ISOVALENT

Optimizing BPF hashmap and friends

FOSDEM 2023

Anton Protopopov

New hash function for BPF

- BPF Summit 2021: Andrii Nakriyko proposed to try new hash functions for BPF hashmap (and other hash-based maps)
- XXH3 – a perfect modern hash function by Yann Colette, but requires vector operations, so no use for BPF
- However, vectorized ops only required for input lengths > 240, and there’s a scalar version which should work better than jhash in any case
- Our use cases in Cilium require key sizes of 4-24 bytes
- (My original intent was to use xxh3 to optimize Wildcard map)

Short contents

- Benchmark howto
- Benchmark hash functions
- Benchmark maps using different hash functions

Reduce noise

- Modern CPUs will do everything to ruin your benchmarking, so
- Disable frequency scaling
- Disable hyperthreading (and multiprocessing if you’re paranoid)
- Benchmark in kernel, so that you can disable preemption and interrupts

How to benchmark

```c
/* assume preemption and interrupts are off */
u64 benchmark(void)
{
    u64 start, end;
    int i;

    start = gimme_time();
    for (i = 0; i < N; i++)
            /* your function */ ;
    end = gimme_time();
    return (end - start - OFFSET) / N;
}
```

How to benchmark

```
/* assume preemption and inter
u64 benchmark(void)
{
    u64 start, end;
    int i;

    start = gimme_time();
    for (i = 0; i < N; i++)
          /* your function */
    end = gimme_time();
    return (end - start - OFFSET) / N;
}
```

OFFSET is how much time gimme_time() takes itself. For small N, e.g., 1, the error of OFFSET/N may be order[s] greater than the function call itself

How to compute OFFSET?

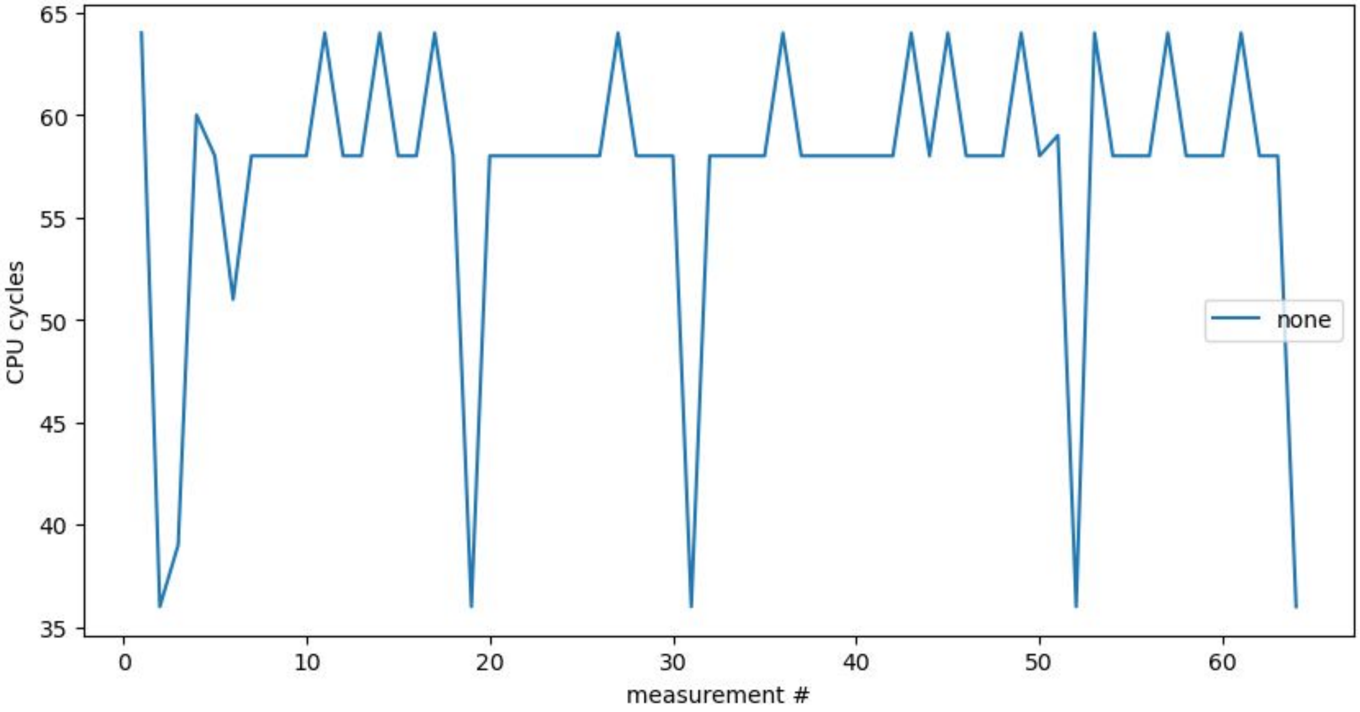
```
/* assume preemption and interrupts are off */
u64 benchmark(void)
{
    u64 start, end;
    int i;

    start = gimme_time();
    /* do nothing */
    end = gimme_time();

    return end - start;
}
```

Benchmark an empty loop

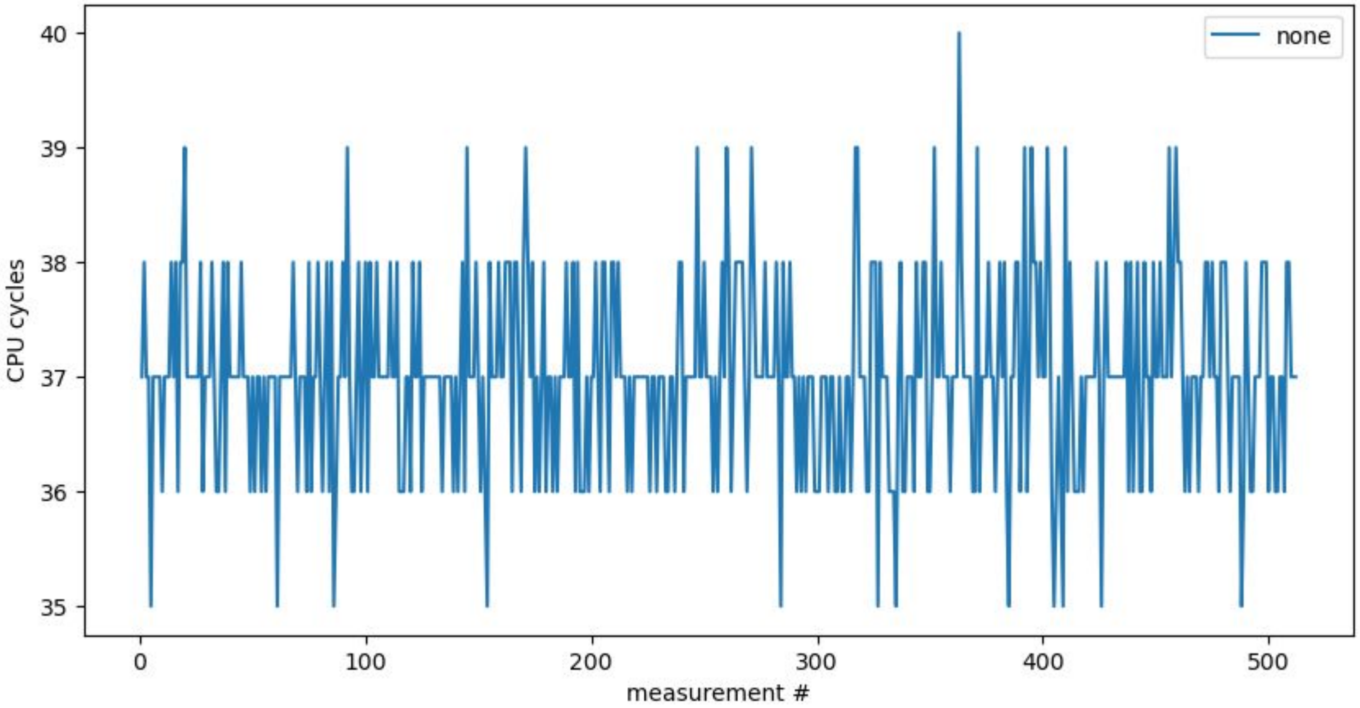
Let’s try with gimme_time=rdtsc

```
/* assume preemption and interrupts are off */
u64 benchmark(void)
{
    u64 start, end;
    int i;

    start = rdtsc();
    /* do nothing */
    end = rdtsc();

    return end - start;
}
```

