Multi-Dimensional Prefix Matching Using Line Search

Marcel Waldvogel

Computer Science Department
Washington University, St. Louis, MO, USA
mwa@arl.wustl.edu

Abstract

With the increasing popularity of firewalls, virtual private networks (VPNs) and Quality of Service (QoS) routing, packet classification becomes increasingly important in the Internet. The high-performance solutions known so far strongly rely on certain properties of the filter database to match against, such as a small number of distinct prefixes or the absence of conflicts. In this paper, we present Line Search *as a two-dimensional generalization of the one-dimensional binary search on prefix lengths [21], exploiting the advantage given by the different approach therein. This algorithm also works best on the filter databases that are expected to occur most often, but degrades gracefully when these assumptions no longer hold. We also show how to efficiently extend the algorithm to a complete five-dimensional Internet Protocol (IP) and transport header match.*

1. Introduction

With the gaining popularity of the Internet, additional services are requested or installed by users to fulfill their particular requirements. Such demands range from understanding the network (through network monitoring) over better QoS (latency, throughput, packet loss) to privacy and security (encrypted VPNs, firewalls and Network Address Translators (NATs [7])). What these services have in common is their requirement for some network flows and thus their constituent data packets to be treated differently from others, based on any combination of static and dynamic rules: Firewalls drop packets or let them pass, VPN gateways additionally en- or decrypt them, and QoS-aware routers select the packet's next hop along the path, it's output queue and queuing parameters. These rules are known as *filters* and the process of identifying the rule best matching a given packet is known as *classification*.

Despite several efforts to limit the complexity of performing QoS functionality in core routers, such as introduced by Differentiated Services [2], still many routers, especially border routers, are expected to perform packet classification. To provide advanced services, including QoS routing, which not only decide upon the priority of a packet or flow, but also on the optimal route, every single router needs to perform some form of packet classification. Today, setting up flow labels in the context of Multiprotocol Label Switching (MPLS [18]) is expected to simplify packet processing in the core by pushing the full-blown packet classification to the border, but requires the additional burden of setting up and distributing flow labels, which are also a comparably scarce resource. This gives hope that packet processing at many routers will be simplified. Nevertheless, packets will need to be classified in a large and increasing number of network devices to satisfy the needs for security and quality. Recent work by Shaikh et al. [19] indicates that dynamic routing of individual long-lived flows in the Internet has significant advantages.

In all the firewalls and routers where packets need to be classified, *all* packets will need to be classified. At line speeds of currently 10 Gb/s, this is not an easy task. At an average packet length of about 2000 bits [13], this results in an average packet arrival rate of 1 every 20 ns. Even with the fastest available static RAM, this only allows for a handful of memory accesses before the next packet needs to be processed. To some extent, this narrow time limit can often be extended by using multiple parallel packet processing engines or pipelining the process. As the routers of the future continue increasing link speed and the number of ports per router, hardware design space gets tighter and tighter.

Besides throwing additional hardware at the problem, the algorithms need to be improved. To improve the algorithm, we need to know more about packet classification. Background knowledge is given in Section 2. Two-dimensional Line Search is presented in Section 3 and analyzed in Section 4. The mechanism will be extended to more dimensions in Section 5. Related work is discussed Section 6, before the paper is concluded in Section 7.

2. Internet Packet Classification

Before we start talking about multi-dimensional packet classification, we introduce the concept by looking at just a single dimension. Whenever an Internet Protocol (IP) packet arrives at a router, this router matches the packet's destination address against its routing database and decides which directly connected node (another router or the ultimate destination) the packet should be send to, to get closer to the destination with the ultimate goal of reaching it.

While this sounds rather simple, a complex process is hidden behind the phrase "match a destination address against the routing database." To avoid having to continuously store and update each of the many million addresses in use at any given time, the entries are heavily aggregated: Nodes connected to the Internet which are neighbors in the network topology (e.g., connected to the same Ethernet, within the same organization, or dialing in to the same ISP) are allocated addresses sharing a common *prefix*. These are typically represented as bit strings, where the leftmost bits are all fully specified, while the remaining bits are all don't-cares. The number of significant bits is known as the *prefix length*. For every destination address that is to be looked up against this database of prefixes, the router has to determine a matching entry. In case multiple entries should match, it has to return the most specific (i.e., longest) match. This entire process is known as a *prefix match* and is used to match packet source and destination addresses.

