Right-sizing is hard, Resizable maps for optimal map size

John Fastabend, Cilium.io

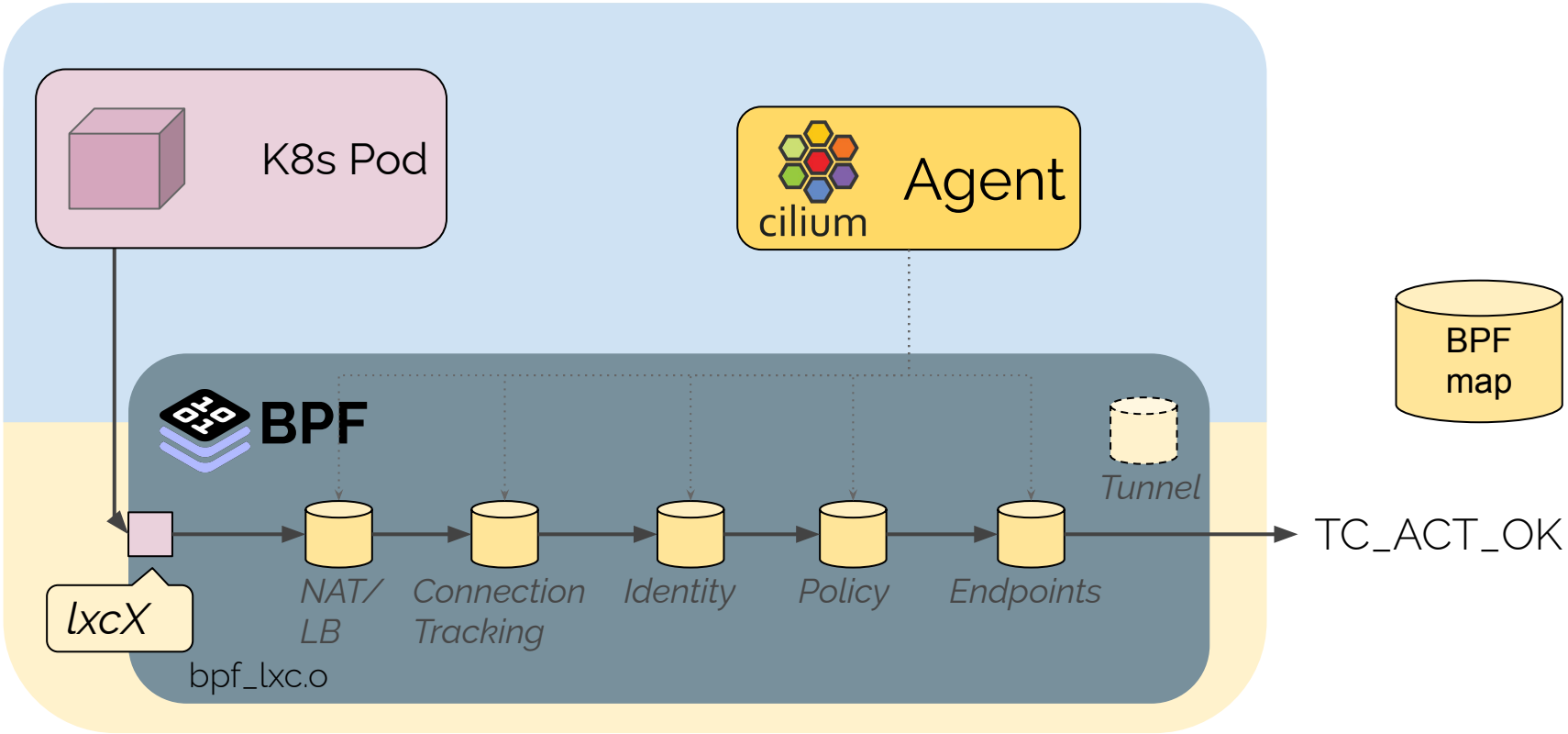
Right-sizing is hard, Resizable maps for ~~optimal~~ Better map size

John Fastabend, Cilium.io

Agenda

- Use Cases
- Current Situation
- Resizable maps
  - Hashmap Resize
  - Array Resize
  - Sockmap
- Conclusion

Use Cases: Cilium



Use Case: Cilium Networking

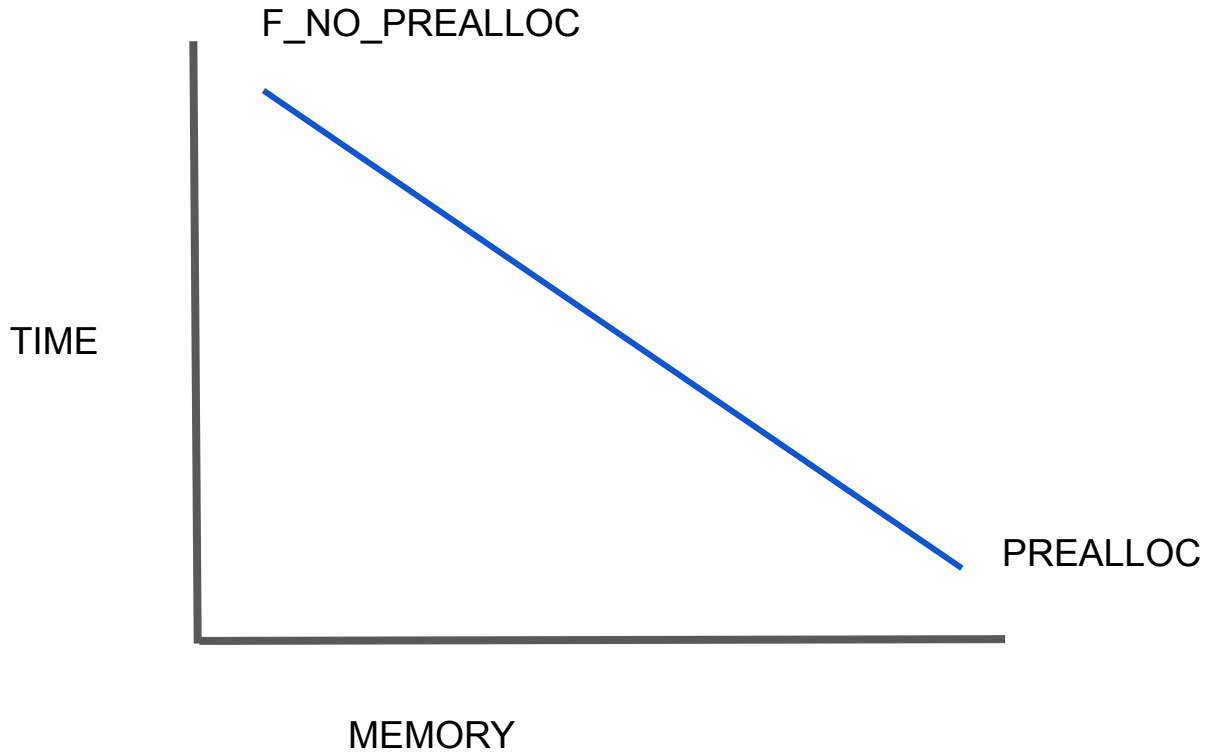
- Connection Tracking:
  - Every flow state is tracked
    - 1st packet, CT_NEW -> map_update_elem()
    - Nth packet, CT_* -> map_lookup_elem()
    - Last packet, CT_CLOSE -> map_delete_elem()
  - Cilium uses LRU or HASH map for CT_MAP, NAT
    - LRU overflows remove possibly live flow
    - Hash map_update_elem() fails
    - Both cases drop flows
  - Right-sizing may by hard
    - Heuristics (GC, NumCPUs, Memory, #nodes, #pods, etc)
    - Worst-case dynamics
    - Over-estimating is not free (56B entries, worse in some other places)
    - Dropping flows is also not nice.