Time sample = rdtsc, noise on



Time sample = rdtsc, noise off, better scale



+- 1 cycles looks ok for your case? Not so fast

```c
/* assume preemption and interrupts are off */
u64 benchmark(void)
{
    u64 start, end;
    int i;

    start = rdtsc();
    /* do nothing */
    end = rdtsc();

    return end - start;
}
```

+- 1 cycles looks ok? Not so fast

```c
/* assume preemption and interrupts are off */
u64 benchmark(void)
{
    u64 start, end;
    int i;

    start = rdtsc();
    /* do nothing */
    end = rdtsc();

    return end - start;
}
```

> The problem here is that rdtsc is not a serializing instructions and can be reordered. For example, it might be executing in the middle of your function or even after

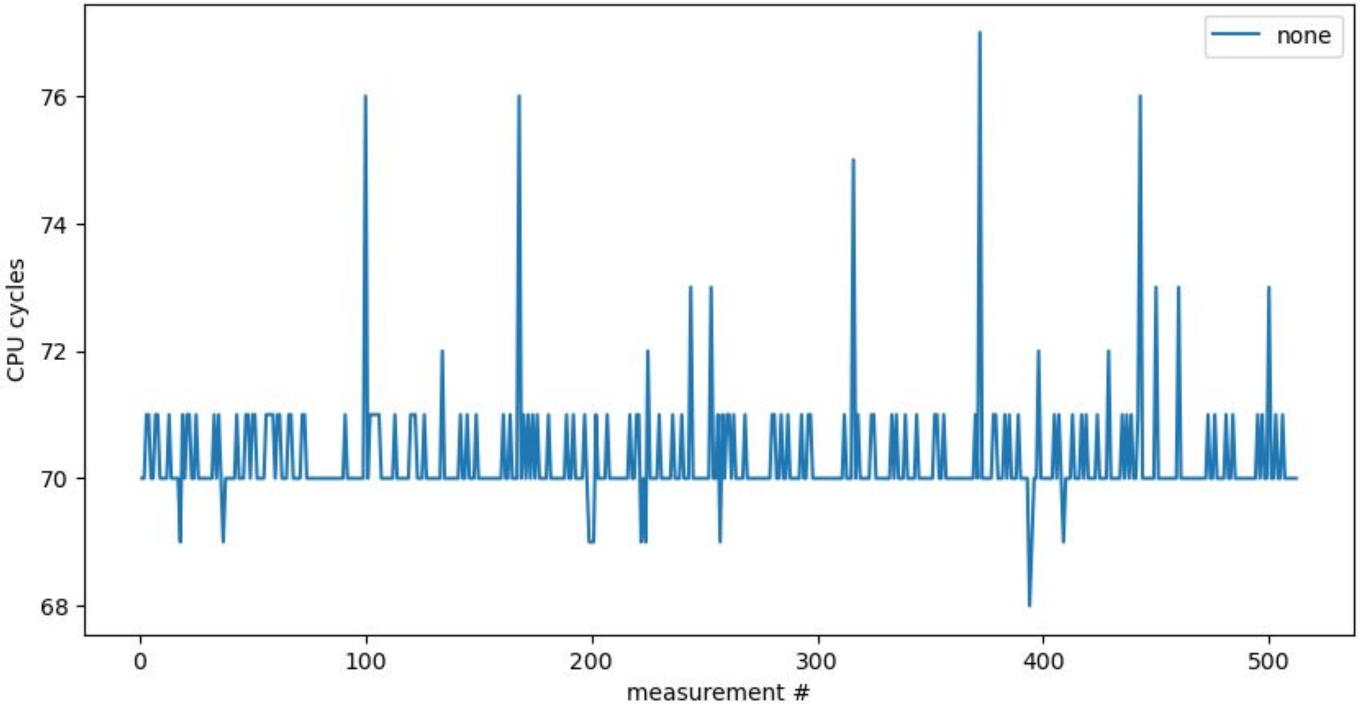
Serialize it!

```
#define time_sample_rdtscp_start() ({
        u32 low, high;

        asm volatile ("LFENCE\n\t"
                "RDTSC\n\t"
                "mov %%edx, %0\n\t"
                "mov %%eax, %1\n\t"
                : "=r" (high), "=r" (low)
                :: "%rax", "%rdx");

        low | (u64) high << 32;
})
```

```
#define time_sample_rdtscp_end() ({
        u32 low, high;

        asm volatile("RDTSCP\n\t"
                "LFENCE\n\t"
                "mov %%edx, %0\n\t"
                "mov %%eax, %1\n\t"
                : "=r" (high), "=r" (low)
                :: "%rax", "%rdx");

        low | (u64) high << 32;
})
```

* See the whitepaper by Gabriele Paoloni from Intel; I’ve replaced CPUID by LFENCE to deal with less regs

lfence+rdtsc+lfence (10x10 measurements)

