Percpu hash table traversal

Measurement study

Presenter: Brian Vazquez

Contributors: Sagarika Sharma, Luigi Rizzo, Stanislav Fomichev

Google

Agenda

- **Motivation:** Traversing bpf pcpu htab is expensive
- **Measurement:** Where is the cost?
- **Experiments:** Result and interpretation
- **Conclusion:** Request for comment

Motivation: traversing a bpf pcpu htab is expensive

Motivation: problem context

- Demon running that **tracks every flow** in the host to shape traffic
- The information is saved in a **bpf percpu hashtab map**
- The table can be **as large as 40K entries**
- The FlowStats table is dumped in userspace for processing **every 5s**

Motivation: the past pitfall

- Not the first time, realizing that traversing a percpu hashtab is **expensive**
  - Remember the batch ops implementation?

https://patchwork.ozlabs.org/project/netdev/patch/20200115184308.162644-6-brianvv@google.com/

- Turns out it **saved about 20% of the total cost** back then (only saving the syscall overhead, but not more)

Motivation: traversing a bpf pcpu htab is expensive

- bpf_map_lookup_batch is **expensive!!!**
  - **.63s** of processing time every **5s**

Measurement: where is the cost?

Measurement: profiling the traversal

Function name	Cycle percentage
DumpStatsBatach (userspace code)	100%
bpf_map_lookup_batch	73%
bpf_long_memcpy	62%

Measurement: bpf_long_memcpy

- Is bpf_long_memcpy suboptimal?
  - Not really, so why it is expensive?

```
static inline void bpf_long_memcpy(void *dst, const void *src, u32 size)
{

      const long *lsrc = src;
      long *ldst = dst;
      size /= sizeof(long);
      while (size--)
            *ldst++ = *lsrc++;
}
```

Measurement: bpf_long_memcpy

Almost all the time is spent in **cache misses** when reading from the per-cpu map values!!!

CPU_LLC_LOAD_MISS  SQL Total [M·accesses] ||| %	pc	Disassembly	Inlined frames
850.0	87.18%	0xffffffff8102adcc	movq (%rdx,%rbx), %rsi	bpf_long_memcpy @ include/linux/bpf.h : 1606

Measurement: using kstats github.com/luigirizzo/lr-cstats

```
struct kstats *ks_memcpy_ns;
void bpf_debug_init(void)
{
        ks_memcpy_ns = kstats_new("memcpy_ns", 3);
}
...
static int __htab_map_lookup_and_delete_batch (...)
{
        ...
        t0 = ktime_get_ns();
        num_elem = 0;
        hlist_nulls_for_each_entry_safe(l, n, head, hash_node) {
                num_elem++;
                memcpy(dst_key, l->key, key_size);
                if (is_percpu) {
                        int off = 0, cpu;
                        void __percpu *pptr;
                        int num_cpus = num_possible_cpus();
                        pptr = htab_elem_get_ptr(l, map->key_size);
                        for_each_possible_cpu(cpu) {
                                bpf_long_memcpy(dst_val + off,
        per_cpu_ptr(pptr, cpu), size);
                                off += size;[
                        }
                        ...
                }
                if (num_elem)
                kstats_record(ks_memcpy_ns, (ktime_get_ns() -
                t0)/num_elem);
                ...
```

```
extern void bpf_debug_init(void);
static int __init bpf_syscall_sysctl_init(void)
{
        register_sysctl_init("kernel", bpf_syscall_table);
        bpf_debug_init();
        return 0;
}
```

Measurement: using kstats (2nd approach)

```
struct kstats *ks_memcpy_ns;
void bpf_debug_init(void)
{
      ks_memcpy_ns = kstats_new("memcpy_ns", 3);
}
...
static int __htab_map_lookup_and_delete_batch (...)
{
      ...

      hlist_nulls_for_each_entry_safe(l, n, head, hash_node) {
            num_elem++;
            memcpy(dst_key, l->key, key_size);
            if (is_percpu) {
                  int off = 0, cpu;
                  void __percpu *pptr;
                  int num_cpus = num_possible_cpus();
                  pptr = htab_elem_get_ptr(l, map->key_size);
                  t0 = ktime_get_ns();
                  for_each_possible_cpu(cpu) {
                        bpf_long_memcpy(dst_val + off, per_cpu_ptr(pptr, cpu), size);
                        off += size;[
                  }
            kstats_record(ks_memcpy_ns, ktime_get_ns() - t0);
                  ...
            }
            ...
```