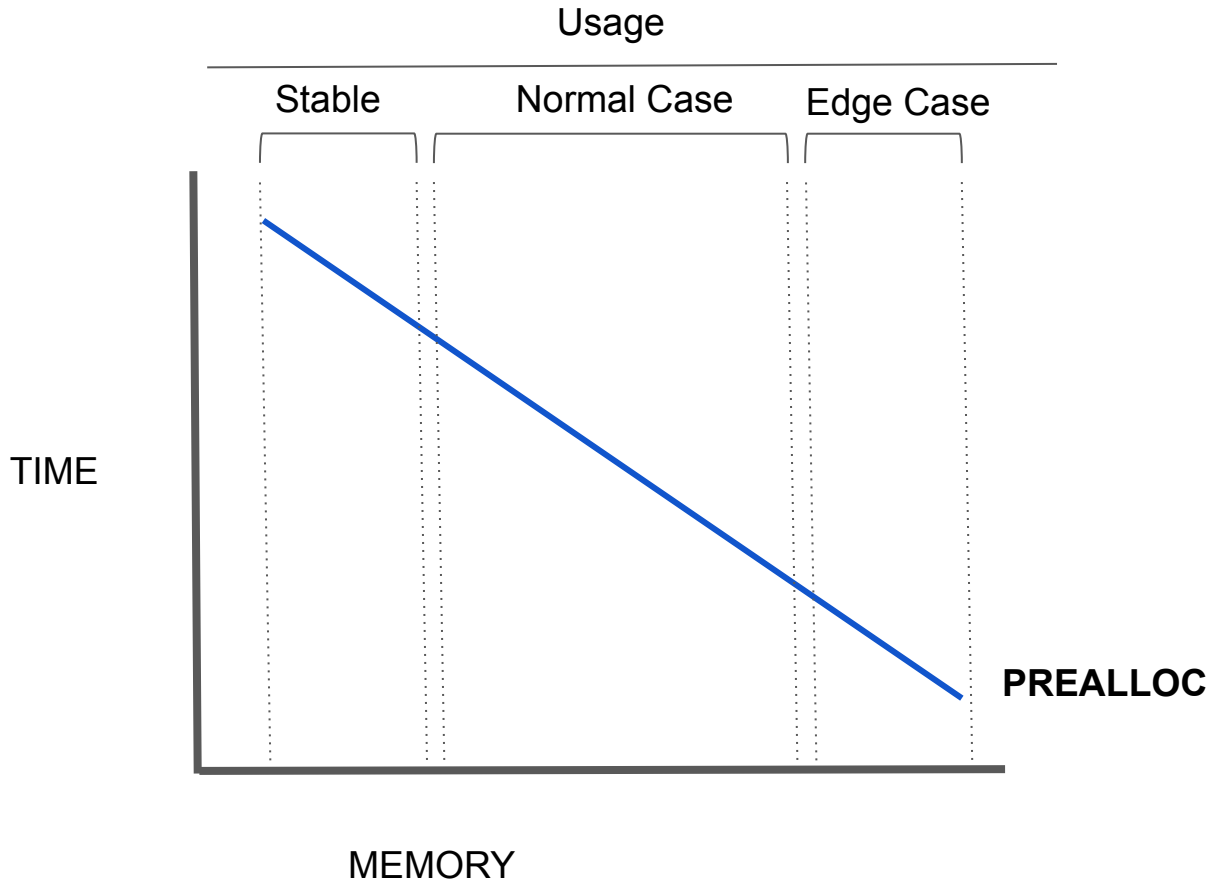
Use Case: Cilium Networking

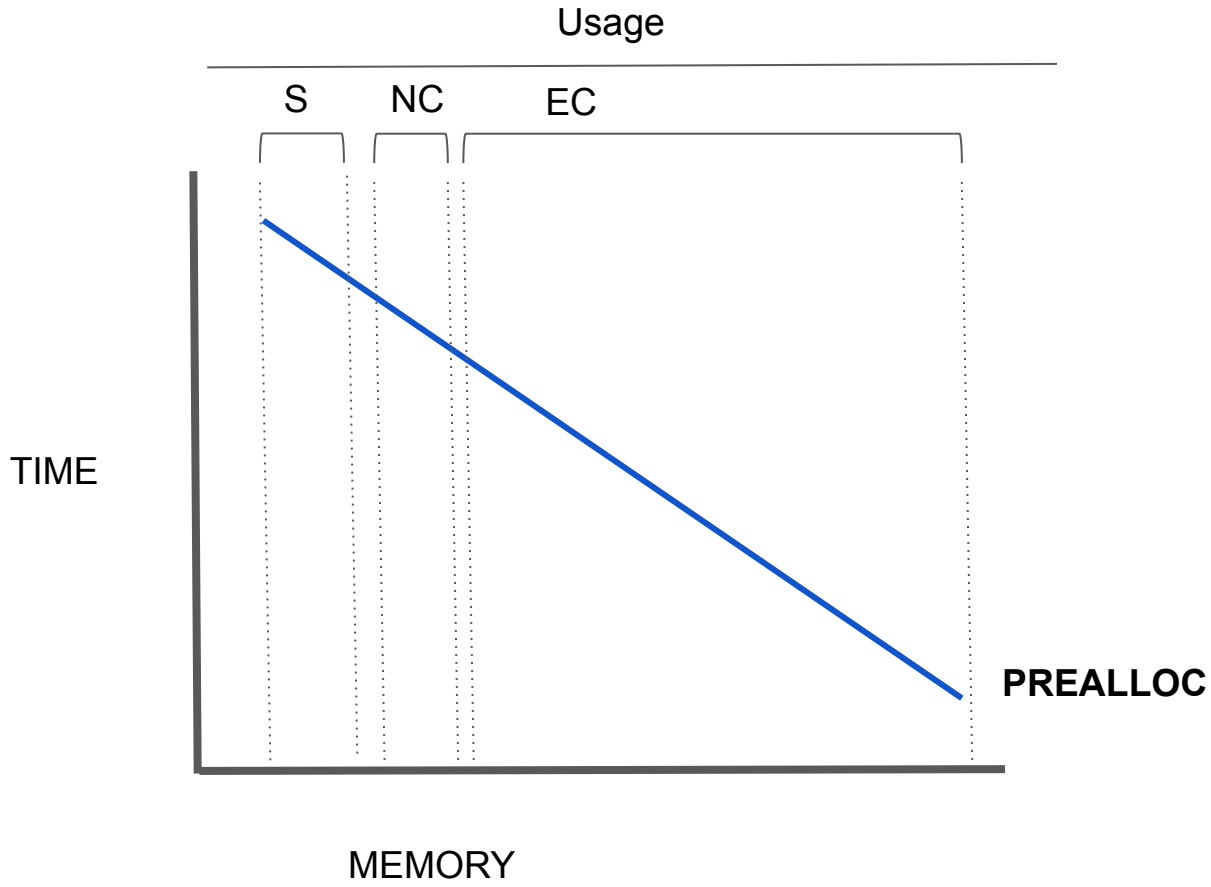
- Sockmap, Sockhash: Socket map in system for policy, load-balancing, etc.
  - “Glues” sockets together for local connections
    - Sockets added to map after 3WHS by sockops hook
    - Sockets deleted automatically when socket unhashed by kernel
  - Cilium uses Sockhash with some large # of elements hopefully an over-estimate
    - Hashmap full errors are tracked and sockets fail back to normal path
  - Cilium right-sizing requires understand max number of local sockets
    - Pre-allocation step, (max_entries * sizeof(struct sock *))
    - Easier to over-estimate, may be less of a penalty (8B)
    - Drops not fatal, cause performance variation
  - Insert **policy** and now critical component!
  - Sockmap, Load-balancing resize to adjust with endpoints
    - Hash table of endpoints, resize number of backends

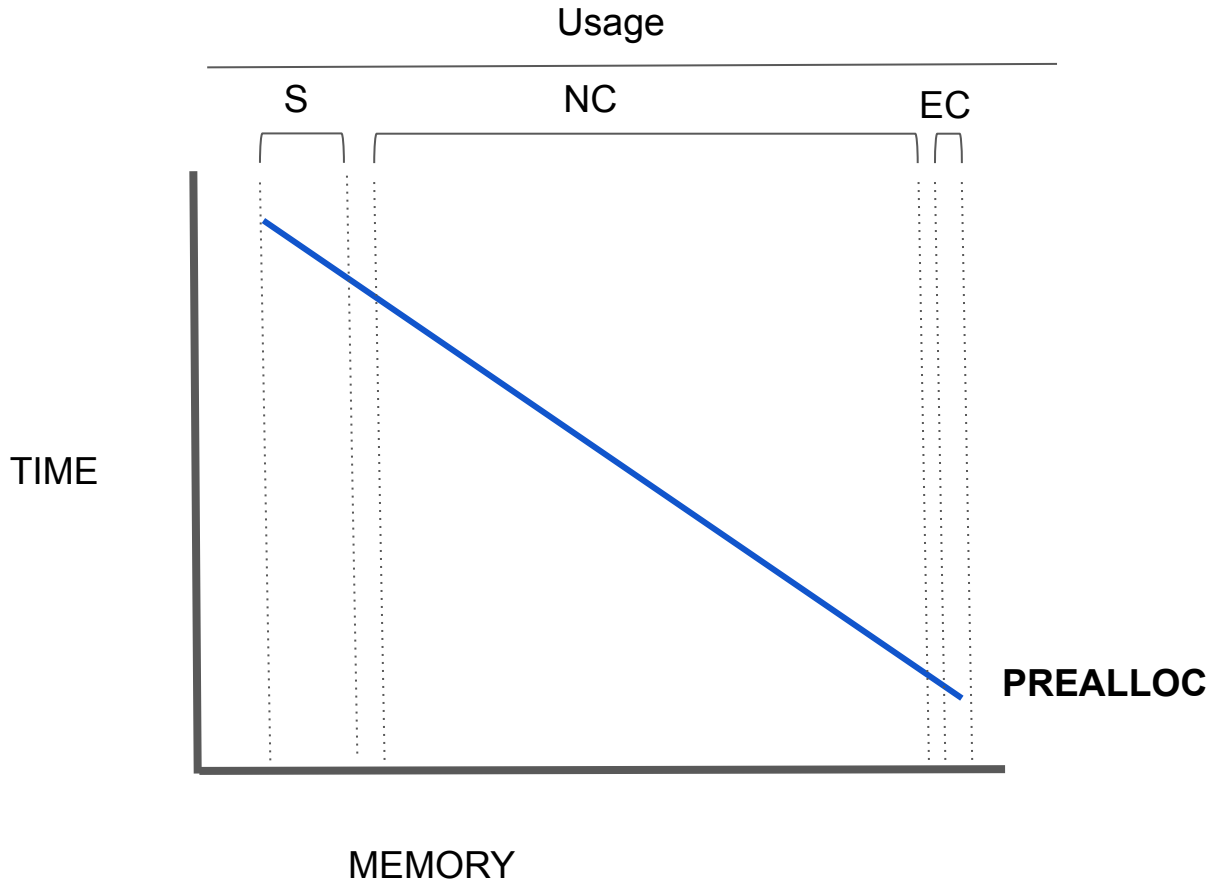
Use Case: Tracing

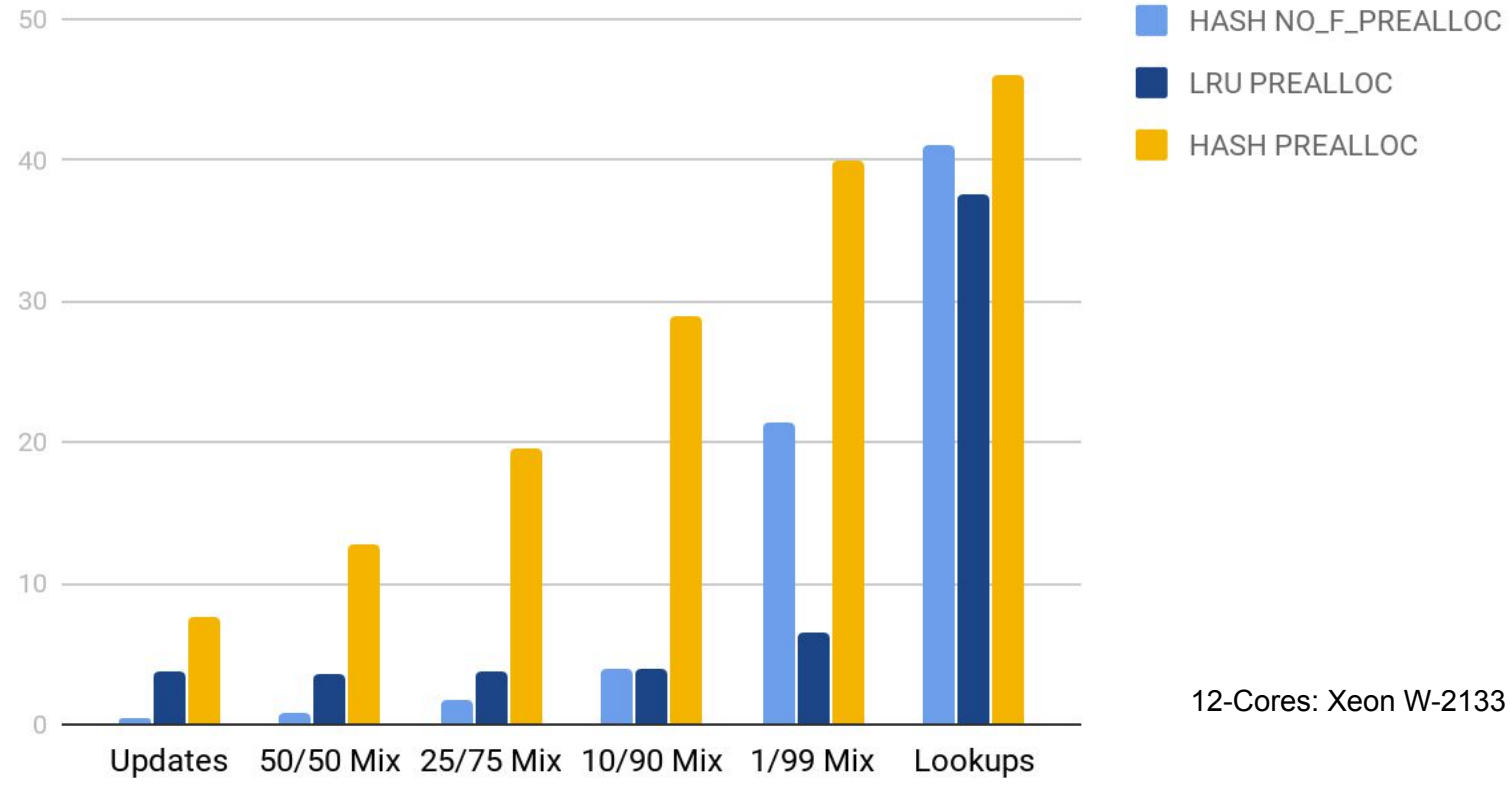
- Tracing: Processes, Sockets, ...
  - Many kernel objects may not have upper bounds so sizing is challenging
    - Process arguments
    - *max_pid > 4M*
    - Pods
    - Sockets
  - May not be tied directly to lifetime of object monitoring
    - Socket local storage
    - Inode local storage
    - Many cases not though
  - May benefit from flexible upper bounds