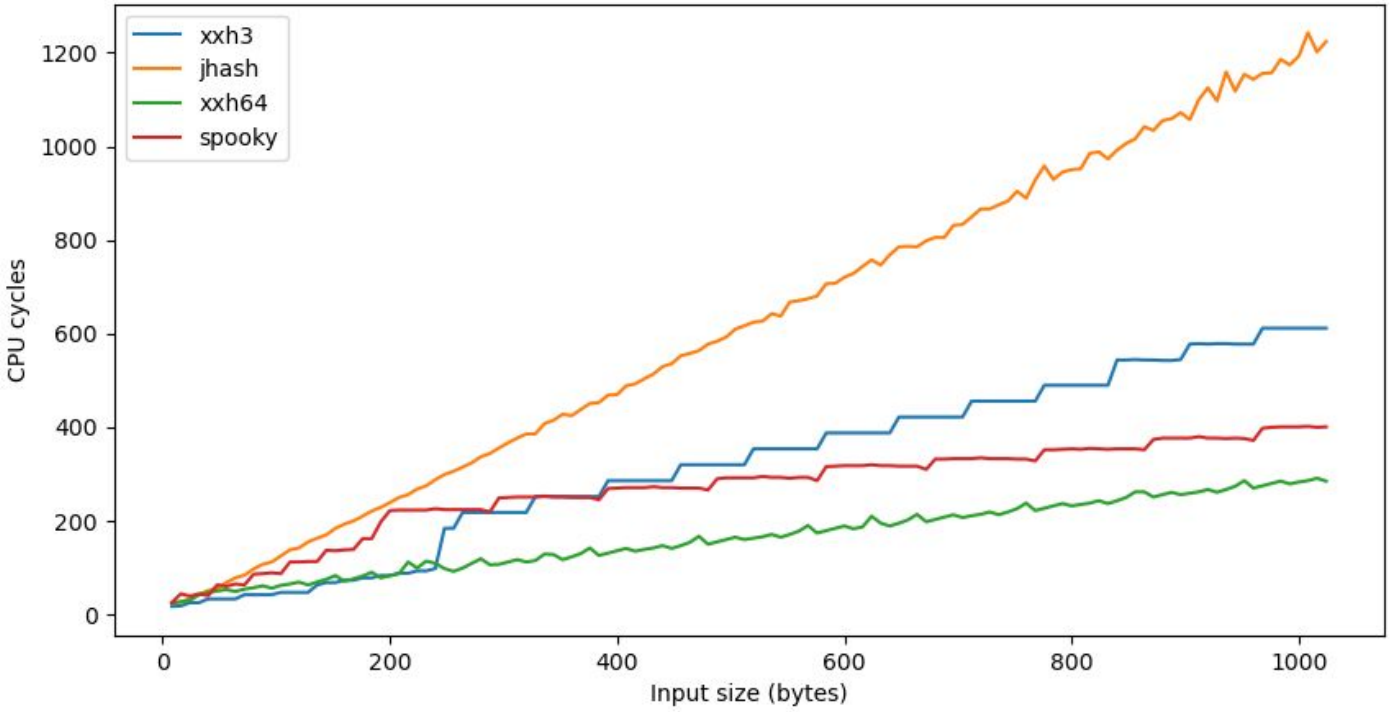


* See the whitepaper by Gabriele Paoloni from Intel; I've replaced CPUID by LFENCE to deal with less regs

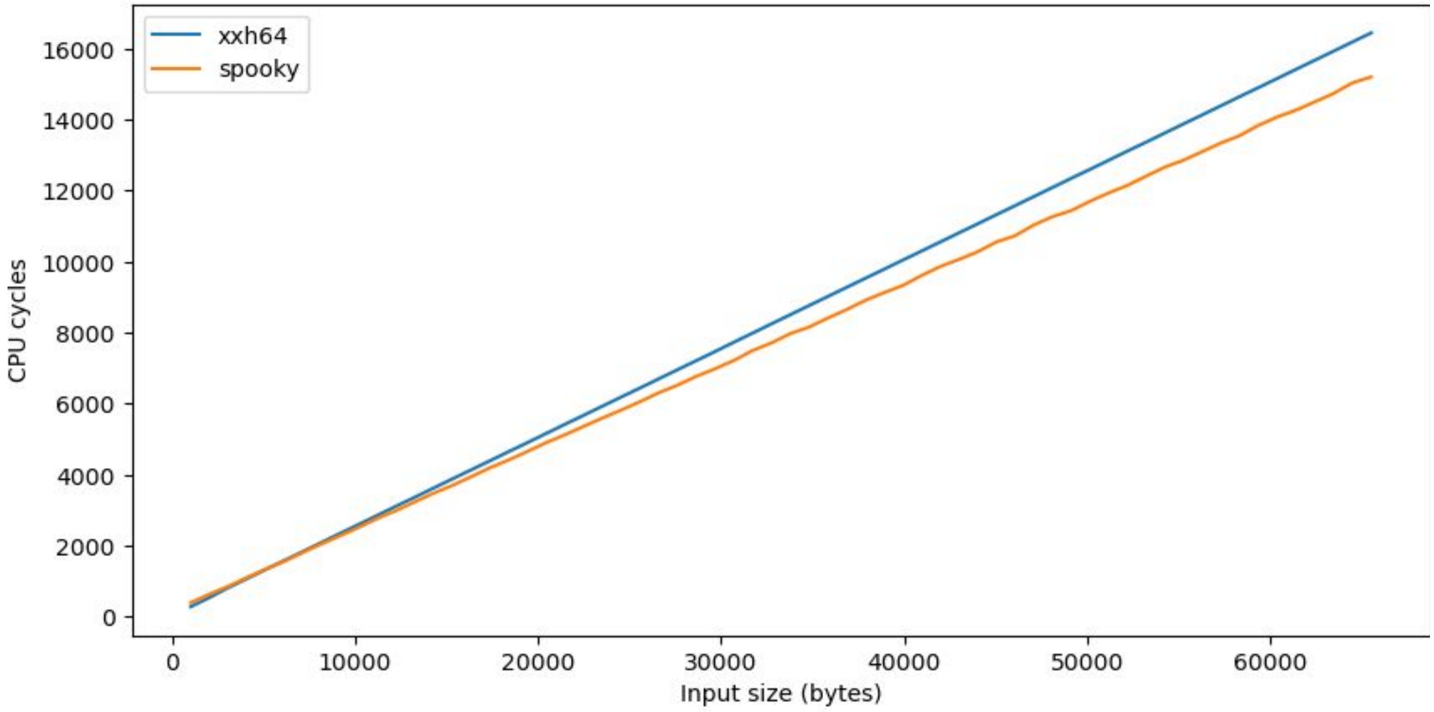
Hash functions of interest

- Jhash: Bob Jenkins hash, used in BPF
- Spooky hash: a newer hash by Bob Jenkins
- XXHash32, XXHash64: modern hash functions by Yann Collet
- XXH3: more modern hash by Yann Collet

Go spooky!

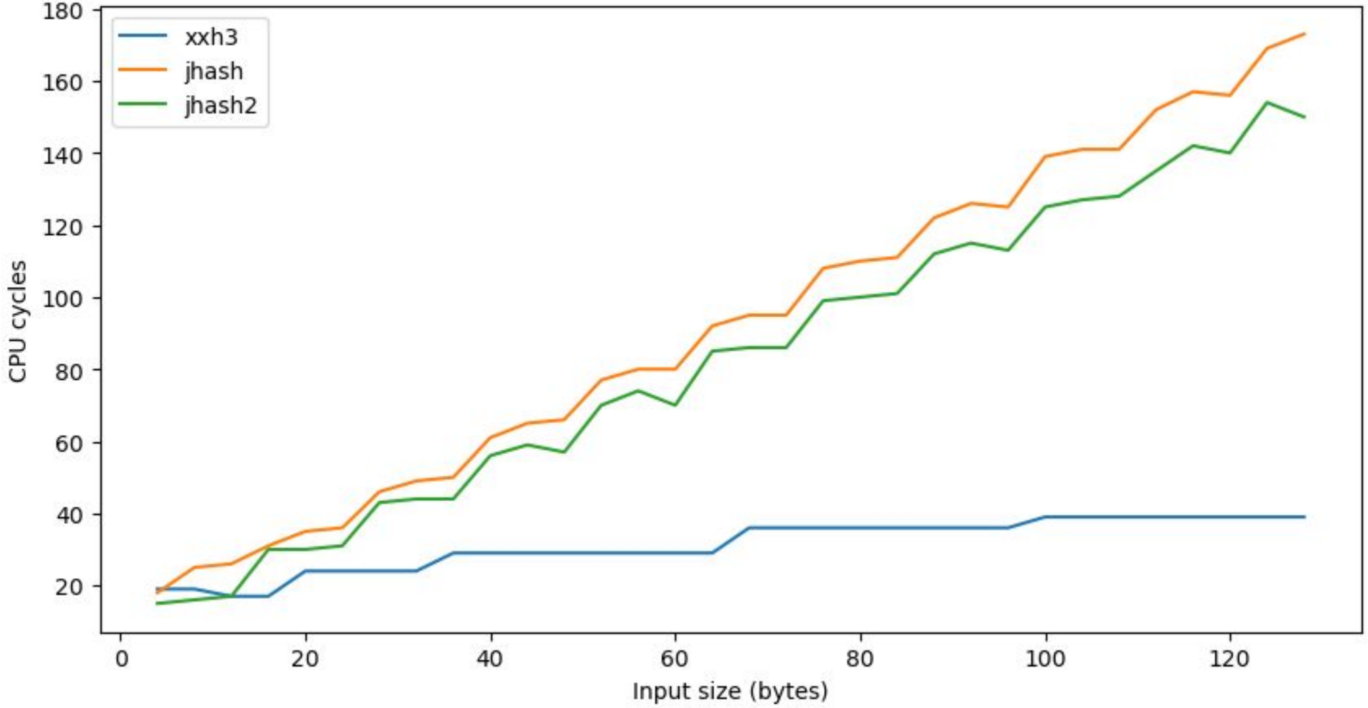

xxh3
jhash
xxh64
spooky
1200
1000
800
600
400
200
0
CPU cycles
0
200
400
600
800
1000
Input size (bytes)

Spooky wins!

