Fast and Scalable Layer Four Switching

V. Srinivasan[*] G. Varghese[†] S. Suri[‡] M. Waldvogel[§]

Abstract

In Layer Four switching, the route and resources allocated to a packet are determined by the destination address as well as other header fields of the packet such as source address, TCP and UDP port numbers. Layer Four switching unifies firewall processing, RSVP style resource reservation filters, QoS Routing, and normal unicast and multicast forwarding into a single framework. In this framework, the forwarding database of a router consists of a potentially large number of filters on key header fields. A given packet header can match multiple filters, so each filter is given a cost, and the packet is forwarded using the *least cost matching filter*.

In this paper, we describe two new algorithms for solving the least cost matching filter problem at high speeds. Our first algorithm is based on a grid-of-tries construction and works optimally for processing filters consisting of two prefix fields (such as destination-source filters) using linear space. Our second algorithm, cross-producting, provides fast lookup times for arbitrary filters but potentially requires large storage. We describe a combination scheme that combines the advantages of both schemes. The combination scheme can be optimized to handle pure destination prefix filters in 4 memory accesses, destination-source filters in 8 memory accesses worst case, and all other filters in 11 memory accesses in the typical case.

1 Introduction

With everyone building Web Sites, Internet usage has been expanding at a rate more commonly associated with nuclear reactions. Internet traffic is exploding because of a growing number of users as well as a growing demand for bandwidth intensive data. Multimedia applications, for instance, can easily consume megabytes of bandwidth. To keep up with increased traffic, link speeds in the Internet core have been increased to 622 Mbps, and a number of vendors are providing faster routers.

A *traditional* router performs two major tasks in forwarding a packet: *looking up* the packet's destination address in the router database, and *switching* the packet from an incoming link to one of the outgoing links. With recent advances [18, 30], the task of switching is well understood, and most vendors use fast buses or crossbar switches. Several new algorithms have been developed recently for address lookup as well [9, 31, 22, 27]. Thus it would appear that there is no inherent impediment to building Gigabit routers for traditional data forwarding in the Internet.

Increasingly, however, users are demanding, and some router vendors are providing, a more discriminating form of router forwarding. To quote John McQuillan [19]:

Routing has traditionally been based solely on destination host numbers. In the future it will also be based on source host or even source users, as well as destination URLs (universal resource locators) and specific business policies ... Thus, in the future, you may be sent on one path when you casually browse the Web for CNN headlines. And you may be routed an entirely different way when you go to your corporate Web site to enter monthly sales figures, even though the two sites might be hosted by the same facility at the same location... An order entry form may get very low latency, while other sections get normal service. And then there are Web sites comprised of different servers in different locations. Future routers and switches will have to use class of service and QoS to determine the paths to particular Web pages for particular end-users. All this requires the use of layers 4, 5, and above.

This new vision of forwarding is called *Layer 4 Forwarding* because routing decisions can be based on headers available at Layer 4 or higher in the OSI architecture. Layer 4 Switching offers increased flexibility: it gives a router the capability to block traffic from a dangerous external site, to reserve bandwidth for traffic between two company sites, and to give preferential treatment to one kind of traffic (e.g., online database transactions) over other kinds (e.g., Web browsing). Layer 4 switching is sometimes referred to in the vendor literature [28] by the phrase "service differentiation". Traditional routers do not provide service differentiation because they treat all traffic going to a particular Internet address in the same way. Layer 4 Switching allows service differentiation because the router can distinguish traffic based on origin (*source address*) and application type (e.g., web

[*]Computer Science Department, Washington University, St. Louis. Research supported in part by NSF Grant NCR-9628145.

[†]Computer Science Department, Washington University, St. Louis. Research supported in part by NSF Grant NCR-9628145 and an ONR Young Investigator Award.

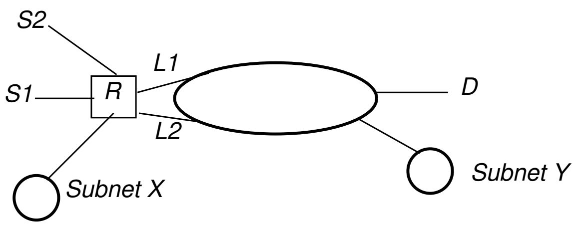
[‡]Computer Science Department, Washington University, St. Louis. Research supported in part by NSF Grant NCR-9628145

[§]Computer Science Department, ETH, Zurich.

traffic vs. file transfer).

Layer 4 Switching, however, does not come without some difficulties. First, a change in higher layer headers will require reengineering the routers, which is why routers have traditionally used only Layer 3 headers. Second, when data is encrypted for security, it is not clear how routers can get access to higher layer headers.

Despite these difficulties, several variants of the Layer 4 switching have already evolved in the industry. First, many routers implement *firewalls* [6] at trust boundaries, such as the entry and exit points of a corporate network. A firewall database consists of a series of packet filters that implement security policies. A typical policy may be to allow remote login from within the corporation, but to disallow it from outside the corporation. Second, the need for predictable and guaranteed service has lead to proposals for reservation protocols like RSVP [32] that reserve bandwidth between a source and a destination. Third, the cries for routing based on traffic type have become more strident recently—for instance, the need to route web traffic between Site 1 and Site 2 on say Route A and other traffic on say Route B. Figure 1 illustrates some of these examples.



Database at Router R

To	From	TrafficType	Forwarding Directive
D	S1	Video	Forward via L1
*	S2	*	Drop all traffic
Y	X	*	Reserve 50 Mbps

Figure 1: Example of filters that provide traffic sensitive routing, a firewall rule, and resource reservation. The first filter routes video traffic from S1 to D via L1; not shown is the default routing to D which is via L2. The second filter blocks traffic from an experimental site S2 from accidentally leaving the site. The third filter reserves 50 Mbps of traffic from an internal network X to an external network Y, implemented perhaps by forwarding such traffic to a special outbound queue that receives special scheduling guarantees; here X and Y are prefixes.

Once users have gotten used to the flexibility and features provided by firewalls, traffic reservations, and QoS routing, it is hard to believe that future routers can ignore these issues. The genie appears to be out of the bottle, or the camel has entered the tent, depending on one's point of view. On the other hand, it seems clear that the ad hoc solutions currently being deployed are not the best, and cleaner and more general techniques are possible. For example, a cleaner solution to the traffic sensitive routing and reservation problem would be to push some form of "traffic classifier" into the routing header to determine application requirements without inspecting higher layer headers[1]. But whatever the final solutions will be, it seems clear that future routers will need to forward at least some traffic based on a combination of destination address, source address and some other classifier fields, whether they are in the routing (Layer 3) or higher layer (Layers 4 and up) headers.

In this paper, we take a neutral stance on the issue of choosing which combination of fields should be used for a particular function, and even on the issue of deciding which functions are most useful. Instead, we concentrate on a general problem where a router forwarding database consists of a number of filters, each of which is a conjunction of either exact, range, or prefix matches on a set of packet fields. We describe a family of efficient algorithms for finding the best matching filter for a given packet, which then determines the packet's route, resource allocation and access rights. We are especially concerned with finding algorithms that are *efficient* (i.e., implementable at Gigabit speeds), but are also scalable to large numbers of filters (say, $100K$ filters) with reasonable memory costs.

Firewalls today contribute only a few (10-100 typically) filters. However, if we consider that backbone routers [20] have 40,000 prefixes, and if we qualify each destination prefix with even a few port numbers (e.g., for QoS routing) or source prefixes (e.g., for resource reservation between sites in a Virtual Private Network), it is not hard to imagine the need for several hundred thousand filters. Today, even firewall processing with 10-100 filters is generally slow because of linear search through the filter set, but is considered an acceptable price to pay for "security". Thus the problem of finding the best matching filter for up to $100K$ filters at Gigabit speeds is an important challenge.

