Lessons learned
Hundreds of millions routes in OpenBGPD

Claudio Jeker

RSSF

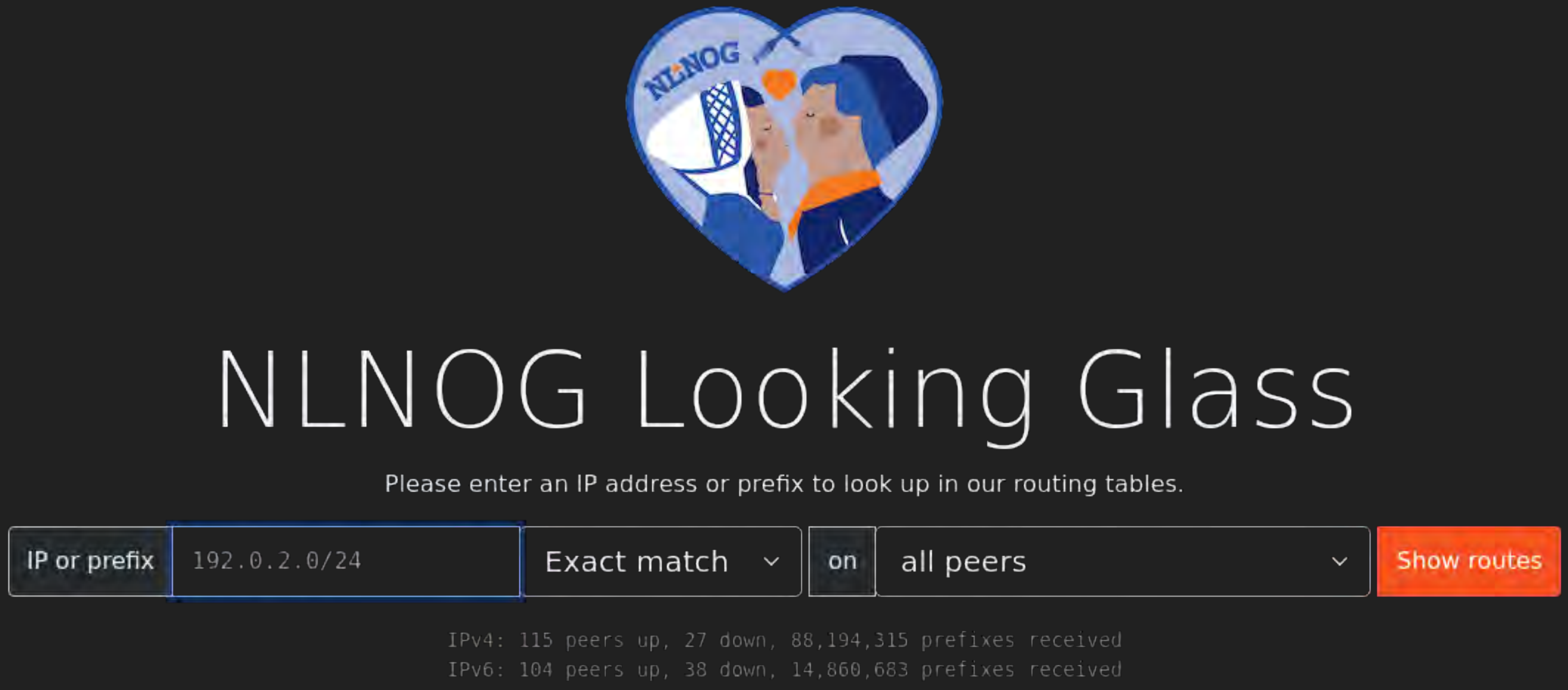
Why does this matter?

Running large scale routers **highlights scalability issues** and helps to fix them before others hit them

Improving BGP performance not only benefits the biggest machines, **everyone else benefits** as well

It also helps to **reduce latency** under heavy load

Good validation for OpenBGPD as a previous version of the looking glass failed to handle the load



NLNOG Looking Glass

Please enter an IP address or prefix to look up in our routing tables.