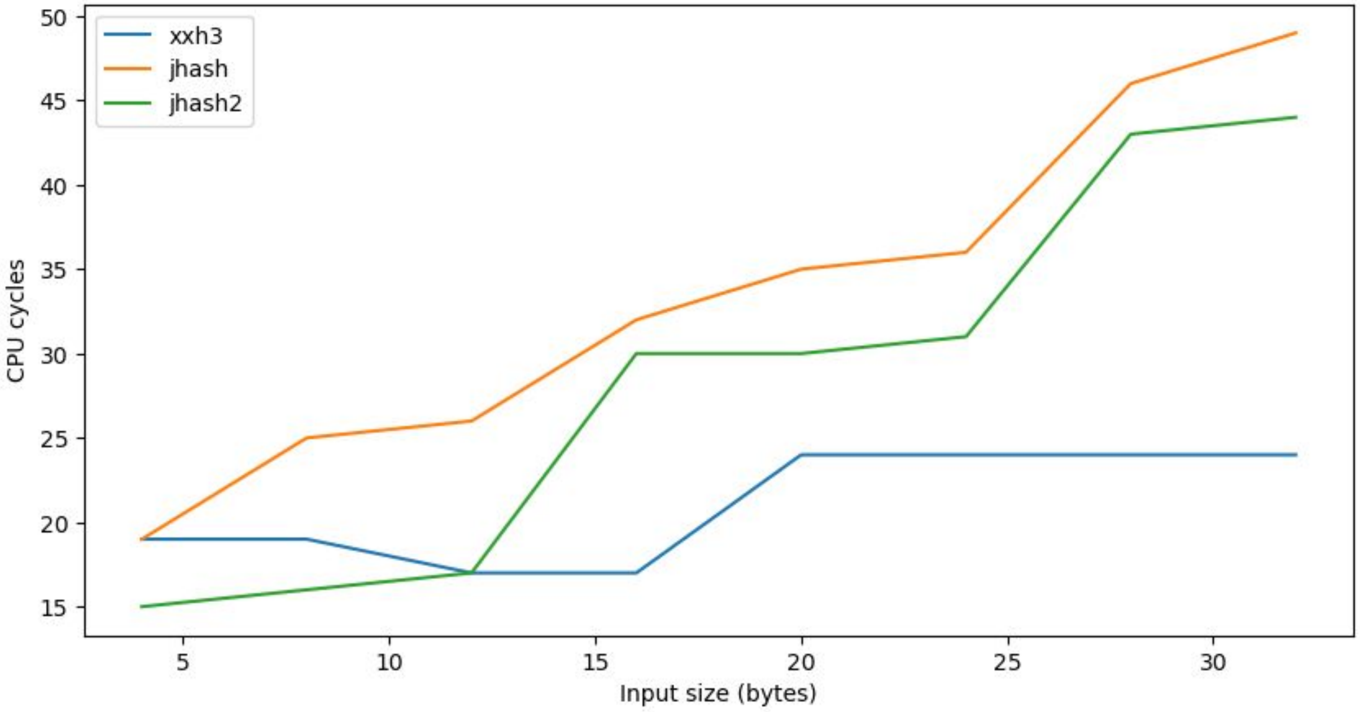


xxh64
spooky
16000
14000
12000
10000
8000
6000
4000
2000
0
CPU cycles
0
10000
20000
30000
40000
50000
60000
Input size (bytes)

xxh3 vs jhash



xxh3 vs jhash



Hash-based maps

- Stacktrace map: the original reason to use xxh3
- Hashmap
- Bloom filters

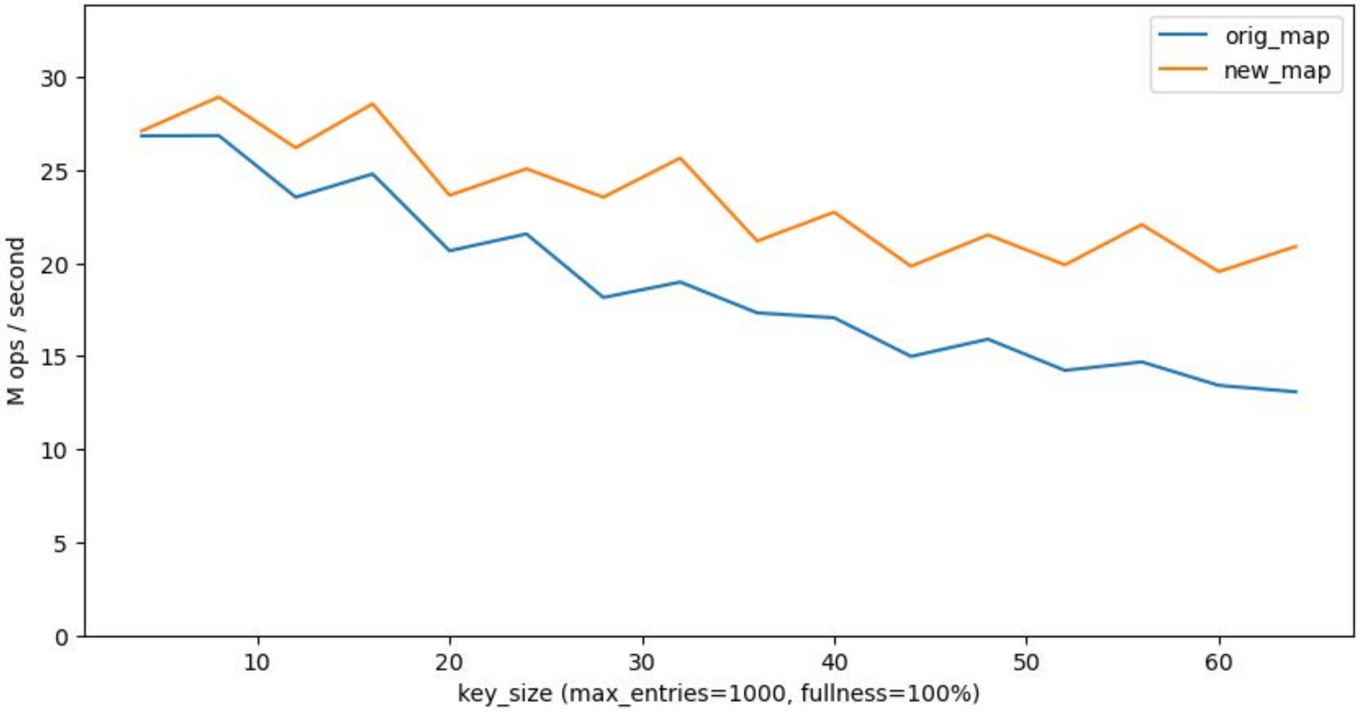
Stacktrace: why to use xxh3?

