TRASH
A dynamic LC-trie and hash data structure

Robert Olsson[*] Stefan Nilsson[†]

August 18, 2006

Abstract

A dynamic LC-trie is currently used in the Linux kernel to implement address lookup in the IP routing table [6, 9]. The main virtue of this data structure is that it supports both fast address lookups and frequent updates of the table. Also, it has an efficient memory management scheme and supports multi-processor architectures using the RCU locking mechanism. The structure scales nicely: the expected number of memory accesses for one lookup is $O(\log \log n)$, where n is the number of entries in the lookup table. In particular, the time does not depend on the length of the keys, 32-bit IPv4 addresses and 128-bit addresses does not make a difference in this respect.

In this article we introduce TRASH, a combination of a dynamic LC-trie and a hash function. TRASH is a general purpose data structure supporting fast lookup, insert and delete operations for arbitrarily long bit strings. TRASH enhances the level-compression part of the LC-trie by prepending a header to each key. The header is a hash value based on the complete key. The extended keys will behave like uniformly distributed data and hence the average and maximum depth is typically very small, in practice less than 1.5 and 5, respectively.

We have implemented the scheme in the Linux kernel as a replacement for the dst cache (IPv4) and performed a full scale test on a production router using 128-bit flow-based lookups. The Linux implementation of TRASH inherits the efficient RCU locking mechanism from the dynamic LC-trie implementation. In particular, the lookup time increases only marginally for longer keys and TRASH is highly insensitive to different types of data. The performance figures are very promising and the cache mechanism could easily be extended to serve as a unified lookup for fast socket lookup, flow logging, connection tracking and stateful networking in general.

Keywords: trie, LC-trie, hash, hashtrie, Linux, flow lookup, garbage collection.

Trita-CSC-TCS 2006:2, ISRN/KTH/CSC/TCS-2006/2-SE, ISSN 1653-7092.

1 Introduction

In this article we introduce TRASH, a general purpose data structure combining a trie with hashing (trie plus hash makes TRASH). TRASH supports fast lookup, insert and delete operations for arbitrarily long bit strings. The idea is very simple. Each key to be stored in the trie is prepended with a header; this header is a hash value based on the complete key. The extended key will, obviously, be longer. But on the other hand, the headers will be uniformly distributed (assuming the hash function is good). This type of data is perfectly suited for an LC-trie. The average and maximum depth does not depend on the length of the keys and a trie structure built from uniformly distributed data will be very well balanced.

We have implemented the scheme in the Linux kernel as a replacement for the dst cache (IPv4) and performed a full scale test on a production router using 128-bit flow-based lookups. The Linux implementation of TRASH inherits the efficient RCU locking mechanisms [11] from the dynamic LC-trie implementation and improves on the garbage collection algorithm. The lookup time increases only marginally for longer keys and TRASH is highly insensitive to different types of data. The performance figures are very promising and the cache

[*]Uppsala University, www.uu.se, `Robert.Olsson@its.uu.se`

[†]KTH, www.kth.se, `snilsson@nada.kth.se`

mechanism could easily be extended to serve as a unified lookup for fast socket lookup, flow logging, connection tracking and stateful networking in general.

The hashtrie [12] is another data structure addressing the drawbacks of a fixed hash table. It is a modification of the standard hash table implementation based on linked lists. The main idea is to replace a list that become too long by additional levels of hashing. The hash trie uses arrays of fixed size (256) and the lookup is based entirely on the hash value. The LC-trie, in comparison, uses nodes of varying size and two different modes of compression, level- and path compression. This creates additional opportunities for reducing both the memory requirements and the depth of the tree structure.

The first section of the paper gives a brief description of the LC-trie data structure. In Section 3 we review the caching mechanism used in Linux to speed up packet forwarding. In Section 4 we presents experiments comparing performance when replacing the standard hash table with TRASH in the Linux dst cache. Section 5 is an account of a full-scale test using TRASH in a large production network.

2 LC-trie

In this section we give a short review of the LC-trie data structure. For a more detailed description we refer the reader to [6] and [7].

The *trie* [4] is a general purpose data structure for storing strings. The idea is very simple: each string is represented by a leaf in a tree structure and the value of the string corresponds to the path from the root of the tree to the leaf. Consider a small example. The binary strings in Figure 1 correspond to the trie in Figure 2a. In particular, the string `010` corresponds to the path starting at the root and ending in leaf number 3: first a left-turn (`0`), then a right-turn (`1`), and finally a turn to the left (`0`). For simplicity, we will assume that the set of strings to be stored in a trie is prefix-free, no string may be a proper prefix of another string.

This simple structure is not very efficient. The number of nodes may be large and the average depth (the average length of a path from the root to a leaf) may be long. The traditional technique to overcome this problem is to use *path compression*, each internal node with only one child is removed. Of course, we have to somehow record which nodes are missing. A simple technique is to store a number, the *skip value*, in each node that indicates how many bits that have been skipped on the path. A path-compressed binary trie is sometimes referred to as a Patricia tree [5]. The path-compressed version of the trie in Figure 2a is shown in Figure 2b. The total number of nodes in a path-compressed binary trie is exactly $2n - 1$, where n is the number of leaves in the trie. The statistical properties of this trie structure are very well understood [3, 10]. For a large class of distributions path compression does not give an asymptotic reduction of the average depth. Even so, path compression is very important in practice, since it often gives a significant overall size reduction.