F_NO_PREALLOC
TIME
PREALLOC
MEMORY


Usage
Stable
Normal Case
Edge Case
TIME
PREALLOC
MEMORY

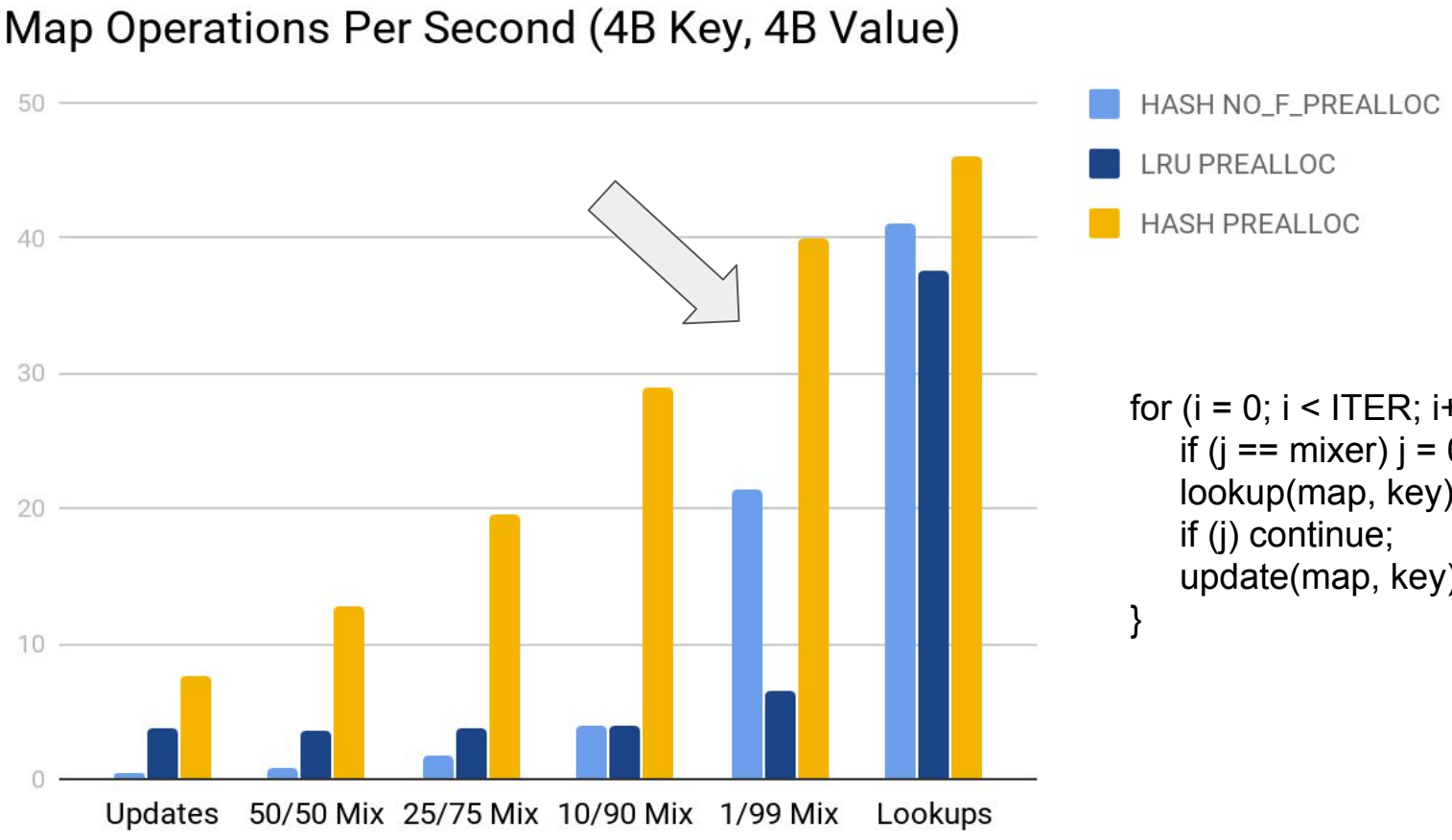

Usage
S
NC
EC
TIME
PREALLOC
MEMORY


Usage
S
NC
EC
TIME
PREALLOC
MEMORY

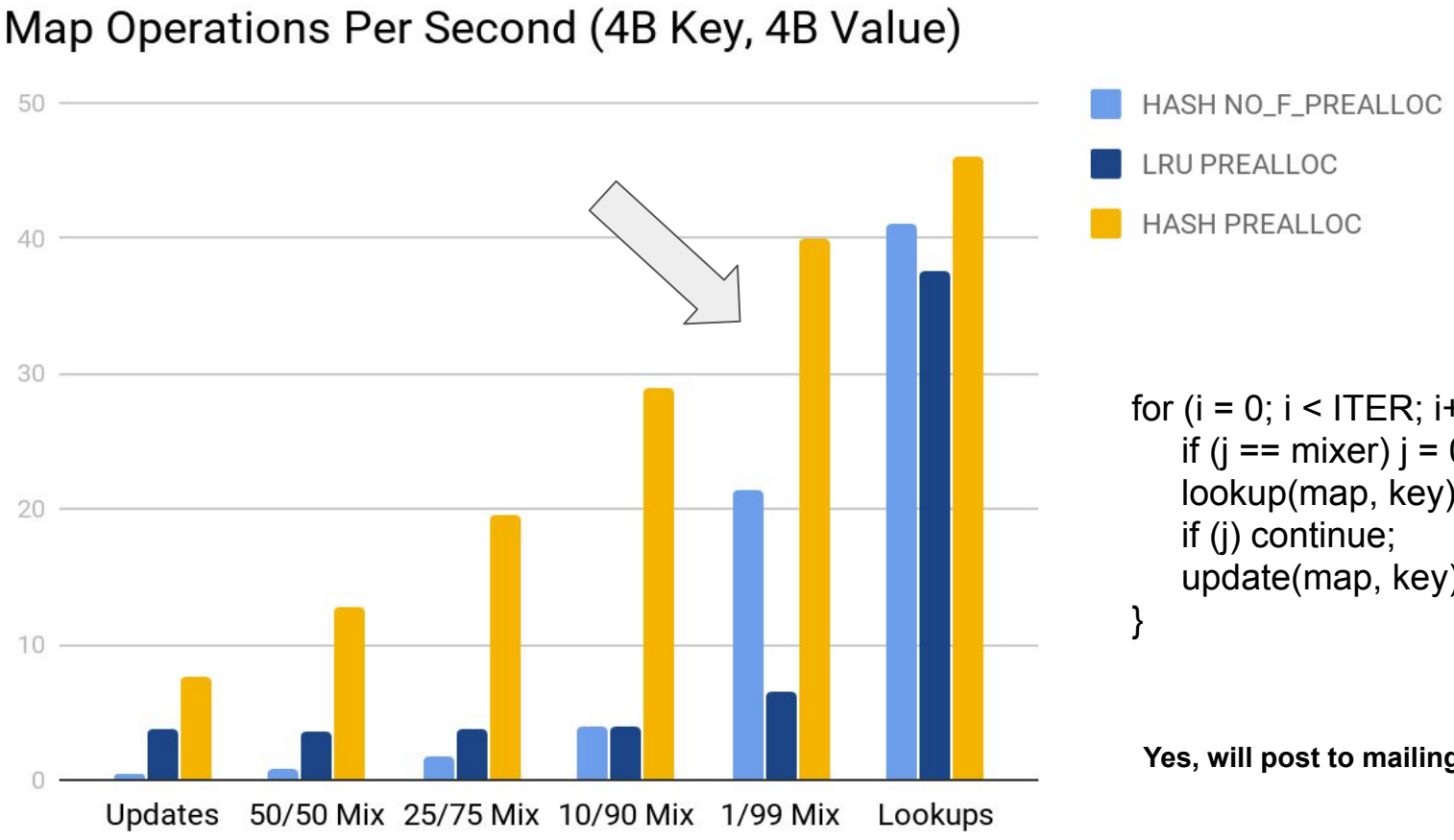


cilium

12-Cores: Xeon W-2133 CPU@3.60

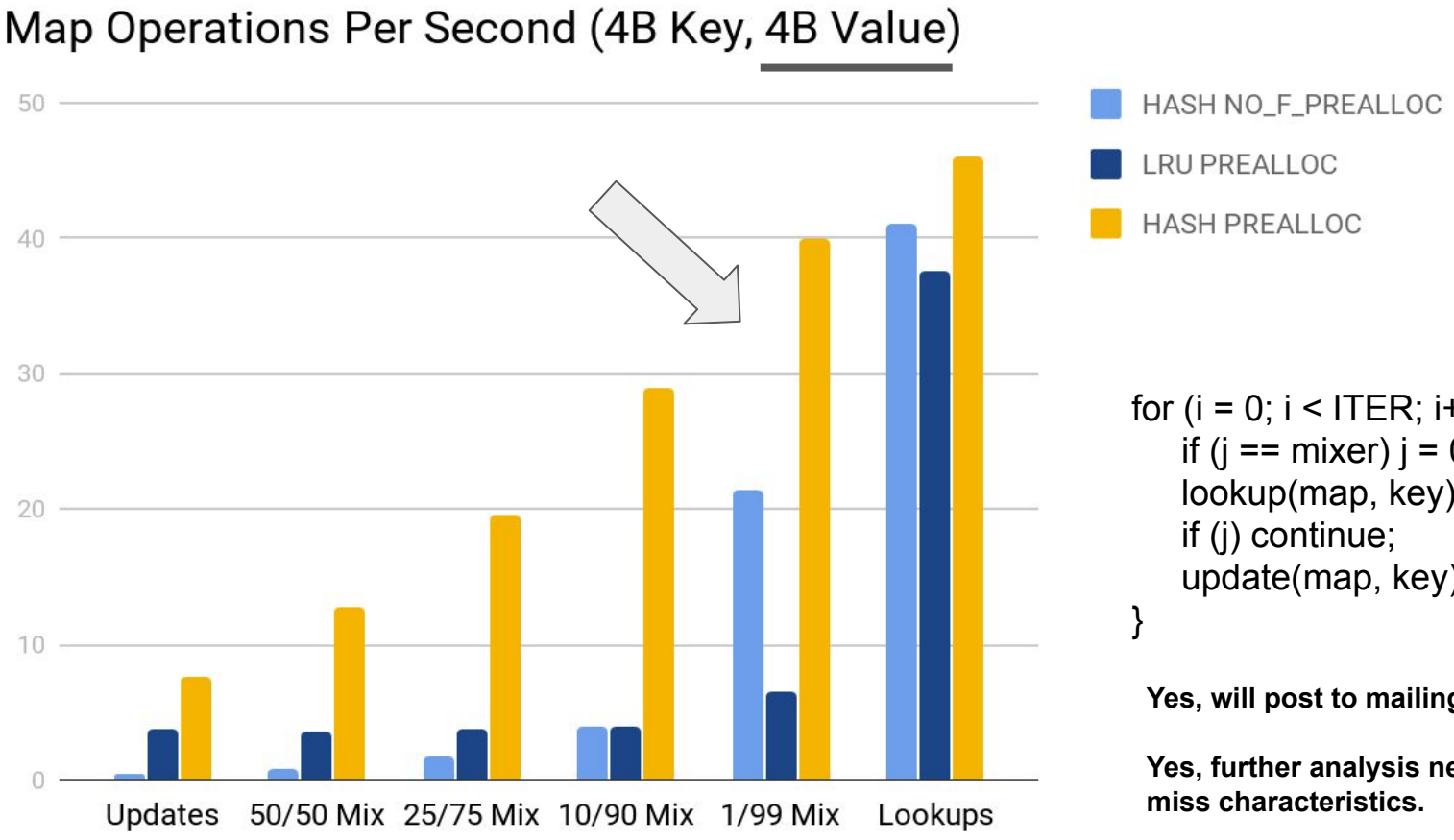



```
for (i = 0; i < ITER; i++, j++) {
    if (j == mixer) j = 0;
    lookup(map, key);
    if (j) continue;
    update(map, key);
}
```




```
for (i = 0; i < ITER; i++, j++) {
    if (j == mixer) j = 0;
    lookup(map, key);
    if (j) continue;
    update(map, key);
}
```