The rest of the paper is organized as follows. We formulate the best matching filter problem precisely in Section 2. In Section 3, we briefly discuss related work. In Section 4, we show how to replace range matches by prefix matches. In Section 5, we describe our first new scheme, the grid-of-tries. Our second scheme, cross-producting, is described in Section 6. We discuss lower bounds that show the difficulty of the general filter problem in Section 7. We present a scheme that combines the best features of grid-of-tries and cross-producting in Section 8. Finally, we discuss implementation results in Section 9, and conclude in Section 10.

2 The Best Matching Filter Problem

Traditionally, the rules for classifying a message are called *filters*, (or rules in firewall terminology) and the Layer 4 Switching problem is to determine the lowest cost matching Filter for each incoming message at a router.

We assume that the information relevant to a lookup is contained in K distinct *header fields* in each message. These header fields are denoted $H[1], H[2], \ldots, H[K]$, where each field is a string of bits. For instance, the relevant fields for an IPv4 packet could be the Destination Address (32 bits), the Source Address (32 bits), the Protocol Field (8 bits), the Destination Port (16 bits), the Source Port (16 bits), and TCP flags (8 bits). The number of relevant TCP flags is limited, and so we prefer to combine the protocol and TCP flags into one field—for example, we can use TCP-ACK to mean a TCP packet with the ACK bit set.[2] Other relevant TCP flags can be represented similarly; UDP packets are represented by $H[3] = UDP$.

Thus, the combination (D, S, TCP-ACK $, 63, 125$), denotes the header of an IP packet with destination D, source

[1] We are grateful to Craig Partridge and John Wrocklawski for sharing their ideas and opinions with us.

[2] TCP flags are important for packet filtering because the first packet in a connection does not have the ACK bit set while the others do. This allows a simple rule to block TCP connections initiated from the outside while allowing responses to internally initiated connections.

S, protocol TCP, destination port 63, source port 125, and the ACK bit set.

The *filter database* of a Layer 4 Router consists of a finite set of filters, $F_1, F_2 \ldots F_N$. Each filter is a combination of K values, one for each header field. Each field in a filter is allowed three kinds of matches:[3] *exact match, prefix match,* or *range match*. In an exact match, the header field of the packet should exactly match the filter field—for instance, this is useful for protocol and flag fields. In a prefix match, the filter field should be a prefix of the header field—this could be useful for blocking access from a certain subnetwork. In a range match, the header values should lie in the range specified by the filter—this can be useful for specifying port number ranges.

Each filter F_i has an associated directive act_i, which specifies how to forward the packet matching this filter. The directive specifies if the packet should be blocked. If the packet is to be forwarded, the directive specifies the outgoing link to which the packet is sent, and perhaps also a queue within that link if the message belongs to a flow with bandwidth guarantees.

We say that a packet P *matches* a filter F if each field of P matches the corresponding field of F—the match type is implicit in the specification of the field. For instance, if the destination field is specified as $1010*$, then it requires a prefix match; if the protocol field is UDP, then it requires an exact match; if the port field is a range, such as 1024–1100, then it requires a range match. For instance, let $F =$ ($1010*$, $*$, TCP, 1024–1080, $*$) be a filter, with $act = block$. Then, a packet with header ($10101\ldots111$, $11110\ldots000$, TCP, 1050, 3) matches F, and is therefore blocked. The packet ($10110\ldots000$, $11110\ldots000$, TCP, 80, 3), 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 determine an unambiguous match. So each filter F in the database is associated with a non-negative number, $cost(F)$, and our goal is to find the filter with the least cost matching a packet's header. Our cost function generalizes the implicit precedence rules that are often used in practice to choose between multiple matching filters. In firewall applications, for instance, rules or filters are placed in the database in a specific linear order, where each filter takes precedence over a subsequent filter. Thus, the goal there is to find the *first* matching filter. Of course, we can get the same effect in our scheme, by making $cost(F)$ equal the position number of F in the database.

As an example of a filter database, consider the firewall database [6] shown in Figure 2, where a screened subnet configuration is assumed. There is a so-called bastion host M within the company that mediates all access to and from the external world. M serves as the mail gateway and also provides external name server access. TI, TO are Network Time Protocol (NTP) sources, where TI is internal to the company and TO is external. S is the address of the secondary name server which is external to the company. All addresses of machines within the company's network start with the CIDR prefix Net. Thus M and TI both match the prefix Net.

As an example, consider a packet sent to M from S with UDP destination port equal to 53. This packet matches Filters 2, 3, and 8, but must be allowed through because the first matching filter is Filter 2.

[3] It is possible to extend the type of matches for greater flexibility; we illustrate our examples using these three most common types.

Destination	Source	Destination Port	Source Port	flags	comments
M	*	25	*	*	allow inbound mail
M	*	53	*	UDP	allow DNS access
M	S	53	*	*	secondary access
M	*	23	*	*	incoming telnet
TI	TO	123	123	UDP	NTP time info
*	Net	*	*	*	outgoing packets
Net	*	*	*	TCP ack	return ACKs OK
*	*	*	*	*	block everythingl

Figure 2: Sample firewall database "for a small company" as described in the book by Cheswick and Bellovin [6]. The $block$ flags are not shown in the figure; the first 7 filters have $block = false$ (i.e., allow) and the last filter has $block = true$ (i.e., block).

3 Related Work

There does not appear to be any work directly related to fast filter processing. Packet filters for demultiplexing have been used for some time (for instance, see [1, 17, 11]), but they solve a somewhat different problem. Filters specify different matching rules, allow wildcards and address ranges in arbitrary fields, and require that we return the first matching filter. The IP address lookup problem is the one most closely related to our problem; however, the IP lookup problem is simpler than and a special case of the filter problem. Our cross-producting scheme uses best matching prefix as a building block for packet filtering.

An unpublished paper by Paul Tsuchiya [29] describes a data structure called *Cecilia tries* for dealing with *non-contiguous* IP net masks. Cecilia tries can be generalized to what we call *set pruning trees*, and can be used for Layer 4 switching [7]. Unfortunately, the scheme suffers from a memory explosion, which makes it impractical when the filter database size becomes large. Figure 5 shows an example for which Tsuchiya's scheme, as well as many other simple methods, have exponential memory blowup. In Section 5.1 we describe the basic idea behind set pruning trees.

Several existing firewall implementations do a linear search of the database, and keep track of the best matching filter. Some implementations use caching to improve performance—they cache full packet headers to speed up the processing of future lookups. Now the cache hit rate of caching full IP addresses in routers is at most 80–90% [23, 21]; cache hit rates are likely to be much worse for caching full headers. Incurring a linear search cost to search through $100,000$ filters is a bottleneck even if it occurs on only 10 to 20% of the packets.

The least cost matching filter can be thought of as a special case of a very general *multidimensional searching problem*. Several general solutions exist for the problem. In particular, each K-field filter can be thought of as a K-dimensional *rectangular box*, and each packet header can be thought of as a *point* in the K-dimensional space. The least cost filter matching problem is to find the least cost box containing the header point. A general result in Computational Geometry offers a data structure requiring $O(N(\log N)^{K-1})$ space, and search time $O((\log N)^{K-1})$, where the logarithms are to the base 2 (for instance, see Section 2.3 in [24]). Unfortunately, the worst-case search and memory costs of this data structure are infeasible, even for modest values of N and K. For instance, when $N = 10,000$ and $K = 4$, the worst-case search cost is at least $13^3 = 2197$ and the memory cost is $2197N$.

A recent approach to Layer 4 switching is described in [15]. We have been unable to determine the details of this scheme. It appears to implement multi-dimensional range matching in hardware.

Another possible technique is to generalize binary search by using quad-tree like construction in higher dimensions. (See, for instance, [25].) Consider, for instance, destination-source filters, which correspond to a two-dimensional search. A filter $F = (D, S)$ can be mapped to a quad-tree cell (i, j) if D is i bits long and S is j bits long. Now, we can try to do a binary search by first matching the packet with the filters in the quad-tree cell $(W/2, W/2)$, where W is the maximum bit length of any destination or source prefix. The problem is that the probe outcome (fail or match) only eliminates one *quadrant* of the search space, and requires *three* recursive calls (not one, as in 1 dimension) to finish the search, which leads to a large search time. One possible way to avoid making three recursive calls is to precompute future matches using *markers*, but that leads to an infeasible memory explosion of $2^{W/2}$. We have also shown a lower bound on hashing schemes like [31] to show that they generalize poorly to multiple dimensions.