One might think of path compression as a way to compress the parts of the trie that are sparsely populated. *Level compression* [1] is a recently introduced technique for compressing parts of the trie that are densely populated. The idea is to replace the i highest complete levels of the binary trie with a single node of degree 2^i; this replacement is performed recursively on each subtrie. The level-compressed version, the *LC-trie*, of the trie in Figure 2b is shown in Figure 2c.

For an independent random sample with a density function that is bounded from above and below the expected average depth of an LC-trie is $\Theta(\log^* n)$, where $\log^* n$ is the iterated logarithm function, $\log^* n = 1 + \log^*(\log n)$, if $n > 1$, and $\log^* n = 0$ otherwise. For data from a Bernoulli-type process with character probabilities not all equal, the expected average depth is $\Theta(\log \log n)$ [2]. Uncompressed tries and path-compressed tries both have expected average depth $\Theta(\log n)$ for these distributions.

3 Linux packet forwarding with caching

Linux uses a caching mechanism (*dst cache*) to speed up the handling of IP packets. Packets are referenced with metadata as an *skb* (`struct sk_buff`). When a new packet arrives, a lookup in the dst cache is performed; depending on the lookup we take either a slow or fast path. The dst cache is implemented as a hash table with *dst entries*.

nbr	string
0	0000
1	0001
2	00101
3	010
4	0110
5	0111
6	100
7	101000
8	101001
9	10101
10	10110
11	10111
12	110
13	11101000
14	11101001

Figure 1: Binary strings to be stored in a trie structure.

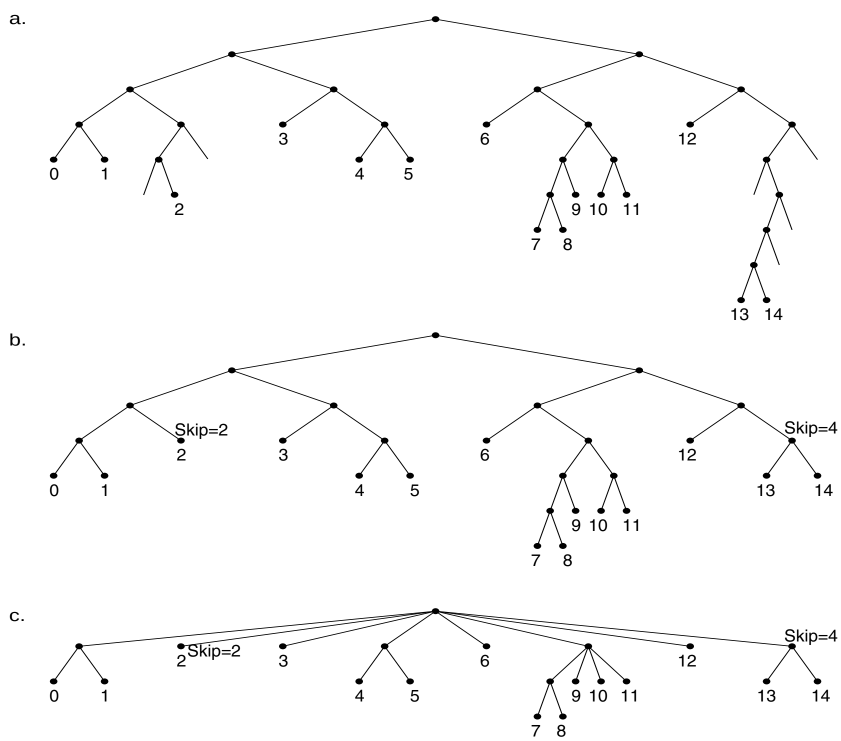

Figure 2: a. Binary trie, b. Path-compressed trie, c. LC-trie.

Figure 3: The lab setup.

A brief description of the fast path, the path taken when forwarding data is available in the dst cache, is as follows. After a successful lookup the skb is tagged with a dst entry from the cache. This entry holds information about how the packet should be further treated, e.g. if it is to be delivered locally or to be routed.

When no data is available in the cache, the packet will have to take the slower long path. After validation and routing table lookup (`fib_lookup`) the skb is tagged with a new dst entry as above and the newly created entry is stored in the dst cache.

Both paths are handled by RX softirq and thus run in the softirq context and use RCU locking. Stale entries are removed via garbage collection (GC). This is described in more detail in Section 4.4. For dynamic routing protocols it is also very important that the cache has an effective flush operation as the dst entries are invalidated when routing changes occur. In the current Linux implementation all the dst entries are invalidated. This is an area for further investigation.

3.1 Our modifications

While the dst cache used destination and source addresses as a key for the lookup, our idea was to extend the cache lookup to full flow data. Our goal was to try this out, gather experimental results and perhaps even produce usable code. The input path of the dst cache was rewritten to use an LC-trie instead of a hash table. The output path (from localhost) has not yet been converted. The dst entry is now a leaf (`struct leaf`) in the trie. The lookup is now performed by constructing a key from the incoming skb. If this key is found in the trie we may take the short path.

