Comparative Evaluation of Software Implementations of Layer-4 Packet Classification Schemes

Vivek Sahasranaman
Inktomi, Inc.
San Jose, CA
vivek@inktomi.com

Milind M. Buddhikot
Center for Network Software Research
Lucent Bell Labs
milind@dnrc.bell-labs.com

Abstract

In recent years, availability of fast network processors [1] and general purpose CPUs has made software implementation of per-packet processing in network elements an attractive option. Given this, a-priori knowledge of performance of software implementations of the well known Layer-4 packet classification will be very useful. In this paper, we compare the performance of three state-of-the-art packet classification schemes namely, Grid-of-Tries (GOT) [7], Packet Classification Algorithms using Recursive Space-decomposition (PACARS) [5], and Tuple-Space-Search (TSS) [8], implemented in FreeBSD 3.3 UNIX kernel. We developed two new OS extensions, namely, the Virtual Filter Database (VFD) framework and the new `routing socket API` *to implement these algorithms. We used real-life as well as synthetic rule databases to evaluate their performance. Our key conclusions are: (1) Compression of trie data structure that is central to a lot of classification algorithms has limited benefits on general purpose CPUs. (2) Static algorithms such as GOT that do not support dynamic updates support very fast search performance of the order of a few microseconds per search and may be adequate for static firewalls. (3) With medium sized databases, PACARS and TSS schemes provide update times of the order of 100s of microseconds and search performance of the order of 10s of microseconds. These algorithms are adequate for dynamic firewalls, traffic directors, and network monitoring applications in enterprise networks.*

1 Introduction

High performance, scalable implementation of new differentiated services such as Quality-of-Service (QOS) guarantees, Virtual Private Networks (VPNs), Network Address Translation (NAT), require space and time efficient solution to the problem of packet or flow classification. The basic *Packet(or Flow) Classification* problem is as follows : a network element that offers differentiated services maintains a database of N rules for processing incoming packets. Each rule R_i consists of a filter F_i and an associated action A_i. Filters are constraints on the values in the fields of the packet header. Each filter F_i has K fields $H[1], H[2], \ldots, H[K]$ corresponding to the fields in packet headers which it should match. Each of the header fields is assigned one of the four match types: *exact match*, *wild card match*, *prefix match*, and *range match*. If more than one filter matches an incoming packet, the tie is broken by using priority P_i assigned to each filter. For every incoming packet, the classification algorithm performs a *search* operation using the fields in the packet header to find the best matching filter in a rule and the executes the action associated with that rule. For IPv4 packets in today's Internet, fields such as IP source address, destination address, protocol ID, transport protocol port numbers, etc. are considered relevant fields. Actions associated with the rules vary with the application that uses classification.

Flow or packet classification algorithms reported in literature fall into four categories [4]: (1) Hardware approaches such as Ternary CAMS, bitmap intersections and special ASICS [4]. (2) Basic techniques such as caching and search data structures such as hierarchical tries and set pruning tries [7]. (3) Computational geometry based approaches such as PACARS (Area-based Quad Trees) [5], and Fat Inverted Segment Tree (FIS) [4]. (4) Heuristic based algorithms such as tuple space search [8] and hierarchical cuttings[4].

These algorithms can be compared using following three metrics: (1) **Search time**: The worst case search time decides the peak bandwidth a classification algorithm can sustain and should be of the order of few microseconds. (2) **Space:** A good algorithm usually stores each filter once or constant number of times in the database and thus requires linear ($O(N)$) space. Algorithms that create smaller data structures benefit from improved cache performance on a software platform. (3) **Insert and Delete times:** Applications, such as dynamic firewalls, monitoring, and QoS guarantees, that dynamically detect start (termination) of flows and add (delete) rules for detected flows, require insertion/deletion latencies of the order of 100s of microseconds to 10s of msecs. On the other hand, applications, such as route lookup, subscription services, (simple) static firewalls etc., that require updates less than once a second, *static* data structures which are optimized for search may suffice.

Interestingly enough, for several of the algorithms,, the average and the worst case values of the performance metrics differ dramatically. Often, heuristic based algorithms attempt to exploit special properties filter databases. If such properties are violated, performance can be dramatically different than the average case.

1.1 Motivation for our Research

The motivation of comparative evaluation of 2-D packet classification schemes is two fold: (1) The most general problem of 5-dimensional layer-4 packet classification is computationally a very hard problem to solve[4]. The 2-D packet classification problem is a restricted version of general 5-D classification problem, which involves prefix constraints on the IP source and destination addresses of a packet. If we have a fast solution for a 2-D filter classification problem, the restricted version of the 5-D problem can be solved using a simple scheme described in [7]. (2) Several commercially important applications such as Virtual Private Networks, Multicast Forwarding, web hosting services that host competing web sites and subscriptions and direct clients to different servers based on content, require fast 2-D packet classification.