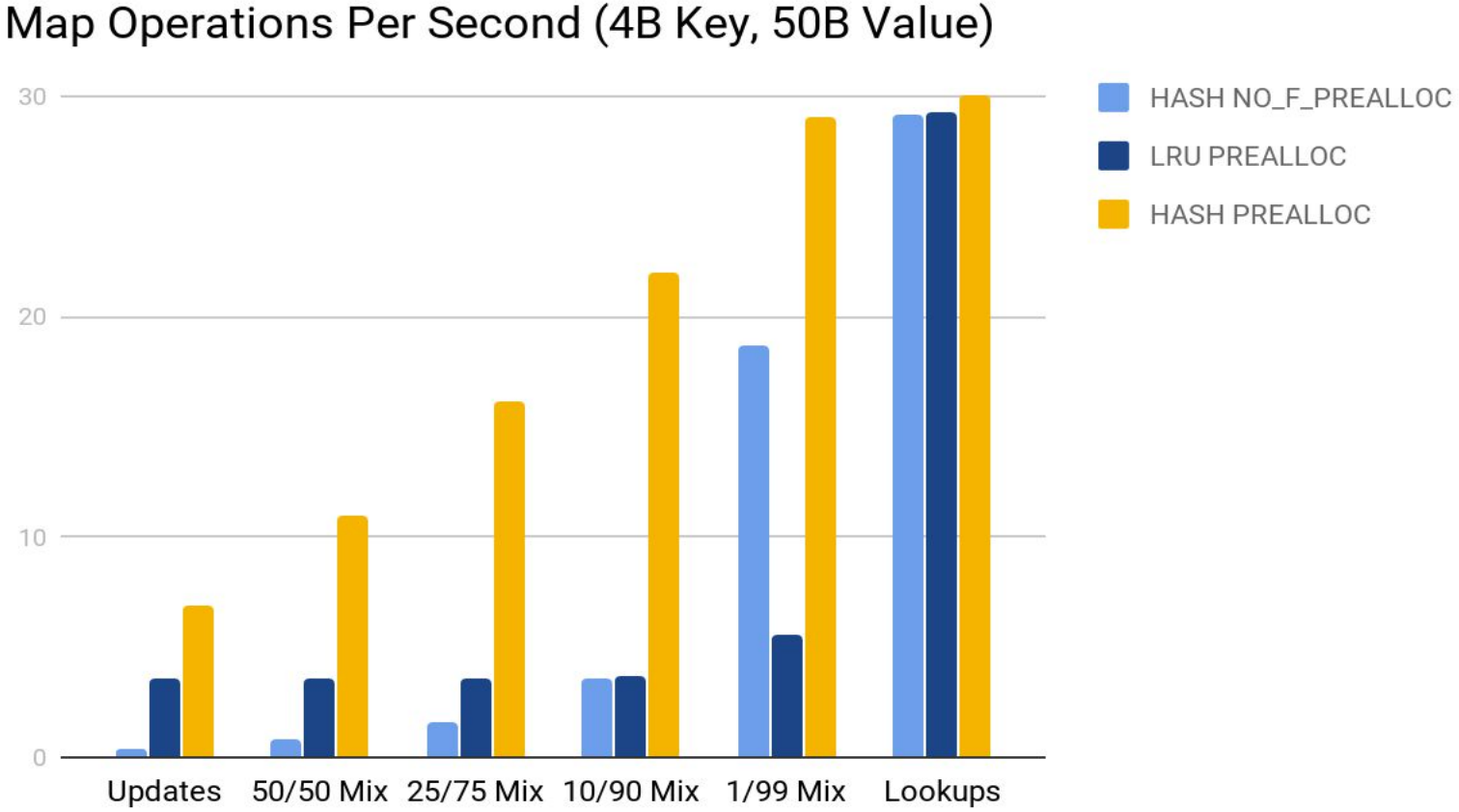
Yes, will post to mailing list!



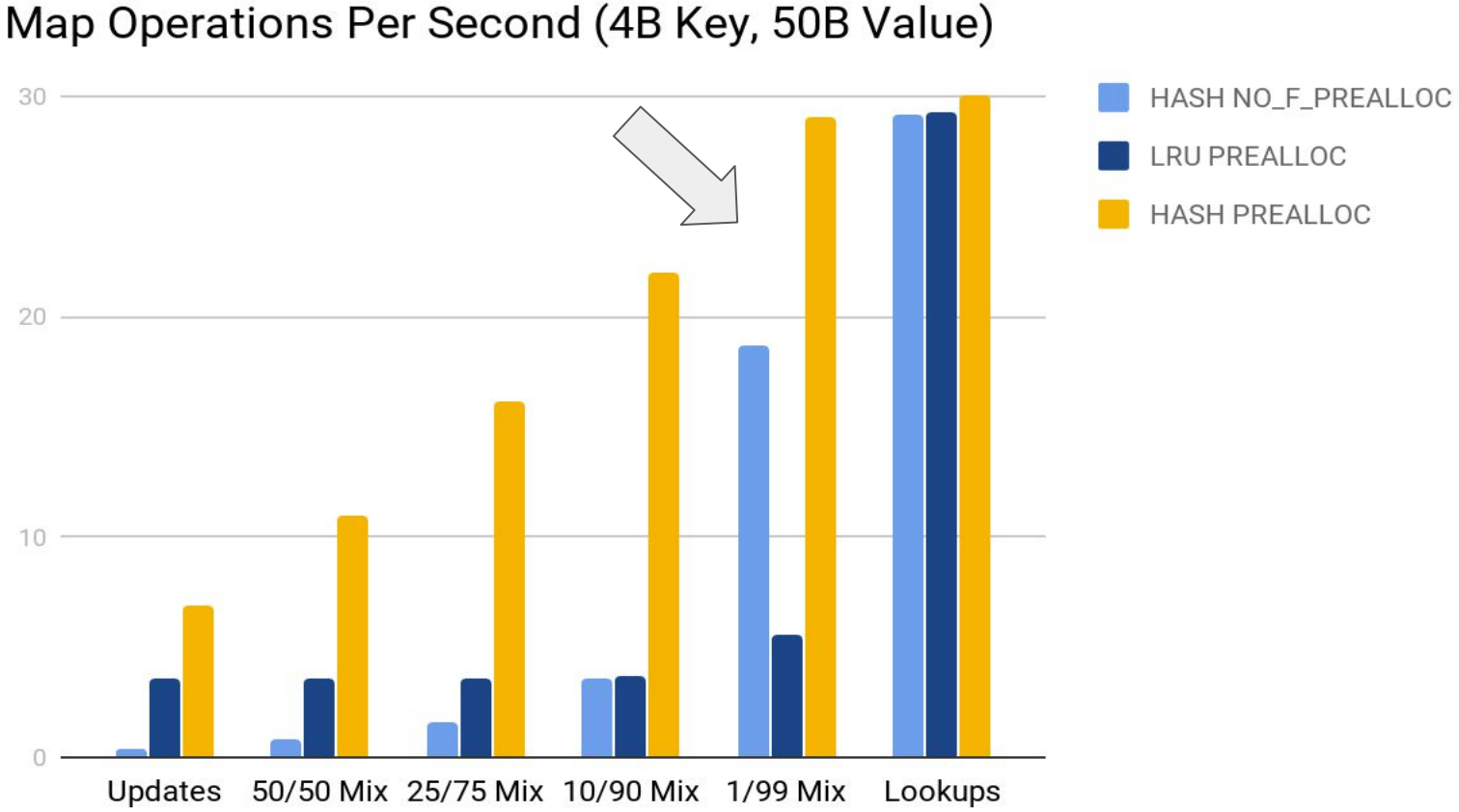

```
for (i = 0; i < ITER; i++, j++) {
    if (j == mixer) j = 0;
    lookup(map, key);
    if (j) continue;
    update(map, key);
}
```

Yes, will post to mailing list!

Yes, further analysis needed for miss characteristics.


Map Operations Per Second (4B Key, 50B Value)
30
20
10
0
Updates
50/50 Mix
25/75 Mix
10/90 Mix
1/99 Mix
Lookups
HASH NO_F_PREALLOC
LRU PREALLOC
HASH PREALLOC

cilium


Map Operations Per Second (4B Key, 50B Value)
30
20
10
0
Updates
50/50 Mix
25/75 Mix
10/90 Mix
1/99 Mix
Lookups
HASH NO_F_PREALLOC
LRU PREALLOC
HASH PREALLOC

cilium

To Prealloc or not to Prealloc

- Where possible can simple
  NO_F_PREALLOC
  max_entries >> worse_case

- Where performance matters we can not afford to allocate

- However, we want to avoid consuming large amounts of memory
  - Best-fit without intimate knowledge of workload (Cilium installed in many environments)
  - Grow memory* use as needed

* Consider DDOS implications for some use-cases, but others local storage for example may be safe

Can we grow the map when needed, instead of drop new entries?

AND only preallocate memory I need

AND keep approximate performance of preallocated map

Can we grow the map when needed, instead of drop new entries?

Yes, but we introduce a new error case -EBUSY.

Can we grow the map when needed, instead of drop new entries?

Yes, but we introduce a new error case -EBUSY and lose direct access*.