IP or prefix: 192.0.2.0/24 — Exact match — on — all peers

```
IPv4: 115 peers up, 27 down, 88,194,315 prefixes received
IPv6: 104 peers up, 38 down, 14,860,683 prefixes received
```

The Hardware

VM running Ubuntu 22.04 LTS

CPU: 4 x AMD EPYC Processor @ 2.8GHz

Memory: 125GB total, currently 74GB used

My largest OpenBGPD setup

IPv4: 115 peers up, 27 down, 88,219,041 prefixes received

IPv6: 104 peers up, 38 down, 14,867,571 prefixes received

RDE memory statistics

```
 412,201,131 prefix entries using 49.1GB
  32,130,792 BGP path attribute entries using 2.4GB
  32,130,792 BGP AS-PATH attribute entries using 1007MB
   2,257,071 entries for 32,376,786 BGP communities using 737MB
RIB using 52.7GB of memory
```

What did I see?

It takes a long time from startup to steady state

CPU usage of RDE process is at 100% for hours

Firehose feeds are almost unable to catch up with input

Some looking glass queries take very long to complete

The problem definition

Ingest as many full feeds as possible Keep the sessions alive, even during crunch time	Not a problem, design of OpenBGPD ensures that keep-alive messages are sent	✅
Converge in reasonable time and keep up with the incoming updates Provide a few firehose feeds using BGP add-path	Unclear reason for the sluggish performance. This needs investigation!	‼️
Provide quick queries of the RIB for the looking glass	Most queries OK, only some take too long	❗

Low hanging fruit

Problem: Some looking glass queries did full table walks even though not needed.

Solution: Implement **subtree walks** and simple lookup loop to **walk only a small part** of the tree.

Benefit: Looking glass queries faster

Problem: Various hash tables did not scale to large number of elements. As a result many lookups were slow.

Solution: Replaced hash tables with a balanced **binary lookup tree**. In one case **remove lookup** in favor of an additional copy.

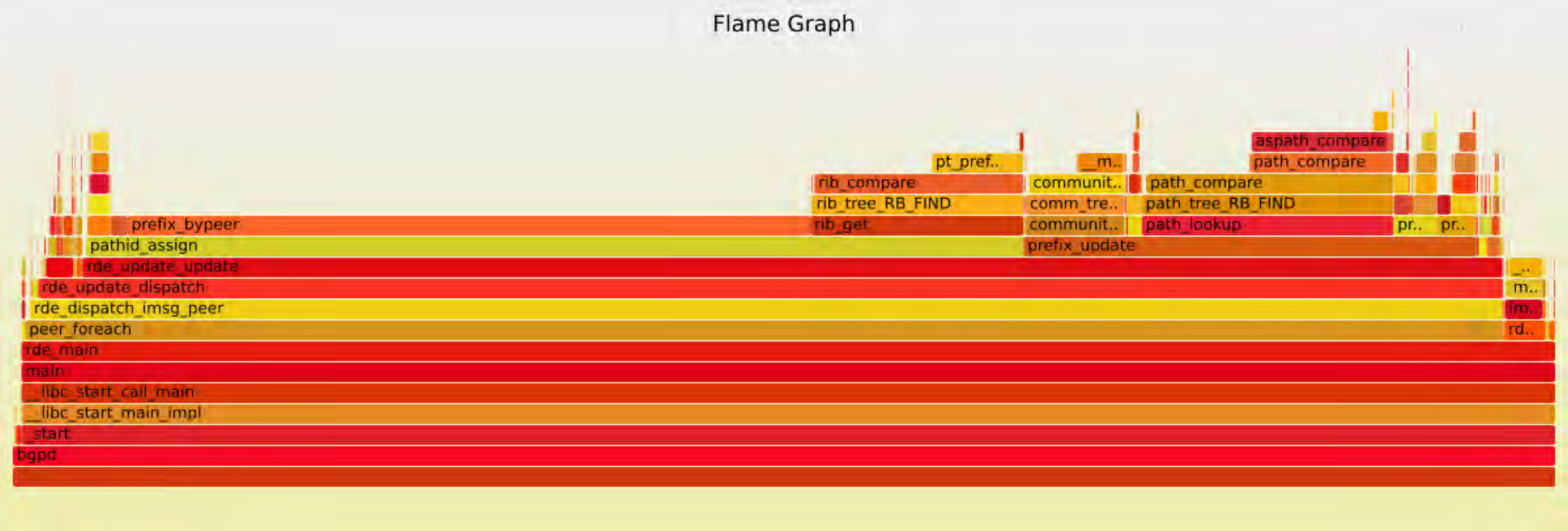
Benefit: Internal lookups faster

Profile don’t speculate

I used **perf utility** from linux to generate flame graphs:

