An $O(n \log n)$ Output-Sensitive Algorithm to Detect and Resolve Conflicts for 1D Range Filters in Router Tables

†Technical Report – Albert-Ludwigs-Universität Freiburg

Khaireel A. Mohamed and Christine Kupich

Institut für Informatik†. Lehrstuhl Algorithmen & DatenStrukturen.
{khaireel, kupich}@informatik.uni-freiburg.de

August 5, 2006

Preamble. We present an output-sensitive solution to the *offline* variant of the conflict detection and resolution problem for a given set of n 1D arbitrary range filters R. This Slab-Detect algorithm is applicable to two popular tie-breaking rules in the event of filter conflicts – the most-specific tie breaking rule (MSTB), and the highest-priority filter rule (HPF). We are able to detect all conflict situations in R and report an optimal number of $O(n)$ slab-resolves to make the set R conflict-free. The algorihm achieves a worst-case time complexity of $O(n \log n)$ and utilises $O(n)$ space. As a direct follow up to the main lemmata, this article serves to support and strengthen the claims and conjectures of our earlier report with robust experiment proofs from data collected on various simulations running the Slab-Detect algorithm for R on IPv4 and IPv6 settings.

Keywords: filter conflict, detection, reporting, resolving, classification, sweep-line, intervals
CR-Classifications: C.2.1 (Network communications), F.2.2 (Geometrical problems and computations), H.3.4 (Information networks)

1 Introduction

The design of IP version 6 (IPv6) is primarily to overcome the inevitable 32-bit address shortage in the current IP version 4 (IPv4), and to address several other important issues that are either lacking in or are problems caused by IPv4. During the transition from IPv4 to IPv6, in which both protocols will coexist, the Internet Engineering Task Force (IETF) has defined several technologies to ensure the smooth metamorphosis while adding value and extending the capabilities of edge routers [1, 2]. Edge routers, in short, are complicated devices that bridges the two protocols, ensuring that packets are delivered correctly and efficiently to and from networks running the two protocols.

A packet flowing into an internet router is assigned an action based on its header fields, typically the destination address field in either IPv4 or IPv6. The actions in the router table are rules that act as *filters* to direct the packet to its next destination. When the header fields of an incoming packet trigger several filters from the table, such that the router *cannot* determine one unique filter to classify the packet, then the set of filters in the router table is said to be conflicting. A variety of aspects exists as to how they collectively define the term 'filter conflict' [3, 4, 5], and there is

† Fakultät für Angerwandte Wissenschaften. Georges-Köhler-Allee 51. D-79110, Freiburg, Deutschland.

still a considerable amount of discussions on how to detect such conflicts in a given set of filters R, how to break the ties when confronted with a conflict situation, and how to 'resolve' unbreakable conflicting pairs of filters [6, 7, 8, 9, 10, 11].

Our approach to the problem above is construed to the offline version in one dimension, where we apply the Slab-Detect algorithm directly to a given set of filters R and report all k non-conflict-free slabs that require resolve filters. We exploit the geometric properties of the router table and regard a range-filter $r \in R$ as an inclusive interval $[r.lo, r.hi]$ which fields denote the low- and high-endpoints on the discrete universe U. An internet packet classification is then a point $p \in U$, so that any filter $r \in R$ *applies* to p if $p \in r$.

The specifics of definitions and terminologies introduced in this report are as a result of the direct follow-up from our earlier report [12]. It lays the foundations to the conflict detection and resolution problem in the generic sense, and highlights the differences in both the online and offline variants through various literary evidence. In this article, we continue to build on those foundations. More specifically, we apply the theories and concepts in Kupich *et al.* [12] on a 1D domain involving range filters in an internet router table.

2 The Output-Sensitive Solution to the 1D Offline Problem

Our solution utilizes a *slab-detect* algorithm running on the sweepline principles to report $O(n)$ resolves for all conflicts it detects based on a given set of 1D range-filters. It achieves a worst case time complexity of $O(n \log n)$ and uses $O(n)$ space on a single left-to-right sweep. The algorithm is output-sensitive in a sense that it reports only those slabs that require a resolve filter.

Definition 1. *A set of filters R is* conflict-free *iff for each point p there is a unique range $r \in R$ such that $p \in r$ and for all other filters $s \in R$ stabbed by p, $r \subseteq s$ [12].*

Let R be a set of n 1D arbitrary range-filters under MSTB. The lo- and hi-endpoints of the filters in R partition the discrete universe into at most $2n$ slabs. Each distinct endpoint makes up the boundary between two slabs and is placed in a linearly ordered set to form the event points of the sweepline paradigm.

Definition 2. *A slab σ_i is a single partition of the set R containing a range of points from the discrete universe between two event points ep_i and ep_{i+1}, where $\sigma_i = [ep_i, ep_{i+1})$ and that all points $p \in \sigma_i$ stab the same subset $S_i \subseteq R$ and are represented collectively by ep_i.*

Definition 3. *Slab σ_i is conflict-free under MSTB iff there is a shortest filter $r \in S_i$ that applies to ep_i, such that r is contained in all other filters $S_i \subseteq R$ stabbed by ep_i.*

It follows that the shortest (most specific) filter r applies to ep_i, and hence to all p in slab σ_i. Thus, we detect a conflict situation at ep_i if the shortest filter r is not contained in at least one $s \in S_i$.