The key is constructed from the IPv4 source and destination addresses, source and destination port numbers, and type of protocol. This key consists of 128 bits or four 32-bit integers. We also added flow accounting and the netlink API to monitor and control both flow and cache behavior. We can also access trie statistics such as number of leaves and average and maximum tree depth. Using device statistics we are able to monitor the number of forwarded packets and estimate aggregated throughput. We also monitor `/proc/net/softnet_stat`. In particular, `time_squeeze` (the third column) tells us how many times the RX softirq has been rescheduled. The indicates the CPU load.

Garbage collection is crucial for performance. We have used a straight forward implementation. During insertion, when `gc_thresh` is reached, we try to reclaim `gc_goal`. in the experiments below we used `gc_thresh`=100,000 and `gc_goal`=100. See Section 4.4 for further details.

4 Experiments in lab setting

To verify our ideas we have implemented the TRASH data structure in the Linux kernel and performed experiments in a lab setup. This section contains a description of these experiments.

4.1 Lab setup

Figure 3 shows the lab setup. The router was a dual 1.6 GHz Opteron with two two-port Intel e1000 adapters based on 82546GB controllers at PCI-X 100MHz/133MHz. Routing and CPU affinity was set up so that incoming packets on eth0 was routed to eth1 using CPU0 and incoming packets on eth2 were routed to eth3 using CPU1. A multi-CPU system was chosen to be able to test locking and other multi-CPU aspects. The reported numbers are aggregate numbers from the two CPU's unless stated otherwise. The `fib_trie` was used for the routing lookups and the routing table had five routes.

In order to stress the lookup, insert and delete functions as well as the GC we injected very short flows: a flow-length of 2 and 2×4096 concurrent UDP flows with 64-byte packets within IP range X.0.0.0 to X.255.255.255 on eth0 and similarly on eth2, but with another IP range. The input rate was the maximum our packet generator with *pktgen* [8] could achieve with 64-byte packets: 2×160 kpps.

4.2 Unmodified LC-trie

In the first run we use an unmodified LC-trie. The key is composed of 128 bits from the IP header.

Setup	Klen	Avg. depth	Max depth	leaves	tsqueze	Tput
LC-trie src,dst	128	2.78	6	99905	56039	221

Notice the relatively high average depth and the 221 Kpps aggregated forwarding performance.

For the second run we swapped the order of the source and destination address in the key. Remember that the destination varies greatly in our setup while the source address in constant.

Setup	Klen	Avg. depth	Max depth	leaves	tsqueze	Tput
LC-trie dst,src	128	1.29	4	99997	16353	317

The performance is much enhanced. We get an increase of more than 40% in packet per second performance. The trie root node increases from 16 bits to 18 bits (not shown here), which makes for a flat tree. This is due to the fact that the first 8 bits of the key are now uniformly distributed. A big root node tree gives a flat tree and good performance.

4.3 TRASH

For the third run we used TRASH: the first 32 bits of the key is a hash value computed on the full 128-bit IP header.

Setup	Klen	Avg. depth	Max depth	leaves	tsqueze	Tput
trash	160	1.30	4	99952	23258	317

This is a big improvement. The average depth is less than half compared to the unmodified LC-trie, and the throughput increases by more than 40%. In fact, even though the key length increased to 160 bits, the tree depth is almost the same as in the optimum case. Throughput is also the same. However, the tsqueeze increased a bit indicating that we use more CPU.

4.4 Garbage collection

In general, garbage collection (GC) is totally independent from the data structure. Nevertheless, GC may have a large effect on overall performance and it needs to be carefully implemented. The main idea is to avoid too much periodic work as this has been seen to cause packet loss and performance drops in systems with high continuous loads.

One should also keep the dynamic characteristics of the TRASH algorithm in mind. It is probably a good idea to do GC incrementally to avoid frequent changes of the root node size. In other words, `gc_goal` should be kept relatively small. This optimization needs to be further studied.

4.5 Passive garbage collection

We use the term "last resort" or passive garbage collection (PGC) for the traditional scheme (See Appendix A). In short, we see the trie as a ring buffer and scan for candidates to be removed. Candidates are selected via the `rt_score` function, which is the same as the destination hash variant uses. The purpose of selecting candidates via `rt_score` is that valuable entries should not be removed. When `gc_thresh` is reached during an insert operation we try to reclaim `gc_goal` entries. This PGC process is relatively costly: afterwards (with cache misses) we have to scan the trie for GC candidates.

4.6 Active garbage collection

In addition to the GC above we have tried an even more active GC approach. Since we now do full flow lookup, and can keep stateful information cheaply, we get new opportunities for GC. For example, we can look for flow termination and do the GC directly. This is possible for TCP and maybe also with other protocols. The code used for active GC (AGC) is in Appendix B. Briefly, this is done by parsing the incoming packets for `FIN` and `ACK` and keeping stateful information in the trie (`struct leaf`). To simplify lab testing, code was added to `pktgen` to signal end of flow. This had to be done since we had no other way of testing TCP.

Here is a comparison between the two GC algorithms.