In summary, we believe that all existing methods lead to either a large blowup in memory or lookup time for the least cost filter problem.

4 Converting address ranges to prefixes

A filter field is sometimes specified as a range. A common example is a range of port numbers; for instance, a firewall filter may require that the source port be greater than 1023. An arbitrary range can be converted into a union of *prefix ranges*, where a prefix range is one that can be expressed by a prefix. For instance, in a 4-bit field, the prefix 10∗ expresses the range $[1000, 1011] = [8, 11]$.

Suppose we want to convert an arbitrary range X that lies within an enclosing binary range $[0, 2^k]$. Define an *anchored range* as one that has at least one endpoint at the end of the enclosing range. Then, the arbitrary range X can be split into at most two anchored ranges that lie within $[0, 2^{k-1} - 1]$ and $[2^{k-1}, 2^k]$. Each anchored range can be split into a logarithmic number of prefix ranges by constantly halving the range—at each stage, the halving contributes at most one prefix range. The net result is that we can represent an arbitrary subrange of $[0, 2^k]$ with at most $2k$ prefix ranges. As an example, with 16-bit port numbers the range ≤ 1023 can be expressed using the prefix range 000000∗. On the other hand, the range > 1023 can be expressed with 6 prefix ranges 000001∗, 00001∗ 0001∗, 001∗, 01∗, and 1∗.

Thus, for the rest of the paper, we assume that each filter field is a prefix.

5 Grid-of-tries

Our first scheme is based on tries. In its simplest form, a trie is a binary branching tree, with each branch labeled 0 or 1. The prefix associated with a node u is the concatenation of all the bits from the root to the node u. In Figure 4, for instance, the leftmost node in the Dest-Trie has prefix value 00; the node on the right has value 10. Our basic data structure, called a *grid-of-tries*, is designed to handle two-dimensional filters, such as destination-source pairs. We believe this is a significant algorithm in its own right because large backbone routers may have a large number of destination-source filters to handle virtual private networks and multicast forwarding.

The grid-of-tries can be extended, albeit with some loss of efficiency, to handle filters on other fields such as port numbers. This is described in Section 5.5. We start by explaining the basic two dimensional data structure using an example database of 7 destination-source filters, shown in Figure 3. Though our examples use destination-source tries, we note that the idea can be abstracted to handle filters with any two prefix fields (and the remaining fields completely wildcarded).

Filter	Destination	Source
F_1	0∗	10∗
F_2	0∗	01∗
F_3	0∗	1∗
F_4	00∗	1∗
F_5	00∗	11∗
F_6	10∗	1∗
F_7	∗	00∗

Figure 3: An example with 7 dest-source filters.

5.1 Set Pruning Trees

To motivate our grid-of-tries scheme, we begin by describing two dimensional set pruning trees. We build a trie on the destination prefixes in the database. Figure 4 illustrates the construction for the example database in Figure 3. Each valid prefix in the Destination Trie (Dest-Trie) points to a trie containing some source prefixes. The question is: which source prefixes should we store?

For instance, consider $D = 00$. Both filters F_4 and F_5 have this destination prefix, and so we need to store the corresponding source prefixes 1∗ and 11∗ at D. But storing only these filters is not sufficient, since filters F_1, F_2, F_3 also match whatever destination D matches. In fact, the wildcard destination prefix ∗ of F_7 also matches whatever D matches. Suppose we get a packet whose destination header starts with 00 and whose source address starts with 101. Then, the least cost filter matching this header is the lowest cost filter among $\{F_1, F_3, F_4\}$. This suggests we need to store at $D = 00$ a source trie containing the source prefixes for $\{F_1, F_2, F_3, F_4, F_5, F_7\}$, because these are the filters *whose destination is a prefix of* D. Figure 4 shows the complete data structure for the database in Figure 3.

In this trie of tries, we first match the destination of the header in Dest-Trie. This yields the longest match on the destination prefix. We then traverse the associated source trie to find the longest source match. As we search the source trie, we keep track of the lowest cost matching filter. Since all filters that have a matching destination prefix are stored in the source trie being searched, we find the correct least cost filter. This is the basic idea behind set pruning trees [29, 7].

Unfortunately, this simple extension of tries from one to two dimensions has a memory blowup problem. The problem arises because a source prefix can occur in multiple tries. In Figure 4, for instance, the source prefixes S_1, S_2, S_3 appear in trie associated with $D = 00∗$ as well as $D = 0∗$. A worst-case example forcing $\Theta(N^2)$ memory is created using the set of filters shown in Figure 5. The problem is that since destination prefix ∗ matches any destination header, each of the $N/2$ source prefixes are copied $N/2$ times, one for each destination prefix.

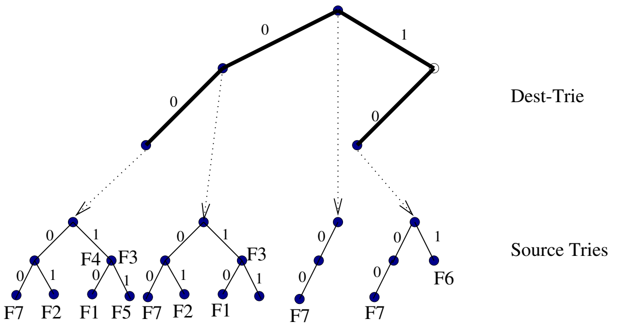


Figure 4: The first idea for grid-of-tries. It may require $\Theta(N^2)$ memory for N filters in the worst case. Dest-Trie is a trie for the destination prefixes. The nodes corresponding to a valid destination prefix in the database are shown as solid; others are shown as circles. Each valid destination prefix D has a pointer to a trie containing the source prefixes that belong to filters whose destination field is a prefix of D.

Filter	Destination	Source
F_1	D_1	$*$
F_2	D_2	$*$
	$\vdots$	
$F_{N/2}$	$D_{N/2}$	$*$
$F_{N/2+1}$	$*$	S_1
$F_{N/2+2}$	$*$	S_2
	$\vdots$	
F_N	$*$	S_N

Figure 5: An example forcing N^2 memory for two dimensional set pruning trees. Similar examples, that apply to a number of other simple schemes, can be used to show N^K storage for K dimensional filters.

5.2 Avoiding the Memory Blowup

In order to avoid the memory blowup of the simple trie scheme, we observe that filters associated with a destination prefix D are copied into the source trie of D' whenever D is a prefix of D'. For instance, in Figure 4, the prefix $D = 00$ has two filters associated with it: F_4 and F_5. The others F_1, F_2, F_3 are copied because their destination field 0 is a prefix of D; similarly, F_7 is copied because its destination field $*$ is also a prefix of 00.

We can avoid the copying by having each destination prefix D point to a source trie that stores the filters whose destination field is *exactly* D. This requires us to also modify the search strategy as follows: instead of just searching the source trie for the best matching destination prefix D, we must now search the source tries associated with all the *ancestors* of D.

In order to search for the least cost filter, we first traverse the Dest-Trie, and find the longest destination prefix D' matching the header. We search the source trie of D', and update the least cost matching filter. We then work our way back up the Dest-Trie, and search the source trie associated with every prefix of D' that points to a nonempty source trie.[4]

Since each filter now is stored exactly once, the memory requirement for the new structure is $O(NW)$, which is a significantly improvement over the the previous scheme. Unfortunately, the lookup cost in the new scheme is worse than the first scheme: in the worst-case, the lookup costs $\Theta(W^2)$, where W is the maximum number of bits specified in the destination or source fields. The $\Theta(W^2)$ bound on the search cost follows from the observation that, in the worst-case, we may end up searching W source tries, each at the cost of $O(W)$, for a total of $O(W^2)$.