Definition 4. *A filter $r = [r.lo, r.hi]$* left-*overlaps filter $s = [s.lo, s.hi]$ iff $r.lo < s.lo < r.hi < s.hi$.*

Definition 5. *A filter r* right-*overlaps s iff $s.lo < r.lo < s.hi < r.hi$.*

Lemma 1. *Let σ_i be a* non-*conflict-free slab. Then σ_i requires only a single "slab-resolve" filter $h_{\sigma_i} = [ep_i, ep_{i+1})$ to make it conflict-free.*

Proof. If σ_i is not conflict-free, then the shortest filter r stabbed by ep_i is not contained in all other filters in S_i. Let h_{σ_i} be a resolve filter that spans σ_i. Then for the same ep_i, lengths $||h_{\sigma_i}|| < ||r||$, so that h_{σ_i} is now the shortest filter that applies to ep_i and is contained in all S_i (including r).

Corollary 1. *Slab σ_i requires h_{σ_i} if $\exists s \in S_i$ that* left-*overlaps r and $s.hi > ep_i$.*

Corollary 2. *Slab σ_i requires h_{σ_i} if $\exists s \in S_i$ that* right-*overlaps r and $s.lo \leq ep_i$.*

Theorem 1. *Let $SlabResolve(R)$ be the set obtained from R by adding a slab-resolve h_{σ_i} for every non-conflict-free slab σ_i. Then $R \cup SlabResolve(R)$ is conflict-free.*

Proof. After adding a slab-resolve h_{σ_i} for every non-conflict-free slab σ_i, there exists a most-specific filter in every single slab. Hence, by Definition 1, the set $R \cup SlabResolve(R)$ is conflict-free.

Therefore, it is sufficient to (**i**) determine the smallest filter r from all S_i stabbed by ep_i, and then (**ii**) check that r is contained in all S_i to deduce that slab σ_i is conflict free.

2.1 Status Structures

A filter $s \in R$ belongs to the status structure T at ep_i iff $ep_i \in [s.lo, s.hi)$, and all such filters are stored in a collective structure $ordLen(T)$, which orders the filters in their ascending lengths. T can be split into two distinct subsets, so that $T = new(T) \cup current(T)$, where $s \in new(T)$ if $s.lo = ep_i$, otherwise, $s \in current(T)$.

The filters in $new(T)$ are ordered in a single structure in their ascending lengths. The filters in the generic set $current(T)$, on the other hand, are maintained in two separate red-black trees; (**i**) $cur_R(T)$ that orders all s in their ascending hi-endpoints, and (**ii**) $cur_L(T)$ that orders all s in their descending lo-endpoints. Furthermore, if $current(T) \neq \emptyset$, then let lhp and hlp be pointers to the lowest of all the hi-endpoints and the highest of all the lo-endpoints in $current(T)$. Respectively, lhp and hlp point to the top of the ordered lists in $cur_R(T)$ and $cur_L(T)$.

2.2 Handling Event Points

At each event point ep_i, we take the shortest filter $r \in T$ from the top of the list in $ordLen(T)$ and check if $r \in new(T)$, otherwise, $r \in current(T)$. Here, there are two cases to consider:

Case I: That the shortest filter $r \in new(T)$.
Case II: That the shortest filter $r \in current(T)$.

Lemma 2 (Case I). *If $r \in new(T)$ and $current(T) = \emptyset$, then slab σ_i is conflict-free.*

Proof. At ep_i all $s \in new(T)$ have the same lo-endpoints, and since $r \in new(T)$, r is completely contained in all $s \in new(T)$.

Lemma 3 (Case I). *If $r \in new(T)$ and $lhp \geq r.hi$, then slab σ_i is conflict-free.*

Proof. From Lemma 2, r cannot conflict with any $s \in new(T)$. Also, by Definition 2, slab σ_i cannot span any longer than $||r||$[3], which then makes it unnecessary to consider the case for $lhp \geq r.hi$ in slab σ_i.

Lemma 4 (Case I). *If $r \in new(T)$ and $lhp < r.hi$, then $\exists$ at least one filter $s \in current(T)$ that left-overlaps r, so that slab σ_i is no longer conflict-free.*

Proof. Straightforward from Corollary 1.

Lemma 5 (Case II). *If $r \in current(T)$ and $new(T) \neq \emptyset$, then every $s \in new(T)$ conflicts with r, so that slab σ_i is no longer conflict-free.*

[3] However, $||\sigma_i||$ can be shorter than $||r||$ if $\exists ep_{i+1}$ that comes before $r.hi$.

Proof. Straightforward from Corollary 2; $\forall s \in new(T)$ right-overlaps with r.

Lemma 6 (Case II). *If $r \in current(T)$ and $new(T) = \emptyset$, then slab σ_i is conflict-free iff $hlp \leq r.lo$ and $r.hi \leq lhp$.*

Proof. It follows from Definition 3 that r is contained in all $s \in current(T) = T$.

In the case that there are several filters at event point ep_i that have the same shortest length $||r||$, all these filters will be at the top of the list in $ordLen(T)$. We still pick the filter at the top of the list and test it against the lemmata above. By now, we would understand that it does not matter which subset of T the shortest filter r comes from, it will be correctly handled by the algorithm. We present it in Algorithm 1, which takes in the current event point ep_i and the new set of filters $new(T)$ stabbed by ep_i.