Two-dimensional prefix matching is very similar to it's one-dimensional cousin, but instead of having a database of prefixes such as $\{1111\_00*, 0110*, \ldots\}$[1], we have a database of prefix pairs, e.g., $\{(1*, 0000\_00*), (1111\_11*, 0*), \ldots\}$. These pairs are ordered tuples, with each of the tuple's fields representing a range of coordinates in the corresponding dimension. This database is consulted for fully-specified tuples, such as—assuming 12 bit address length—$(1100\_0000\_1111, 0000\_1111\_1111)$. Extending this to $d > 2$ dimensions is straightforward, but instead of 2-tuples, d-tuples are being used.

Obviously, each prefix (or prefix tuple) can also be represented by the set of addresses (or address tuples) it matches. In one-dimensional matching, when multiple matching entries exist in the database, the sets representing these entries can always be completely ordered by a subset relation. In other words, from each pair of matching entries, one of the representing sets was a subset of the other. Therefore, the most specific entry could be determined easily and unambiguously.

For d-dimensional matching (with $d \geq 2$), ambiguities may—and in general, will—exist. Assume again our two-dimensional prefix database $\{(1*, 0000\_00*), (1111\_11*, 0*)\}$. If we would search this database for entries matching $(1111\_1111\_1111, 0000\_0000\_00000)$, both entries would match. Neither of them can be considered more specific: The second entry is more specific in the first dimension, but the first entry is more specific in the second dimension. Also, the size of the sets represented by either tuple is the same. Therefore, it is impossible to find a natural ordering between the two; the ambiguity cannot be resolved.

If it is known in advance that only few entries will contain ambiguities, it may be possible to split the entry into several sub-entries to resolve ambiguities, as described in [1].

To resolve ambiguity, several solutions have been proposed:

Unspecified There is no simple way to know in advance which of the matching entries will be returned. This is the simplest solution, but seldom satisfactory, unless ambiguities can be prevented to appear in the database in the first place ([1]). Unfortunately, a general solution requires $O(N^d)$ memory, with N the number of filters and d the number of dimensions.

Priorities of Dimensions The dimensions are prioritized against each other. Without loss of generality, it can be assumed that the dimensions are sorted in order of decreasing priority. When resolving ambiguities, the prefix lengths of the individual numbers are concatenated as digits in a $W + 1$-ary number and the entry with the highest number wins. W is the number of bits in a given dimension.

Although this clear hierarchy of dimensions seems sensible at a superficial inspection, the problem is effectively reduced to a single dimension: A second dimension is only evaluated if inspecting the first dimension results in a tie. This greatly reduces the usefulness and generality of the filters, requiring filters to be made unambiguous before inserting them into the database, which leads to the same memory explosion problem.

Filter Priorities Each entry in the filter database is assigned an explicit priority, which can be considered constant during the presence of that entry in the database. This priority is then used to resolve ambiguities. We assume that entries having a subset relation will have priorities set so they do not conflict with the subset relation. If they ever do, i.e., if a more specific entry should have a lower priority than a less specific entry, the lower priority entry will never be consulted. Instead of solving this problem at search time, it can be

[1]Underscores are used as a group separator and the asterisk indicates that the remaining bits are don't-cares.

avoided at database build time by ignoring the hidden entry.

This scheme is the the most flexible of these three, and includes the others as subsets. Providing a solution for this problem therefore implies having a solution for the others.

In the following, we will assume that explicit priorities have been assigned and provide solutions for this case.

3. Line Search

[21] introduced binary search on prefix lengths to match a single address against a database of one-dimensional prefixes using just $O(\log W)$ hash table probes (typically equivalent to the number of memory accesses) and $O(N \log W)$ storage. The paper also discusses several improvements which reduce the search time even further in current routing databases. Unlike most other approaches, which were derived from binary tries, this solution stores prefixes in hash tables organized by prefix length, allowing for an $O(1)$ membership test.