For now consider

BPF_MAP_TYPE_HASH_RESIZE

HASH_RESIZE

- Implement new map type BPF_MAP_TYPE_HASH_RESIZE
- Backed by rhashtable* object

  Author: Thomas Graf <tgraf@suug.ch>
  Date: Sat Aug 2 11:47:44 2014 +0200
  lib: Resizable, Scalable, Concurrent Hash Table

- Rhashtable: Resizable, Scalable, Concurrent Hash Tables
  - Lookups: lockless
  - Update: bucket locks, may EAGAIN, EBUSY during resize
  - Delete: bucket locks, may EAGAIN, EBUSY during resize
  - Grow/Shrink: bucket locks

[0] https://www.usenix.org/legacy/event/atc11/tech/final_files/Triplett.pdf

HASH_RESIZE

- **Optimized BPF Rhashtable**
  - Grow hash table using watermarks or bpf_grow()
  - Shrink hash table using watermarks or bpf_shrink()
  - Keep upper bound to avoid DDOS
  - Configurable initial limit
  - Removed generics
  - Updated some error codes
  - Moved some bits around

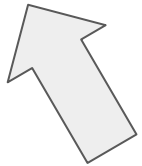
- During resize update operations may receive **new error EBUSY**

Implementation Note: No percpu, mmap, flags, implemented in initial draft.

HASH_RESIZE

- **Optimized BPF Rhashtable**
  - Grow hash table using watermarks or bpf_grow()
  - Shrink hash table using watermarks or bpf_shrink()
  - Keep upper bound to avoid DDOS
  - Configurable initial limit
  - Removed generics
  - Updated some error codes

- During resize update operations may receive **new error EBUSY**



Cost of having resize operations

HASH_RESIZE

Aside: RESIZE Dynamics

- Current benchmarks/draft using hard-coded watermarks

- Map library maintains statistics with map
  a. Presumably could use to grow/shrink map
  b. Track size of current map
  c. Track BUSY errors to correct for too frequent resizes

HASH_RESIZE

-EBUSY

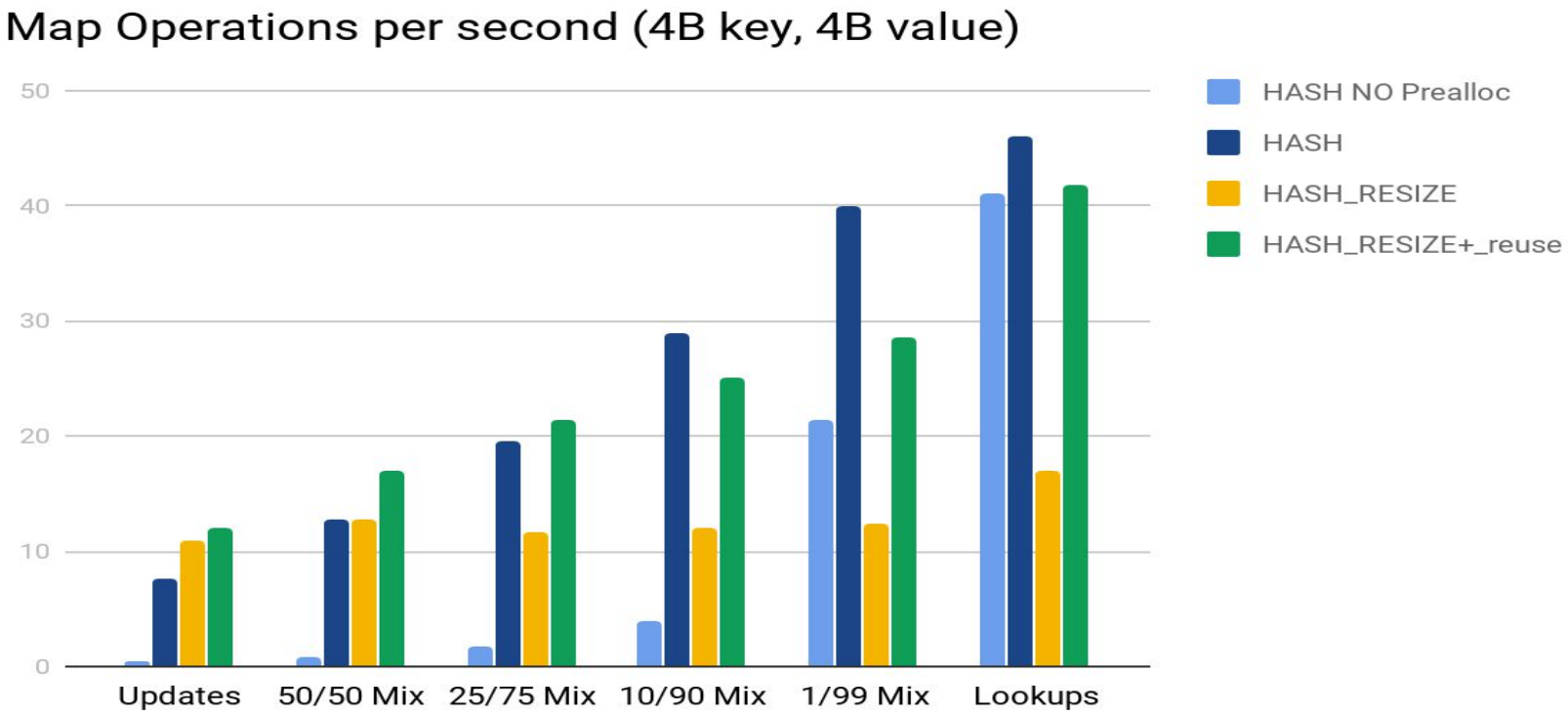
- Retry? If (err == -EBUSY) do_again;
- Drop entry, same case as when the map overflows
  - Assumption: Better to drop a few in transient case
- Recover, may be difficult in CONNTRACK
  - Experimenting with small overflow LRU
  - Also appears nicer to drop initial 3WHS instead of established session
  - If we manage resize appropriately do we see these in-practice? Need to deploy to find out.

- Better than over-running an LRU table and dropping everything.

HASH_RESIZE

How did we do?

HASH_RESIZE Benchmark

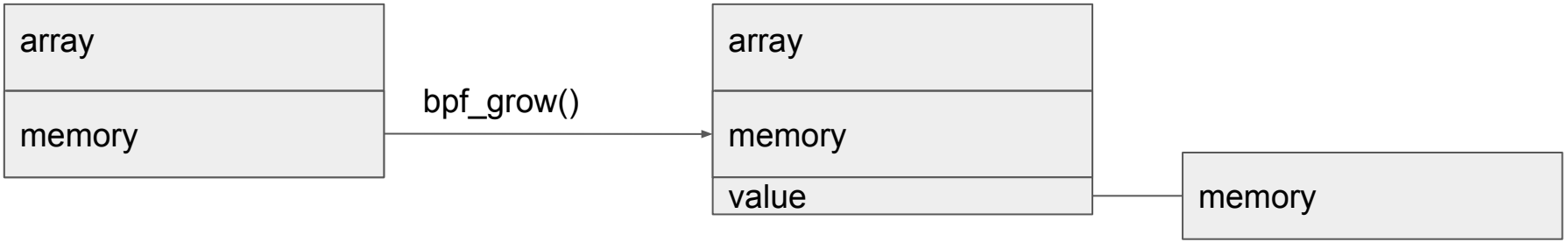


HASH_RESIZE Conclusion

- Provides resizable variant of hash table with good performance characteristics
- Introduces a new error type EBUSY and retries. These appear manageable in our use cases.
- Better than dropping packets on the floor.

ARRAY_RESIZE

- Implement new map type BPF_MAP_TYPE_ARRAY_RESIZE



ARRAY_RESIZE

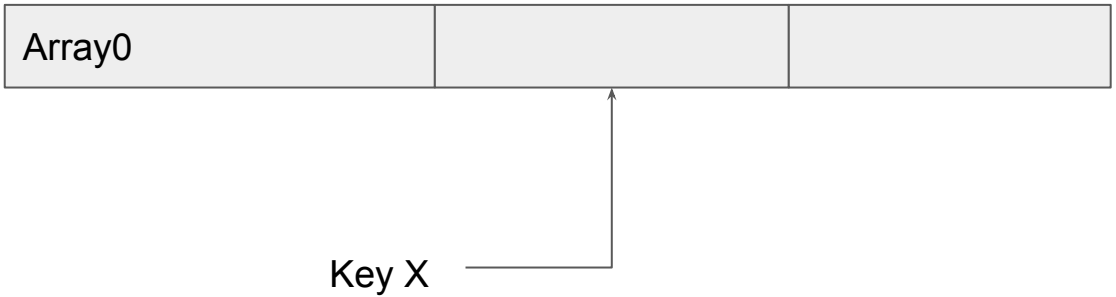
- ~~Implement new map type BPF_MAP_TYPE_ARRAY_RESIZE~~