Setup	Klen	Avg. depth	Max depth	leaves	tsqueze	Tput
trash	160	1.30	4	99952	23258	317
trash+AGC	160	1.33	3	5983	27	319

First we notice that the trie only holds 5983 entries with AGC as we try to detect end of flow (ideally every flow should be removed here). The average depth is about the same, while the maximum depth is reduced by one. Throughput increases somewhat, but we see no packet drops at RX. The packet generator is not fast enough to cause packet drops. The tsqueeze value improves dramatically as the RX softirq rarely needs to be rescheduled. In summary, the AGC seems to be very efficient.

4.7 Long keys

In theory, the properties of the LC-trie are independent of key length. To test this claim in practice, we performed a simple experiment where we filled the key with multiple IPv4 addresses to get a 384-bit key. This should be long enough to hold flow data for IPv6 headers with 128-bit source and destination addresses.

Setup	Klen	Avg. depth	Max depth	leaves	tsqueze	Tput
trash+AGC	160	1.33	3	5983	27	319
trash+AGC	384	1.33	4	5915	22	321

The numbers speak for themselves. Furthers details can be found in the profile in Appendix C. Notice that `get_offset_pmtmr` shows up in the profile. This is due to the fact that our implementation also does full flow accounting with timestamps (start of flow) for logging via netlink. If we skip this verbose flow accounting, we would get even better performance. In fact, timestamps were enabled during all experiments described in this paper. Once again, we see from tsqueeze that our packet generator is not capable of pushing the router to its limit.

4.8 Comparison of hash table and trie

Next we do a comparison with the standard hash-based destination cache in the Linux kernel. The two schemes are quite different and it's hard to compare the performance as it depends on many factors, i.e. key length, distribution of data and distribution of updates, GC algorithm, hashtable size and chain length. In particular, the hash table could possibly be even better tuned to this particular workload in terms of bucket size, hash-chain length and GC. We used 32768, 131072 buckets and `gc_elasticity` of 4 and 8. An important advantage of the TRASH data structure is that it doesn't require any explicit tuning.

A profile is found in Appendix D.

Setup	Klen	Avg. depth	Max depth	leaves	tsqueze	Tput
trash	160	1.30	4	99952	23258	317
hash	-	-	-	-	2108	308

From the profile we see, not surprisingly, that we spend most of the time in `ip_route_input`. This is where the hash lookup occurs. Even though it's difficult to directly compare the performance of the two schemes, we note that the difference in throughput is small.

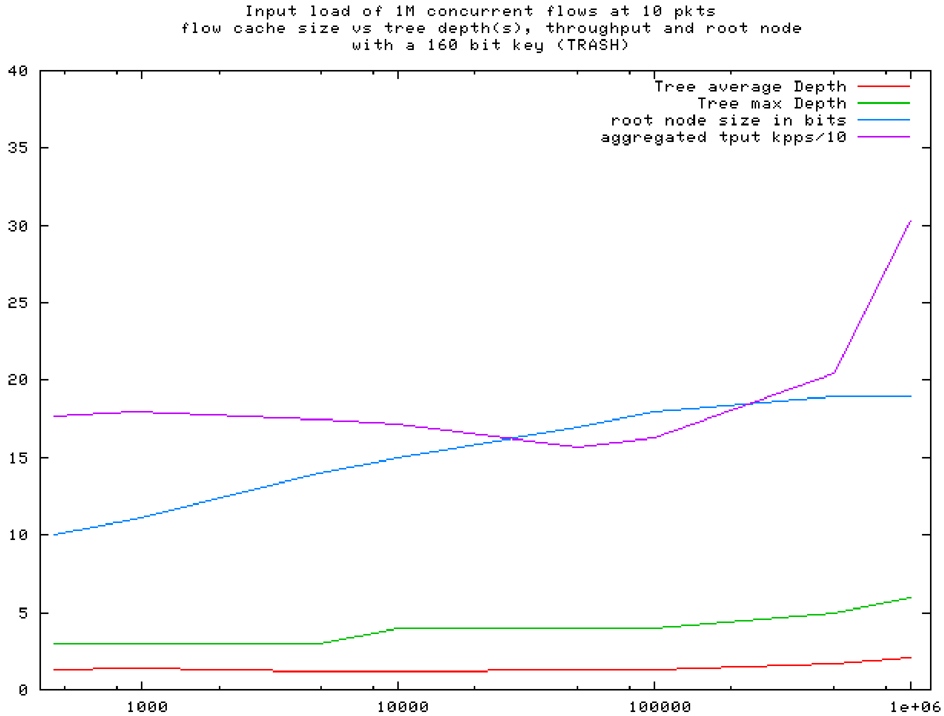

Figure 4: Input of 1M concurrent flows each consisting of 10 packets, 160-bit key cache implemented with TRASH.

4.9 Single-flow comparison of hash table and trie

Finally, we look into the single-flow performance. We use the same lab setup, but inject traffic with constant source and destination addresses. Just as before, two flows are injected to eth0 and eth2 using different CPU's and aggregate numbers are reported.