[4]In this scheme, we could search each of the source tries corresponding to prefixes of the destination in any order without changing the search time; we used this particular order in order to motivate our final scheme.

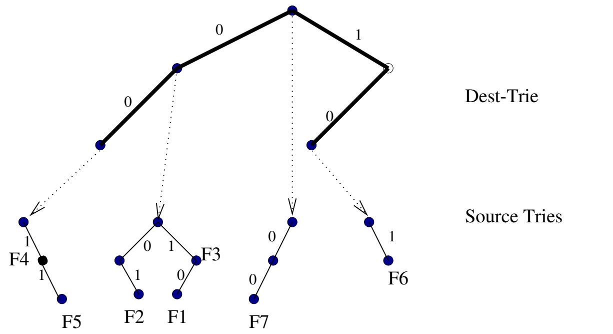


Figure 6: Avoiding the memory blowup, by storing each filter in exactly one trie.

5.3 Improving Search Time: Basic Grid-of-Tries

We now describe our key ideas for improving the search cost in two-dimensional tries from $O(W^2)$ to $O(W)$, while keeping the memory requirement linear. The key idea is to use *precomputation* and *switch pointers* to speed up search in a later source trie based on the search in an earlier source trie. Figure 7 shows the construction with switch pointers. The switch pointers are shown using dashed lines between source tries. This is to distinguish the switch pointers from the dotted lines that connect the Dest-Trie nodes to the corresponding source tries.

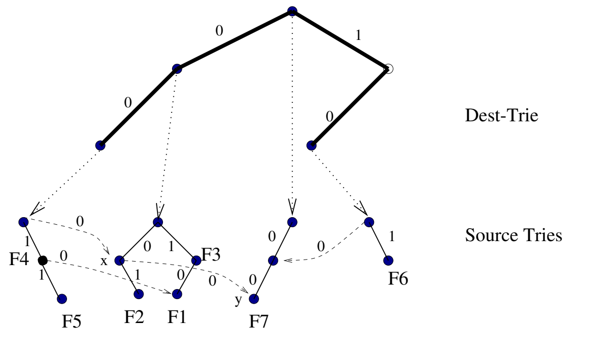


Figure 7: Improving the search cost with the use of switch pointers.

In order to understand the role of switch pointers, consider matching a packet with destination address 001 and source address 001. The search in the Dest-Trie gives $D = 00$ as the best match. So we start our search for the matching source prefix in the associated source trie, which contains filters F_4 and F_5. However, the search immediately fails, since the first bit of the source is 0. In the previous scheme, we would back up along the Dest-Trie and *restart* the search in the source trie of $D = 0*$, the parent of $00*$.

In the new scheme, however, we use the switch pointer to *directly jump* to the node x in source trie containing

$\{F_1, F_2, F_3\}$. Similarly, when the search on the next bit of the source fails again, we jump to the node y of the third source trie (associated with the destination prefix $*$). Intuitively, the switch pointers allow us to jump directly to the lowest point in the ancestor source trie which has at least as good a source match as the current node. This allow us to skip over all filters in the next ancestor source trie whose source fields are shorter than the current source match. This in turn improves the search complexity from $O(W^2)$ to $O(W)$.

We now define the *switch pointers* more precisely. We say that destination string D' is an *ancestor* of D if D' is a *prefix* of D. We say that D' is the *lowest ancestor* of D if D' is the *longest* prefix of D in the Destination Trie. Let $T(D)$ denote the source trie pointed to by D. (Recall that $T(D)$ contains the source fields of exactly those filters whose destination field is D.) Let u be a node in $T(D)$ that *fails* on bit 0; that is, if u corresponds to the source prefix s, then the trie $T(D)$ has no string starting with $s0$. Let D'' be the *lowest* ancestor of D whose source trie contains a source string *starting with prefix* $s0$, say, at node v. Then, we place a switch pointer at node u pointing to node v. If no such node v exists, the switch pointer is nil. The switch pointer for failure on bit 1 is defined similarly. For instance, in Figure 7, the node labeled x fails on bit 0, and it has a switch pointer to the node labeled y.

The switch pointers allow us to increase the length of the matching source prefix, without having to restart at the root of the next ancestor source trie. In particular, they allow us to skip over all filters in the next source trie whose source fields are shorter than the current source match.

For instance, consider the packet header $(00*, 10*)$. We start with the first source trie, pointed to by the destination trie node $00*$. We match the first source bit 1, which gives us filter F_4. But then we fail on the second bit, and therefore follow the switch pointer, which leads to the node in the second trie labeled with the filter F_1. The switch pointers at the node containing F_1 are both nil, and the search terminates. Note, however, that we have missed the filter $F_3 = (0*, 1*)$, which also matches the packet. While in this case F_3 has higher cost than F_1, in general the overlooked filter could have lower cost.

We solve this problem by having each node in a source trie maintain a variable *storedFilter*. Specifically, a node v with destination prefix D and source prefix S stores in *storedFilter*(v) the least cost filter whose destination field is a prefix of D and whose source field is a prefix of S. With this precomputation, the node labeled with F_1 in Figure 7 would store information about F_3 if F_3 had lower cost than F_1.

Finally, we argue that the search cost in the final scheme is at most $2W$. The time to find the best destination prefix is at most W. After that all the time is spent traversing the source tries. However, in each step, the length of the match on the source field increases by one—either by traversing further down in the same trie, or following a switch pointer to an ancestral trie. Since the maximum length of the source prefixes is W, the total time spent in searching the source tries is also W. The memory requirement is $O(NW)$, since each of N filters is stored only once, and each filter requires $O(W)$ space.

5.4 Further Improvements

Several improvements to the previous scheme are possible. First notice that the only role played by the Dest-Trie is in determining the longest matching destination prefix. The longest matching destination prefix tells us in which source trie to start searching. From that point on, the Dest-Trie plays no role, and we move among source tries using switch pointers. Thus, the first improvement is to replace the Dest-Trie with a fast scheme for determining the *best matching prefix* [9, 31] of the destination address. The scheme proposed in [31] requires $O(\log W)$ time in the worst-case for finding the longest matching prefix. Combining this scheme with the grid-of-tries leads to a total lookup time of $(\log W + W)$ for destination-source filters.

Second, instead of using 1-bit source tries, we can use multi-bit tries [27]. In multi-bit tries, we first expand each destination or source prefix to the next multiple of k. For instance, suppose we use $k = 2$. Then, in the example of Figure 7, the destination prefix $0*$ of filters F_1, F_2, F_3 is expanded to 00 and 01. The source prefixes of F_3, F_4, F_6 are expanded to 10 and 11. If we use k-bit expansion, a single prefix might expand to 2^{k-1} prefixes. The total memory requirement grows from $2NW$ to $NW2^k/k$, and so the memory blows up by the factor $2^{k-1}/k$. On the other hand, the depth of the trie reduces to W/k, and so the total lookup time becomes $O(\log W + W/k)$. Depending on the memory available, one can optimize the time-space tradeoff as in [27].

5.5 Extending Grid-of-tries to Handle Protocol and Ports

We now describe how to handle more general filters (with the protocol type and port number fields specified) using the grid-of-tries. We will assume that the port number field in each filter is either a single port number or a wild card.[5]

We partition the filters into a small number of classes, each of which only requires a lookup using the destination-source combination. First, we eliminate the Protocol field at the cost of increasing the memory by a factor of 3, as follows. There are two main protocols, TCP and UDP; all other protocols are grouped under the class "Other" for the purpose of packet forwarding. Note that port numbers are only defined for TCP and UDP, and not for the other protocols. Thus, we replicate three times any filter with a $*$ in the protocol field, using 3 values of the protocol, TCP, UDP, and Other. So we now have only two remaining port fields. We build 4 hash tables, one for each possible combination of port fields (both unspecified, destination only, source only, and both specified). The hash tables are indexed by the combination of port fields *and* the protocol field (TCP, UDP, or Other). See Figure 8.