- The hash computations for stacktrace work about twice faster with xxh3 (as stacktrace keys are 8 x stack depth long)
- This doesn't affect the speed much, because get_perf_callchain() runs >> longer than hash
- However, xxh3 should be better when considering hash collisions
- For stacktrace [speed] benchmarking see my drafts one, two

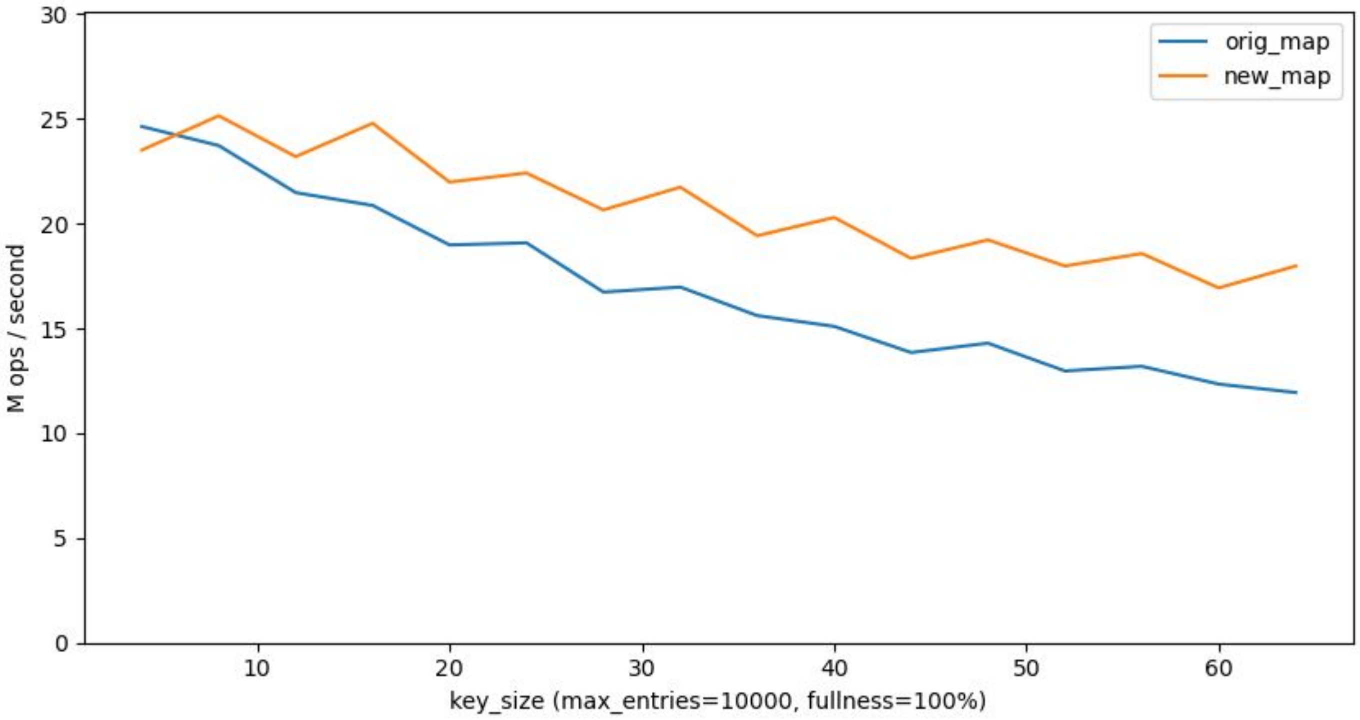
Hashmap benchmark

- I was primarily interested in lookup times, so used a new hashmap benchmark for bpf bench utility
- A lot of output, so I wrote scripts to plot the results

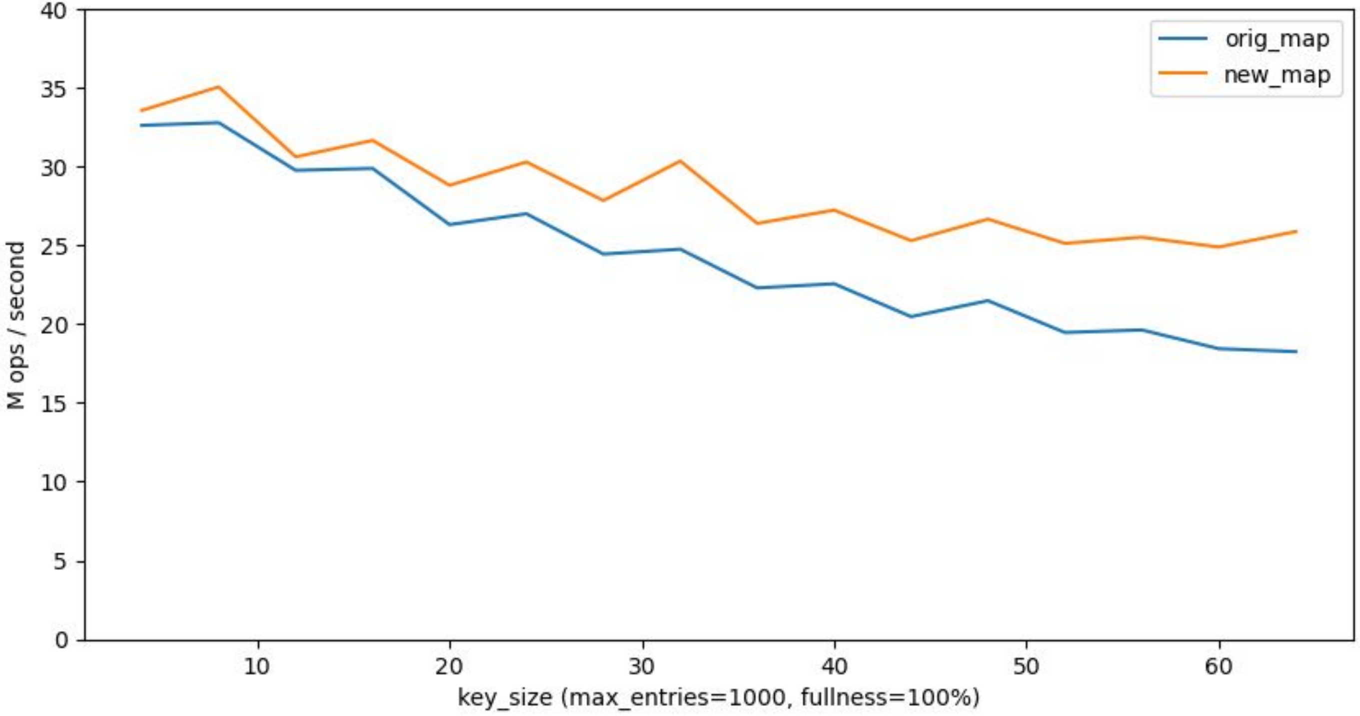
Hashmap (max_entries=1000, 100% full, Intel i7)