Setup	Klen	Avg. depth	Max depth	leaves	tsqueze	Tput
hash	-	-	-	-	54429	1308
trash+AGC	160	1.00	1	2	51210	1218
trash+AGC	384	1.00	1	2	47506	1193

We see that the overall throughput is higher and that the hash table cache performs better here. Also notice that there is a small penalty for handling longer keys. The profiles in Appendix F (hash) and Appendix G (384-bit trie) indicate that the difference is due to key handling. The trash implementation has not yet been reviewed and improved by other researchers and developers so there might be room for improvement.

4.10 Large number of flows

In this experiment we use the same hardware setup but inject 2×524288 concurrent flows, each consisting of 10 packets, i.e. more than 1 Mflows. The size of the flow cache was varied with `gc_thresh` and tree data and throughput was collected. The result is presented in Figure 4.

Again we see the average depth (red) is amazingly low and nearly constant ranging from a few hundred flows to one million flows. This is also true for the maximum depth (green). The size of the root node (blue) does not increase above 19 bits. This is due to an upper limit in the code. To conserve memory the root node is never allowed to have more than 2^{19} entries. When we reach this limit the tree depth starts to grow as can be seen in the graph.

Also, notice that the throughput (purple) goes up when the cache size gets close to the number of concurrent flows. If the cache is too small we always get cache misses, just like a DoS attack, but when the size of the flow cache gets close to the number of concurrent flows we see a dramatic improvement in performance.

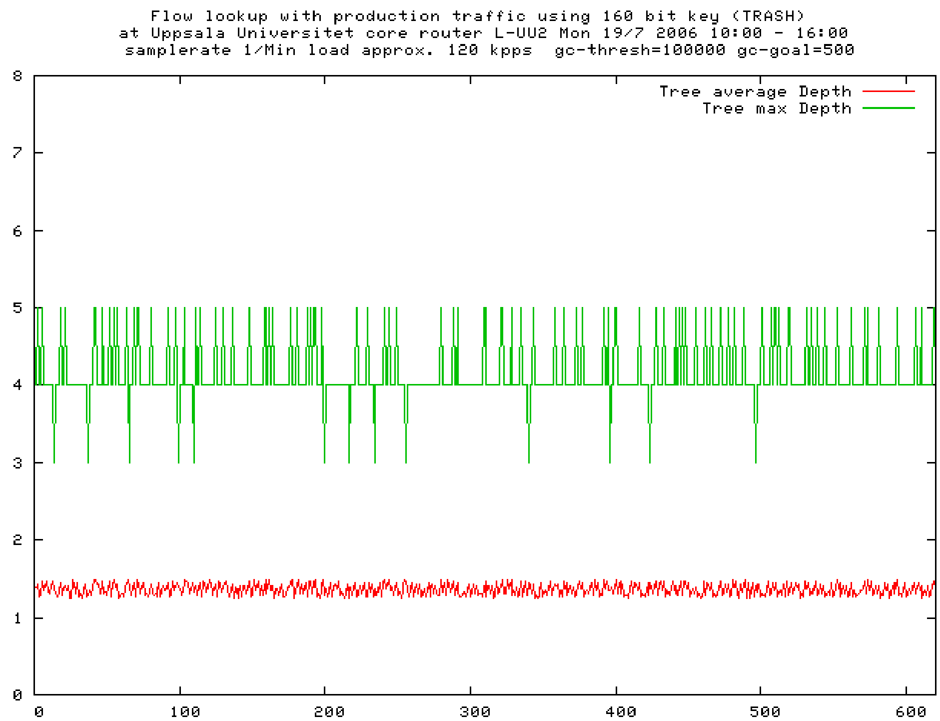

Figure 5: Flow lookup with production traffic using 160-bit key (TRASH) at Uppsala University core router L-UU2, Mon 19/7 2006 10:00-16:00, sample rate 1/min, load approx. 120 kpps, `gc_threshold` = 100000, `gc_goal` = 100.

4.11 Discussion

The comparison in Section 4.8 shows that the hash table and trie has very similar performance. This is good news. In fact, it's hard to imagine a more efficient data structure than a perfectly sized and perfectly distributed hash table. The advantage of the LC-trie comes from the dynamic implementation described in [7]. This means that we can expect close to optimum performance regardless of size, as the trie is continuously rebalanced. In particular, we don't have to reserve, and perhaps waste, memory in advance. In addition, with the trash technique described in this paper we have a simple and very straightforward way of constructing well-balanced trees that are highly insensitive to input data. The average number of memory accesses per lookup is less than two and the worst-case behavior is also well behaved.

5 Full scale test

After carefully testing the new implementation with 24 hours of continuous denial of services we mustered the courage to deploy the trie cache in a live production network. We were particularly interested in trie statistics and profiling data.

After discussions with Uppsala University (UU) network manager Hans Wassen, who had overseen this work, it was decided to test a Linux kernel with TRASH-based lookup in one of the UU core routers. UU is a pioneer in Linux routing and has used Linux routers for a period of seven years in a high-speed production environment. UU has dual BGP-peering at GIGE speed with the upstream provider SUNET. Furthermore, UU has BGP peering with several other organizations and companies in the same Linux router. In total there were 191052 routes in the kernel. With the dual access connection, all traffic could be forced to use only one of the two routers. BGP MED metric to the upstream provider controls the incoming traffic and default route metric for the internal routers changes the outgoing traffic. This enabled us to install a new Linux kernel with TRASH lookup. By changing these routing protocol metrics, we could force traffic to use this router, using the other router as a backup.