- **MAP_OF_ARRAY + ARRAY**

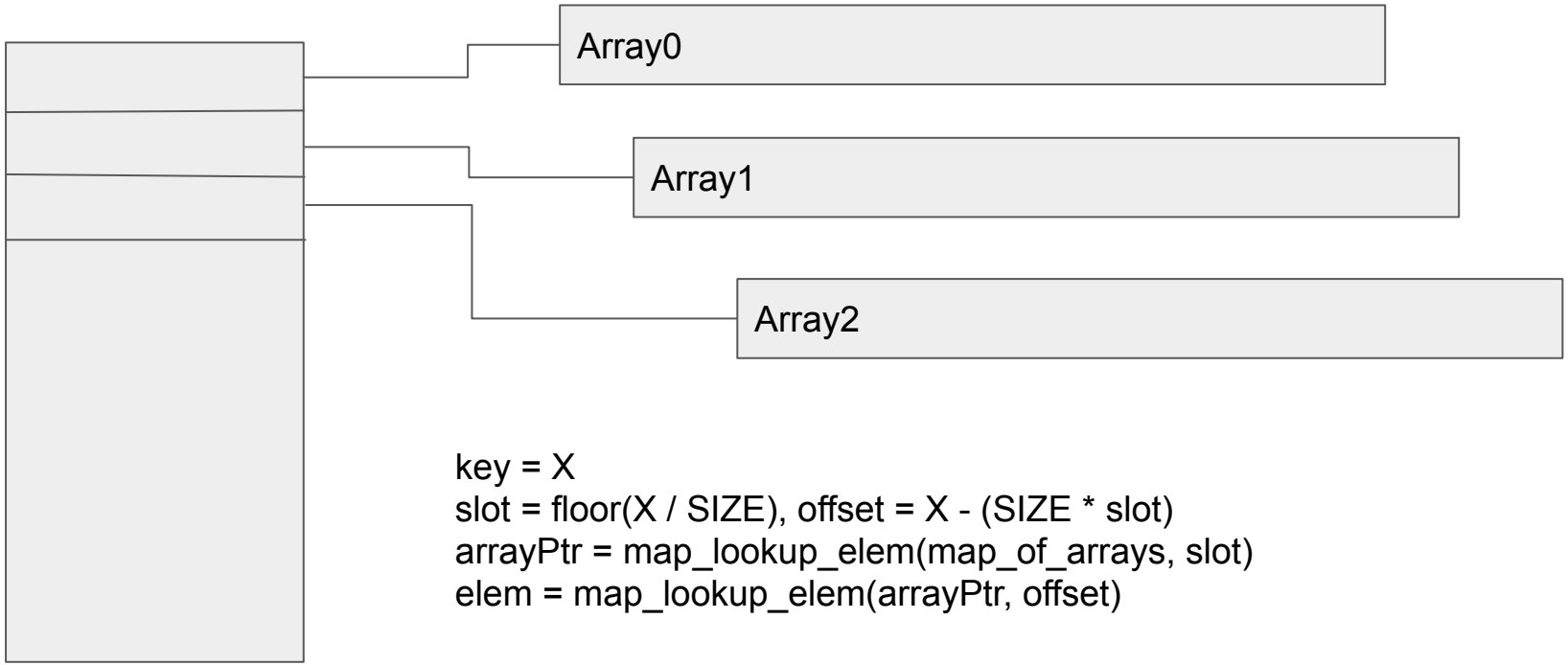
ARRAY_RESIZE

- ~~Implement new map type BPF_MAP_TYPE_ARRAY_RESIZE~~

- **MAP_OF_ARRAY + ARRAY**



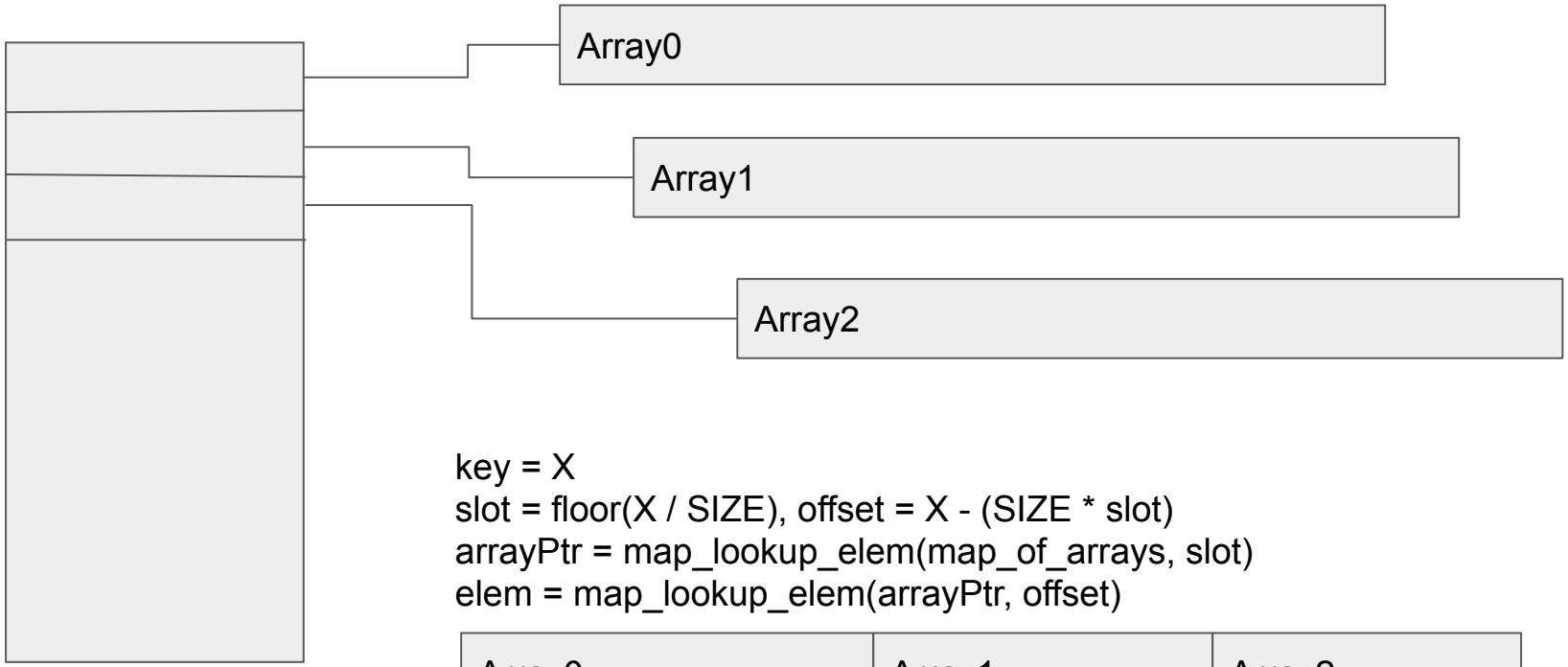
ARRAY_RESIZE



key = X
slot = floor(X / SIZE), offset = X - (SIZE * slot)
arrayPtr = map_lookup_elem(map_of_arrays, slot)
elem = map_lookup_elem(arrayPtr, offset)

MAP_OF_ARRAYS

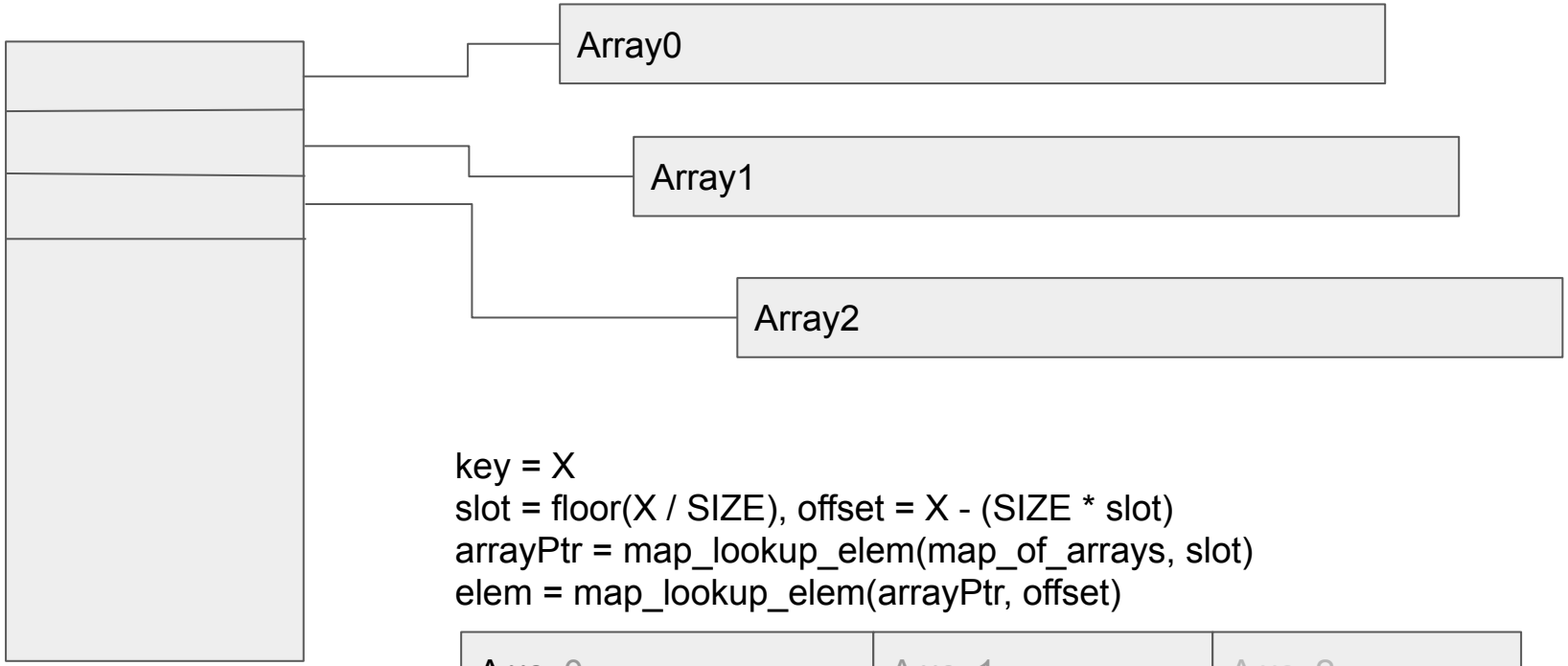
ARRAY_RESIZE



key = X
slot = floor(X / SIZE), offset = X - (SIZE * slot)
arrayPtr = map_lookup_elem(map_of_arrays, slot)
elem = map_lookup_elem(arrayPtr, offset)