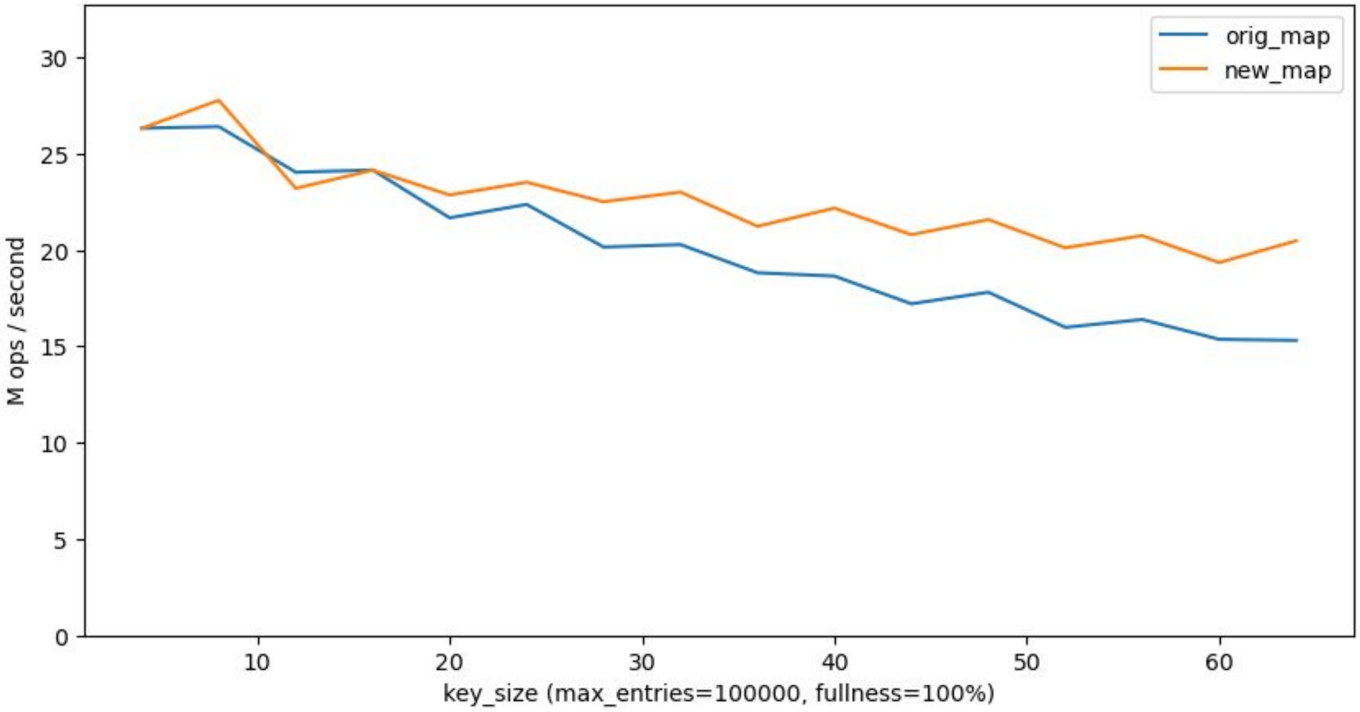
Hashmap (max_entries=10K, 100% full, Intel i7)



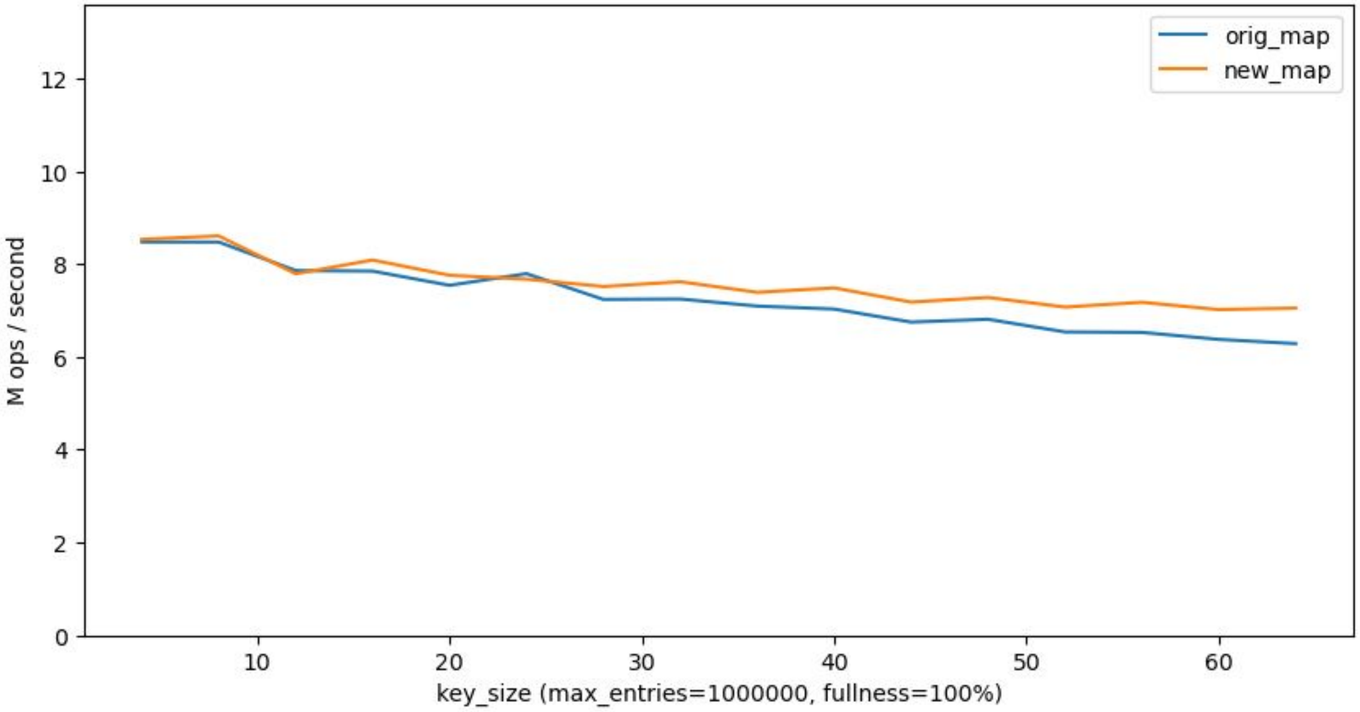
Hashmap (max_entries=1000, 100% full, Ryzen 9)



Hashmap (max_entries=100K, 100% full, Ryzen 9)



Hashmap (max_entries=1M, 100% full, Ryzen 9)



Hashmap: composite hash

- I’ve used the same trick as in Bloom filter: just use jhash2 for key sizes which are divisible by 4
- How to combine jhash2 and xxh3? Use jhash2 for small keys which are multiple of 4, and xxh3 otherwise