After the switch-over we monitored kernel variables such as memory leaks and, if needed, could have switched back. In a worst case scenario we could have had a break in traffic until BGP timers expired. Luckily no problems were seen and the test was run for a period of four days. Traffic rates were about 100-130 kpps and approximately 250 Mbit/s. The router hardware is based on a Dual Opteron running at 2.6 GHz with two Intel 82546 Dual adapters. The Linux-kernel is based on Dave Miller's git tree 2.6.17-rc1 with TRASH patches for flow lookup and GC. Various data was collected from the router. A time plot of average and maximum tree depth is shown in Figure 5. The number of entries in the trie was held around 100k (`gc_thresh`).

From the data we can see that the tree depth is amazingly constant and very low, around 1.3-1.4. Recall that this router carries IP-traffic for thousands of users. The worst-case behavior is also very good; the maximum depth over time is 5 in the data above with a normal case of 4. A profile is available in Appendix E. Unfortunately the driver and netfilter code was not profiled. From the profile we see that the router is lightly loaded as `netif_rx_schedule` is high on the list. The profile contains the expected functions and the lookup and GC functions are behaving nicely.

6 Conclusions

We have shown that it is possible to implement flow-based lookup with long keys, keeping stateful information for a large number of flows. The TRASH technique substantially improves the worst-case behavior of lookups as demonstrated in the experiments. The behavior seen in the experiments is quite robust since the algorithm uses randomization. A degradation of the performance is only likely in case of a DOS attack engineered to fool the hash function and the LC-trie balancing mechanism. If necessary, it's possible to guard against this by using a high-grade cryptographic hash function.

We have also discussed how to perform garbage collection based on flow information. Of course, this technique is by no means limited to Linux or this particular application. On the contrary, the LC-trie and TRASH ideas described here are very general and could be used in many applications.

7 Acknowledgements

We would like to thank Alexey Kuznetsov, David Miller and Jens Låås for indispensable discussions in preparing this report.

References

[1] Arne Andersson and Stefan Nilsson. Improved behaviour of tries by adaptive branching. *Information Processing Letters*, 46(6):295–300, 1993.

[2] Arne Andersson and Stefan Nilsson. Faster searching in tries and quadtrees – an analysis of level compression. In *Proceedings of the Second Annual European Symposium on Algorithms*, pages 82–93, 1994. LNCS 855.

[3] Luc Devroye. A note on the average depth of tries. *Computing*, 28(4):367–371, 1982.

[4] Edward Fredkin. Trie memory. *Communications of the ACM*, 3:490–500, 1960.

[5] Gaston H. Gonnet and Ricardo A. Baeza-Yates. *Handbook of Algorithms and Data Structures*. Addison-Wesley, second edition, 1991.

[6] Stefan Nilsson and Gunnar Karlsson. IP routing with LC-tries. *IEEE Journal on Selected Areas in Communications*, 17(6):1083–1092, 1999.

[7] Stefan Nilsson and Matti Tikkanen. An experimental study of compression methods for dynamic tries, 1998. Submitted to Algorithmica.

[8] Robert Olsson. pktgen the linux packet generator. In *Proceedings of the Linux Symposium, Ottawa, Canada*, volume 2, pages 11–24, 2005.

[9] Robert Olsson, Jens Låås, and Hans Liss. LC-trie implementation in linux. Linux v2.6.16.27, net/ipv4/fib_trie.c.

[10] Bonita Rais, Philippe Jacquet, and Wojciech Szpankowski. Limiting distribution for the depth in Patricia tries. *SIAM Journal on Discrete Mathematics*, 6(2):197–213, 1993.

[11] `http://en.wikipedia.org/wiki/RCU`.

[12] SoftComplete Development, `http://www.softcomplete.com`.

A GC

```
static int unicache_garbage_collect(struct trie *t)
{
    struct leaf *l, *ll = NULL, *cand;
    int i, j;
    int goal;
    int old_size = t->size;
    u32 min_score;

    if(t->size < t->gc_thresh)
        return 0;

    l = t->nleaf;
    goal = t->gc_goal;

    for (i = 0; i < goal; i++) {
        min_score = ~(u32)0;
        cand = NULL;

        for (j = 0; j < 4; j++) {
            l = trie_nextleaf(t, l);

            if(!l)
                l = trie_nextleaf(t, NULL); /* Starting over */

            if(ll) {
                struct dst_entry *dst = (struct dst_entry *) ll->obj;

                if (!atomic_read(&dst->__refcnt)) {

                    u32 score = rt_score((struct rtable *)&dst);

                    if (score <= min_score) {
                        cand = ll;
                        min_score = score;
                    }
                }
            }
            ll = l;
        }
        if(cand == ll)
            ll = trie_nextleaf(t, ll);

        trie_remove(t, cand, &unicache_ops);
    }
    t->nleaf = ll;
    return old_size - t->size;
}
```