- **Stack traces** collected using 99 samples per second for 300 second of runtime
- Flame graphs show **statistical distribution** of samples (stack traces)
- Flame graph does not show if a function takes a lot of time or is frequently called

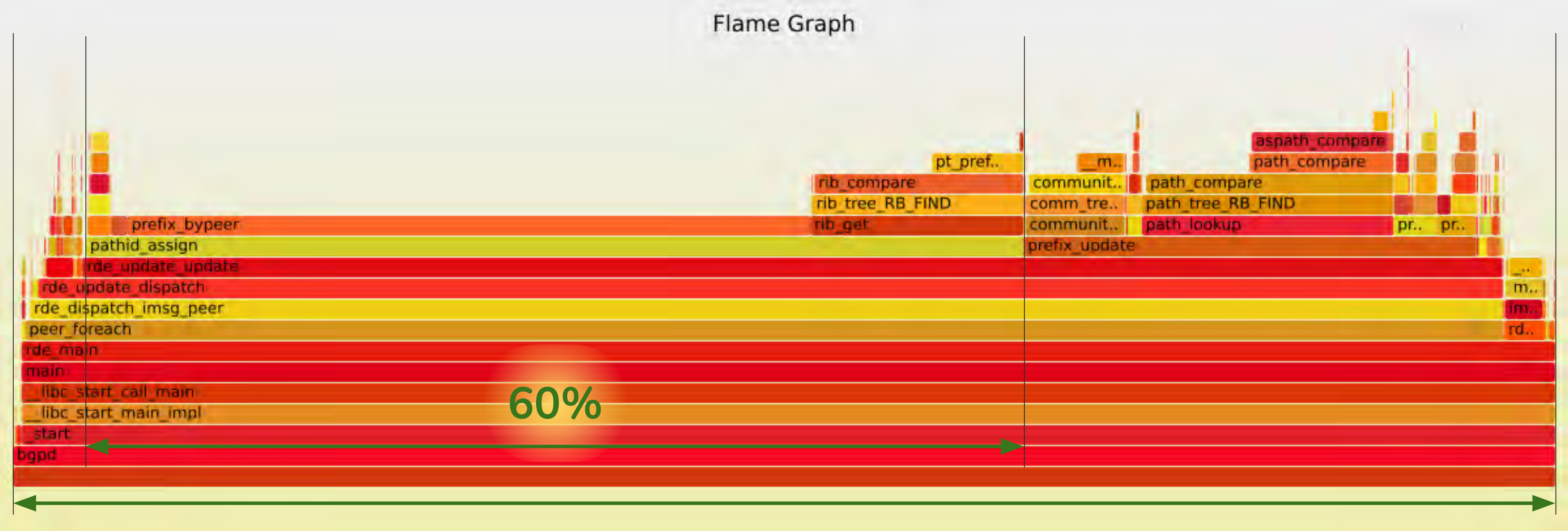
What is a flame graph



Shows path to currently running function (bottom to top), color does not matter.

The longer a bar is the more time is spent in that function.

Input processing - 60% time spent in pathid_assign



Code before - lots of work for something trivial

```
pathid_assign(struct rde_peer *peer, ...)
{
  /* Assign a send side path_id to all paths. */
  re = rib_get(rib_byid(RIB_ADJ_IN), prefix,
        prefixlen);
  if (re != NULL)
    p = prefix_bypeer(re, peer, path_id);
  if (p != NULL)
    path_id_tx = p->path_id_tx;
  else {
    do {
      /* assign new local path_id */
      path_id_tx = arc4random();
    } while (pathid_conflict(re, path_id_tx));
  }
  return path_id_tx;
```

First lookup route in the RIB

Then locate the prefix of this peer if it exists.