Hashmap: composite hash

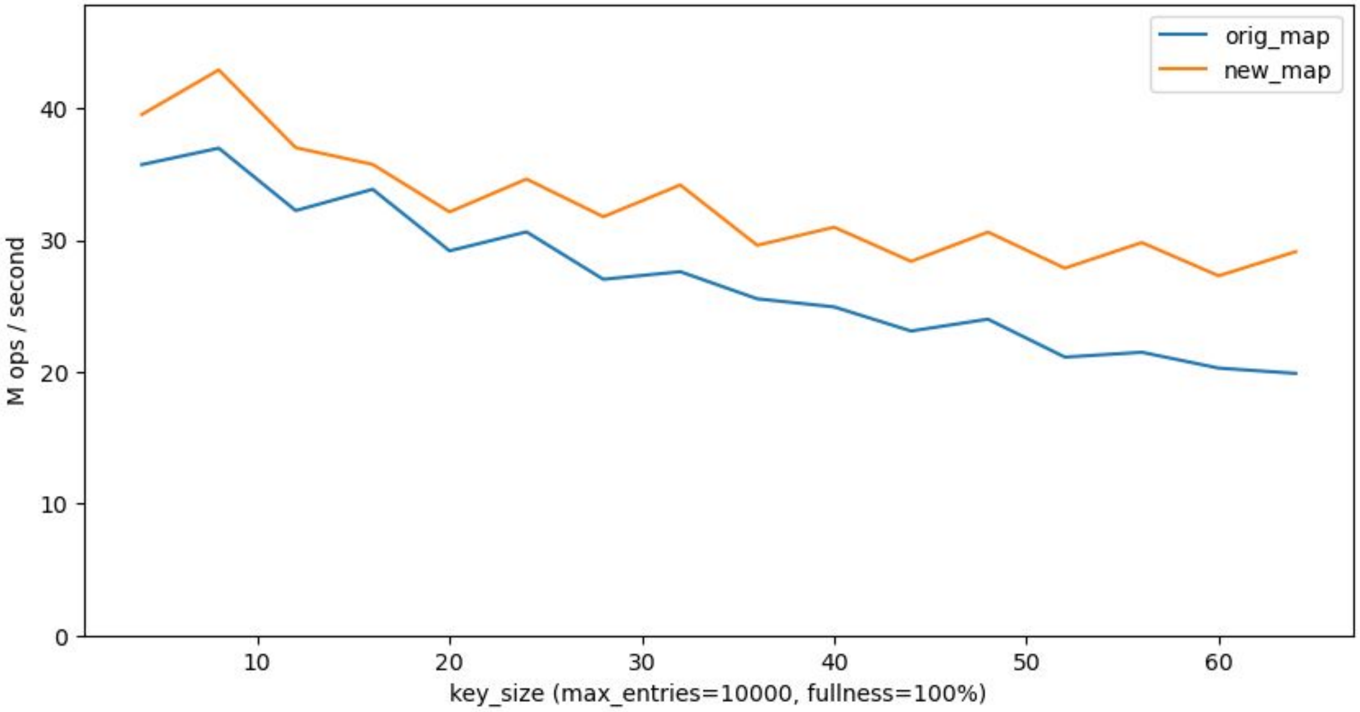
```
static inline u32 htab_map_hash(struct bpf_htab *htab,
                                const void *key,
                                u32 key_len)
{
        if (htab->key_len_32)
                return jhash2(key, htab->key_len_32, htab->hashrnd);
        return xxh3(key, key_len, hashrnd);
}
```

Hashmap: composite hash

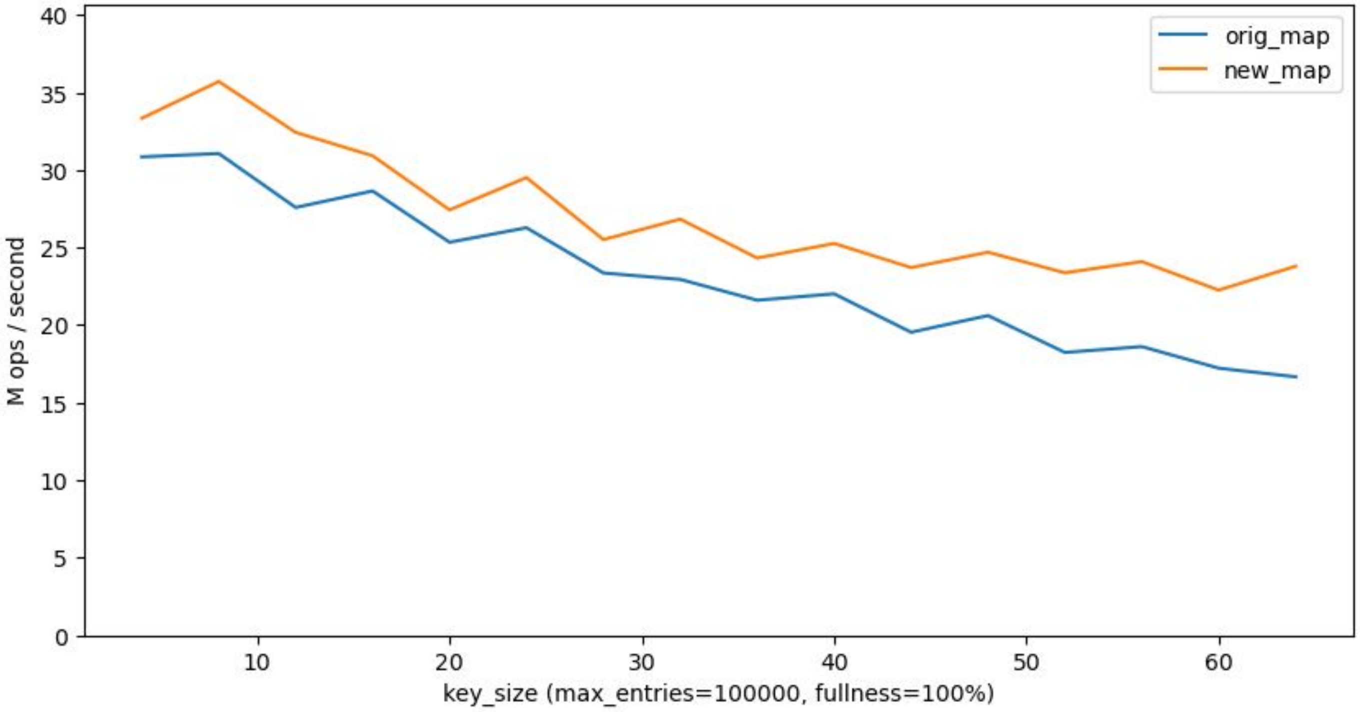
```
static inline u32 htab_map_hash(struct bpf_htab *htab,
                                const void *key,
                                u32 key_len)
{
        if (htab->key_len_32)
                return jhash2(key, htab->key_len_32, htab->hashrnd);
        return xxh3(key, key_len, hashrnd);
}
```

The key_len_32 = key_len/4, and is computed once when hash is initialized

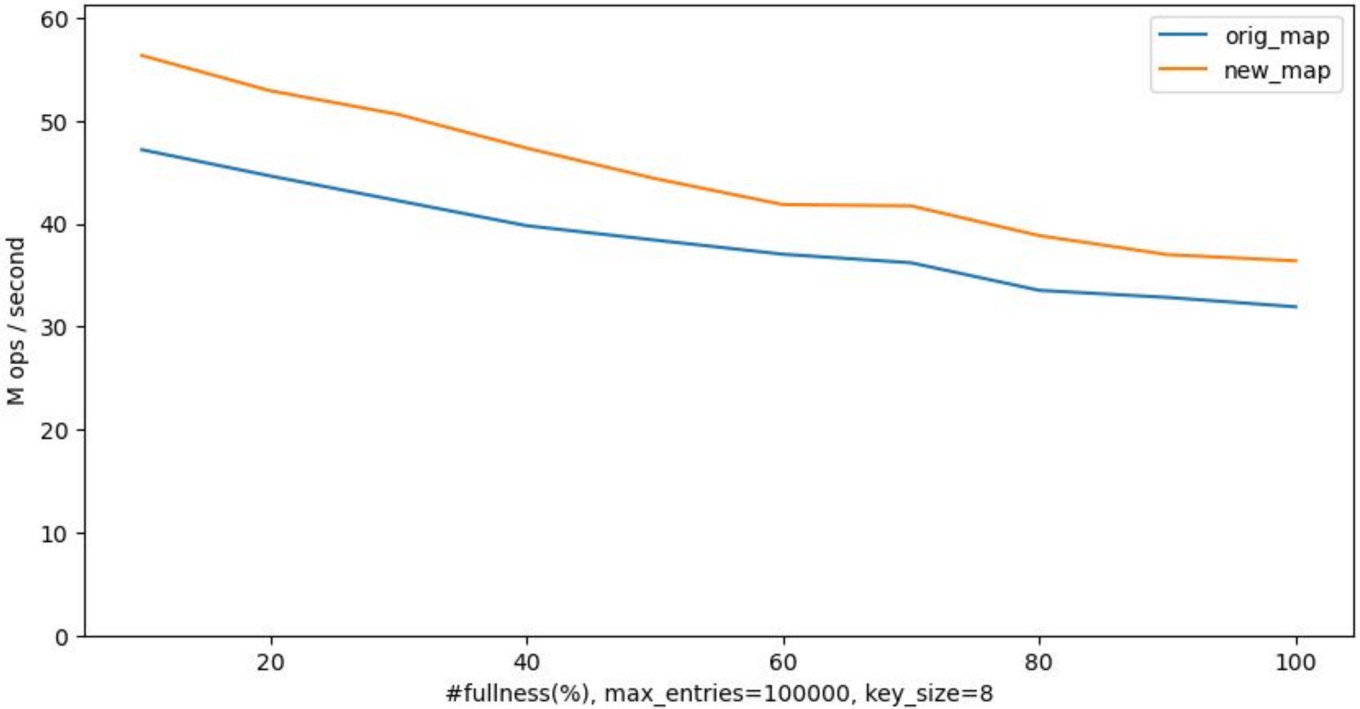
Hashmap: 10K, 100% full (worst case)