B AGC

State machinery used for snooping and doing GC based on TCP FIN. In the code snippet below the variable `state` holds TCP state bits for the current packet. The pointer `l` (`struct leaf`) is the trie result node and keeps the stateful information (as well as the dst entry).

```
/*
 *  1)       A -----> FIN+ACK B  | CLOSE_WAIT  Flow AB
 *  2)       A FIN+ACK <----- B  | LAST_ACK    Flow BA
 *  3)       A ---------> ACK B  | CLOSED      Flow AB
 */

if(state & UNICACHE_TCP_FIN_RX)
    l->state |= UNICACHE_TCP_FIN_RX;
else {
    if(state & UNICACHE_TCP_ACK && l->state & UNICACHE_TCP_FIN_RX) {

        unicache_create_key_ipv4_reverse(key, l->key);

        trie_remove_by_key(t_unicache, l->key, &unicache_ops);

        /*
         * We're at #3 for Flow AB. We can remove Flow BA too
         */
        trie_remove_by_key(t_unicache, key, &unicache_ops);
    }
}
```

C Profile for long keys

Profile for TRASH with active GC (AGC) and 384-bit key.

```
Cpu speed was (MHz estimation) : 1598.9
Counter 0 counted CPU_CLK_UNHALTED events
vma      samples  %          symbol name
c02c5f5d 2643     7.61781    tkey_extract_bits
c010b43f 2636     7.59764    get_offset_pmtmr
c021df18 2428     6.99813    e1000_clean_rx_irq
c030847a 1656     4.77302    fn_trie_lookup
c02c6d6d 1196     3.44718    trie_lookup
c01476a2 1062     3.06096    kmem_cache_alloc
c021c8ef 1061     3.05808    e1000_xmit_frame
c02e2628 964      2.7785     ip_output
c02dcdb5 926      2.66897    ip_route_input_slow
c02df8ef 801      2.30869    ip_rcv
c02e0d8c 787      2.26834    ip_forward
c02dd8b4 780      2.24816    ip_route_input
c01478f3 717      2.06658    kfree
c02c9002 666      1.91958    netif_receive_skb
c02c21a6 591      1.70342    __alloc_skb
c01478b9 589      1.69765    kmem_cache_free
c034d227 588      1.69477    _spin_lock
c01477a0 582      1.67748    __kmalloc
c02c62c8 580      1.67171    resize
c02c8bfb 533      1.53624    dev_queue_xmit
c02d2b93 530      1.5276     qdisc_restart
c02c6e6e 503      1.44978    trie_insert
c030a338 499      1.43825    fib_lookup
c02d278c 494      1.42384    eth_header
c02d4cac 467      1.34602    qdisc_dequeue_head
c021dd43 461      1.32872    e1000_clean_tx_irqc
```

D Profile for hash table

Profile for Linux ordinary destination hash lookup with 131072 hash buckets and `gc_thresh`=8.

```
Cpu speed was (MHz estimation) : 1598.9
Counter 0 counted CPU_CLK_UNHALTED events
vma      samples  %          symbol name
c02db5b6 3913     13.5286    ip_route_input
c021df18 2023     6.99419    e1000_clean_rx_irq
c0305e9a 1805     6.24049    fn_trie_lookup
c02de7ac 1147     3.96556    ip_forward
c01b937e 975      3.3709     memcmp
c02dd30f 972      3.36053    ip_rcv
c02e0048 937      3.23952    ip_output
c02dad97 934      3.22915    ip_route_input_slow
c02d9861 884      3.05629    rt_intern_hash
c01478f3 877      3.03208    kfree
c021c8ef 851      2.94219    e1000_xmit_frame
c01478b9 750      2.593      kmem_cache_free
c01476a2 695      2.40285    kmem_cache_alloc
c02c711a 608      2.10206    netif_receive_skb
c02c21a6 586      2.026      __alloc_skb
c02ca14d 544      1.88079    dst_alloc
c021dd43 520      1.79781    e1000_clean_tx_irq
c01477a0 479      1.65606    __kmalloc
c02d0cab 460      1.59037    qdisc_restart
c01ba6ef 457      1.58       memset
c0147240 441      1.52469    slab_put_obj
c02ca259 429      1.4832     dst_destroy
c0307d48 422      1.459      fib_lookup
c02d2dc4 405      1.40022    qdisc_dequeue_head
c02d08a4 365      1.26193    eth_header
c02c69f6 323      1.11672    __netif_rx_schedule
```

E Profile for production router

Profile from Uppsala University production router running at 120 kpps. Driver and netfilter was not profiled.

```
Cpu speed was (MHz estimation) : 2588.72
Counter 0 counted CPU_CLK_UNHALTED events
vma      samples  %          symbol name
c02b948e 1285     12.4275    __netif_rx_schedule
c014786f 796      7.69826    kfree
c02b9f0c 592      5.72534    net_rx_action
c011be14 509      4.92263    __do_softirq
c02ce37c 432      4.17795    ip_route_input
c02b3296 388      3.75242    kfree_skb
c02ce2fb 352      3.40426    unicache_hash_code
c02b6c85 341      3.29787    tkey_extract_bits
c02b7a95 337      3.25919    trie_lookup
c02b31bb 325      3.14313    __kfree_skb
c02b2ec6 310      2.99807    __alloc_skb
c02b9bb2 298      2.88201    netif_receive_skb
c02b30c4 286      2.76596    skb_release_data
c02c332a 282      2.72727    eth_type_trans
c02d30c8 262      2.53385    ip_output
c02c55a3 253      2.44681    pfifo_enqueue
c02c35f7 252      2.43714    qdisc_restart
c02d1828 237      2.29207    ip_forward
c02d0383 178      1.72147    ip_rcv
c02b97ab 161      1.55706    dev_queue_xmit
c010b45f 154      1.48936    get_offset_pmtmr
c0100afc 145      1.40232    default_idle
c02ca7ff 141      1.36364    nf_iterate
c014771c 139      1.34429    __kmalloc
c02c5714 121      1.17021    qdisc_dequeue_head
c014761e 118      1.1412     kmem_cache_alloc.
```

