Hashtrie

An early experiment

Martin Josefsson <gandalf@wlug.westbo.se> Netfilter Workshop 2005 (051005)

What is it?

- It's a not very known datastructure, think of it as a tree of multiple hashtables.
- It's more complicated than a regular hashtable but nicer to caches and with todays machines where cpus are so much faster than memory it's a nice feature.

Do we need it?

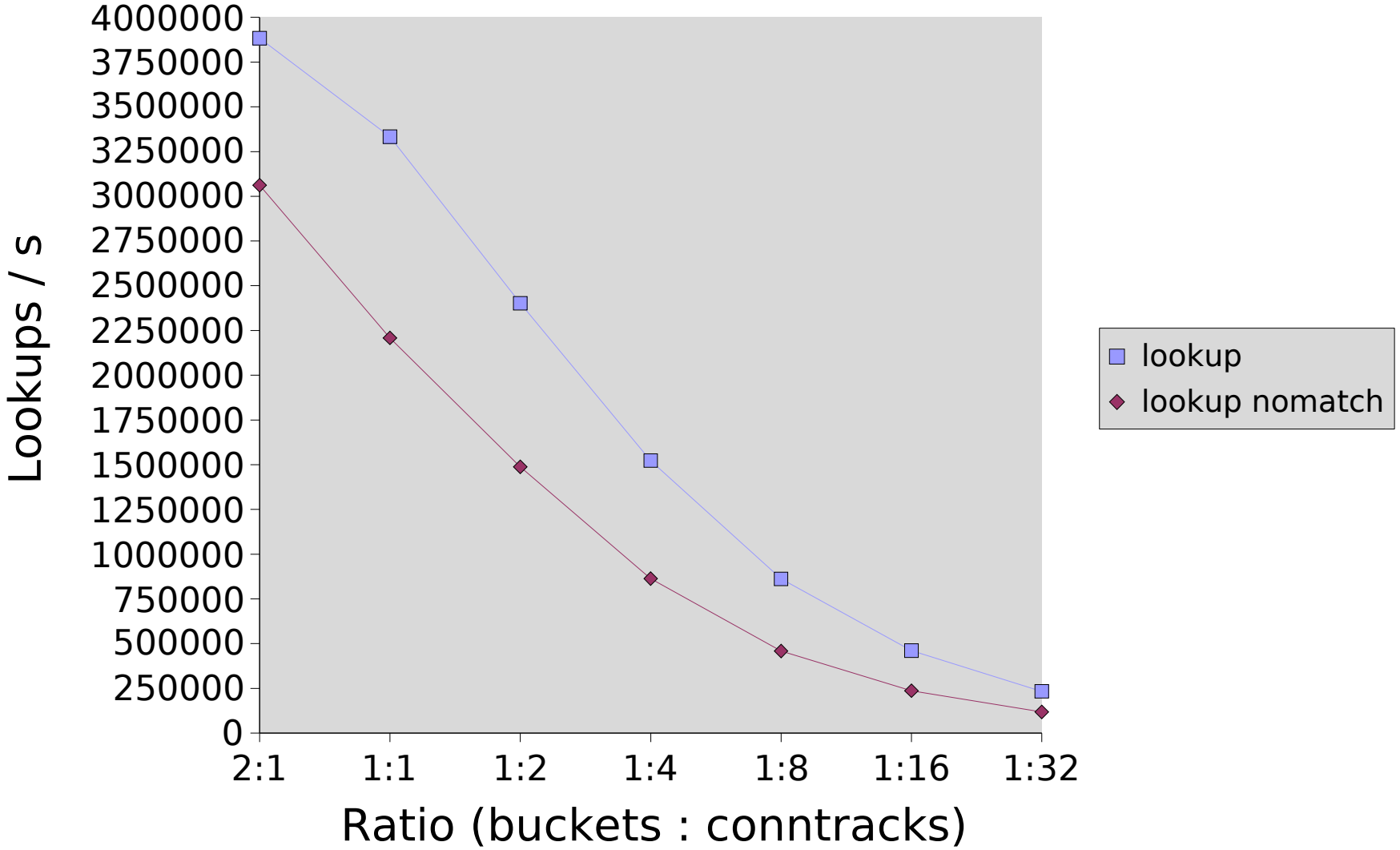
- What is the problem with current hashtable?
- What about performance?
- Many questions, no real answers, the hashtrie is still in development.

Current hashtable situation

- Current conntrack uses a regular hashtable.
- It's performance is good when sized properly.
- But when improperly sized its performance is awful.
- Generally users don't understand that they have to increase the size of the hashtable in addition to increasing the number of buckets, this also goes for network admins.