Algorithm 1 HandleEventPoint(ep_i, $new(T)$)

Update $current(T)$
Update $ordLen(T)$
Set $r \leftarrow ordLen(T)$.getTopElement()
Set $h_{\sigma_i} \leftarrow [ep_i, ep_{i+1})$
if $current(T) \neq \emptyset$ **then**
  Set $lhp \leftarrow cur_R(T)$.getTopElement()
  Set $hlp \leftarrow cur_L(T)$.getTopElement()
  if $r \in current(T)$ **and** $new(T) = \emptyset$ **and** $hlp \leq r.lo$ **and** $r.hi \leq lhp$ **then** Reset $h_{\sigma_i} \leftarrow$ null
  if $r \in new(T)$ **and** $lhp \geq r.hi$ **then** Reset $h_{\sigma_i} \leftarrow$ null
else
  if $r \in new(T)$ **then** Reset $h_{\sigma_i} \leftarrow$ null
end if
return h_{σ_i}

2.3 The Sweepline Environment

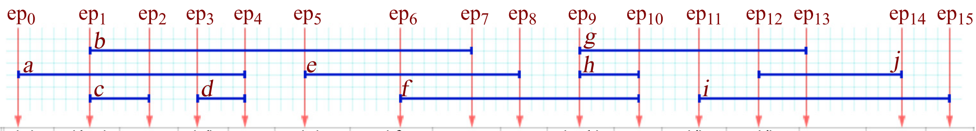
We conceptually extend all intervals by half the size of the distance of points in the discrete raster U, such that all stabbing queries become queries for points falling into the interior of the intervals. Figure 1 shows this transformation on filters r, s, t, and u, such that for any range filter x there is a mapping function $f : x[x.lo, x.hi] \longmapsto x'[x.lo - 0.5, x.hi + 0.5]$. This is important for the Slab-Detect algorithm so as not to miss any crucial event points that make up a valid slab.

For example, the point $ep_m = 19$ in Figure 1(a) stabs *both* filters r and s. If the filters are not mapped as above, then only s exists in the status structure at ep_m, and worse still, the conflict between r and s is not detected. Also, we will miss t completely at $ep_n = 21$. Figure 1(b) shows the solution to these problems, and, as a consequence, all event points ep_i lie on the 0.5 mark.

2.4 Running Slab-Detect

Figure 2 illustrates the Slab-Detect algorithm applied on a given set $R = \{a, b, ..., j\}$, where R partitions the discrete universe into 17 slabs, separated by 16 distinct event points. Sweeping from left to right, Slab-Detect maintains the status structure $T = new(T) \cup current(T)$ at every event point ep_i, determines the shortest filter $r*$ and notes which subset of T it comes from, and then reports whether or not slab σ_i requires a resolve filter. In the given example, Slab-Detect finds that non-conflict-free slabs $\sigma_2, \sigma_5, \sigma_6, \sigma_7, \sigma_{11}$, and σ_{12} require resolve filters, and the reasons for the requirements are indicated by the associated lemmata.

Figure 1. Extending each filter in (a) by half the size of the distance of points to (b) so as not to miss a crucial event point during a sweep in Slab-Detect.



new(T)	{a}	{b,c}	-	{d}	-	{e}	{f}	-	-	{g,h}	-	{i}	{j}	-	-	-
current(T)	-	{a}	{a,b}	{a,b}	{b}	{b}	{b,e}	{e,f}	{f}	{f}	{g}	{g}	{g,i}	{i,j}	{i}	-
r	a	c	a	d	b	e	e	e	f	h	g	g	j	j	i	-
Lemma	2	3	6^{-1}	3	6	4	5	6^{-1}	6	3	6	6^{-1}	4	6	6	-
h_σ_i	-	-	[2,3)	-	-	[5,6)	[6,7)	[7,8)	-	-	-	[11,12)	[12,13)	-	-	-

Figure 2. Running from left to right, the Slab-Detect algorithm reports six resolves required for the non-conflict-free slabs $\sigma_2, \sigma_5, \sigma_6, \sigma_7, \sigma_{11}$, and σ_{12}.

Lemma 7. *Let g and h be two adjacent resolve filters reported by the Slab-Detect algorithm that are required in and span two consecutive slabs σ_i and σ_{i+1}. Then g and h cannot be merged.*

Proof. Every adjacent event points on the sweepline contains a unique set of filters in $new(T)$ and $current(T)$; that is, no two consecutive event points ep_i and ep_{i+1} have the same elements in the status structure T. Thus it follows that the conditions that make the filters in slab σ_i conflict is different from the conflict conditions in σ_{i+1}. Therefore, g and h cannot be merged.

However, while it is obvious that Slab-Detect does not report duplicated resolve filters, in its worst-case scenario, we get a situation where *every other* event point on the sweepline having the same status structure T. For example, in Figure 3(a), event points ep_4, ep_6, and ep_8 all have the exact same filters in their status structures, and which exact same conditions make it necessary to require resolve filters for the respective slabs σ_4, σ_6, and σ_8. But if we are to ignore filters r and s, then it is sufficient to have only one resolve filter $h = p \cap q$; a reported $O(1)$ output of essential[4] resolve filters, versus $O(n)$ slab-resolves by Slab-Detect.