Given a filter of the form $(D, S, TCP, P1, *)$, we first place an entry, if it does not already exist, in the $(DstPort, *)$ hash table with a key of $(TCP, P1)$. This points to a grid-of-tries structure representing the destination and source prefixes of all the filters that have $Prot = TCP$, $DstPort = P1$ and $SrcPort = *$. This is shown in Figure 8. Each filter is placed in exactly one grid-of-tries structure, which keeps the memory linear in the number of filters.

Finally, to search for a header, we search each of the four hash tables in turn. When searching a hash table, we use the actual port numbers and the protocol field to follow a pointer to a grid-of-tries, where we perform the search we described. For each of the four grid-of-tries we search, we keep track of the lowest matching filter. A simple optimization is to combine the hash of the port number fields with the lookup

[5] While grid-of-tries can be extended to handle port number ranges by creating further two dimensional "planes", this causes further loss of efficiency. A better scheme for firewall filters that use port number ranges is cross-producting, described later.

in the first trie node of the grid-of-tries (see Figure 8). This saves 4 hashes.

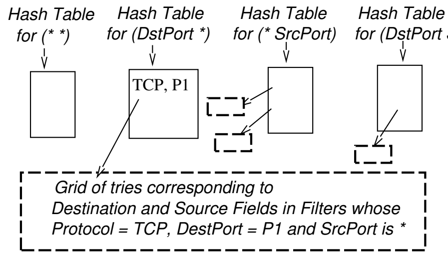


Figure 8: Extending basic grid-of-tries to deal with port number fields.

6 Cross-Producting

The grid-of-tries scheme has excellent performance for two dimensional prefixes matches such as Destination-Source filters. It requires only linear memory and takes time equivalent to doing IP lookups on both the source and destination address. While this is extremely useful in many important cases such as Virtual Private Networks, we need to consider more general filters for applications like firewalls. The grid-of-tries can be extended to handle other fields by replicating the grid-of-tries structure. In the last section, we showed that if the port number fields were either wildcarded or fully specified (no ranges), then we could do so using four grid-of-trie structures. While this is itself expensive, it gets worse if we have to handle filters with port number ranges.

We now describe our second algorithm, *cross-producting*, for the filter matching problem. Unlike the grid-of-tries, cross-producting can easily handle arbitrary filters (including filters with range specifications) at high speeds. However, either its memory needs or search times are less predictable than grid-of-tries. Thus our final scheme will combine the best features of grid-of-tries and cross-producting. We proceed to describe cross-producting.

The main idea behind cross-producting is the following: we start by slicing the filter database into columns, with the ith column storing all distinct prefixes in field i. Then, given a packet P, we determine the best matching prefix for each of its fields separately, and combine the results of the best matching prefix lookups on individual fields. The main problem, of course, lies in finding an efficient method for combining the lookup of individual fields into a single compound lookup.

To this end, we start by slicing the database of Figure 2 into individual prefix fields. In the sliced columns, from now on we will refer to the wildcard character $*$ by the string *default*. Recall that the mail gateway M and internal NTP agent TI are full IP addresses that lie within the prefix range of Net. The sliced database is shown in Figure 9.

At the top of each column, we have indicated the number of elements in the column. Consider a 5-tuple, formed by taking one value from each column. We call this a *cross-product*. Altogether, we have $4 * 4 * 5 * 2 * 3 = 480$ possible cross-products. Some sample cross-products are shown in Figure 10. If we consider the destination field to be most significant and the flags field to be least significant, and if we pretend that values increase down a column, we can order the cross-products from the smallest to the largest, as in any number system.

Our key insight is as follows: *given a packet P, if we do a best matching prefix operation for each field $P[i]$ in the corresponding sliced prefix database, and concatenate the results to form a cross-product C, then the least cost filter matching P is the same as the least cost filter matching C.* This can be formalized by the following simple theorem:

4 * 4 * 5 * 2 * 3 = 480

Destination Prefixes	Source Prefixes	DstPort Prefixes	SrcPort Prefixes	Flags Prefixes
M	S	25	123	UDP
T1	TO	53	Default	TCP–ACK
Net	Net	23		Default
Default	Default	123		
		Default		

Figure 9: The database of Figure 2 "sliced" into columns where each column contains the set of prefixes corresponding to a particular field.

Num	CrossProduct	Matching Filter
1	M, S, 25, 123, UDP	Filter 1
2	M, S, 25, 123, TCP–ACK	Filter 1
3	M, S, 25, 123, default	Filter 1
4	M, S, 25, default, UDP	Filter 1
5	M, S, 25, default, TCP–ACK	Filter 1
6	M, S, 25, default, default	Filter 1
•	• • • •	•
•	• • • •	•
479	default,default,default,default,TCP–ACK	Filter 8
480	default,default,default,default,default	Filter 8

Figure 10: A sample of the cross-products obtained by cross-producting the individual prefix tables of Figure 9

Theorem 6.1 *For any packet P and its associated cross-product $C = M_1 \ldots M_K$, the best matching filter of C is the best matching filter of P*

Proof: Suppose not. Since each field in C is a prefix of the corresponding field in P, every filter that matches C also matches P. Thus the only case in which P has a different matching filter is if there is some filter F that matches P but not C. This implies that there is some field i such that $F[i]$ is a prefix of $P[i]$ but not of M_i. But since M_i is a prefix of $P[i]$, this can only happen if $F[i]$ is longer than M_i. But that contradicts our assumption that M_i was the longest matching prefix in column i. ∎

Thus, the basic cross-producting algorithm is to build a table of all possible cross-products and *precompute* the least cost filter matching each cross-product. This is shown in Figure 10. Then, given a packet header, we can determine the least cost matching filter for the packet with K best matching prefix operations plus a single hash table lookup of the cross-product table. For small databases, the individual prefix lookups may reside in cache and result in K cache accesses together with a single memory lookup. In hardware, each of the K prefix lookups can be done in parallel.

As an example, consider matching a packet with header $(M, S, \mathrm{UDP}, 53, 57)$ in the database of Figure 2. The cross-product obtained by performing best matching prefixes on individual fields is $(M, S, \mathrm{UDP}, 53, \textit{default})$. One can easily check that the precomputed filter for this cross-product is Filter 2—although Filters 3 and 8 also match the cross-product, Filter 2 has the least cost.

This simple cross-producting algorithm suffers from a memory explosion problem: in the worst case, the cross-product table can have N^K entries where N is the number of filters and K is the number of fields. Thus, even for moderate values, say, $N = 100$ and $K = 5$, the table size can reach 10^{10}, which is prohibitively large. In the following, we describe a simple optimization that can reduce the memory requirement considerably.

6.1 On Demand Cross-Producting

The major idea to reduce memory is to build the cross-products on demand: instead of building the complete cross-product table at the start, we incrementally add entries to the table. The prefix tables for each field are built as before. When a packet P arrives, we perform best matching prefixes on the individual fields to compute a cross-product term C. If the cross-product table has an entry for C, then of course the associated filter is returned. However, if there is no entry for C in the cross-product table, we find the best matching filter for C (possibly using a linear search of the database), and insert that entry into the cross-product table. Of course, any subsequent packets with cross-product C will yield fast lookups. Figure 11 shows pseudo-code for build and search for on-demand cross-producting.

```
Build DataStucture: (* called whenever a filter changes *)
  for i = 1 to K (* K is number of packet fields *)
    Let S_i be the set of distinct prefixes in field i of any Filter
    PrefixTable[i] := BuildTable(S_i) (* prefix table for field i*)

CrossSearch(P) (* called on arrival of packet P *)
  for i = 1 to K
    M_i := PrefixLookup(P[i], PrefixTable[i]);
  C := M_1 M_2 ... M_K; (* cross-product for P *)
  R := (HashLookup(C, CrossProductTable))
  if R = nil then (*not in table*)
    Find the first filter R' matching C
    HashInsert(C, R', CrossProductTable) (* insert filter for C *)
    Return (R')
  else Return (R);
```

Figure 11: Pseudo Code for On Demand Cross Producting

On-demand cross-producting can greatly improve both the building time of the data structure as well its storage cost. In fact, we can treat the cross-product table as a cache and remove all cross-products that have not been recently used. We have discovered a number of optimizations to allow incremental computation of the cross-product database when filters are added, but we defer these results to another paper.