Array0	Array1	Array2

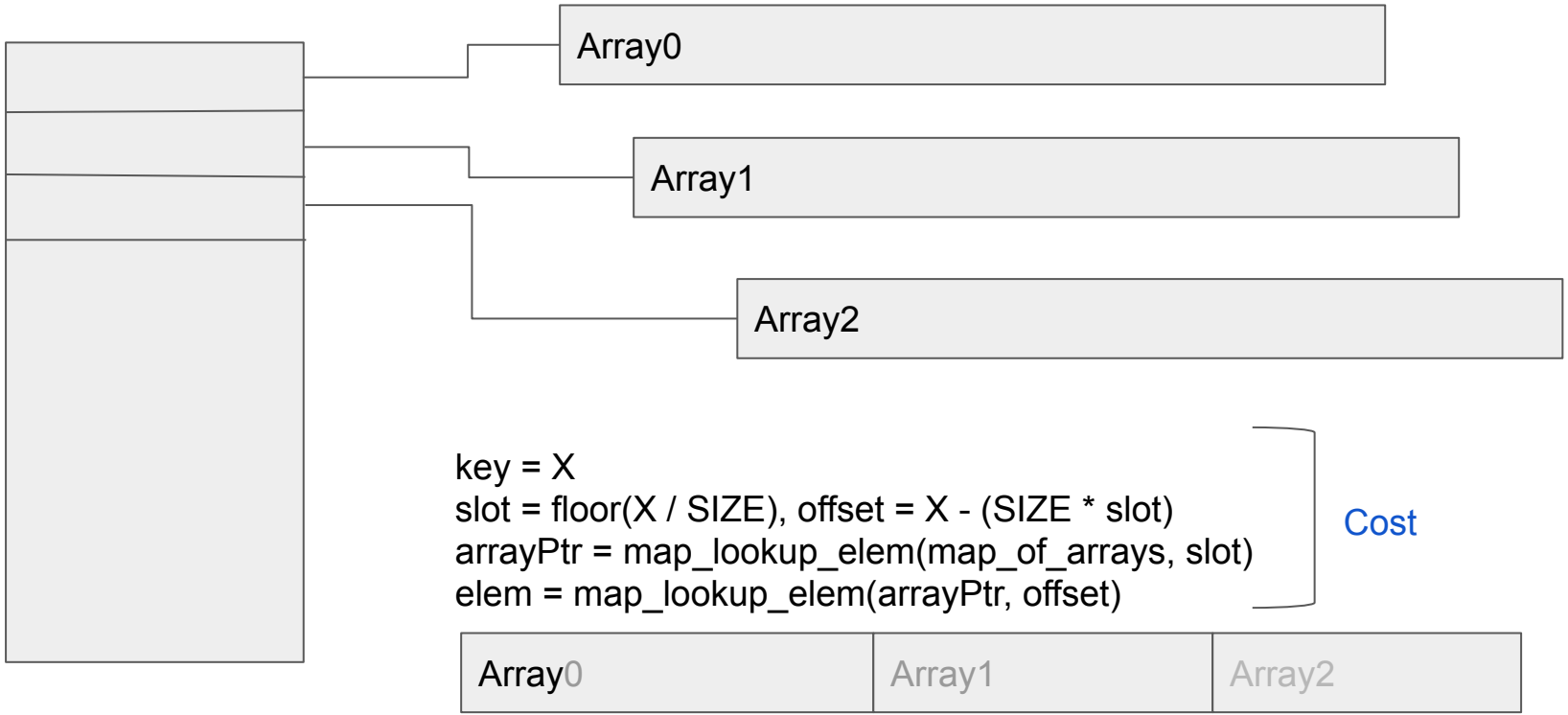
ARRAY_RESIZE

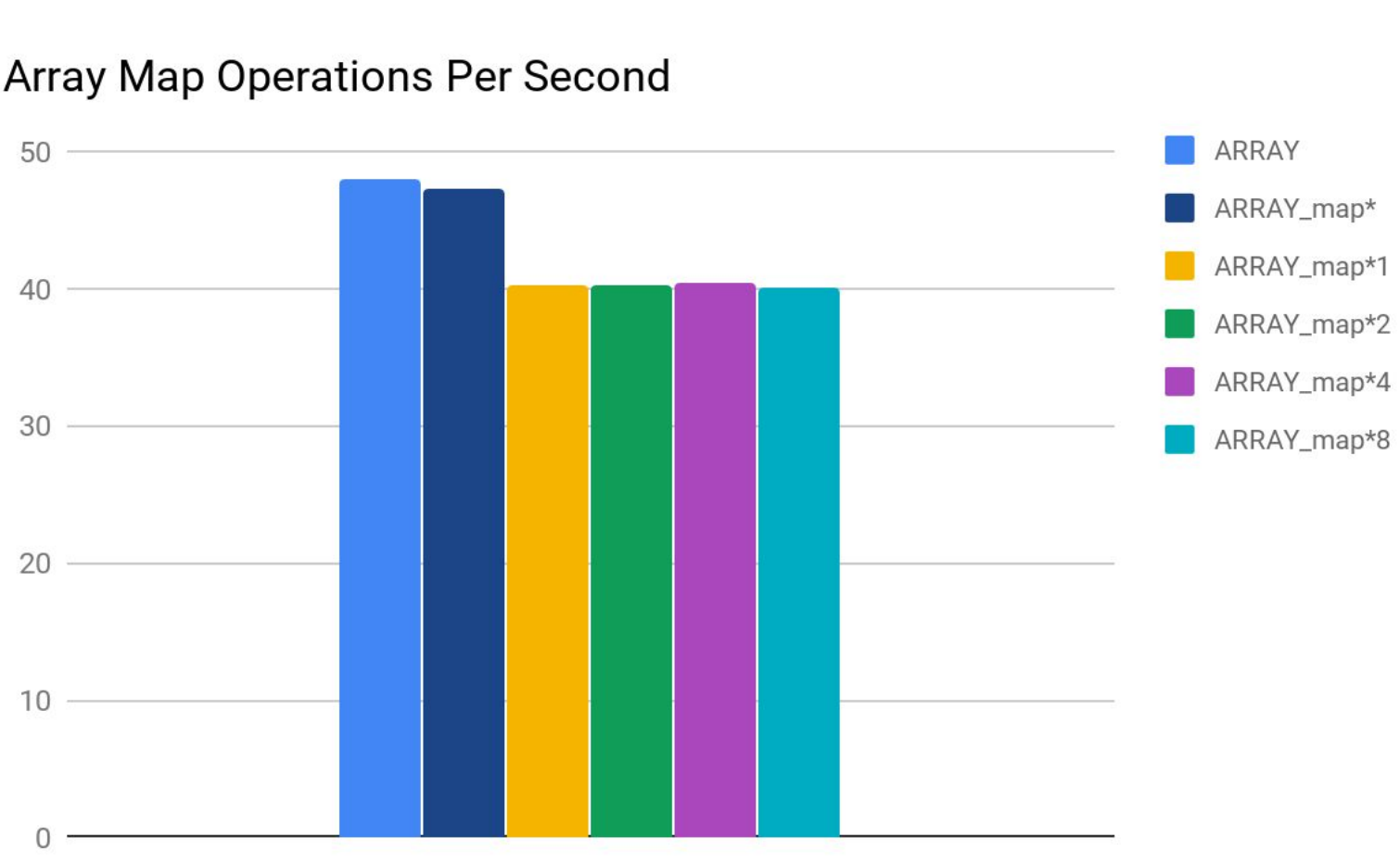


key = X
slot = floor(X / SIZE), offset = X - (SIZE * slot)
arrayPtr = map_lookup_elem(map_of_arrays, slot)
elem = map_lookup_elem(arrayPtr, offset)

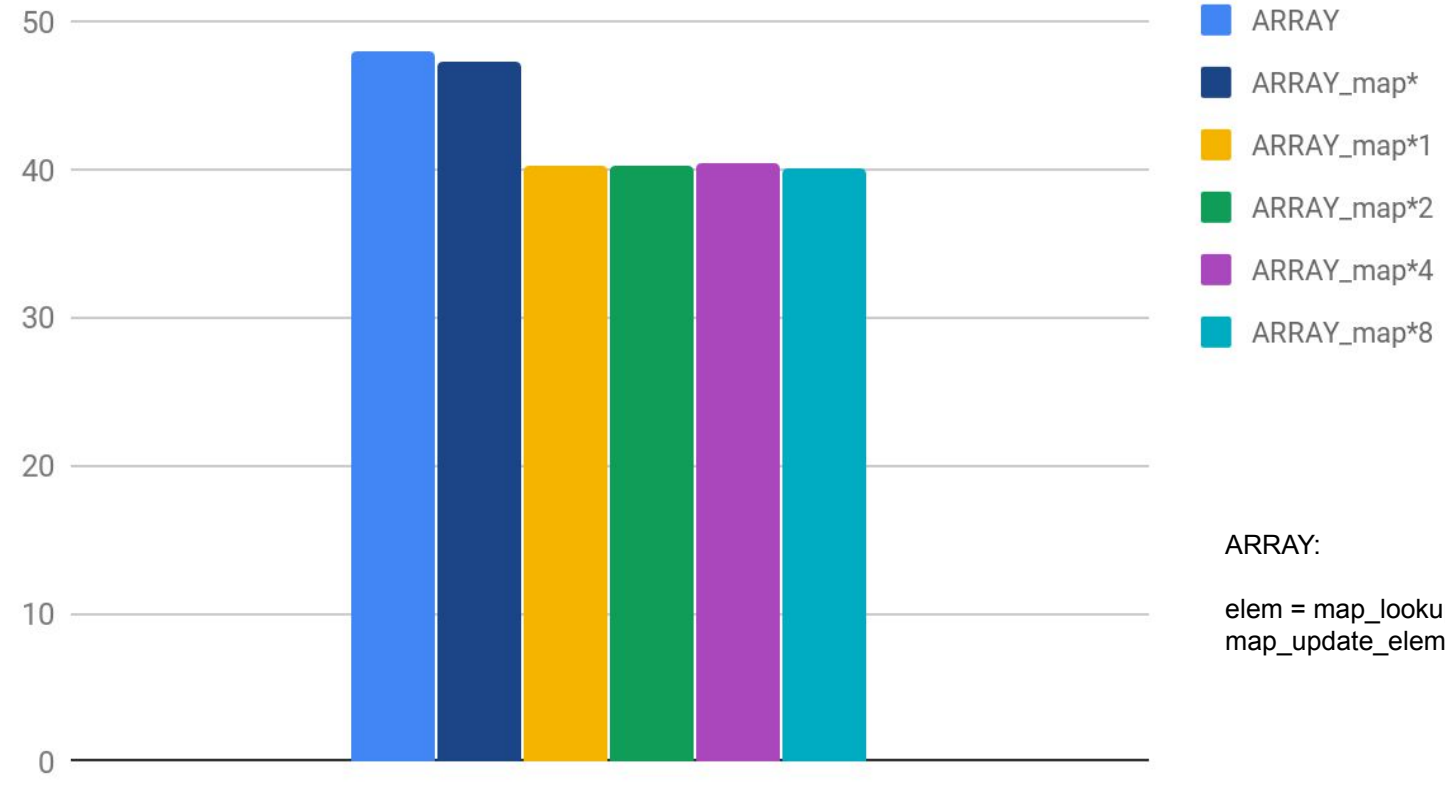
Array0	Array1	Array2

ARRAY_RESIZE




Array Map Operations Per Second
50
40
30
20
10
0
ARRAY
ARRAY_map*
ARRAY_map*1
ARRAY_map*2
ARRAY_map*4
ARRAY_map*8