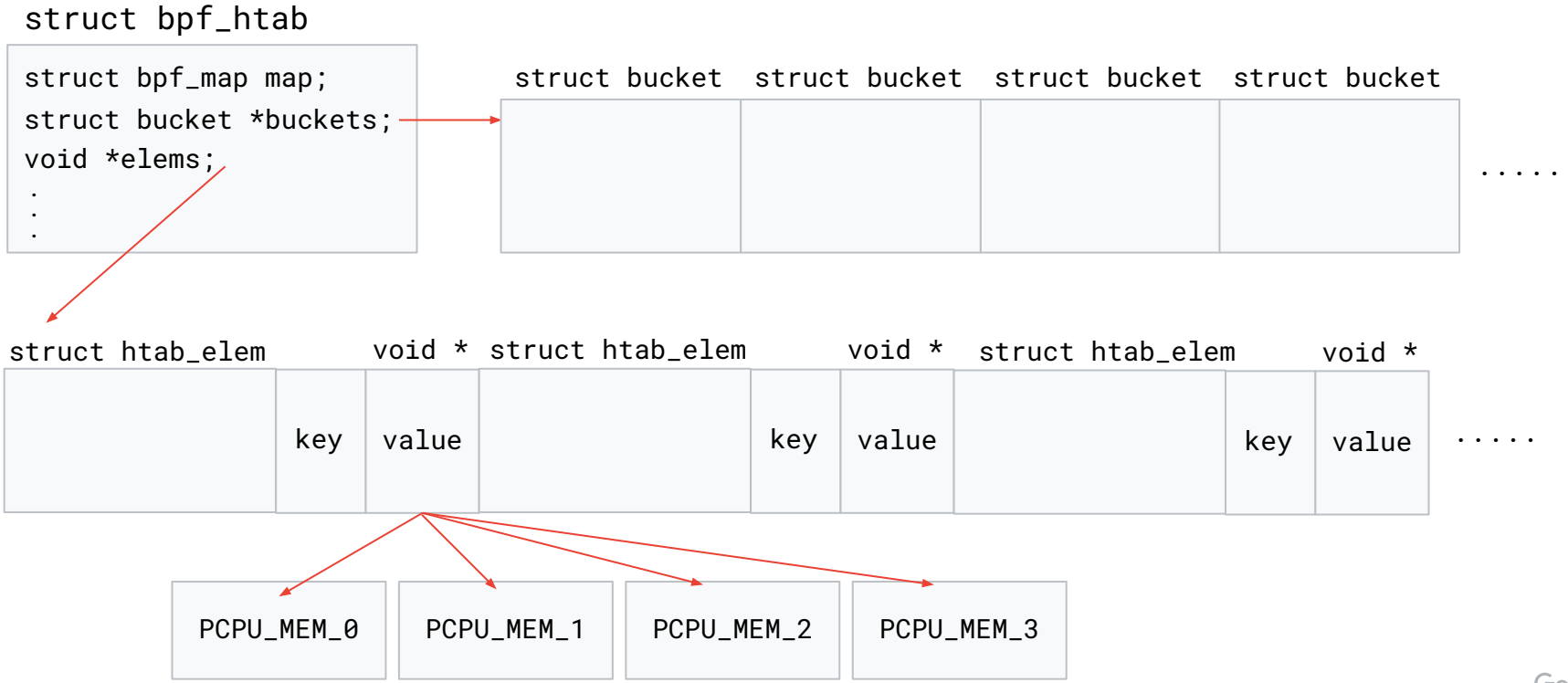
Measurement: kstats results

Percentile	AMD Rome, 256 cores bucket avg (memcpy_ns)	Intel skylake 112 cores bucket avg(memcpy_ns)
10th percentile	15062	2689
25th percentile	15829	3972
50th percentile	**15829**	**4890**
90th percentile	16816	5344
95th percentile	16816	5835
99.9 percentile	**25077**	**19678**

Percentile	AMD Rome, 256 cores bucket avg (memcpy_ns)	Intel skylake 112 cores bucket avg(memcpy_ns)
10th percentile	13852	2946
25th percentile	15089	3971
50th percentile	**15460**	**4900**
90th percentile	16302	5346
95th percentile	16902	5830
99.9 percentile	**20994**	**15834**