If we view these hash tables as sets, two ordered comparisons with the operators $x \subset y$ and $x \succ y$ can be defined, ordering them both in gestalt and priority. Incidentally, these two relations return the same ordering for both criteria. It seems that this definition is both necessary and sufficient to enable binary search over hierarchical prefixes.

The algorithm to perform such a binary search over these hash tables organized by prefix lengths is simple: On a hit, the hash tables containing shorter prefixes can be ruled out, whereas on a miss, the longer tables were ruled out. For correctness, a routing table entry would require up to $(\log W) - 1$ helper entries to direct the search, called *markers*.

How can this definition be applied to two or even more dimensions? Figure 1 shows the increasing lengths and thus specificity for one-dimensional matching on the left-hand side. Each square represents a hash table with all the prefixes of length i. Moving up results in a more specific prefix. A natural placement of the length pairs can be seen on the right-hand side. Again, the tuples represent the number of significant bits in each of the two dimensions, and label the hash table represented by the enclosing square. Moving right or up in this matrix results in a more specific entry (one of the prefixes becomes more specific). Moving left or down results in a less specific entry. Moving two steps, one left and one up (or one right and one down), results in a more specific entry along one dimension, and in a less specific entry along the other, resulting in ambiguity.

As seen in Section 2, ambiguity cannot be avoided in structural ways. We therefore apply the proven *divide and conquer* strategy. Each of the columns (or rows) in

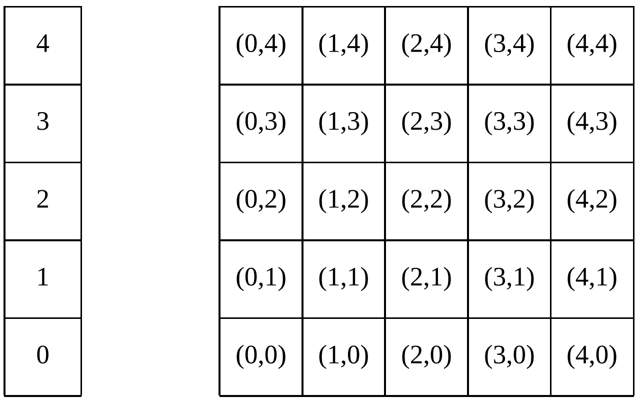

Figure 1. One- vs. Two-Dimensional Search. Each square represents a hash table containing all the prefixes (prefix pairs) with the prefix length (prefix length pair) indicated.

Figure 1's matrix fulfills the non-ambiguity criteria, when taken by itself. An obvious solution would be to search each of the columns (or rows) using the one-dimensional binary search scheme. For two addresses of W bits each, this would require searching a $(W + 1) \times (W + 1)$ matrix, using binary search in one dimension and linear search in the other. Thus, the number of steps would be $O(W \log W)$ or, more concrete, $(W + 1) \cdot \lceil \log_2(W + 1) \rceil$. For $W = 32$, this would amount to 198 search steps, too much for modern routers.

Fortunately, there is hope. Not only is a better solution available, we also expect the classification databases to exhibit a large amount of structure, which can be exploited.

Please note, that unlike the one-dimensional case discussed in [21], the row (and column) corresponding to prefix length zero is necessary in the multi-dimensional search. This is due to the fact that—except for the prefix length pair $(0, 0)$—the other prefix length is non-zero, providing for non-zero information.

3.1. Faster Than Straight

To improve on the row-by-row scheme presented above, recall that the number of memory accesses for binary search grows logarithmically to the number of prefix pairs covered. It is thus better to use fewer binary searches, each covering more ground.

Further recall that the entries get more specific both in vertical (up in Figure 1) and in horizontal direction (left). By combining a path in both directions, it is possible to create a sequence of completely order prefix lengths which is longer than a single row or column. Figure 2 shows a set of such longest paths. Let us call such a path collecting unambiguous prefix length pairs a *Line*.