On the other hand, with the presence of r and s, we get resolve filters $h_1 = r \cap s$, $h_2 = r \cap q$, $h_3 = p \cap s$, and $h_4 = p \cap q$. The resolve filters h_2, h_3, and h_4 are duplicates of h_1, and additional steps are needed to ascertain with the set $R \cup resolve(R)$ to ensure that only essential[4] resolves are reported – which for a set of non-prefix-filters may prove to be very costly. In general, this is a situation similar to Figure 3(b), where without the additional checking steps, we get a reported $O(n)$ output of duplicated resolve filters for slab σ_i versus $O(1)$ non-duplicate, slab-resolves by Slab-Detect.

Another extreme condition discussed in Kupich *et al.* [12] is given in Figure 3(c), where all n filters in R are staggered in such a way that any one filter overlaps all other filters in R. We showed in the previous report that for every resolve between two pairwise conflicting filters in the set R, the union of all these resolves with the set R gives a conflict-free set. However, we may end up

[4] By Lu and Sahni [4], a filter $h \in resolve(R)$ is an *essential* resolve filter iff $R \cup resolve(R) - \{h\}$ has no subset whose union equals h.

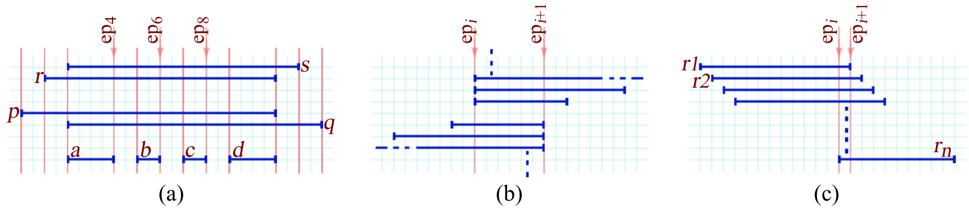


Figure 3. Slab-Detect: (a)Worst-case example. (b) Duplicates not reported. (c) Only one slab-resolve required for slab $\sigma_i = [ep_i, ep_{i+1})$, instead of $\Omega(n^2)$.

with $\Omega(n^2)$ many resolves between ep_i and ep_{i+1}, which is the worst case in the example given in Figure 3(c). The alternative solution offered by the Slab-Detect algorithm is to simply introduce a single slab-resolve $h_{\sigma_i} = [ep_i, ep_{i+1})$ at the same slab.

2.5 Effect on Packet Classification

Based on the discussions above, we argue that the solution offered by the Slab-Detect algorithm will not impede on the original packet classification performance when forwarded with the new conflict-free set of filters. That is, after running Slab-Detect on a non-conflict-free set of filters R, an attempt to classify any incoming packet p in $R' = R \cup SlabResolve(R)$, the number of filters stabbed by p in R' is at most *one* more than the number of filters stabbed by p in the original R. This is because *all* the resolve filters reported by Slab-Detect only add one extra layer to any offending slab; as opposed to $O(n)$ many per slab in the situation above, or $\Omega(n^2)$ many discussed previously and pointed out Kupich *et al.* [12].

3 Experimental Setup

As was with the case of Lu and Sahni [4], we were also unable to get either synthetic or large real data used by the network administrators and others in their research despite substantial efforts. Thus, we too generated our own sets of random data to simulate the real-world situation, while also creating discrepancies (at certain times) in the generated sets to assess the limits of the Slab-Detect algorithm. The random samples are created depending on two main parameters: the total number of filters to generate n, and the bit-length of a filter-field w (where $w = 32$ bits in IPv4, and $w = 128$ bits in IPv6). This gives a maximum range of the filter r to be $[0, 2^w - 1]$.

3.1 Generating and Mixing Filters

An arbitrary range filter $r = [r.lo, r.hi]$ is formed using the function $random(i, j)$ that generates a random integer that is normally distributed between i and j, and is described is follows:

$$r.lo \texttt{ = random(0, } 2^w - 1\texttt{)}, \; r.hi \texttt{ = random(}lo\texttt{, } 2^w - 1\texttt{)}$$

We added variations of discrepancy into the generated set R by introducing an additional parameter $percentPrefix$, which determines the percentage-*mix* of 1D prefix-filters in R. That is, by setting $percentPrefix = 1.0$, we generate all n non-conflicting prefix-filters in R. By varying this parameter between 0.0 and 1.0, we can indirectly control the number of conflicting pairs of filters generated by the base set R, and in doing so, test the Slab-Detect algorithm on its output-sensitivity. Due to the randomness of the distribution of the normally generated filters, we can expect more pairs of conflicting filters for smaller values of $percentPrefix$ than higher ones.

A prefix-filter is described by $f = [b/prefixLen]$, where b is a random bit-pattern, and $prefixLen$ is the number of prefix-bits to retain in b. We generate it as follows:

$$b = \texttt{random(0, } 2^w - 1\texttt{)},\ prefixLen = \texttt{random(0, } w - 1\texttt{)}$$

3.2 The Naïve Algorithm

Here, the Naïve algorithm reports all pairwise conflicting filters following an earlier definition by Kupich *et al.* [12] stated below. Following which is the pseudo codes that explains how the final set of $resolve(R)$ is reported from a given set of n range filters R in Algorithm 2.