Experiments: results and interpretation

Context: pcpu htab data structure

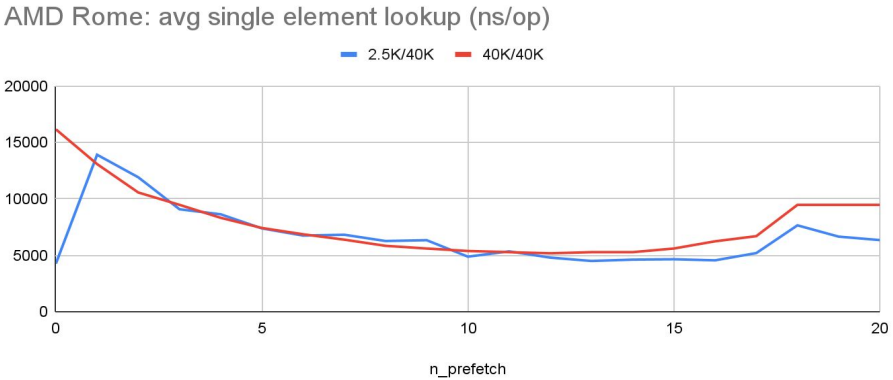


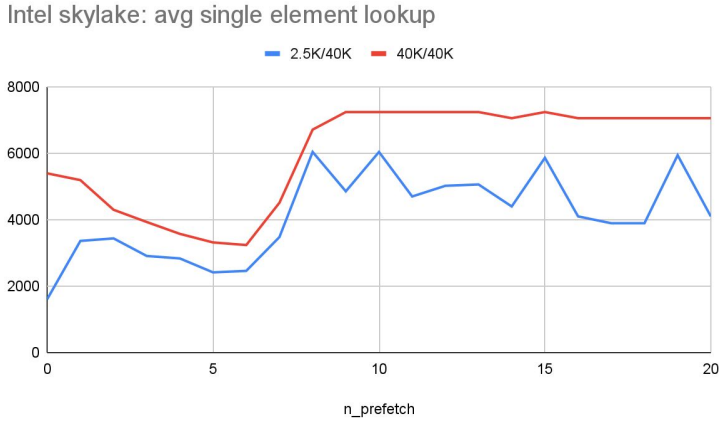
Optimization: prefetch to the rescue!?

```
static int __htab_map_lookup_and_delete_batch (struct bpf_map *map ...)
{
      ...
      void * dst_key;
      struct hlist_nulls_head *head;
      struct hlist_nulls_node *n;
      struct htab_elem *l;
      ...
      hlist_nulls_for_each_entry_safe(l, n, head, hash_node) {
            memcpy(dst_key, l->key, key_size);
            if (is_percpu) {
                  int off = 0, cpu;
                  void __percpu *pptr;
                  int num_cpus = num_possible_cpus();

                  pptr = htab_elem_get_ptr(l, map->key_size);

                  for_each_possible_cpu(cpu) {
                        if (n_prefetch > 0 && (cpu + n_prefetch) <= num_cpus)
                              prefetch(per_cpu_ptr(pptr, cpu + n_prefetch))

                        bpf_long_memcpy(dst_val + off, per_cpu_ptr(pptr, cpu), size);
                        off += size;
                  }
                  ...
            }
            ...
```

Optimization: Conclusion

- Setting **n_prefetch = 10** reduces average single entry lookup on hashmaps with 40k/40k entries full by at least **50%** on AMD platform but doesn't seem Intel benefits from it.
- Experimental results: when looking up fewer entries, **prefetching introduces a time cost.**





Conclusion: Request for comment

Conclusion

- Traversing per-cpu data structures will become **more expensive as platform grow in cpu count.**

- **Current hashtable implementation** (pointer chasing and non-contiguous memory allocation) **doesn't allow to exploit any traversal trick**.

Alternatives

Switching to bpf_iter to save aggregation helps, but bpf_iter **suffers on cache misses** when traversing pcpu elem

```
static int __bpf_hash_map_seq_show(struct seq_file *seq, struct htab_elem *elem)
{
      ...
       if (prog) {
              ...
                          roundup_value_size = round_up(map->value_size, 8);
                          pptr = htab_elem_get_ptr(elem, map->key_size);
                          for_each_possible_cpu(cpu) {
                                bpf_long_memcpy(info->percpu_value_buf + off,
                                            per_cpu_ptr(pptr, cpu),
                                            roundup_value_size);
                                off += roundup_value_size;
                          }
                    ...
              }
...
}
```

Alternatives

- Implement another data structure? Instead of percpu htab, a **per networking queue** htab?
  - Cpus seems to go up and up
  - Does tc-bpf run with networking queue lock held? Could it be used in our favor?

Questions/Comments?