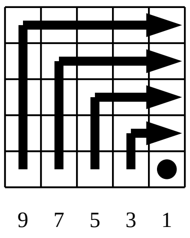

Figure 2. Longest Possible Lines in a 5×5 Matrix

In the naïve solution, a 5×5 matrix was covered by 5 Lines of length 5 each, each requiring 3 search steps, totaling 15 search steps. Now, the matrix is covered with 5 Lines of varying length, summing up to $4 + 3 + 3 + 2 + 1 = 13$ search steps. Larger matrices allow for a higher yield. So the ratio for an 8×8 matrix equals $32 : 24$.

Against the intuition presented earlier, making lines as long as possible is not the optimal solution. Recall that the number v of binary search steps required to cover C prefix length pairs is $v = \lceil \log_2(C + 1) \rceil$. Going from a Line of length 7 to one covering 8 cells also increases the number of steps from 3 to 4, going from a coverage of $7/3$ cells per search step down to $8/4$.

Therefore, it is not only advisable to make the lines as long as possible, but also to cut some off just below powers of two. Figure 3 shows an example with a better line configuration. The longest line has been cut off at length 7 to save a search step, the second line is kept at that size, but changes path to keep nestling up against the first. The third line is extended to length 7 to cover the two cells freed by the other two lines, at no additional cost. Thus, the total number of search steps amounts to $3 + 4 + 3 + 2 + 1 = 12$, a further improvement.

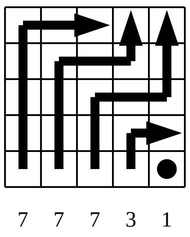

Figure 3. Optimal Lines in a 5×5 Matrix

It can be shown that this solution is optimal. Lines with optimal length can be built by the algorithm in Figure 4. “Spare” cells are cells that could be covered at no additional cost, if the current lines would be extended to the maximum length

```
**Function** OptimalLines($W$) (* Build Lines for Size $W$ *)
(* Calculate memory accesses for each Line *)
Initialize $s$ to 0; (* spare cells *)
**For** $l \leftarrow 1$ to $W$ do
    (* Calculate longest Line along the outer border of a square *)
    (* of length $l$, taking into account and updating spare cells. *)
    (* $2l - 1$ is border length, $c_l$ is coverage, $m_l$ is search steps. *)
    $c_l \leftarrow$ smallest (power of 2) $-1, \leq 2l - 1 - s$;
    $m_l \leftarrow \lceil \log_2(c_l + 1) \rceil$;
    (* Update spare counter by surplus/borrowed cells *)
    $s \leftarrow s + c_l - (2l - 1)$;
    **If** cells were borrowed then
        Extend the most recent Lines which can cover more cells
            ($m_i^2 - 1 > c_i$) until borrows are satisfied;
    **Endif**
**Endfor**
```

Figure 4. Build Optimal Lines For Full Matrices

Now we have reached one goal, making lines longer. But we haven’t yet reduced the number of lines. Unfortunately, the number of elements on the co-diagonal is the limiting factor for any fully populated matrix. Since all of the prefix length pairs on the co-diagonal are ambiguous to each other, they provide a lower bound for the number of Lines. So do the other cuts parallel to the co-diagonal and generally all other sets of mutually ambiguous prefix length pairs.

3.2. Lines for Sparse Matrices

In Figure 4, we have seen how to build optimal Lines for full matrices. For sparse matrices, no algorithm for building optimal Lines is known, short of exhaustive search. We have devised a number of heuristics. Each of these tries to build the largest possible Lines, but some of them cut Lines down.

To find the longest Lines, a directed acyclic graph of subset relations is built. By labeling each vertex with its depth in the graph, the vertex with the highest number is the end of the longest Line. This Line is removed, and the process repeated, until the graph is exhausted.

The algorithms for cutting Lines are as follows:

Simple No cutting is done, the longest Lines are used.

Log All Lines are cut to the maximal length in the form $2^x - 1$, optimizing the coverage per search step.

AlmostLog Only Lines "just above" an optimal length are cut down, i.e., those with lengths of the form $2^x \dots 1.5 \cdots 2^x$.