The problem of performing flow classification in software is of great interest for several reasons: (1) With the advent of GHz speed general CPUs and introduction of new network processors [1], flow classification can be performed in software at faster rates. (2) Several devices such as edge routers, web traffic directors network monitoring tools, small firewalls, NAT devices, and bandwidth managers, often require limited throughput of the order of few 10s or 100s of Mbps. Compared to expensive specialized ASICs common in high end routers and switches, software implementations are more cost effective for these applications. (3) Software implementations allow reconfigurability, on-going fine tuning, and continued performance improvements with faster versions of CPUs, limited only by the memory bandwidth. Clearly, given the performance metrics discussed earlier, a priori knowledge of how well-known classification algorithms reported in literature perform will be of immense use to designers of such systems.

1.2 Contributions of our Research

In this paper, we report implementation and performance evaluation of three representative state-of-the-art 2-D packet classification schemes, namely (1) Grid-of-Tries (GOT) (Category 2)[7], (2) PACARS (Area Based Quad Tree (AQT)) (Category 3)[5], and (3) Tuple Space Search (TSS) (Category 4) [8], in 4.4 BSD UNIX (FreeBSD) kernel. The two new OS extensions: (1) the *Virtual Filter Database* framework and (2) the new `routing socket API` we developed to implement these schemes can be used for any layer 4/7 classification schemes. Our experimental evaluation sheds light on suitability of the three algorithms for various applications.

1.3 Outline of the rest of the paper

The rest of the paper is organized as follows : In Section 2, we provide an overview of three representative packet classification algorithms we chose to evaluate in our research. Section 3 describes our prototype implementation of these algorithms in 4.4 BSD (FreeBSD) Unix kernel. Section 4 presents performance measurements. We conclude in Section 5.

2 Overview of Algorithms Evaluated

In this section we summarize the three schemes we evaluate.

2.1 Grid of Tries (GOT)

Grid-of-Tries (GOT) algorithm, proposed in [7], is a simple generalization of the binary Patricia trie data structure [4] to two dimensional rule matching. In case of simple one dimensional prefix matching used in IP packet forwarding, Patricia trie stores forwarding rules described by prefixes at trie nodes, and uses the destination IP address in a packet to traverse the trie.

Grid-of-Tries extends basic trie to two dimensions, by maintaining two tries – a trie for destination address and a trie for source address in the packet. Each node in the destination trie, instead of storing a rule, now points to a relevant source trie, and each node of the source trie contains a rule that matches the appropriate destination and source prefix pair. An example, (taken from [7]), is shown in Figure 1. Since GOT stores each filter at only one location, its space requirement is $O(N)$. The search may visit all the source tries reachable from every destination trie node in the search path. Therefore, the filter search in a naive implementation

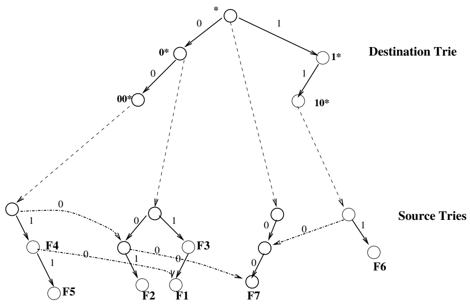


Figure 1. Grid of Tries with forwarding pointers

can in the worst case take $O(W^2)$ time. In our example, to search for the source and destination address pair $(00, 00)$, we have to traverse down each source trie for the destination trie nodes corresponding to $00*$ $0*$ and $*$, in that order.

GOT reduces the search time to $O(2W)$ by using simple pre-computation. If the source trie traversal fails, then, instead of backtracking along the destination trie and starting a search at the root of the next source trie, we can maintain a forwarding pointer, called *switchover pointer* to the appropriate node in the next source trie. Intuitively, this allows us to jump directly to the source trie node which has at least as good a source match as the current node.

When a new filter is added or deleted, in the worst case, GOT may require recomputation of all switchover pointers forcing the entire data structure to be reconstructed. Clearly, this precludes dynamic incremental updates and requires $O(N)$ time to construct the data structure.

2.2 Packet Classification using Recursive Space-decomposition (PACARS)

The PACARS scheme uses a quad tree – a tree where each node has four children to represent the hierarchically decomposed search space. With respect to the square space, if a node represents a square, then each of its children represents one of the four squares obtained by dividing the parent's square into four equal sub-squares. The concept of *crossing filter set of a region* enables PACARS to store every filter only at constant number of locations. Given a region A in the search space, the set of all filters that cross the region or terminate on its boundaries is called as its Crossing Filter Set (CFS). Filters that are fully contained in the region however, are not part of this set. For example, in Figure 2, filters $R_1, R_2, R_3, R_4, R_{10}$ belong to CFS for the square corresponding to entire region. but other filters don't as they are contained in the region.