If it exists reuse that pathid

Else find new pathid by starting with a random one

Check if there is a conflict which calls prefix_bypeer() again

Code after - using precomputed value

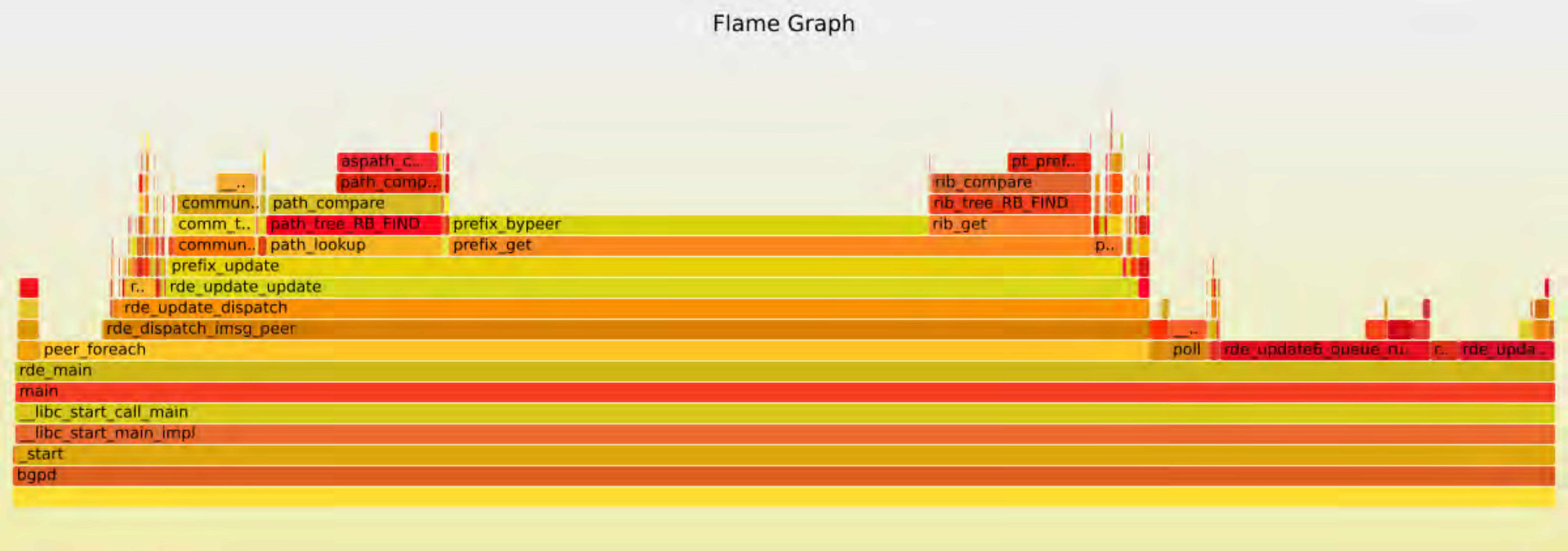
```
pathid_assign(struct rde_peer *peer, ...)
{

  /* If peer has no add-path use the
     * per peer path_id */
  if (!peer_has_add_path(peer, prefix->aid,
        CAPA_AP_RECV))
    return peer->path_id_tx;

  /* peer uses add-path, therefore per path
       path_id needs to be assigned */
  ...
  /* more or less the old code follows */
```

Check if peer is not using add-path.