Definition 6. *Let $r, s \in R$ be two conflicting filters. Then we call the overlapping filter $r \cap s$ the* resolve filter *for r and s with respect to R. We denote* resolve(R) *as the set obtained from R by adding a resolve filter for every pair of conflicting filters in R.*

Algorithm 2 RunNaïve(R)

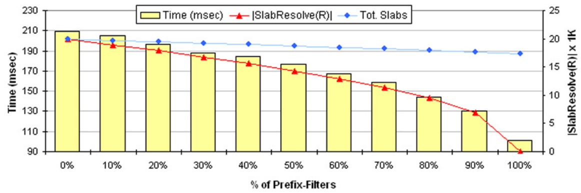
Initialise $resolve(R) \leftarrow \emptyset$
for $i = 0$ to $n - 1$ **do**
  Set filter $r \leftarrow R[i]$
  for all j such that $i \leq j \leq n$ **do**
    Set filter $s \leftarrow R[j]$
    Set resolve filter $h \leftarrow r \cap s$
    if $h \neq \emptyset$ **then** Add h to $resolve(R)$
  end for
end for
return $resolve(R)$

3.3 Simulating Slab-Detect and Naïve under MSTB

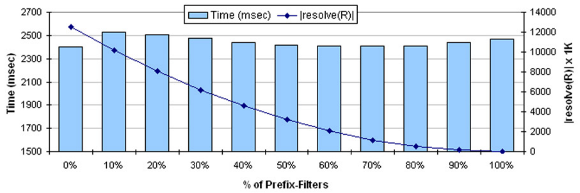
All of our simulations are performed on (Sauron) a Pentium IV, 3GHz machine with 2GBytes RAM running on Java 5.0. We generated arbitrary range filters for $w = 32$ and $w = 128$ with various mixes of prefix-filters in steps of 10% increments, for sample sizes $|R| =$ 10K, 20K, 30K, 40K, and 50K. At each epoch of one sample size $|R|$ and one preset value of $percentPrefix$, we noted the total runtime, the resolve filters reported, and the memory consumption for running R on Slab-Detect and Naïve algorithms. We repeated each epoch 20 times, each time generating a new set of random samples for R, and then calculated the averages for the noted values above.

4 Results

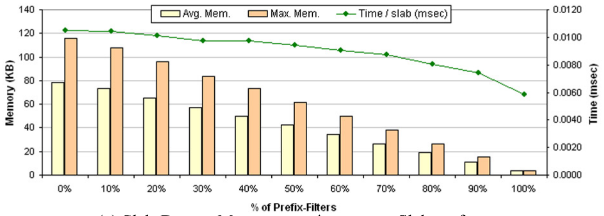
Figure 4 gives a break down of the elements of our simulations for $|R| = 10$K and $w = 128$. We see in Figure 4(a) that the number of reported $SlabResolve(R)$ is never more than the number of slabs as a result of the generated filters in R. Also, we observe that the total runtime for Slab-Detect decreases as the number of $SlabResolve(R)$ decreases, which is a consequence of increasing the percentage mix of prefix-filters in the sample R. This shows that Slab-Detect is indeed *output-sensitive*, as the amount of time taken to report all necessary resolve filters in the set R is proportional to the number of non-conflict-free slabs in R. This, as compared to the total runtime of the Naïve algorithm, is unaffected by the number of conflicting pairs within the sample R as shown in Figure 4(b). From both these figures, we see that the runtime for Slab-Detect is significantly faster than the Naïve algorithm. Bearing in mind that the time reported in Figure 4(b) is only the *base* time taken by Naïve to report the raw set of $resolve(R)$, and additionally, it will take more time for Naïve to remove all unnecessary filters in $resolve(R)$ to make it on par with Slab-Detect's solution.



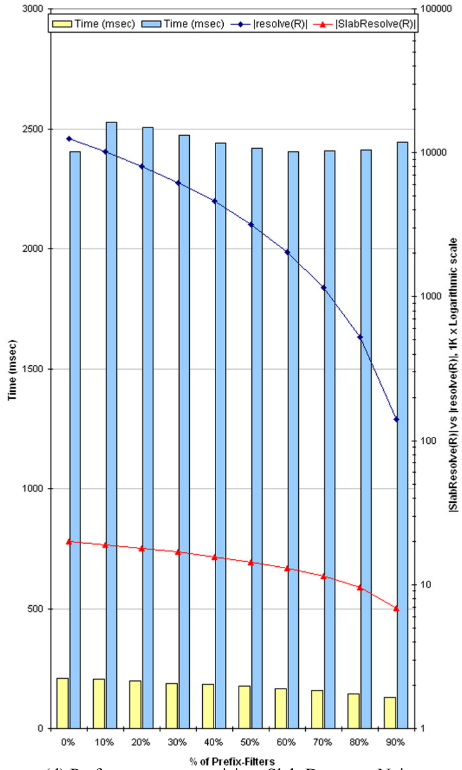
(a) Slab-Detect's performance: Runtime vs $|SlabResolve(R)|$



(b) Naive's performance: Runtime vs $|resolve(R)|$



(c) Slab-Detect: Memory requirement vs Slab performance



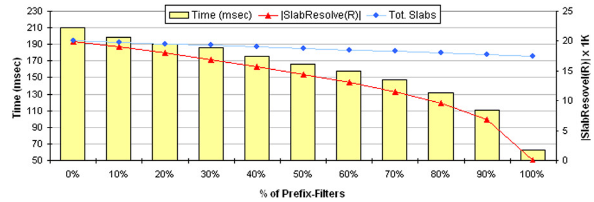
(d) Performance comparision: Slab-Detect vs Naive

Figure 4. Simulation results for Slab-Detect and Naïve. $|R|$ =10K, w = 128.