The Area-based Quad Tree (AQT) combines binary space decomposition with a simple rule for prefix based filters to form CFS for each region of the decomposed space. Each node in a AQT at depth h has a square region of size $2^{32-h} \times 2^{32-h}$ associated with it. The rule to form its CFS is as follows: if a given square associated with a node is the *smallest* square that *terminates* or *fully-contains* a filter, then the filter is assigned to the CFS for that square.

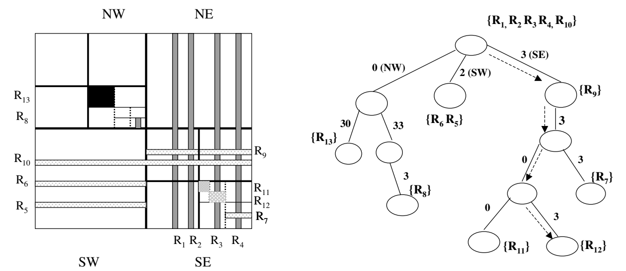


Figure 2. Example of Area-based Quadtree

Figure 2 illustrates an example of an AQT with 13 rectangles constructed using this rule [5]. It also illustrates the search operation for packet (1101,1101) which results in traversal of path shown by dotted line and a rule match with filter R_{12}.

The worst case run time of this simple data structure is $W \log N$. However, this can be improved to $O(W + logN) \leq O(2W)$ using the well known technique of fractional cascading [5].

Since a filter belongs to only one AQT node, filter addition or deletion requires only the search lists at a single node need to be changed with O(N) complexity. PACARS achieves faster incremental updates using the idea of *prefix partitioning* that puts N prefixes into $\sqrt(N)$ buckets each with $\sqrt(N)$ prefixes and attempts to maintain the partitioning in face of updates. In the amortized worst case, such partitioning leads to $O(2\sqrt(N))$ update time.

In our FreeBSD implementation of PACARS algorithm, we implemented search lists at each node as skiplists [6] and did not implement the fractional cascading. Therefore, the performance numbers we report for PACARS represent lower bounds.

2.3 Tuple Space Search

Tuple Space Search (TSS) algorithm [8] is a heuristic based algorithm that exploits properties of address aggregation in the internet. The main idea in TSS is to group filters into sets that have the same prefix lengths. In case of 2-D filters, two filters belong to the same set if they have the same prefix lengths in both the dimensions. For example, filters (01*, 01*) and (11*, 00*) both belong to the set characterized by the lengths (2,2) and therefore, are said to belong to the *tuple* (2,2). Now, given a search packet, we only have to find which tuples a filter that might match the packet can belong to. If we can find these tuples, we can search each of these by hashing with the appropriate number of bits from the source and destination addresses(which is fast in the common case). To find the tuples, we maintain two tries, searchable by the destination and source address respectively. Every node of each trie has a prefix X associated with it which corresponds to the search path from the root of the trie to the node. At each trie node, we maintain a bitmap of the sets the filters that match a prefix X. For example, on a destination trie node at 001*, we maintain a bitmap of sets that have a destination prefix of length 3, and contain at least one filter that has destination prefix 001*(and any source prefix). Similarly, we also populate the source trie.

To search for a particular packet, we traverse each trie, and obtain a set of potential tuples that matching filters may lie in. Since the filter must match in both the dimensions, the matching filter must lie in the intersection of the two sets. We search this set of potential tuples by hashing bits of the source and destination address of the packet. For example, if one of the tuples is (2,2), and if the source and destination addresses are 6fffffff and cf0cda04 respectively, then we choose two bits each from the addresses (which in this case are 01 and 11 respectively) and hash into the hash table with key (TupleID, 01,10), where *TupleID* is a unique number assigned to each tuple.

Insertion and deletion into this data structure are fast if we maintain an augmented list of the number of filters in each tuple at each trie node. This helps us remember, for each tuple t, the number of filters that belong to tuple t at any trie node. While deleting a filter, if the number of filters for a tuple goes to 0, then we can also delete the tuple from that node. Similarly, we increment the tuple count for tuple t when we insert a new filter belonging to the tuple t. The TSS scheme requires $O(W^2)$ worst case search time, $O(N)$ space, and $O(N^2)$ worst case update time. Clearly, its performance depends on the hashing functions used.

3 Implementation in 4.4 BSD UNIX

In the following, we will first describe a scheme for compressing trie data structure and then present some details of implementation of the three schemes in a 4.4 BSD UNIX kernel, namely FreeBSD Release 3.3.