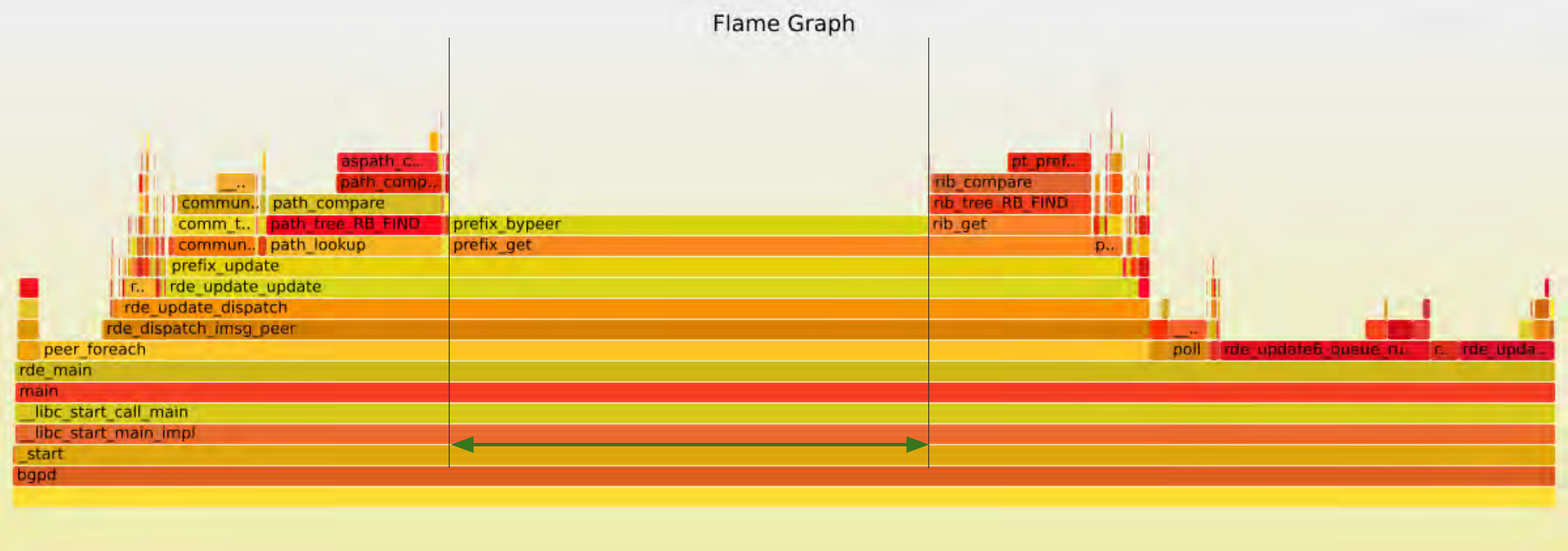
Use precalculated path_id since the peer can only send a single path

Peers using add-path receive still need to use the old more costly algorithm.