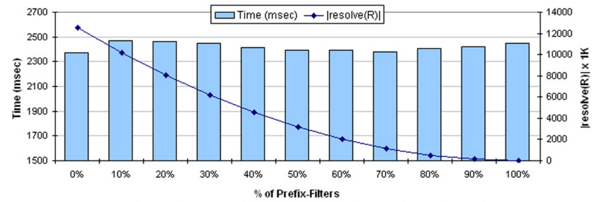
Memory requirement measures the total number of filters retained in memory per event point; that is, the total number of filters in the status structure T as implemented by the Slab-Detect algorithm explained earlier in Section 2.2. Each filter is assumed to have one *source*-address field and one *destination*-address field, amounting to 256 bits = 64 Bytes of memory. Throughout the simulations, we noted the average and maximum memory requirements to handle Slab-Detect on $w = 128$, and depict this in Figure 4(c).

Finally, Figure 4(d) directly compares the Slab-Detect and Naïve algorithms on their runtime performance and the number of resolves each algorihm offers. Clearly, we see that Slab-Detect outperforms Naïve in terms of runtme, and it also conserves the number of reported resolves (compared to Naïve) that will ensure optimal memory management for the system.

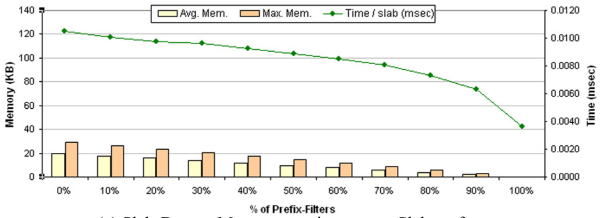
We also simulated different sets of $|R|$ = 10K for $w = 32$, to compare the overall performance of the Slab-Detect between IPv4 and IPv6. The complete roundups, similar to the above observations are illustrated in Figure 5. We do not see a difference in terms of runtime – that is, Slab-Detect runs as if the number of bits of address space makes no difference to its performance. Although, in the performance by Naïve, we see that it requires a little more time to handle $w = 128$ than it would handle $w = 32$.



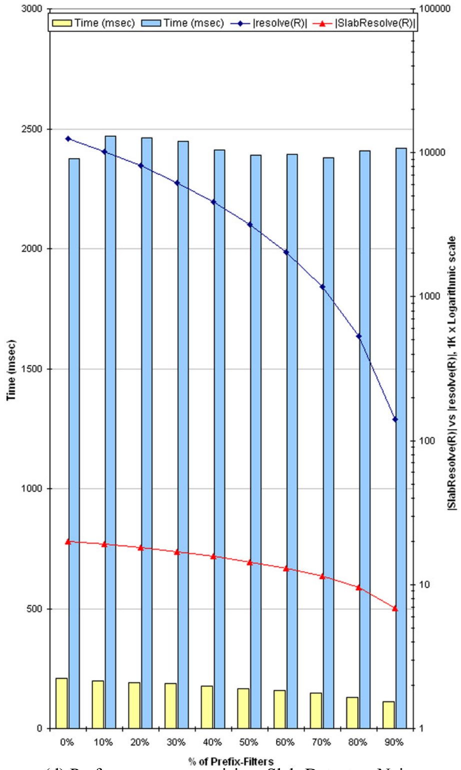
(a) Slab-Detect's performance: Runtime vs $|SlabResolve(R)|$



(b) Naive's performance: Runtime vs $|resolve(R)|$

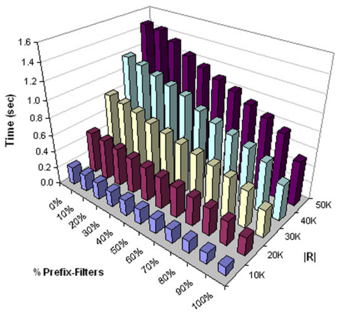


(c) Slab-Detect: Memory requirement vs Slab performance

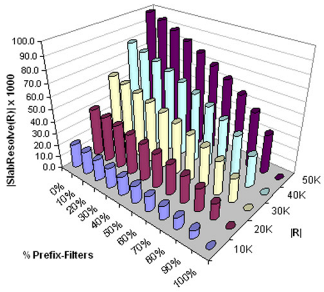


(d) Performance comparision: Slab-Detect vs Naive

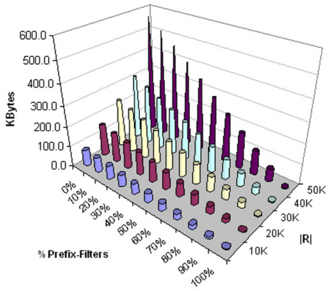
Figure 5. Simulation results for Slab-Detect and Naïve. $|R|$ =10K, $w = 32$.