F Profile for single-flow with hash table

Profile of single-flow forwarding with destination hash table.

```
Cpu speed was (MHz estimation) : 1599.56
Counter 0 counted CPU_CLK_UNHALTED
vma      samples  %           symbol name
c021df18 8698     19.4265     e1000_clean_rx_irq
c021c8ef 3408     7.61156     e1000_xmit_frame
c02e0048 2916     6.51271     ip_output
c02c21a6 2332     5.20838     __alloc_skb
c02c711a 2311     5.16148     netif_receive_skb
c01478f3 2148     4.79743     kfree
c02de7ac 2071     4.62545     ip_forward
c02dd30f 1941     4.33511     ip_rcv
c01477a0 1888     4.21673     __kmalloc
c01476a2 1888     4.21673     kmem_cache_alloc
c02d2dc4 1757     3.92415     qdisc_dequeue_head
c02db5b6 1616     3.60924     ip_route_input
c021dd43 1287     2.87444     e1000_clean_tx_irq
c02c6d13 1030     2.30044     dev_queue_xmit
c02d0cab 991      2.21334     qdisc_restart
c02c2576 828      1.84929     kfree_skb
c01478b9 775      1.73092     kmem_cache_free
c021e929 705      1.57457     e1000_alloc_rx_buffers
c02d2c53 592      1.3222      pfifo_enqueue
c02d09de 561      1.25296     eth_type_trans
c034ac65 500      1.11672     _spin_unlock_irqrestore
c02d8eb4 474      1.05865     rt_hash_code
c02d08a4 453      1.01175     eth_header
c011bf43 445      0.99388     local_bh_enable
c02c7092 415      0.926877    ing_filter
c02c6a6d 408      0.911243    dev_kfree_skb_any
```

G Profile for single-flow with trie

Profile of single-flow forwarding with destination trie. Key length 384 bits.

```
Cpu speed was (MHz estimation) : 1599.82
Counter 0 counted CPU_CLK_UNHALTED
vma samples       %           symbol name
c021df18 7400     17.3108     e1000_clean_rx_irq
c021c8ef 3174     7.42491     e1000_xmit_frame
c02e2628 2691     6.29503     ip_output
c02c5f5d 1996     4.66922     tkey_extract_bits
c02c9002 1973     4.61542     netif_receive_skb
c01477a0 1957     4.57799     __kmalloc
c02c21a6 1945     4.54992     __alloc_skb
c02df8ef 1861     4.35342     ip_rcv
c02e0d8c 1783     4.17096     ip_forward
c01476a2 1679     3.92767     kmem_cache_alloc
c02d4cac 1641     3.83878     qdisc_dequeue_head
c01478f3 1538     3.59783     kfree
c02c6d6d 1393     3.25863     trie_lookup
c02dd8b4 1230     2.87733     ip_route_input
c021dd43 1137     2.65977     e1000_clean_tx_irq
c02d2b93 1051     2.45859     qdisc_restart
c02c8bfb 863      2.01881     dev_queue_xmit
c01478b9 670      1.56732     kmem_cache_free
c02c2576 651      1.52288     kfree_skb
c021e929 644      1.5065      e1000_alloc_rx_buffers
c02d278c 607      1.41995     eth_header
c02d28c6 546      1.27725     eth_type_trans
c011bf43 485      1.13456     local_bh_enable
c02d4b3b 476      1.1135      pfifo_enqueue
c034d255 426      0.996538    _spin_unlock_irqrestore
c02dd833 391      0.914663    unicache_hash_code
```

H Summary of experiments

Table summarizing all experiments.

RdoS

Setup	Klen	Avg. depth	Max depth	leaves	tsqueze	Tput
LC-trie src,dst	128	2.78	6	99905	56039	221
LC-trie dst,src	128	1.29	4	99997	16353	317
trash	160	1.30	4	99952	23258	317
trash+AGC	160	1.33	3	5983	27	319
trash	352	1.30	4	99950	62680	271
trash+AGC	384	1.33	4	5915	22	321
hash	-	-	-	-	2108	308

Note that in runs with active garbage collection (AGC) we are not able to saturate the RX softirq (see tsqueeze), consequently we expect higher packet rates with a faster sender. Therefore the table is somewhat biased.

Single flow

Setup	Klen	Avg. depth	Max depth	leaves	tsqueze	Tput
hash	-	-	-	-	54429	1308
trash+AGC	160	1.00	1	2	51210	1218
trash+AGC	384	1.00	1	2	47506	1193