We said earlier that caching was not every effective, so why should caching based on cross-producting be more effective? Consider the database of Figure 2, and imagine a series of web accesses from an internal site to the external network. Suppose the external destinations accessed are $D_1, \ldots, D_M$. All these addresses correspond to *two* cross-product terms $(*, Net, *, *, \text{TCP-ACK})$ and $(*, Net, *, *, *)$. While full-header caching will result in $2M$ distinct entries in the cache, cross-producting cache will need only two entries. Examples like these lead us to believe that the hit rates for the cross-product cache should be much better than standard header caches. Clearly the benefits of on-demand cross-producting need to be validated with actual packet traces. We plan to do so in future work.

7 Lower Bounds

We have seen that the grid-of-tries scheme works well for two-dimensional prefix matches (such as destination-source pairs), but it requires multiple planes (grid-of-tries) to solve the problem for general filters. On the other hand, pure cross-producting is very fast but can require a prohibitive amount of memory. On-demand cross-producting appears to offer a good caching solution but does not guarantee worst case performance. Set pruning trees are also very fast but require a prohibitive amount of memory in the general case. These observations raise a natural question: are there schemes that can handle hundreds of thousands of arbitrary filters with bounded memory and fast worst-case search times?

It seems unlikely that a fast and scalable scheme exists for *completely arbitrary* multi-dimensional filters. It is known that general multidimensional range searching over N ranges in d dimensions requires $\Omega((\log N)^{K-1})$ worst-case time if the memory is limited to about linear size [4, 5]. Notice that this lower bound allows the two dimensional case to be as fast as $O(\log N)$. Once again, the two dimensional case seems to be special, and allows a fast and scalable solution.

The lower bounds of [4, 5] hold in an arithmetic model of computation, and do not apply to schemes based on hashing and tries. The repeated hashing scheme of [31], for instance, offers an $O(\log W)$ solution for the one-dimensional prefix matching problem, where W is the maximum prefix length. Thus it seems plausible to look for general solutions based on the techniques of [31]. We did pursue such an approach based on repeated hashing for generalized filter matching. We defer the detailed description of those results to another paper [26], but summarize our main results in the following paragraph.

First, we were able to devise a hashing scheme that takes $2W - 1$ hashes in the worst-case for the two-dimensional prefix matching problem (e.g., source-destination prefixes). We called this scheme *rectangle search*. More importantly, we were able to show a matching lower bound of $2W - 1$ hashes. The lower bound was then extended to show that for any dimension $k > 1$, schemes based on the techniques in [31] would require W^{k-1} hashes. This fits in nicely with the lower bound for multidimensional range matching. The bottom line is that the one and two dimensional cases appear to be special, and extensions to higher dimensions appear to be slow.

Our lower bound does not apply to schemes based on tries, and thus to grid-of-tries. However, it seems plausible that schemes based on tries can be emulated by schemes based on hashing. Suppose the trie scheme is at a node N (that was reached using some string P from the root of trie search) and follows a pointer at location I. A hash based scheme can determine the same pointer by looking up a hash table indexed by the complete path PI. While this is only a very rough plausibility argument and applies only to certain types of trie search schemes, it does make us suspect that it is infeasible to find a more efficient generalization of grid-of-tries to higher dimensions.

Do these theoretical arguments imply that Layer 4 switching cannot be implemented in real routers at high speeds without requiring infeasible amounts of memory? We do not think so. This is because we believe that in practice filter databases will only have a small number of completely general filters (e.g., firewall filters); the vast majority of the filters will be restricted to destination prefixes, destination-source prefixes, and filters with all 5 fields completely specified. If this assumption is true, we can leverage off the assumed distribution of filters to construct an efficient combined scheme that we describe next.

8 A Combined Scheme

We envisage filter databases of the future to consist of a large number (say $80K$) of pure destination prefix filters (standard IP forwarding), a fairly large number (say $20K$) of fully specified filters (destination, source and both port fields fully specified for say bandwidth reservations), a fairly large number (say $20K$) of destination-source prefix filters (e.g., for multicast forwarding and virtual private networks), and a smaller number (say $1K$) of completely arbitrary filters with port ranges (e.g., for firewalls). Thus rather than have a flat worst case figure for all types of filters, it makes sense to have a scheme that can optimize the important special cases (e.g., pure destination prefix filters). We have seen in the previous sections that the grid-of-tries works optimally for destination and destination-source prefix filters. On the other hand, on-demand cross-producting can handle arbitrary filters but with less predictable speed (because of possible cache misses). Thus it makes sense to combine the two schemes.

The simplest combination is to divide the filters into two sets. The first set of filters with pure destination and destination-source prefixes is handled by a single grid-of-tries. The second set containing the remaining filters is handled by cross-producting. This simple scheme has two disadvantages. First, the common case of destination only and destination-source prefixes requires a cross-producting search on the remaining filters to ensure that there is no lower cost filter in the second set. Second, cross-producting search requires a destination and source prefix lookup which is also done in the grid-of-tries search; this is wasteful. Instead, our combined scheme will attempt to terminate the grid-of tries search in the common cases; it will also avoid redundant destination and source lookups if we have to fall back on cross-producting.

A key idea required for early termination is the concept of *filter overlap*. We say that two filters F and F' overlap if there is some packet header that matches both F and F'. Suppose, during our search, we find a filter F that matches packet P. If we can ensure that no other filter in the database overlaps with F, then we can terminate the search and output F as the least cost filter. Our search will match against progressively more complex filters. Initially, we will try to see if the packet matches a destination-only filter $(D, *, *, *, *)$, which does not overlap any other filter. Failing this, we will look for a destination-source filter $(D, S, *, *, *)$. If that also fails, we will do a cross-product search, but will only need to do the best matching prefix on the remaining $K-2$ fields.

We need to modify slightly both the grid-of-tries as well as the cross-producting algorithm for our combined scheme. We divide the set of filters into two sets. We allocate filters that have $(*, *)$ in the port fields to the first set, which we call the *port-free* filter set G_0. All other filters are allocated to what we call the *port-full* filter set G_1.

For the combined scheme, we need to *project* port-full filters into G_0. That is, for each port-full filter $F \in G_1$, we create a projection filter F' obtained by wild-carding the port entries of F. In order to distinguish the original port-free filters from the projected filters, we add a bit *port* to each filter, which is set to 0 for the port-free filters, and to 1 for the projection of port-full filters. The reason for adding the projections of port-full filters is that now filters in the enhanced group G_0 contain all destination and source prefixes in the database. This allows cross-producting to avoid a redundant computation of destination and source prefix matches.

We now build a single grid-of-tries structure for this enhanced group G_0. For each port-free filter $F \in G_0$ (that is, $port(F) = 0$), we associate an additional bit, called the *overlap bit*, which is set if F overlaps with some other filter in the filter database; otherwise the bit is false. For each port-free filter F, we compute $F' = storedFilter(F)$, where F' is the least cost port-free filter whose destination and source fields are prefixes of the corresponding fields of F.

Given a packet P, we start with the grid-of-tries search. As usual, we begin by finding the best matching prefix D_{bmp} for the destination field $P[1]$. If the source trie associated with D_{bmp} has a filter $F = (D_{bmp}, *, *, *, *)$, with $port(F) = 0$ and $overlap(F) = 0$, then we output F as the least cost filter for P and stop.

Otherwise, we perform the normal grid-of-tries search, starting at D_{bmp}. We initialize an overlap bit $overlap = 0$. Whenever we arrive at a new node that has a port-free filter F stored with it, we update the least cost filter, and set $overlap = \max\{overlap(F), overlap\}$. When the search for the group G_0 ends, if $overlap = 0$ and if the temporary variable containing the least cost filter is non-nil, we output that filter and terminate the search. If either $overlap = 1$ or the least cost filter variable is nil, we initiate the cross-producting search.