(a) Slab-Detect: Runtime performance



(b) Slab-Detect: Reporting $|SlabResolve(R)|$

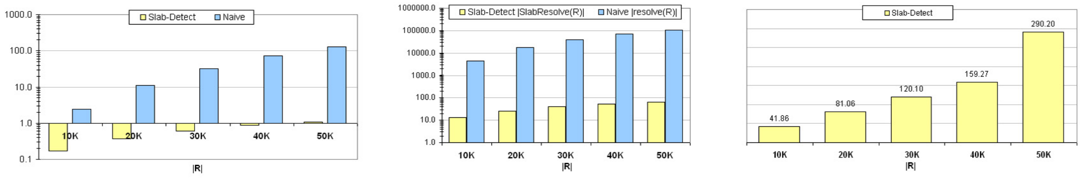


(c) Slab-Detect: Memory requirements

Figure 6. Slab-Detect: Simulation results for $|R|$ = 10K, 20K, 30K, 40K, 50K, and $w = 128$.

The starkest difference between Figure 4 and Figure 5, is the memory requirement. As obviously, we would require more space to store filters in IPv6 than filters in IPv4.

The complete round of simulation outputs for the Slab-Detect algorithm on $|R|$ = 10K, 20K, 30K, 40K, and 50K are summarised in the graphs shown in Figure 6. Its overall performances compared to the Naïve algorithm is shown in Figure 7. As the number of samples $|R|$ increases, we see that the runtime of Slab-Detect very much outperforms the Naïve algorithm, and the slab-resolves offered will not cause a strain in system memory as would the Naïve algorithm to accomodate $R \cup resolve(R)$.



(a) Runtime comparison (seconds x log. scale) (b) Reporting resolves (1K x log. scale) (c) Slab-Detect: Average memory requirement

Figure 7. Average performance comparisons: Slab-Detect vs. Naive for $|R|$ = 10K, 20K, 30K, 40K, 50K, and $w = 128$.

5 Adapting Slab-Detect under HPF Rule

In this section, we show that the original concepts of the Slab-Detect algorithm under MSTB is adaptable for use under the highest-priority filter (tie-breaking) rule (HPF), which also achieves the same runtime performances and space complexities as discussed in Section 4. Under HPF, each filter $r \in R$ is assumed to have been assigned a priority value $p(r)$.

Definition 7. *A filter r has a higher priority than filter s iff $p(r) < p(s)$.*

Definition 8. *A set of filters R is conflict-free under HPF iff for each point p there is a unique filter r of the highest priority that applies to p, such that $p(r) < p(s)$, $\forall s \in S \subseteq R$ stabbed by p.*

Definition 9. *Slab σ_i is conflict-free under HPF iff there is a highest priority filter $r \in S_i$ that applies to ep_i, such that $p(r) < p(s)$, $\forall s \in S_i$ stabbed by ep_i.*

A filter $s \in R$ belongs to the status structure T at ep_i iff $ep_i \in [s.lo, s.hi)$. All such filters are stored in a single red-black tree, $ordPrio(T)$, which orders the filters in their ascending priorities.

At each event point ep_i, we query the top two filters r_0 and r_1 in $ordPrio(T)$.

Lemma 8. *If $p(r_0) \neq p(r_1)$, then slab σ_i is conflict-free.*

Proof. This implies that filter r_0, which applies to ep_i, has the highest priority compared to all others in $ordPrio(T)$ stabbed by ep_i. Otherwise, if $p(r_0) = p(r_1)$, then slab σ_i is no longer conflict-free and requires a resolve filter h_{σ_i}, and assign $p(h_{\sigma_i}) < p(r_0)$.

Lemma 9. *If two adjacent slabs σ_i and σ_{i+1} require resolve filters g and h respectively, then under HPF g and h can be merged, and assign $p(g \cup h) = p(g)$, if $p(g) < p(h)$; otherwise assign $p(g \cup h) = p(h)$.*

Proof. In both σ_i and σ_{i+1}, $g \cup h$ is the highest priority resolve filter stabbed by ep_i and ep_{i+1}.

6 Setting up Packet Classification with Slab-Detect

Here, we discuss packet classification for use under MSTB where we utilise the structure $ordLen(T)$ from Section 2. Alternatively, we can substitute $ordLen(T)$ with $ordPrio(T)$ from Section 5 to allow packet classification under HPF.

Ordinary data structures are *ephemeral* in the sense that an update on the structure destroys the old version, leaving only the new version available for use. A data structure is called *partially persistent* if all versions can be accessed, but only the newest version can be modified, and *fully persistent* if every version can be both accessed and modified. Refer to Driscoll *et al.* [13] for a systematic study of persistence.

We maintain $ordLen(T)$ partially persistent, and insert the resolve filters as and when we detect the necessity for these. After Slab-Detect completes its full run, let T^p refer to the partially persistent version of $ordLen(T)$.

Let R be a set of n 1D range filters and p be an incoming packet to be classified. Version ep_i of T^p consists of the intervals in R that intersect the line $x = p = ep_i$. We know that the intervals in each version of T^p are ordered with respect to their ascending lengths. Hence, the shortest (most specific) filter in each version can be reported in $O(1)$ time.

For a stabbing point p, we search the highest version less than or equals to p. We also need an auxiliary data structure to store the access pointers to the various versions. When the pointers are stored in a balanced binary search tree, initiating access into any version takes $O(\log n)$ time. Packet classification can thus be performed in $O(\log n)$ time.