3.3. Expected Two-Dimensional Classification Databases

Until very recently, no approach for multi-dimensional classification was available, short of slow and tedious linear search through the entire database. This algorithm could not support more than a few filters except for very high processing power to line speed ratios. Therefore, no one has started creating large classification databases. The only databases in use are small firewall databases, which are not openly available due to security concerns. Nevertheless, we expect demand for using such databases to become real within the next few years. Until then, we cannot but generate our own sample databases. To provide for a wide variety of databases, covering a large part of the possible spectrum, we devised four benchmark scenarios, described below.

Full This is the simplest scenario, but the most expensive to solve: All possible prefix length pairs will show up in the database, giving a full $(W+1) \times (W+1)$ matrix.

Chess In the manner of a checkerboard, only every alternating matrix cell of prefix pair lengths contains prefixes.

CIDR This pattern consists of the prefix lengths that are most likely to appear, so all (x, y), where $x, y \in \{0, 8 \dots 30, 32\}$ are assumed to contain prefixes. Lengths $1 \dots 7$ are excluded since they are not part of the CIDR [16, 9] specification. Length 31 is not part of the set since most of the checked one-dimensional routing databases do not contain entries of that length. This is due to the fact that the two addresses included in that range cover more than a single host, but not enough to cover a reasonable network (the first and last address in each network cannot be assigned to machines).

Random This is actually based on real entries, and comes in two flavors, Random1000 and Random5000. To create this database, 1000 (5000) random prefixes were picked from the Mae-East database. Of these, 1000 (5000) random pairs were constructed. 10% of these pairs had one entry prefix replaced by a default prefix (with zero length). This is based on the assumption that classifiers will be biased towards some prefixes, and that tuples only specifying either source or destination filters will also be common. Additionally, the database contains the "default" prefix pair with a (0,0) length tuple.

These five benchmarks (Full, Chess, CIDR, Random1000, and Random5000) will be used for analysis below.

4. Evaluation

4.1. Performance Analysis

In the previous sections, five benchmark databases (full, chess, CIDR, Random1000, and Random5000) have been introduced. Table 1 compares the performance of different Line selection algorithms for these benchmark scenarios. The algorithms are called Original (Lines are parallel, either all rows or all columns), Optimal (Figure 4), and the three heuristics for sparse matrices (Simple, Log, and AlmostLog).

As can be seen, Selection according to the AlmostLog criteria is up to 25% faster than our first, naïve idea, especially on the random distributions, those we deem most representative for future classification databases.

Note that for reasonably sparse matrices, such as the results from the random distributions, two-dimensional classification is only an order of magnitude more expensive than the fastest one-dimensional lookups. We expect that a two-dimensional generalization of the Rope paradigm would give an additional performance boost.

Benchmark	Full	Chess	CIDR	Rnd1000	Rnd5000
DB Size	1089	544	625	1001	5001
Prefix Pairs	1089	544	625	88	141
Original	198	165	125	42/40[a]	56
Simple	168	140	119	30	46
Log	164	132	111	31	48
AlmostLog	159	130	111	30	45
Optimal	159	*Unknown*			

[a] 42 was achieved when splitting along the columns, 40 when splitting along the rows. This is the only test where data organization made a difference.

Table 1. Line Search Performance (Memory Accesses)

4.2. Memory Consumption

By using the algorithm from [21], memory requirements are very much alike: Each entry can lead to up to $\log W$ markers, leading to a total memory requirement of $O(N \log W)$. The hash table optimizations described in [22] also apply.

Header Field	Matching type
Source Address	Prefix
Destination Address	Prefix
Protocol ID	Wildcard
Source Port	Range
Destination Port	Range
Type of Service/Traffic Class	Wildcard
Flow ID	Wildcard
TCP SYN	Wildcard

Table 2. Header Field Matching. The top five fields are often used together, the remaining fields are typically only used for one specific purposes, combined with a true subset of the other fields.