We need to modify the normal cross-producting search as follows. Instead of using the best matching prefix for the source address $P[2]$, we use the best matching prefix of $P[2]$ *among the filters whose destination field is a prefix of the packet's destination* $P[1]$. It is not hard to show that this modification preserves correctness.

We already know the best matching prefix D_{bmp} for the destination field. Let S_{bmp} be the source prefix at the node where the grid-of-tries search terminated. We claim that S_{bmp} is the best matching prefix of the source field among all filters whose destination field is compatible with $P[1]$. Therefore, we do not repeat the best matching prefix computation for destination and source addresses. We perform the prefix computation for the remaining fields, protocol type and port numbers, and concatenate the best matching prefix into a cross-product term C. Next, we hash into the cross-product table to see if C exists. If it does, we output the filter stored there. Otherwise, we do some other search algorithm (e.g., linear search) among the port-full filters. When the search finishes, we add the corresponding entry to the cross-product table.

Recall that we said that fully specified filters (where all four fields are full specified) may be commonly used for reserving voice and video bandwidth. The combination scheme described so far would allocate such filters to the port-full set, and thus would require a cross-producting search for such filters. This can greatly increase the number of possible cross-products and so reduce the effectiveness of the cross-product cache. If we assume that the destination and source fields of such filters are not prefixes, and the port numbers have no wild cards or ranges, a simple trick is to place such filters in a third set that can be handled by a single hash on all four fields. We can do this search before we fall back on the cross-producting search. If we get a match, we can terminate. This is because with every match in this fully specified set we can precompute the associated best matching filter.

The net result is that the combined scheme will process packets that map to destination filters that have no overlap with other filters in time equal to one IP lookup (3-4 memory accesses using multibit tries [27, 9]), process packets that map to destination-source filters (that have no further overlap) in time equal to two IP lookups using the grid-of-tries (6-8 memory accesses), process packets that map to fully specified filters in one more hash (a total of 7-9 memory accesses), and finish all other filters using two more port number field lookups followed by a hash into the cross-product table (a total of 10-12 memory accesses) if the cross-product is cached. Since the cross-product table only corresponds to filters that are not either in the port-full or fully specified sets (corresponding to what we hope is a small number of firewall filters), this should allow good caching performance for these remaining filters.

We note that that several other combination schemes are possible. For instance, a hardware scheme might implement each of the four planes of the extended grid-of-trie search in parallel. Since the extended grid-of-tries does not handle port number ranges, filters with port number ranges could be handled by (say) a small additional content addressable memory (CAM).

9 Implementation and Measurements

For our implementation platform, we chose a 300 Mhz Pentium II (system cost under 5000 dollars) running Windows NT that had a 512 KBytes L2 cache and a cache line size of 32 bytes. We believe the results would be similar if run on other comparable platforms such as the Alpha. We use a tool called Vtune [13] that gives us access to dynamic instruction counts, cache performance, and clock cycles for short program segments. We did evaluations for the grid-of-tries scheme as well as for cross-producting. We did not finish an implementation of the combined scheme; thus we can only provide estimates of the performance of the combined scheme.

9.1 Grid-of-Tries Implementation Measurements

First, we report on the worst case time for a simple grid-of-tries implementation that can process destination-source filters. Our implementation used multibit tries [27] sampling 8 bits at a time for the Destination trie; each of the source tries started with a 12 bit trie node, followed by 5 bit trie nodes. This yields a worst case of 9 memory accesses (we could easily have done the source tries 8 bits at a time to yield a worst case of 8 memory accesses but that increases storage.)

Destination-Source Filters: Using VTune on the 300 Mhz Pentium II, we measured the worst case path as taking 870 nsec using a memory access time of 60 nsec and a clock tick interval of 3.333 nsec. The numbers for a single IP lookup reported in, for example, [9] are around 400 nsec, and so this roughly corresponds to two IP lookups. For destination-source filters this appears to be optimal as it is hard to find the lowest cost matching filter any faster than doing an individual best matching prefix on both source and destination addresses. The memory required for 20,000 filters was around 2 Mbytes and the time taken to construct the entire data structure was 8 seconds.

General Filters: We built a 4 plane grid-of-tries that can handle more general filters (Section 5.5) with fully specified port numbers. Since there are no layer 4 databases available, we started with a publically available database of pure destination prefixes (D, *, * , *, *) entries, and added further entries which specialize some of these entries. For our experiment, we took the publicly available MaeEast database [20] (around 40000 prefixes). We randomly chose 5000 destination prefixes to create further filters. For each (D, *, *, *, *) prefix chosen, several filters were added which were of the form (D, *, TCP, P1, *), (D, S, *, *, *), (D, S, TCP, P1, *) and (D, S, TCP, P1 P2). The source prefixes were chosen randomly from the set of 40000 prefixes. From each destination prefix in MaeEast, 20 filters were generated. The number of filters of each form was varied, but together 20 filters were generated for each chosen destination prefix. Port numbers were generated randomly.

The following table was obtained using the following distribution of filters. For each of the 5000 pure destination prefix (D, *, *, *, *) filters that we specialized, we made up five (D, S, *, *, *) filters, four (D, S, TCP, P1, *) filters, five (D, S, TCP, P1 P2) filters, and five (D, *, TCP, P1, *) filters. Together with the original destination prefix, the total adds up to twenty filters.

Filters	Memory (KB)	Build Time sec	Worst case search per plane in μsec
1000	240	0.4	0.9
2000	836	0.7	0.9
5000	2033	1.5	0.9
10000	3951	3.5	0.9
20000	7489	32	0.9

Table 1: 4 planes grid-of-tries implementation measurements on a 300 Mhz Pentium

The worst case time for the 4 plane grid-of-tries search was measured using VTune to be 0.9 usec per plane, or a total of 3.6 usec. The number per plane (0.9) is slightly more than the measured number for a single grid-of-tries search because of the need for the additional hash of the port number fields (see Section 5.5).

9.2 Cross-producting Implementation Measurements

Since we expect cross-producting to be used with small filter databases (with arbitrary port number ranges), we used a firewall database to test cross-producting. The firewall filters we used are generated based on a 20 filter database in [6], which is described as a sample firewall database for a university. To create larger databases, we added similar filters to the base 20 filter database, while maintaining the ratio of the number of distinct longest matching prefixes in a field with the total number of prefixes in the field.

Note that the longest matching prefix in the source and destination fields can be found by any technique. We used a multibit trie [27] approach. The port lookups are implemented as full arrays. Note that only the final cross-product table is in main memory; the structures for the individual fields for such a small database can be in the L2 cache. The final cross-product table can be implemented either as an array or as a hash table. We used simple cross-producting for small filter sets; in that case the final table can be implemented as an array which can be looked up by an index that is the concatenation of indexes returned by the individual column lookups. For databases with more than 50 filters, on-demand cross-producting is essential. For on-demand cross-producting to reduce memory, the final cross-product table must be a hash table.

Number of Filters	Memory for cross-product (KBytes)	Average Search (nsec)	Worst case time (nsec)
20	40	360	475
50	1525	405	475
256	on-demand	480	540
1024	on-demand	480	540

Table 2: Cross-producting implementation measurements on a 300 Mhz Pentium

For worst case time measurement, we use Vtune based clock cycle counts. If L2 is the access delay from the L2 cache (=15 nsec), then the worst case filter lookup time when using an array for the cross-product table was 475 nsec. When we use hashing and on-demand cross-producting the worst case is harder to evaluate. Since we used a hash function that gave almost no collisions and we expect the on-demand cache hit rate to be high, the "worst case" figure shown when using hashing and on-demand assumes no hash collisions and a cache hit in the cross-product table.