3.1 Compressed Tries with Bitmaps

All of the three schemes described earlier use binary or quad-tries. Since the trie search time is proportional to the trie height, compressing the trie by allowing a node to have more children reduces search time. If we use a naive trie representation, every node requires up to four pointers, each potentially word in size. Combining such a representation with path compression however potentially blows up space required at each trie node and has limited benefits in a sparsely populated search trie. If we reduce the quadtree

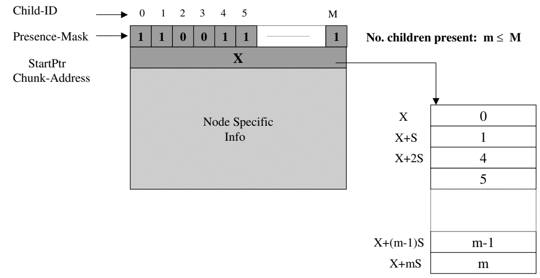


Figure 3. Node representation

height by a factor of k, we require a $2k$ bit trie where each node has $2^{2k} = 4^k$ children. So with $k = 4$, a naive implementation would require 256 pointers, one per child in each node. We can avoid this by using a better node representation where a node's children are stored contiguously in memory. Assume that each trie node is S bytes in size. If a node has m children, we allocate mS bytes of contiguous memory. The compressed representation of each node contains three fields (Figure 3): (1) a **start pointer(X)** that points to the start of the memory chunk that stores all the children of the node. (2) a **Child Presence-Mask** a bitmap p with 4^k bits – one bit to record presence or absence of each child. The children are numbered 0 to $4^k - 1$ from left to right and the presence (absence) of i^{th} child is indicated by marking the i^{th} bit in the childmap to 1 (0). If there are j bits before i^{th} bit that are zero, then the information about the i^{th} child is located at the address $X + S * (i - j)$. (3) Node specific information which depends on the nature of the classification algorithm. We call trie with such node representation as *Compressed Trie* (CT) or *Compressed Quad Tree* (CQT).

Table 1. Savings from CQT with bitmap

Type	Space
AQT without bitmaps	$4 * (4^k) = 4^{k+1}$
AQT with bitmaps	(4 (One word of start pointer) + $(4^k/8)$(Bytes for bitmap)) or $\lceil 0.5(4^{k-1} + 8) \rceil$ bytes
Savings of: $(4^{k+1} - 0.5(4^{k-1} + 8))/4^{k+1}$ = $1 - (0.5(4^{k-1} + 8)/4^{k+1})$	
For 8-bit tries (4-bit AQT), $k = 4$**, Savings=**$\approx 97\%$	

Table 1 characterizes the savings of compressed quadtrees with bitmaps. However, the CQT representation suffers from increased search processing and memory management overheads. Specifically, search requires bitmap computations at each node to locate the next node in the search path. These added bit operations may slowdown the search. The memory management overheads surface when a new child is added to a node. If the node has i children, addition of $(i+1)^{th}$ child requires following steps: (1) Allocate a new chunk of size $(i+1)S$, (2) copy contents of previous chunk of size iS to the new chunk at appropriate locations, (3) modify **start** pointer, and (4) modify the **childmap** variable using bitmap operations. Clearly, the memory allocation routines must support allocation of contiguous chunks of size ranging from S to $4^k * S$.

In the PACARS approach, tree compression destroys the Crossing Filter Set property, and the filter in a CFS at a node has to be duplicated in multiple children to regain this property. The space explosion resulting from this is unavoidable and bitmap representation does not reduce it any further. Clearly, compression may save space but may affect search and even update performance. In our experiments, we identify the optimal point in this time-space tradeoff for compressed tries in section 4.1.

3.2 FreeBSD In-Kernel Implementation

The two main ideas in our implementation are: (1) Virtual Filter Database (VFD) that resembles the Virtual File System (VFS) implementation in BSD UNIX. (2) A modified routing socket API that unifies operations on routing and filter databases.

Our basic VFD implementation architecture offers generic API consisting of two sets of functions on a virtual filter database: (1) Control path functions such as *insert, delete, update*, and (2) Data path functions invoked on every packet, namely *search*. Any classification algorithm implementation that exports these basic functions can be integrated by linking the generic API functions to the specific functions.

In the current 4.4 BSD UNIX OS, a *routing sockets* API is provided for routing daemons to add or delete forwarding entries from the IP forwarding table. We modified this API to enable user level applications such as RSVP daemon (rsvpd) to perform control path operations on filter database and receive special events generated during per-packet datapath operations such as search. Since routing sockets can only be opened by processes with special privileges, this method also provides protection against unauthorized access.

The original function call graph and the new additions in our implementation are shown in Figure 4. As shown, we add a special socket flag (SS_MONITOR), which can be set by an *ioctl* command on the routing socket. The new signaling daemons for layer-4 services, set this flag for their routing sockets. When such a daemon issues a `read` call on a routing socket, the `my_soreceive` also reads an event queue in which special packets or events are enqueued by the *search* process. If there are pending events or packets, they are returned to the daemon, which can process them and take appropriate action. Example scenarios where this capability is useful are events corresponding to start or end of a new TCP connection, in response to which the daemon can ad or /delete a new filter Existing routing daemons such as `routed` do not set SS_MONITOR flag for their routing sockets and are therefore unaffected.