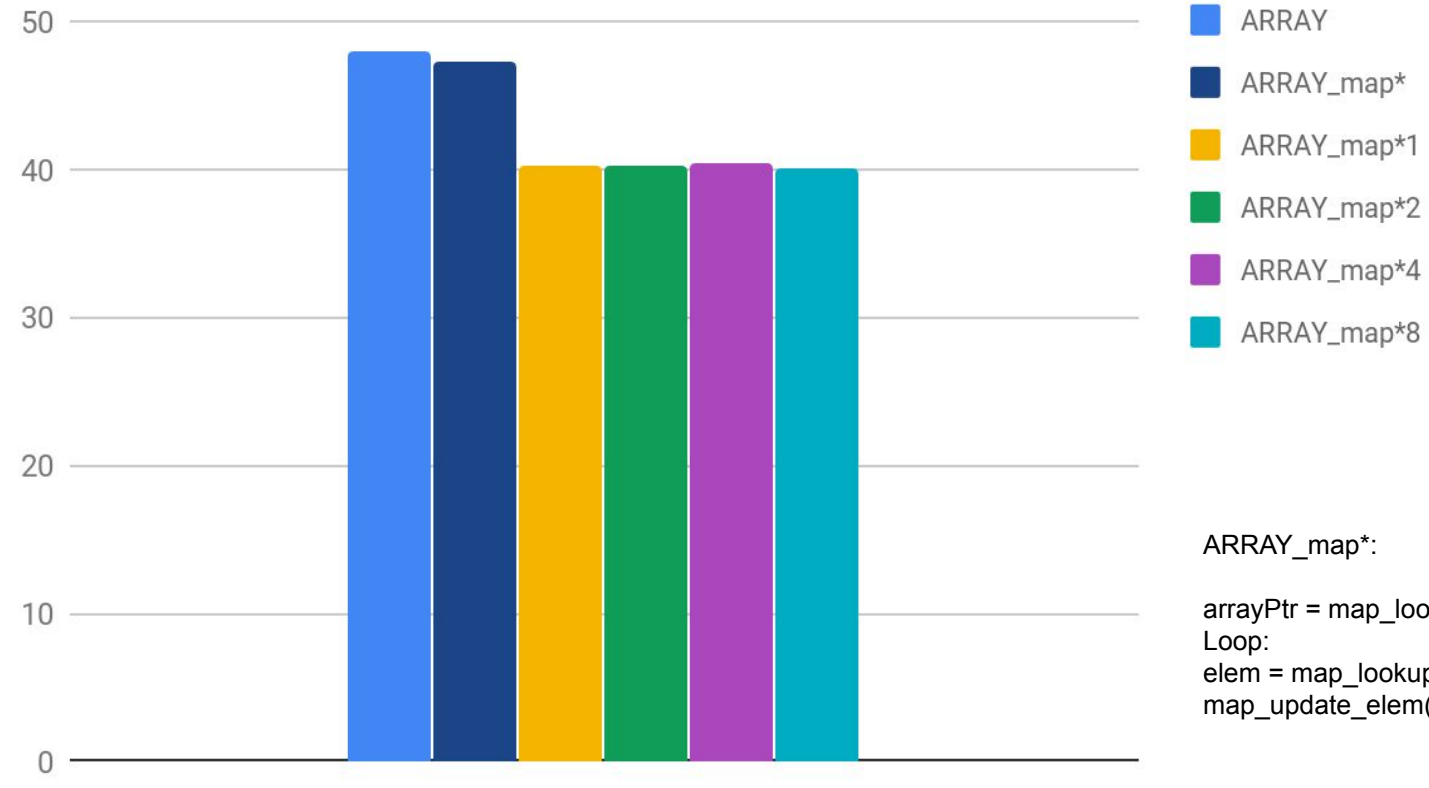
cilium



cilium

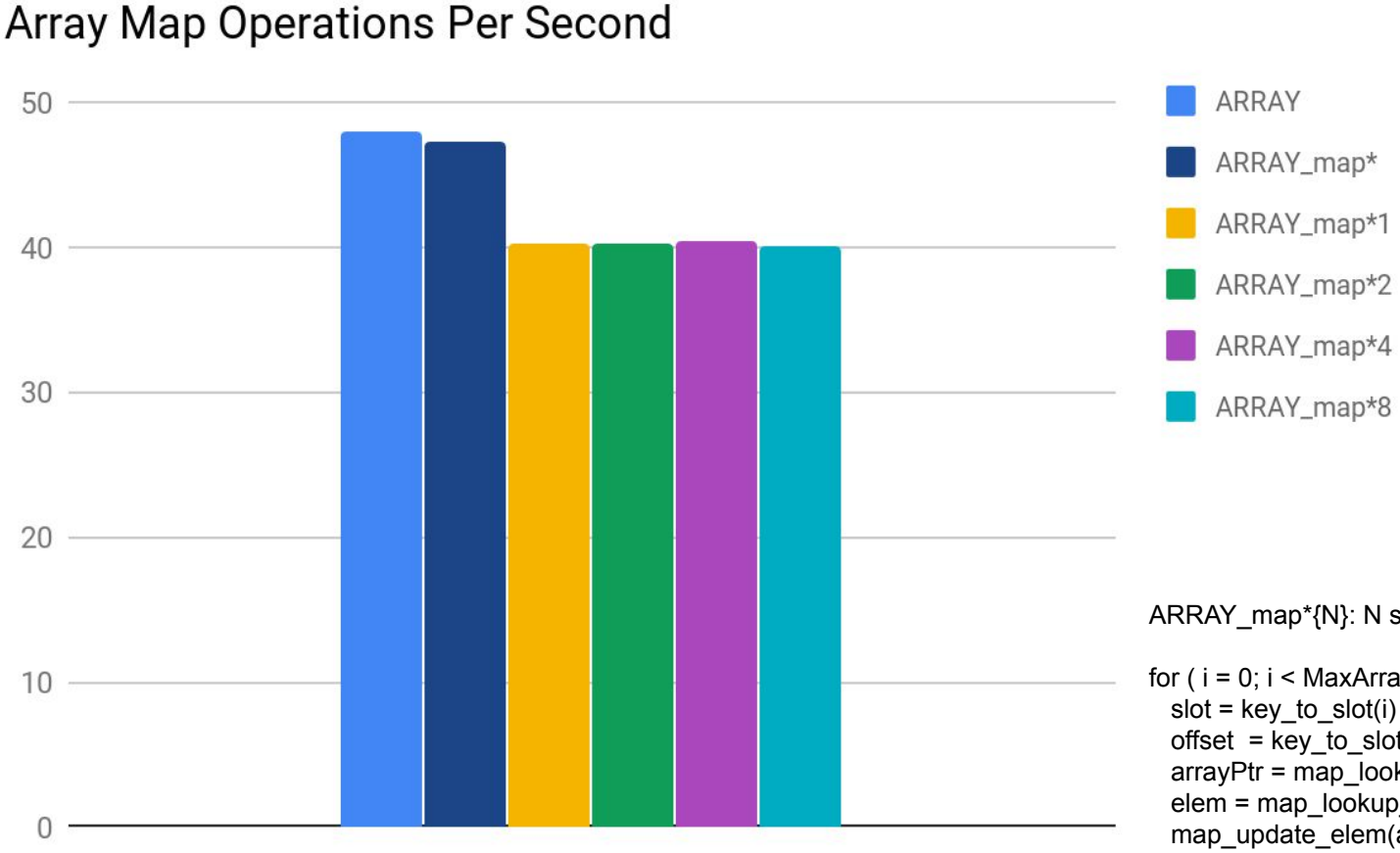
ARRAY:

elem = map_lookup_elem(arrayPtr, offset)
map_update_elem(arrayPtr, offset)



ARRAY_map*:

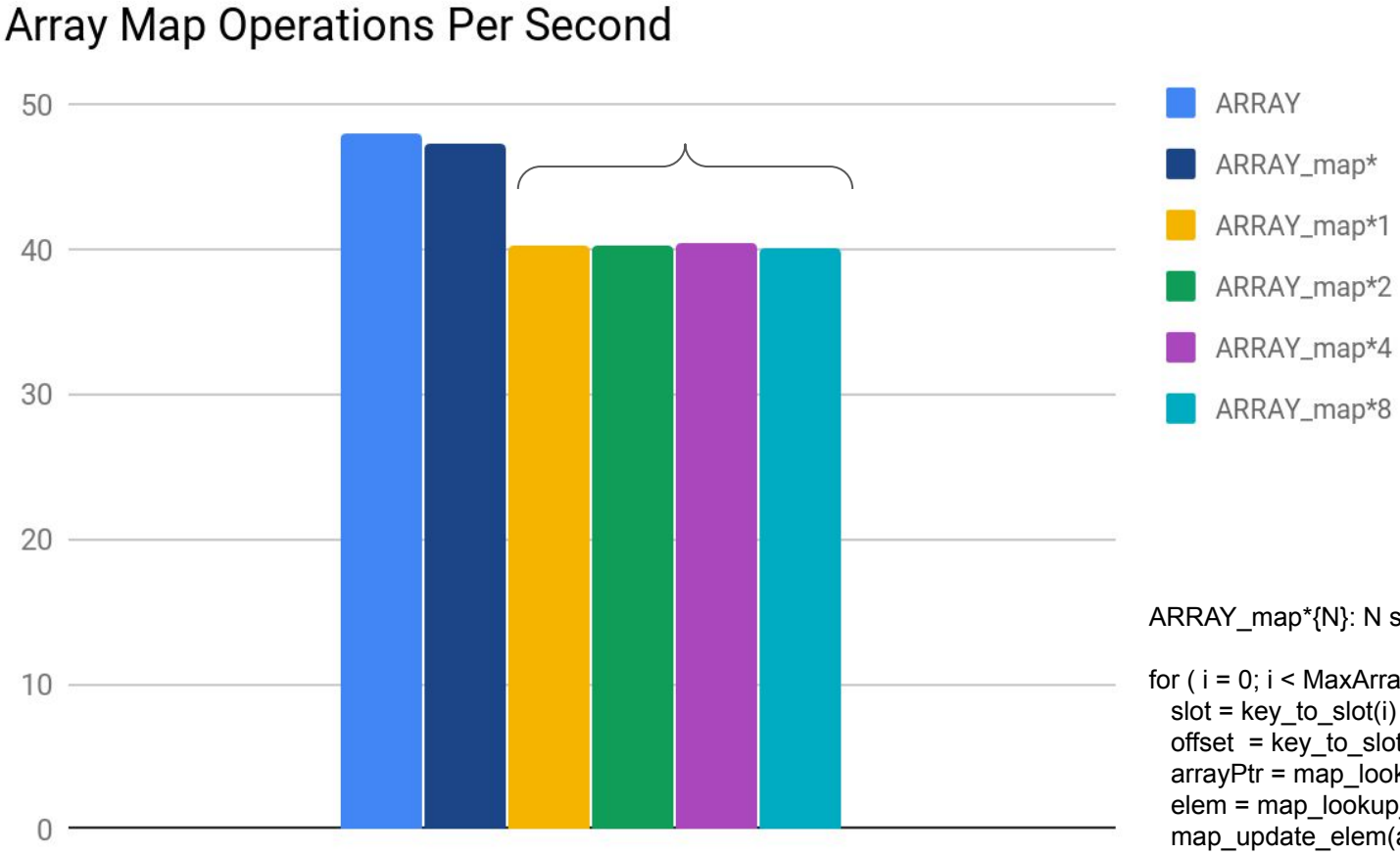
```
arrayPtr = map_lookup_elem(map_of_arrays, 0)
Loop:
elem = map_lookup_elem(arrayPtr, offset)
map_update_elem(arrayPtr, offset)
```



cilium

ARRAY_map*{N}: N sub-arrays

```
for ( i = 0; i < MaxArraySz; i++)
   slot = key_to_slot(i)
   offset  = key_to_slot(i)
   arrayPtr = map_lookup_elem(map_of_arrays, slot)
   elem = map_lookup_elem(arrayPtr, offset)
   map_update_elem(arrayPtr, offset)
```



ARRAY_map*{N}: N sub-arrays

```
for ( i = 0; i < MaxArraySz; i++)
   slot = key_to_slot(i)
   offset  = key_to_slot(i)
   arrayPtr = map_lookup_elem(map_of_arrays, slot)
   elem = map_lookup_elem(arrayPtr, offset)
   map_update_elem(arrayPtr, offset)
```

ARRAY_RESIZE: Claw back performance

- Performance
  - Obvious tricks, division is expensive so use a jump table, if/else, etc.
  - Also obvious, size arrays to make math easy
- TBD direct access
  - Should be able to remove both CALLS with inline insn
  - Now how close would we get?

ARRAY_RESIZE: Conclusion

- Appears promising, any ideas?
- TBD inline instructions

SOCKHASH_RESIZE:

- Load Balancer case resolved where agent controls map with BPF map iter
  - Lorenz Bauer, submitted series under review now!
- Cilium sockmap case
  - BPF map iter seems unusable when map may be updated in parallel
  - Similar to HASH_RESIZE case experiment with RESIZE operation
  - TBD, POC not implemented yet

Conclusion

- BPF_MAP_TYPE_HASH_RESIZE
  - Useful and good performance characteristics, RFC coming soon.
  - Solves a real problem around sizing tables
  - Will send bench patches so folks can reproduce charts

- BPF_MAP_TYPE_ARRAY_RESIZE
  - Looks promising if we inline the helpers

- BPF_MAP_TYPE_SOCKMAP
  - Presumably above can be used here as well

Cilium:

https://cilium.io
https://cilium.io/slack
https://github.com/cilium/cilium

Email:
john@isovalent.com
john.fastabend@gmail.com