Data

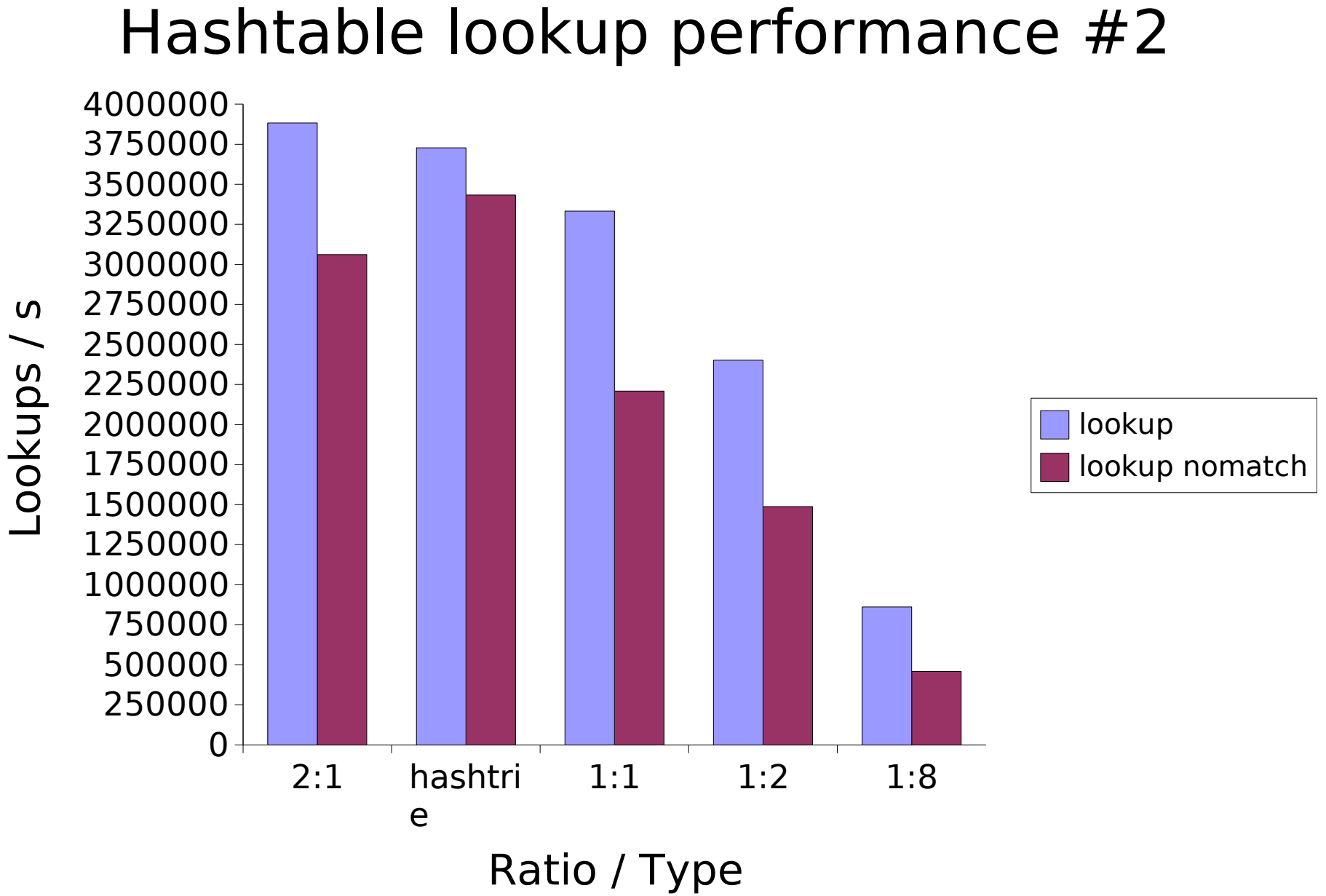
Hashtable lookup performance



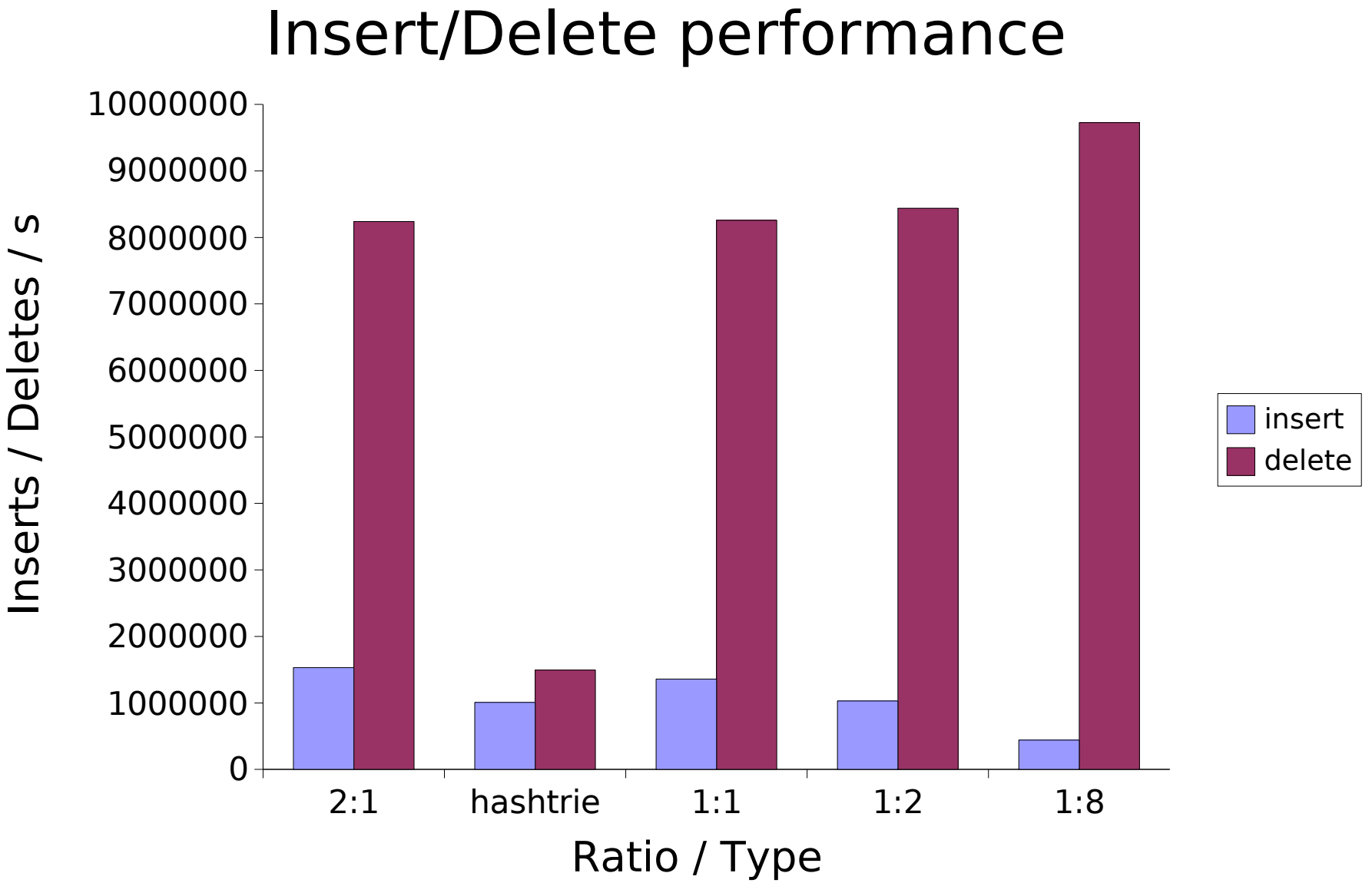
Hashtable performance

- The default configuration is the 1:8 ratio of buckets:conntracks.
- Each conntrack results in two entries in the hashtable, one for each tuple (direction). This gives an average of 16 entries per bucket when the ratio is 1:8
- Increasing the number of buckets, thus lowering the ratio, will increase the performance greatly for lookups. But it also adds other problems.

Data #2


Hashtable lookup performance #2
Lookups / s
4000000
3750000
3500000
3250000
3000000
2750000
2500000
2250000
2000000
1750000
1500000
1250000
1000000
750000
500000
250000
0
2:1
hashtrie
1:1
1:2
1:8
Ratio / Type
lookup
lookup nomatch

Data #3


Insert/Delete performance
Inserts / Deletes / s
10000000
9000000
8000000
7000000
6000000
5000000
4000000
3000000
2000000
1000000
0
2:1
hashtrie
1:1
1:2
1:8
Ratio / Type
insert
delete

Details

```
struct hashentry {
   struct hashentry           *child;
   u8                         counter;
   hashbits_t                 hashbits[NUMENTRY];
   struct ip_conntrack_tuple  *members[NUMENTRY];
   unsigned char              filler[PADNUM];
} __attribute__((packed));

hashtable = malloc(NUM * sizeof(struct hashentry));
bucketnr = hash & (NUM - 1);
bucket = &hashtable[bucketnr];
```

Details #2

When a bucket gets full we expand from that bucket into a new hashtable that is identical to the toplevel hashtable.

```
bucket->child = (void *)malloc(NUM * sizeof(struct hashentry));
```

Then we add the new entry to the new hashtable just as we did with the toplevel hashtable, the only diffrence is which bits in the hashvalue we use.

End

Code in development can be found at:

http://people.netfilter.org/gandalf/graht/