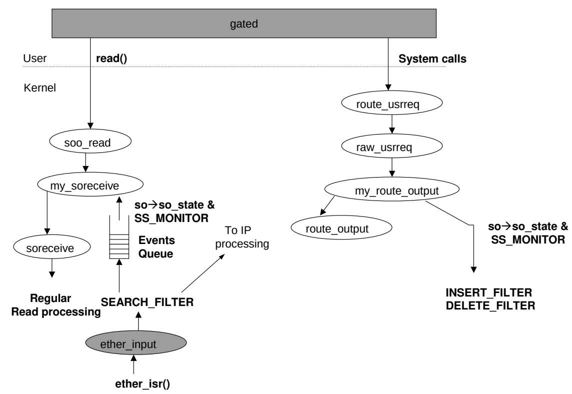


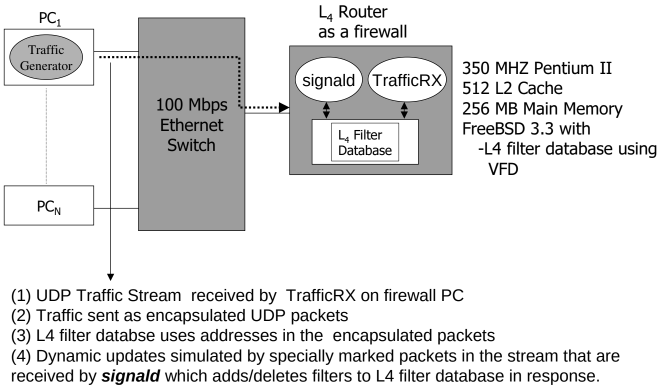
Figure 4. Modified control and data paths

Our implementation currently supports `Allow`, `Deny` and `Add_event` actions for rules and implicitly drops packets that do not match any rules. It calls search function in the ethernet Interrupt Service Routine (ISR) instead of higher layer (such as IP layer), which allows us to drop a packet if it does not match an allow rule and save on higher layer processing. Clearly, the classification speed defines the bottleneck bandwidth and the largest burst that we can sustain equals the size of the Ethernet card buffer.

4 Performance Evaluation

One problem with testing flow classification algorithms is the lack of standardized test data. The most commonly available test data are firewall databases, which are small (not more than a thousand rules) and specialized (they usually contain more constraints on port numbers than on IP address prefixes). Given this, in addition to using real-life firewall databases obtained from authors of [7, 8], we also use synthetic rule databases generated based on the type of application that is expected to use our candidate algorithms. We used two sets of synthetic databases: (1) first set was generated using our own scripting tool, (2) the other set was obtained from the research paper by Woo [9].

Figure 5 illustrates and explains our test setup consisting of two Pentium II PC's, linked by a 100 Mbps switched Ethernet.



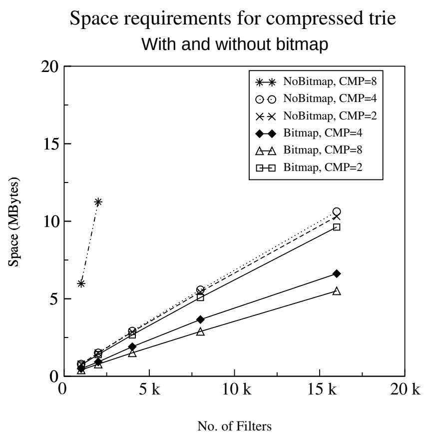


Figure 6. Trie size vs. degree of compression

4.1 Experiment I: Effects of Compression on Optimal Trie

As explained in Section 2, search operation in each of the algorithms we evaluate involves a trie lookup at some point. The compressed trie scheme we discussed in Section 3.1, attempts to reduce the search time by reducing the trie height. However, compression, if used without bitmaps can lead to a blowup of space due to allocation of superfluous space for more empty nodes. On the other hand, using bitmaps increases search time due to the additional cost of doing bit operations on the bitmap to decide next child in the search path. Here, we quantify the overhead of compressed tries with bitmaps to decide the tradeoff of space and search time for optimal trie. For this experiment, we plot the time taken to traverse the trie in case of the PACARS algorithm. We use filters generated randomly using a routing table database to take into account address aggregations on the Internet. In all the cases we also insert fully defined filters which cause traversal of full search path down to the leaf of the trie.