pathid_assign() fixed

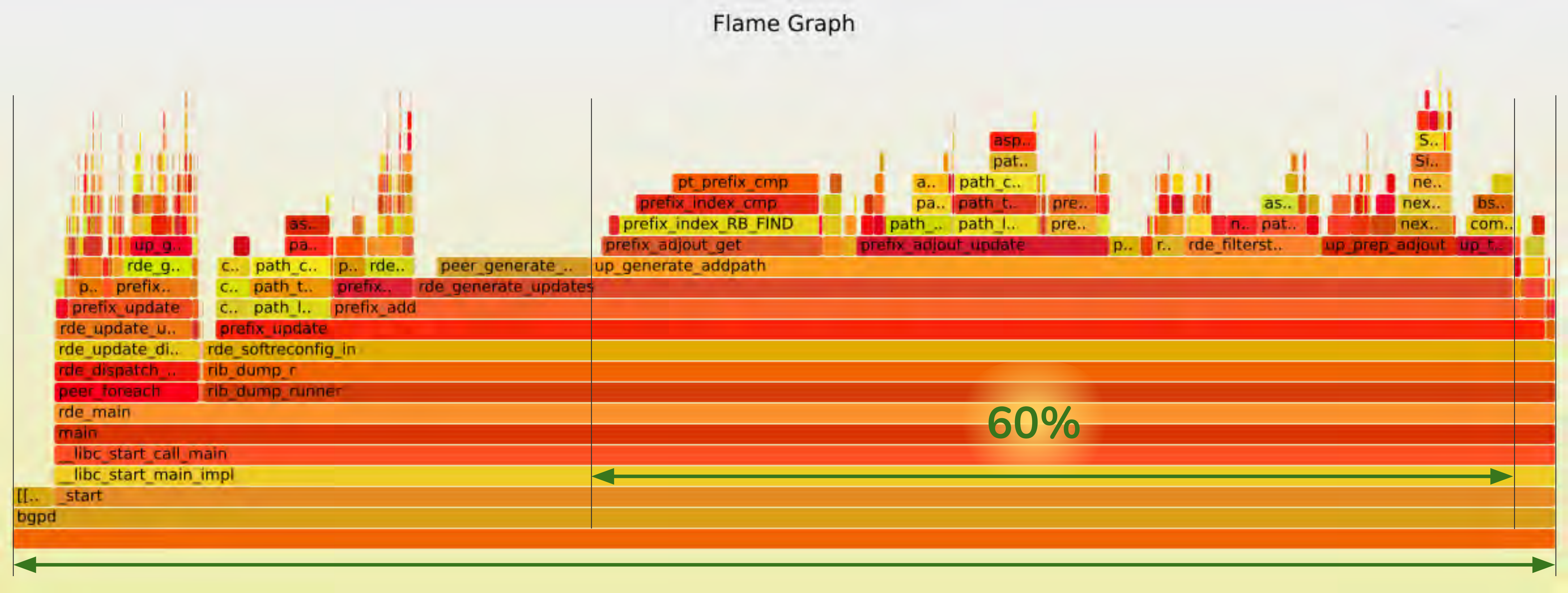


prefix_bypeer() is still an issue



now in prefix_get() called by prefix_update()

Sending updates is too expensive



60% time spent in up_generate_addpath

Sending updates is too expensive

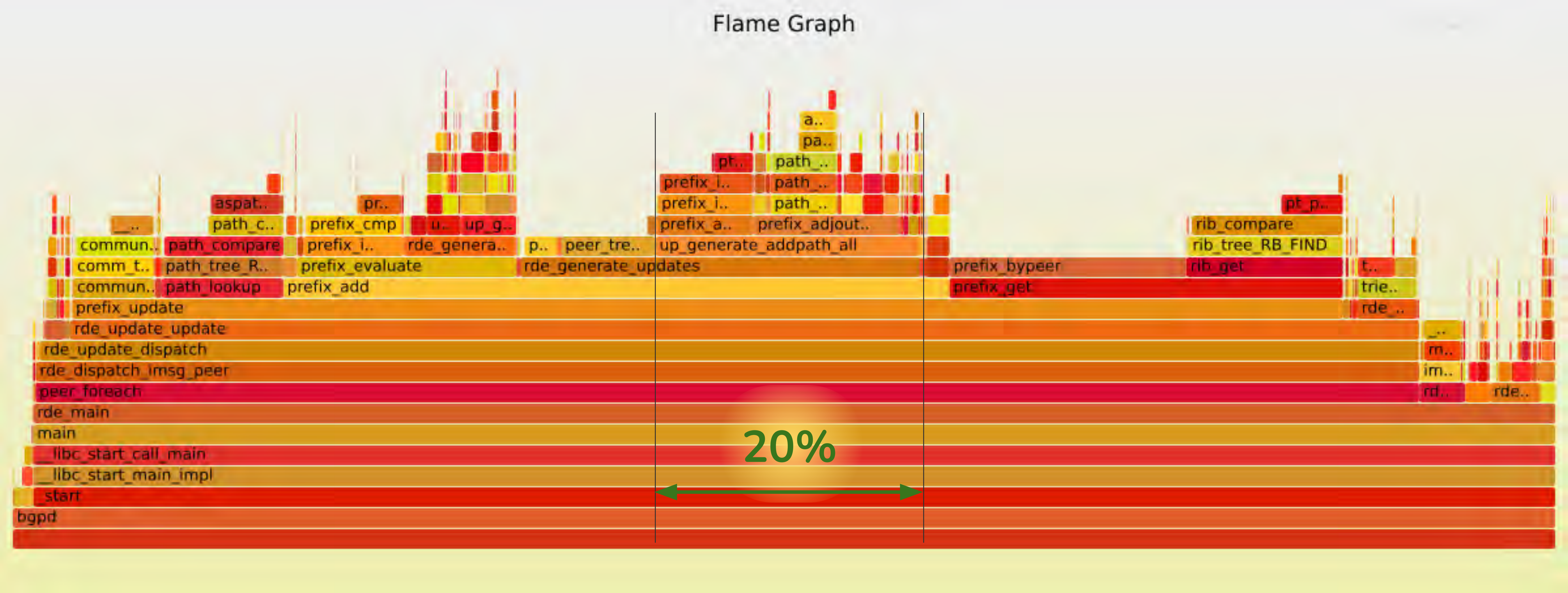
Problem: up_generate_addpath() is complex