For many firewall databases, the destination and source addresses are often full addresses instead of prefixes. In this case the destination and source column lookups would take a single hash each (instead of several memory accesses needed to do a longest matching prefix). Assuming a 10 clock cycle hash function, filter resolution in this special case (no prefixes) can be done in a worst case of 200 nsec which is twice as fast as the general case (400-500 nsec)

10 Conclusions

We have described two algorithms for packet filtering at Gigabit speeds. The grid-of-tries solution provides a scalable (linear storage) and fast (worst case time equal to two IP lookups, 870 nsec on a Pentium II) for destination-source filters. Such filters can be used to implement Virtual Private networks and multicast forwarding efficiently. The grid-of-tries solution can be extended to handle more general filters but at a high lookup cost (3.6 μsec for filters without even allowing port number ranges). On the other hand, the cross-producting solution provides fast lookup times (around 500 nsec) for small (up to 1000 filters) but has less predictable lookup times because of the need for caching cross-products to make the storage needs manageable.

We then described a simple combined scheme that uses grid-of-tries to handle all destination-source and pure destination or source filters. We anticipate that this will handle a large majority of the filters, and that packets matching such filters will terminate after a grid-of-tries search. Packets whose best matching filter is a pure destination prefix filter can be further optimized to terminate after a search in the destination trie (one IP address lookup). Fully specified filters, used for say IP telephony, can be handled with a single extra hash on all five fields. Finally, the remaining filters (e.g., firewall filters) can be handled by on-demand cross-producting. Information from the initial grid-of-tries search can be used to prevent the cross-producting step from re-computing longest matching prefixes for the destination and source addresses.

Based on the measurements for the individual components on the Pentium II, we estimate 450 nsec lookup times for packets that map to pure destination prefixes, 900 nsec for packets that map to destination-source filters, 1000 nsec for packets that map on to fully specified filters, and 1500 nsec (assuming a hit in the cross-product cache) for packets that map on to more general filters. Hardware engines can do better because of increased opportunities for parallelism and pipelining, Given that the average packet size is around 2000 bits [2], a worst case lookup time of 1500 nsec allows 0.75 million packets per second, which allows a Layer 4 router to keep up with a Gigabit link.

As best matching prefix is a special case of lowest cost matching filter, it is not surprising that filter search schemes are generalizations of prefix search schemes. Thus, the grid-of-tries and set pruning trees [29, 7] generalize trie schemes for prefix matching [9, 27, 22]. Multidimensional range matching schemes [15] generalize prefix matching schemes based on range matching [16]. Rectangle search and Tuple Search [26] generalize binary search on hash tables [31]. While cross-producting is not a generalization of an existing prefix matching scheme, it can be specialized for prefix lookups as well.

For future work, we would like to create other filter lookup algorithms that are specialized for certain important filter types (e.g., the way grid-of-tries is specialized for two dimensional filters). It would be useful to have benchmark filter databases to compare lookup schemes. We also hope to be able to do trace-driven evaluation of the effectiveness of on-demand cross-producting. Finally, we have made no effort to have fast filter insertion. Because of issues like BGP implementation instabilities [14] (which can add destination prefixes in the order of milliseconds), and RSVP [32] reservations (which can add other filters in the order of seconds), it is important to have faster filter insertion algorithms.

Despite the work that remains to be done, we believe that Layer 4 Switching is feasible for high performance routers. We do not know whether routers of the future will forward based on Layer 4 headers or based only on fields in the routing header. In either case, we believe that applications like QoS Routing, Firewalls, Virtual Private Networks, and Large Scale Multicast will require a more flexible form of forwarding based on multiple fields, whether they be in the routing header or elsewhere. We believe the techniques in this paper indicate that such forwarding flexibility can go together with high performance.

11 Acknowledgements

The observation that the Protocol field can be eliminated in many cases of interest is due to Hari Adiseshu. We would like to thank Hari for initial discussions that led to the grid-of-tries scheme. We thank Zubin Dittta, Will Eatherton, and Jon Turner for valuable discussions.

References

[1] M. Bailey et al. PathFinder. *Proceedings of OSDI 94.*

[2] S. Bradner. Next generation routers overview. *Proc. of Networld Interop 97.*

[3] B. Chapman and E. Zwicky. Building Internet Firewalls. *O'Reilly and Associates*, 1995.

[4] B. Chazelle. Lower bounds for orthogonal range searching, I: The reporting case. *J. of the ACM*, 37, pp. 200–212, 1990.

[5] B. Chazelle. Lower bounds for orthogonal range searching, II: The Arithmetic model. *J. of the ACM*, 37, pp. 439–463, 1990.

[6] W. Cheswick and S. Bellovin. Firewalls and Internet Security. *Addison-Wesley*, 1995.

[7] D. Decasper, Z. Dittia, G. Parulkar, B. Plattner. Router Plugins: A Software Architecture for Next Generation Routers" *Proc. ACM Sigcomm 98*, Sep 1998.

[8] S. Deering and R. Hinden. Internet protocol, Version 6 (IPv6) specification RFC 1883. `http://ds.internic.net/rfc/rfc1883.txt.`

[9] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink. Small forwarding tables for fast routing lookups. *Proc. ACM Sigcomm 97*, October 1997.

[10] Digital Electronics Corporation. The DEC Alpha. `http://www.dec.com.`

[11] D. Engler and M. Kaashoek. DPF: Fast Flexibible Message Demultiplexing using Dynamic Code Generation. *Proceedings of ACM Sigcomm 96.*, August 1996

[12] Intel. The pentium processor. `(www.pentium.com.)`

[13] Intel. The Vtune performance measurement tool. `(www.intel.com/design/perftool/vtune)`

[14] C. Labovitz, G. Malan, and F. Jahanian. Internet Routing Instability *Proc. ACM Sigcomm 97*, October 1997.

[15] T.V. Lakshman and D. Stiliadis. High Speed Policy-based Packet Forwarding Using Efficient Multi-dimensional Range Matching. *Proc. ACM Sigcomm 98*, Sept 1998.

[16] B. Lampson, V. Srinivasan, and G. Varghese. IP Lookups using Multiway and Multicolumn Search. *Proc. Infocom 98*, March 1998.

[17] S. McCanne and V. Jacobson. The Berkeley Path Finder. *Proc. of Winter USENIX 1993.*

[18] N. McKeown, M. Izzard, A. Mekkittikul, B. Ellersick and M. Horowitz. The Tiny Tera: A Packet Switch Core *IEEE Micro* Jan/Feb 1997, pp 26-33

[19] J. McQuillan. Layer 4 Switching. *Data Communications,* October 21, 1997

[20] Merit Inc. IPMA statistics. `(nic.merit.edu.)`

[21] P. Newman, G. Minshall, and L. Huston. IP Switching and Gigabit Routers. *IEEE Communications Magazine*, January 1997.

[22] S. Nilsson and G. Karlsson. Fast Address Look-Up for Internet Routers. *Proceedings of IEEE Broadband Communications 98*, April 1998.

[23] C. Partridge. Locality and route caches. In *NSF Workshop on Internet Statistics Measurement and Analysis*, San Diego, CA, USA, February 1996.

[24] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, New York, NY, 1985.

[25] H. Samet. The Design and Analysis of Spatial Data Structures. *Addison-Wesley*, 1989.

[26] S. Suri, G. Varghese, M. Waldvogel, and V. Srinivasan. Layer Four Switching using Rectangle and Tuple Search. *In preparation*, 1998.

[27] V. Srinivasan and George Varghese. Faster IP Lookups using Controlled Prefix Expansion. *Proc. ACM Sigmetrics 98*, June 1998

[28] Torrent systems, Inc. http://www.torrent.com

[29] P. Tsuchiya. A search algorithm for table entries with non-contiguous wildcarding. *Unpublished report, Bellcore.*

[30] J Turner. Design of a Gigabit ATM Switch. *Proc. SIGCOMM 97*, October 1997.

[31] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable high speed IP routing lookups. *Proc SIGCOMM 97*, October 1997.

[32] L. Zhang et al. RSVP: A New Resource Reservation Protocol. *IEEE Networks Magazine*, Sept 1993.