We vary the *compression factor* of the trie, which is defined as the logarithm of the number of children that a node can have. We should note that for PACARS, the smallest compression factor is 2, because a node has at least four children. The search time and the space required for the trie st

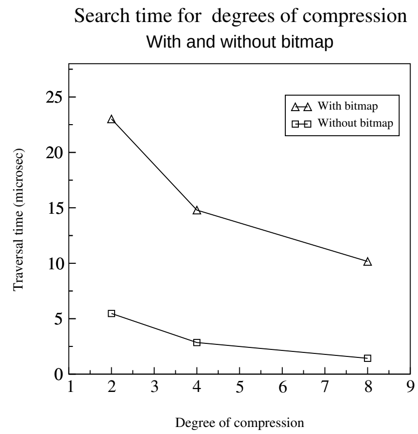


Figure 7. Trie traversal time vs. degree of compression

lowing observations from our experiments: (1) Increasing *compression factor* from two to four makes little difference to the size of the data structure, but a huge difference to the traversal time. This is because with a smaller number of children, we are not allocating too much more extra space when we compress from two to four. However, we notice that a compression to 8 causes a huge memory blowup, and hence a bitmap will be required if we compress by a factor of 8. (2) Bitmap versions take three to four times more time to traverse the trie. The reason for this is that we do not exploit any special instruction support for bitmap operations in software. We believe that the bitmaps are potentially useful in hardware implementations.

We conclude that a tree with a compression factor of 4 that results in a node branching factor of 16 seems to be an optimal trie for the given set of filters. Increasing compression factor to 8 to save time, or to a bitmap version to save space, seems to provide limited gains. The performance of all other schemes is expected to be similar because they all have the same trie implementation, and only differ in what is stored in each trie node or the trie depth. Therefore, we use a compressed trie without bitmaps and with compression factor of 4 in all our experiments.

4.2 Experiment II: Static filter sets

Several applications such as simple firewalls, web subscription services update filter sets infrequently, once every several seconds. In such cases, the database can be rebuilt in the event of an update and incremental update capability is unnecessary. We simulate this scenario by inserting two datasets: (1) a set of real firewall filters [7, 8], and (2) a set of random filters generated using a Perl script, in the databases before any searches are performed. The packets generated choose their addresses such that they match a filter in the database with uniform probability. Real life packets will probably match few select filters with higher probability than others, so our numbers are a conservative estimate of the reality. Search times are reported for tries with compression factor of 4 and are an average over two hundred thousand packets. The performance for each algorithm for the firewall filters is shown in Table 2.

Table 2. Performance for firewall databases

Search time (microsec)			
Firewall Filter Set	Grid-of-Tries	TSS	PACARS
F1(72 Filters)	1.63	6.25	5.511
F2(128 Filters)	2.017	6.72	6.608
F3(60 Filters)	1.798	6.07	5.38
Build time (microsec)			
F1 (72 Filters)	1212	53.39	52.11
F2(128 Filters)	3033	54.60	88.34
F3(60 Filters)	781	56.41	51.34
Space required (KBs)			
Firewall Filter Set	Grid-of-Tries	TSS	PACARS
F1 (72 Filters)	32.06	229.48	13.7
F2(128 Filters)	63.376	257.44	34.989
F3(60 Filters)	25.88	224.88	9.30

We observe that Tuple Space Search performs worst for the firewall databases. Further analysis revealed each search takes a large number of hashes (2.5 hashes for the second firewall which had 22 tuples) relative to the total number of tuples for these databases. This is expected, as the addresses for a firewall database are very localized, and hence tuples are expected to be concentrated along the search path. This suggests that tuple space search may not be the best for a firewall application.

The search performance of the three schemes for random filters is shown in Figure 8. The large search times for tuple space search on random filters affirms that this algorithm can perform poorly in absence of address aggregation assumed in its design. The search times correlate directly with the number of hashes required per search.

The values for PACARS are a bit surprising. Further investigation revealed that high search times were a result of complex skiplist searches at large number of lists on the search path. We noted in section 2.2 that a search in PACARS has to potentially search every node along the search path. In case of random filters, a large fraction of quadtree nodes are populated, causing a lot of skiplists searches to happen along the search path. The time to search

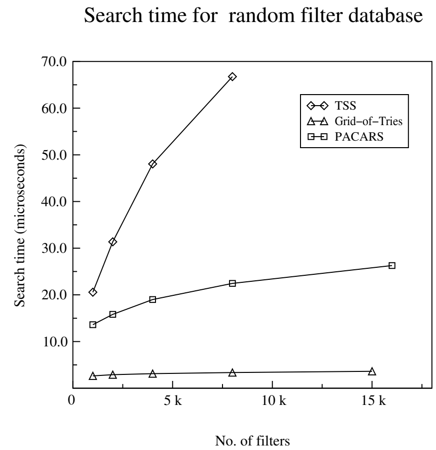


Figure 8. Search time for random filters

these lists is expected to grow logarithmically, but the complexity of the data structure adds a large constant factor for *every non empty list search*. Table 3 shows the variation of skiplist search time with number of elements at the root node, which clearly shows the large constant factor. As the

Table 3. PACARS correlation of root occupancy to search time : random filters

# of filters	# at root	Search Time(μsec)
1000	59	3.412
2000	117	4.287
4000	206	5.004
8000	408	6.089

number of filters increases, more nodes in the search path start to have large occupancies resulting in increasing search time. Our measurements show that PACARS performance will improve significantly with the use of fractional cascading, which threads lists on a single search path to reduce the search to just searching the root node, plus a constant lookup at each subsequent node on the search path.