5. More Dimensions

Analogous to adding a second dimension, further dimensions may be added. Unfortunately, the lower bound on the number of Lines grows impracticable. In the two-dimensional case, we have seen that the number of occupied cells in the co-diagonal, any of its parallels, and in fact any group of cells which are mutually ambiguous imposes a lower bound on the number of Lines. Similarly, the co-diagonal plane in the three-dimensional cube and all its relatives provide a lower bound for three dimensions. Thus, with all prefix length triples in use, there are $O(W^2)$ lines of $O(\log W)$ search steps each, totaling $O(W^2 \log W)$, clearly impractical, even if many databases will perform much better than that. Generally, for d dimensions, $O(W^{d-1} \log W)$ effort is required. If the dimensions should differ in size, with W_i the number of bits necessary to represent the address range in dimension i, $O(\log W_d \prod_{i=1}^{d-1} W_i)$.

5.1. Collapsing Additional Packet Classification Dimensions

As can be seen from the previous section, adding dimensions after the second does apparently not lead to efficient solutions. Therefore, we try to change our goal and reduce the number of dimensions needed, instead of adding dimensions we can support.

5.1.1. Collapsing a Wildcard Matching Dimension

As can be seen from Table 2, full prefix matching is only required for the source and destination addresses. For all the other fields, much more limited matching methods are sufficient: Wildcard matching allows a filter to either specify an exact match for a field or a don't-care and range matching extends this with ranges.

Assume the addition of the *wildcard* fields, such as the protocol ID. Instead of adding this as full-fledged dimension in its own right, we add it as an additional layer to dispatch between multiple two-dimensional search structures. To dispatch, all the valid protocol IDs are stored in an array or a hash table. Each of these entries points to a Line search structure, to perform the source/destination address matching. Each of these Line search structures only contain the entries of the database which contain the appropriate protocol ID. Additionally, there is a Line search structure containing all entries where the protocol ID is a wildcard. Given a packet, it is classified as follows. First, the protocol ID is looked up in the initial array or hash table. If found, the referenced two-dimensional structure is searched and the best match remembered. Independent of the execution of the previous step, the additional structure for wildcard protocols is searched. Then, the search result with the higher priority is used to further process the packet.

In the worst case, this approach only doubles the search steps, compared to an eightfold slow-down if the protocol ID were considered a full-fledged eight-bit prefix dimension. By sacrificing some memory, the addition of the third dimension may not even affect the performance. By including all the relevant data from the wildcard structure into the individual fully-specified sub-databases, the additional search of the wildcard structure can be avoided entirely. Although no large databases are available to support this claim, we believe that the search structures associated with the defined protocol IDs will not be extended significantly.

5.1.2. Collapsing a Limited Range Matching Dimension

Some fields, such as the port fields, do not only require exact matching and wildcard fall-back, they also require a small number of ranges. The universally accepted ranges used for ports are $[1, 1023]$ (privileged ports and well-known services [15, 17]) and $[6000, 6063]$ (X Window System (X11) [14, 17]).[2] Extending wildcard matching to support these is straight-forward. After searching the exact match (if it exists), a range is searched (if appropriate), and then the wildcard default is searched. This requires only three times as many searches as a plain two-dimensional classification.

Since the number of supported ranges is a small constant, it makes sense to avoid the third search in the wildcard database, by including the relevant entries into the two range databases, sacrificing a small amount of memory for a significant worst-case speed improvement.

[2] Some sources refer to the X11 reserved range as $[6000, 6100]$. According to both [17, 11], the authoritative sources for Internet number assignments, only the first 64 ports are in fact reserved for X11.

5.2. Collapsing Multiple Dimensions

Obviously, this strategy can also be extended to matching multiple of these fields. The classical problem in Internet packet classification is the five-tuple matching: source/destination addresses, protocol ID, and source/destination ports. One of the factors that simplify this five-tuple matching is that only two protocols, UDP and TCP, do have port numbers defined. All others have no notion of protocols, so port matching is not necessary when searching the wildcard protocol ID. Instead, these entries are just added to both the UDP and TCP databases, again sacrificing a negligible amount of memory for a significant speedup.

