGCOMM 1998, pp.25-36, Cannes, France.

[22] T. Woo, "A Modular Approach to Packet Classification: Algorithms and Results," Proc. Infocom 2000.

[23] Packet Classification Repository. http://www.cs.ucsd.edu/~baboescu/repository/