The search times for GOT are very low and almost constant. If we refer back to the GOT search procedure, we see that the worst search times should remain constant independent of the number of filters. The small increase we see is attributed to two factors : (1) The search times we measure are average times, hence they might not search the entire depth of the tries. As we increase the number of filters, the probability of occupancy at greater depths increases, causing more searches to have the worst case performance. (2) Cache causes a significant difference in performance at such small search times. If only one fully specified filter is inserted into a Grid of Tries, the time taken for search is 1.88 μsec. With larger filter sets, we are expected to go along different paths on the search trie, and hence have worse cache performance.

4.3 Experiment III: Dynamic updates

The dynamic filter database updates are common with applications that insert/delete filters per flow after a flow is detected or terminated. Since, GOT does not support incremental updates, in this study we consider only TSS and PACARS schemes.

In the first experiment, the flows as well as the static filters choose their source and destination addresses from a MAE EAST router database [3]. The probability of choosing the prefix lengths for flows is taken from an empirically observed distribution of prefix lengths for traffic([2]). Static filters choose their source and destination prefixes with uniform probability from the router database.

In the second experiment, we use the filters from [9]. In this filter set, the addresses are chosen randomly, but the address aggregations are roughly characteristic of what we would see at the network edge. For example, there is a fixed fraction of fully specified filters corresponding to VPN like applications, and there are fixed proportions of filters with aggregations at levels corresponding to single subnet, multiple subnets and an entire domain. We choose the filters for flows to have slightly different characteristics: they have no wild-cards and no VPN like filters, since we would not expect dynamic updates to have these characteristics. Several real applications such as QoS applications like RSVP [10], and route lookup frameworks that populate the routing table on demand, will use the above setup.

The insert and delete numbers reported here are averaged over approximately 10000 updates, whereas the search times are averages over four hundred thousand classifications.

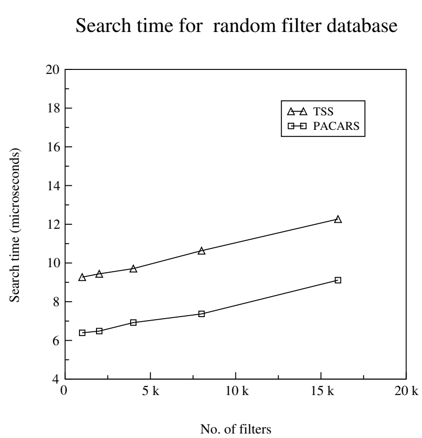


Figure 9. Search time for dynamic databases

The insert/delete performance of Tuple Space Search and PACARS for both experiments are shown in Figures 10. We notice that PACARS and TSS support very low insert and delete times that are always in the range of less than a hundred microseconds and PACARS seems to outperform TSS. With both filter sets, barring a few anomalous filter sets, the update performance of PACARS stays relatively stable. This is mainly because of the characteristics of the filter set, which has no wild-card filters, and usually has filters of high prefix lengths(since prefix length probabilities are high around the class B and C domains). In this case, filters are inserted and deleted at nodes that are deeper in the quad tree and have low list occupancy (usually just 1). This results in very few non-empty lists on any search path and a vast improvement in search performance compared to the case of random filters. The degradation of insert performance with random filters is minimal. In fact, the delete performance is better for random filters. These variations can be due to filter set dependencies. Also, note that tu-

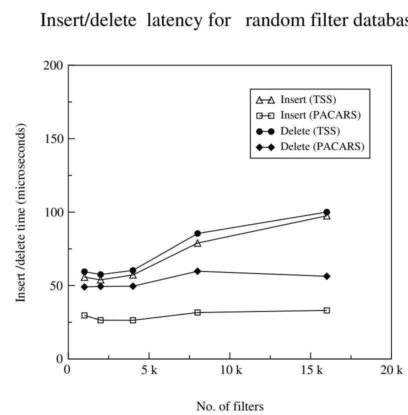


Figure 10. Insert/delete time for dynamic databases

ple space search improves with ”real” address prefixes compared to random filters as the address aggregation assumptions hold. The slight degradation of search performance as we increase the number of filters is partly because of higher number of hashes required on an average and partly because of overhead of tuple bitmap operations.