Figure 5 shows the decision tree that is to be used, limiting the two-dimensional searches for any path to at most 5. If each of these dimensions were treated as prefix matches, the three fields of length 8, 16, and 16 bits (protocol, source port, destination port, respectively), would have resulted in an increase in the number of search steps by several orders of magnitude. Although Figure 5 shows the lookup steps in sequential order, the branching decisions do not rely on any lookup results. Also, the lookups do not depend on each other and may thus be parallelized or pipelined efficiently, making the algorithm both suitable for implementation in hardware and software.

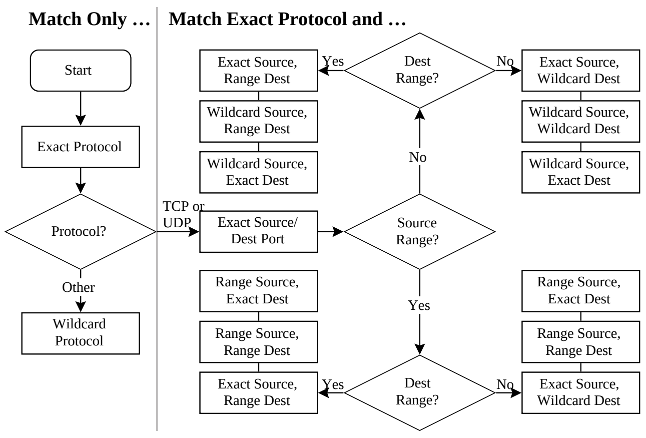

Figure 5. Collapsing Multiple Dimensions

Figure 5 clarifies and extends the idea first published in [20] to allow for a limited amount of ranges on port numbers.

6. Related Work

Multi-dimensional matching is a specialization of the generic point location problem known from computational geometry. Table 2 showed that typical rulesets require five or six dimensions. Unfortunately, the best known general point location algorithms, such as the two- and three-dimensional solution given in [4], require either space or time exponential to the number of dimensions, which is clearly impractical. In addition, the data structures used are often complicated to handle, making them asymptotically optimal, but very expensive for most practical purposes.

Recently, packet classification research finally started to bloom. The solutions found can be grouped as follows: *Directed Acyclic Graphs* (DAGs) are used by [8, 5, 23]. Depending on the implementation, they suffer from backtracking or memory explosion problems. [8, 23] dynamically select the bit(s) which narrow the solution space down most. [5] heavily relies on caching to reduce the number of full-blown packet classifications needed, which does not perform well in the backbone.

Grid of tries [20] cleverly meshes two tries, one for each supported dimension. Although one dimension could be replaced by a more efficient algorithm, at least one dimension must be a pure binary trie, which is slow and does not scale well to larger address sizes, such as for the Internet Protocol version 6 (IPv6) [6].

Quadtree solutions [3] are easy to update, but suffer from a similar problem, that they need to be linear in the number of address bits.

Hardware solutions include the main proposal given in [12]. Unfortunately, its requirement for logical operations on extremely wide RAM (N bits wide) and memory requirements approaching $O(N^2)$ bits for fast response renders it impractical for large filter databases.

Combination is the key to [10]: Each field is looked up individually and the results of these lookups are combined pairwise and iteratively, narrowing the equivalence classes until the final solution is found.

7. Conclusions and Future Work

In this paper, we have shown an efficient technique enabling efficient two-dimensional longest prefix matching in general. To test its performance, we developed models for possible future two-dimensional classification patterns. For more than two dimensions, a native three-dimensional algorithm has been introduced that is able to perform fast searches when the sizes of sets of mutually ambiguous prefix length tuples remains small. More importantly, we have shown an efficient scheme to match 5-tuples used for Internet packet classification. The algorithms lend themselves very well to parallelization, pipelining, and thus to implementation in hardware. Unlike other approaches, the results also do not rely too much on the properties of the data set and are thus largely immune to changes in the database. Thanks to the underlying algorithm, Line search also has a strong performance component logarithmically to the number of bits in the prefix length. We therefore expect it to scale better to the longer addresses in IPv6 than other approaches.