The function **re-evaluates all prefixes** every time

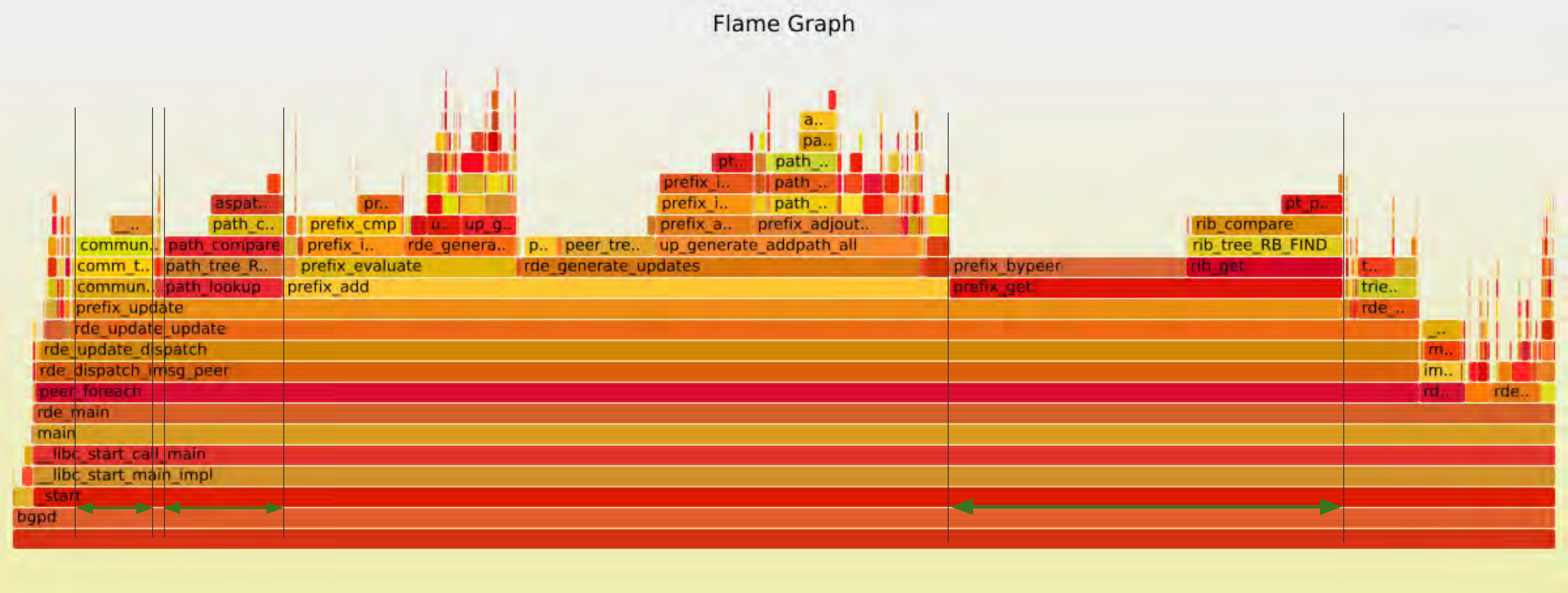
For “add-path all” this is **extra unnecessary work**

Solution: Introduce up_generate_addpath_all() that is **optimized for “add-path all”** and just adds or removes the changed prefix

Less time spent generating updates



Various lookup functions still a problem



prefix_get() with prefix_bypeer() most prominent one

Future work

Problem: prefix_get() and prefix_bypeer() are still slow

Solution: Redesign part of the RIB ‘database’ model to reduce lookups

Problem: Lookup functions slow because of CPU cache misses

Solution: Replace binary trees with a more cache friendly lookup function but first verify that this is actually the case. 🤓

Conclusion

After solving some scalability issues the **system is up and running effectively**

Initial total convergence time is below 90 minutes

Sessions are **up and stable**

Even possible to frequently update the ROA tables and show the origin validation state in the looking glass

Thanks

NetNod for inviting me to this event

Siri Brenden for all the support

Job Snijders and NLnog to let me play with the looking glass machine

Questions?