Insert and delete performance of TSS degrade with more filters because of use of tuple bitmaps to maintain tuple aggregate counts. Recall from section 2.3 that we need to maintain reference counts of each tuple on each node. Since each node is expected to hold a very few tuples, allocating a large array to hold all the tuple counts is a waste of space. So we only allocate space for tuples that are already present, which requires counting ones in the tuple bitmap to determine the tuple counts. As the number of filters increases, more tuples will be stored at each node, forcing increased counting overhead.

In comparison with static filter sets, we can see that the penalty for fast inserts is slower search (by a factor of 3-4 compared to Grid of Tries). We also find that performance of PACARS and TSS does not differ much for realistic address aggregations. Tuple space search performance improves dramatically compared to completely random filter sets if we adopt the filter generation strategy of [9], which restricts the prefix lengths while choosing the addresses at random. However, Tuple Space Search still performs worse than PACARS on these filters. In such a case, it is always preferable to use PACARS, because of its predictable and more uniform behavior for any kind of filter database.

4.4 Discussion

If we revisit the Section 2, on the face of it, the PACARS scheme has all the desirable properties, whereas suboptimal worst-case update performance of Grid-of-Tries (GOT) makes it undesirable for dynamic update applications. Our performance results confirm this: we showed that GOT provides superior search performance of the order of a few microseconds per search but its insert/delete or build performance deteriorates linearly for large database sizes making it a poor choice for dynamic applications. PACARS without fractional cascading seems to provide excellent update performance of the order of 10s-100s of microseconds and a modest search performance. We showed that for large databases adding fractional cascading to PACARS quadtree search will improve its performance significantly.

Table 4. Code size for various algorithms

Algorithm	Code size (lines of C code)
GOT	1074
TSS	1385
PACARS	3854

Table 4 shows the code size for our implementations of three algorithms obtained by excluding the common code base and counting only distinct code. Note that adding fractional cascading to PACARS will increase its foot print further. We see that though PACARS and GOT seem to have same performance in algorithmic notation, GOT is much simpler to implement than PACARS. Also, though all algorithms required $O(N)$ space, GOT is more space efficient. Clearly, in case of applications that use static filter databases GOT will be a clear winner.

A comparison of the performance of PACARS, and TSS schemes that are capable of *dynamic updates* to a static GOT algorithm reveals that these algorithms provide search performance that is four times slower on realistic databases. Their search performance on random databases can be even worse. Even though Tuple-space-search supports dynamic updates, its search performance is dependent on perfect hashing and its dynamic update performance is inferior to PACARS. On the other hand, PACARS provides uniform and predictable performance, as the quad tree data-structure is deterministic and invariant for a given database. We conclude that PACARS with fractional cascading though more complicated than TSS should be algorithm of choice when high search and update performance is required and modest increase in implementation complexity is acceptable.

5 Conclusions

In this paper, we reported our experiences in implementation and evaluation of three representative 2-dimensional packet classification schemes, namely Grid-of-Tries, PACARS with Area Based Quad Trees, and Tuple Space Search, in a 4.4 BSD UNIX kernel. Our measurements showed that GOT scheme is simple to implement and should be algorithm of choice for static applications that need fast search and infrequent updates. Similarly, for dynamic applications that require fast update and modest search performance, PACARS represents an excellent choice. We also showed that without the use of specialized bitmap instructions, compressed tries trade off search performance for reduction in space and the compression factor needs to be carefully selected.

References

[1] http://developer.intel.com/design/network/products/npfamily/ixp1200.htm,

[2] http://www.caida.org/outreach/resources/learn/prefixlengths/, “Prefix Lengths Distribution at FixWest for 1998/03/12”

[3] Routing Databases, www.merit.edu/ipma/routing_table/mae-east/.

[4] *Special Issue of IEEE Network on Fast Layer-3/4 Packet Classification*, ed. Buddhikot, M., Mckeown, N., and Varghese, G.

[5] M. Buddhikot, et al., “Space layer four switching”, *IFIP pfHSN'99*, Aug. 1999.

[6] Bill Pugh, “Skiplists : A probabilistic Alternative to balanced trees”, *CACM*, June 1990.

[7] V. Srinivasan, et al., “Fast and Scalable Layer Four Switching”, *ACM SIGCOMM*, Sept. 1998.

[8] V. Srinivasan, et al. “ Packet Classification using tuple Space Search”, *ACM SIGCOMM*, Sept. 1999.

[9] T. Woo , “A Modular Approach to Packet Classification : Algorithms and Results”, *IEEE Infocom*, Mar. 2000.

[10] Zhang, L., et al., “RSVP : A new ResourceReServation Protocol”, *IEEE Network*, September 1993.