Many of the benchmarks above do not deal with prefixes, but only with the utilized prefix lengths. We are working on improving our model to see how future classification databases could look like. We will also see whether the adaptive search using Ropes [21] can be used to improve search speed by narrowing down the solution set much faster. Preliminary analysis suggests that the improvements in the two-dimensional case will be significantly higher than those for one-dimensional lookups. While we believe in the representativeness of the prefix length pair simulations above, we doubt that Rope search results based on our synthetic data would bear any resemblance to real data.

References

[1] H. Adiseshu, S. Suri, and G. Parulkar. Detecting and resolving packet filter conflicts. In *Proceedings of IEEE Infocom 2000*, Mar. 2000.

[2] S. Blake, D. Black, M. A. Carlson, E. Davies, Z. Wang, and W. Weiss. An architecture for differentiated services. Internet RFC 2475, Dec. 1998.

[3] M. M. Buddhikot, S. Suri, and M. Waldvogel. Space decomposition techniques for fast Layer-4 switching. In *Proceedings of the IFIP Sixth International Workshop on Protocols for High Speed Networks (PfHSN '99)*, pages 25–41, Salem, MA, USA, Aug. 1999.

[4] M. de Berg, M. van Kreveld, and J. Snoeyink. Two- and three-dimensional point location in rectangular subdivisions. *Journal of Algorithms*, 18(2):256–277, 1995.

[5] D. S. Decasper, Z. Dittia, G. Parulkar, and B. Plattner. Router plugins: A modular and extensible software framework for modern high performance integrated services routers. In *Proceedings of ACM SIGCOMM '98*, Sept. 1998.

[6] S. Deering and R. Hinden. Internet protocol, version 6 (IPv6) specification. Internet RFC 2460, 1998.

[7] K. Egevang and P. Francis. The IP network address translator. Internet RFC 1631, May 1994.

[8] P. Francis Tsuchiya. A search algorithm for table entries with non-contiguous wildcarding. Known as "Cecilia," available from the author <francis@aciri.org>, 1991.

[9] V. Fuller, T. Li, J. Yu, and K. Varadhan. Classless Inter-Domain Routing (CIDR): an address assignment and aggregation strategy. Internet RFC 1519, Sept. 1993.

[10] P. Gupta and N. McKeown. Packet classification on multiple fields. In *Proceedings of ACM SIGCOMM '99*, pages 147–160, Cambridge, Massachusetts, USA, Sept. 1999.

[11] IANA. Internet Assigned Number Authority. http://www.iana.org/.

[12] T. V. Lakshman and D. Stiliadis. High speed policy-based packet forwarding using efficient multi-dimensional range matching. In *Proceedings of ACM SIGCOMM '98*, pages 203–214, Sept. 1998.

[13] National Laboratory for Applied Network Research. WAN packet size distribution. http://www.nlanr.net/NA/Learn/packetsizes.html, 1997.

[14] A. Nye, editor. *X Protocol Reference Manual: Volume Zero for X11, Release 6*, volume 0 of *Definitive Guide to X Windows*. O'Reilly and Associates, Feb. 1995.

[15] Transmission control protocol. Internet RFC 793, Sept. 1981.

[16] Y. Rekhter and T. Li. An architecture for IP address allocation with CIDR. Internet RFC 1518, Sept. 1993.

[17] J. K. Reynolds and J. Postel. Assigned numbers. Internet RFC 1700, Oct. 1994.

[18] E. C. Rosen, A. Viswanathan, and R. Callon. Multiprotocol label switching architecture. Internet RFC 3031, Jan. 2001.

[19] A. Shaikh, J. Rexford, and K. G. Shin. Load-sensitive routing of long-lived IP flows. In *Proceedings of ACM SIGCOMM '99*, Cambridge, Massachusetts, USA, Sept. 1999.

[20] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel. Fast and scalable layer four switching. In *Proceedings of ACM SIGCOMM '98*, pages 191–202, Sept. 1998.

[21] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable high speed IP routing table lookups. In *Proceedings of ACM SIGCOMM '97*, pages 25–36, Sept. 1997.

[22] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable high-speed prefix matching, 2000. Submitted for Publication.

[23] T. Y. C. Woo. A modular approach to packet classification: Algorithms and results. In *Proceedings of Infocom 2000*, Mar. 2000.