The node-copying method [13] makes a linked structure partially persistent at an amortized time and space cost of $O(1)$ per update step. Thus, the original sweep in Slab-Detect can still be performed in $O(n \log n)$ time and does not require extra storage.

7 Work In Progress

The concepts explained in Sections 5 and 6 are our current work in progress. We are designing experiments on filters and data packets to test the outcome of the theories stated above. We are also looking to benchmarking our future results to further test the robustness of the Slab-Detect algorithm in its adaptation under the HPF rule, and using it for packet classifications. In due time, we will report our progress.

8 Concluding Remarks

The solutions we presented in this article detect all conflict situations and reports only $O(n)$ slab-resolves to make a given set of 1D arbitrary range filters in an internet router table conflict-free, under both MSTB and HPF rules.This work is unique in a sense that there are no similar previous works to benchmark our results on, particularly in the "offline" variant of the main problem. This is because all the literary evidence we came across either deal exclusively with arbitrary range filters in the "online" variant, or, if the authors deal with the "offline" variant, the data sets involved are prefix-filters in higher dimensions. The closest comparable article to our approach and results is by Eppstein and Muthukrishnan [10], which was discussed in our previous article [12], that only reports whether or not a set of 2D arbitrary range filters conflicts in time $O(n^{\frac{3}{2}})$. This time, however, but does not include the time to report all resolves.

By making a small conjecture, that if our set of 1D arbitrary range filters is to run on Eppstein and Muthukrishnan's 1D *k*-*d*-tree settings, it will still require $O(n^{\frac{3}{2}})$ time to determine if our set contains any conflict. Compared to the Slab-Detect algorithm, which achieves a worst case time complexity of $O(n \log n)$, we can expect an improvement of results based on the above conjecture.

Furthermore, what is reported by the Slab-Detect algorithm is an *optimal* number of slab-resolves required to make a set of filters conflict-free. What is not reported, is the *minimal* number of resolve filters. We pointed this out earlier in Figure 3(c). Along with this open problem, we are also working to see if the methods highlighted in this article are extendable to higher dimensions, and at what costs.

Clearly, during this transition period between IPv4 (32 bits) and IPv6 (128 bits), routers bridging these protocols play an extremely important role in delivering packets correctly and efficiently. And by being able to detect and resolve conflicts for any arbitrary range filters in router tables holding numerous forwarding rules, we can be assured that the algorithm introduced in this report will not suffer under the transitions as it does not manipulate filters by their prefixes or bit patterns.

9 Acknowledgement

This research is funded by the Deutschen Forschungsgemeinschaft (DFG) as part of the research project ,,Algorithmen und Datenstrukturen für ausgewählte diskrete Probleme (DFG-Projekt Ot64/8-3)“.

References

1. D. Preston, “Edge routers for IPv6 migration,” in *Network World* [online] http://www.networkworld.com/news/tech/2001/0423tech.html, Apr 2001.
2. E. Carmès, “The transition to IPv6,” in *Internet Society Member Briefing* [online] http://www.isoc.org/briefings/006/, Jan 2002.
3. A. Hari, S. Suri, and G. Parulkar, “Detecting and resolving packet filter conflicts,” in *INFOCOM 2000: Proceedings of the Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies*. IEEE Press, 2000, pp. 1203–1212.
4. H. Lu and S. Sahni, “Conflict detection and resolution in two-dimensional prefix router tables,” *IEEE/ACM Transactions on Networking*, vol. 13, no. 6, pp. 1353–1363, 2005.
5. A. Datta and T. Ottmann, “A note on the IP table lookup problem,” Dec. 2004, unpublished.
6. S. Sahni, K. S. Kim, and H. Lu, “Data structures for one-dimensional packet classification using most-specific-rule matching,” in *ISPAN 02: Proceedings of the 2002 International Symposium on Parallel Architectures, Algorithms and Networks*. Washington, DC, USA: IEEE Computer Society, 2002, p. 3.
7. S. Sahni and K. S. Kim, “Efficient construction of multibit tries for IP lookup,” *IEEE/ACM Transactions on Networking*, vol. 11, no. 4, pp. 650–662, 2003.
8. K. S. Kim, H. Lu, and S. Sahni, “Prefix and interval-partitioned dynamic IP router-tables,” *IEEE Transactions on Computers*, vol. 54, no. 5, pp. 545–557, 2005.
9. S. Sahni and K. S. Kim, “An O(log n) dynamic router-table design,” *IEEE Transactions on Computers*, vol. 53, no. 3, pp. 351–363, 2004.
10. D. Eppstein and S. Muthukrishnan, “Internet packet filter management and rectangle geometry,” in *SODA 01: Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2001, pp. 827–835.
11. F. Baboescu and G. Varghese, “Fast and scalable conflict detection for packet classifiers,” *Computer Networks*, vol. 42, no. 6, pp. 717–735, 2003.
12. C. Kupich and K. A. Mohamed, “Conflict detection in internet router tables,” Institut für Informatik, Albert-Ludwigs-Universität Freiburg, Tech. Rep. 225, Aug 2006.
13. J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan, “Making data structures persistent,” in *STOC 86: Proceedings of the eighteenth annual ACM symposium on Theory of computing*. New York, NY, USA: ACM Press, 1986, pp. 109–121.