Hashmap: 100K, 100% full (worst case)

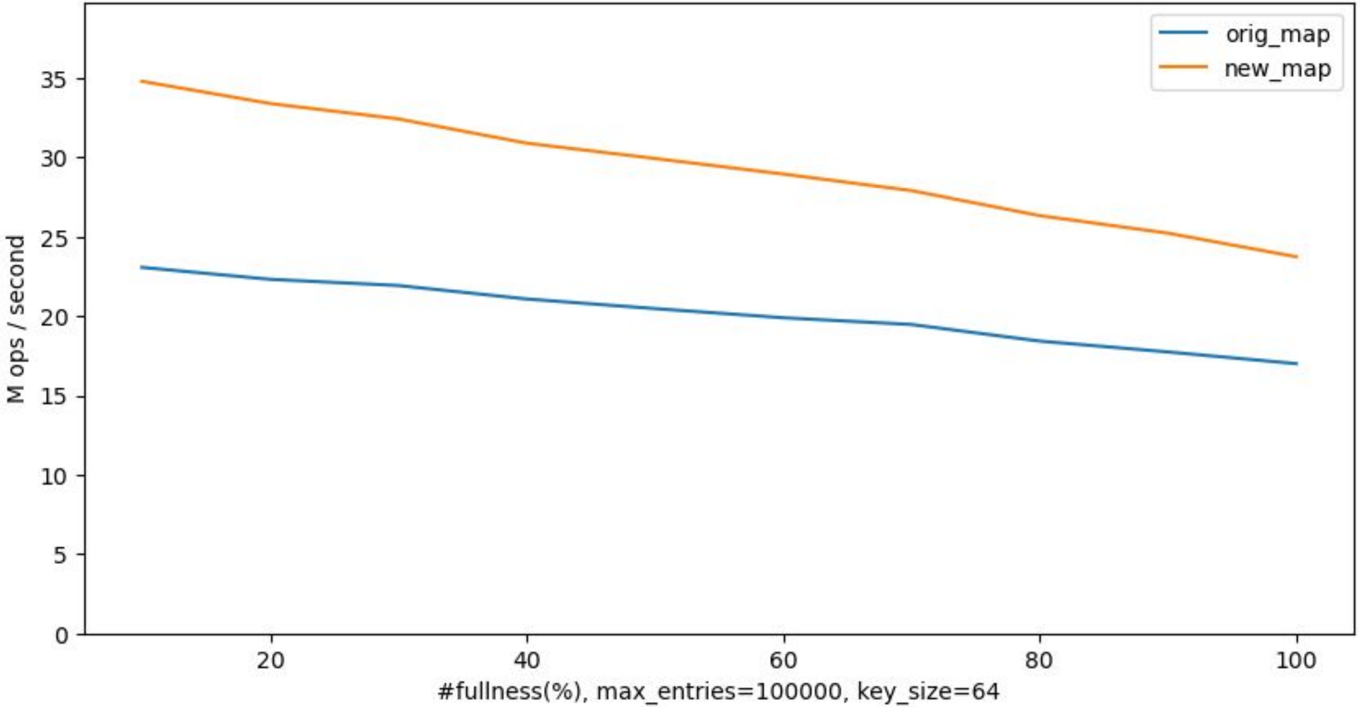


Hashmap: 100K, key_size=8

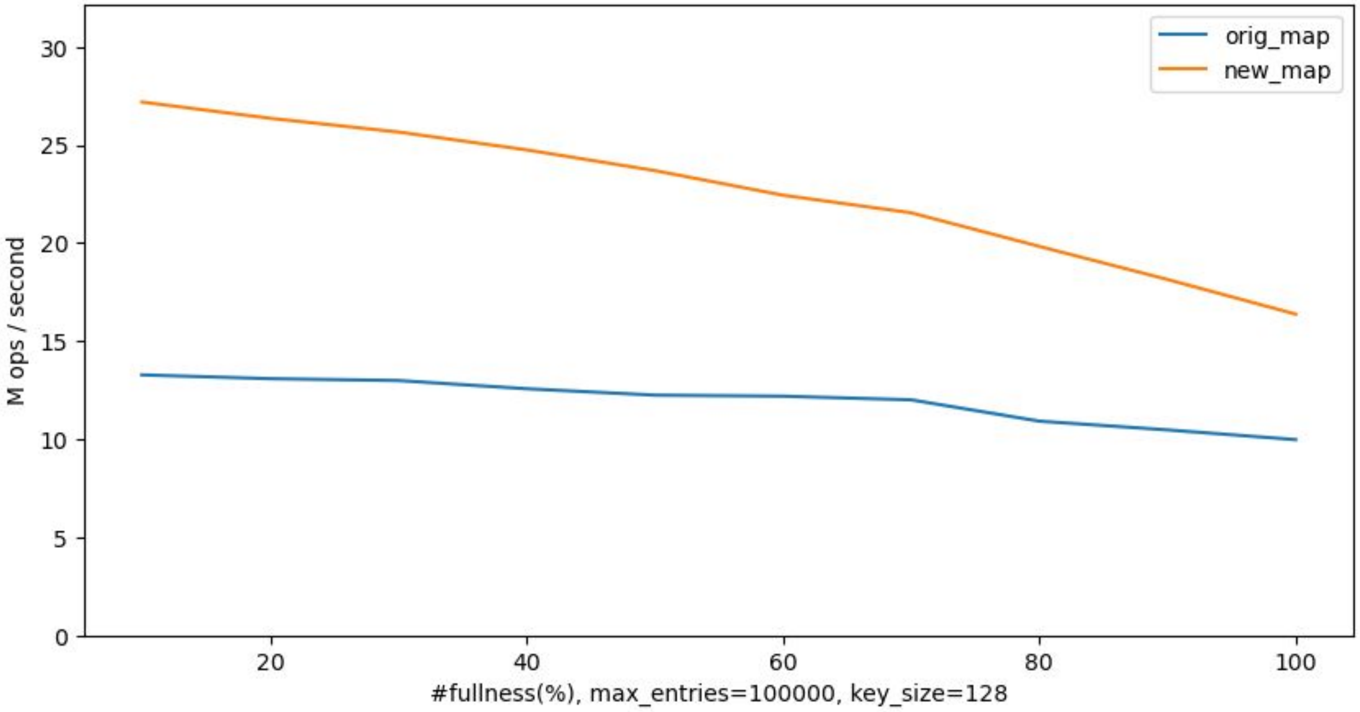


* **Tip:** always use key lengths divisible by 8 in BPF maps

Hashmap: 100K, key_size=64



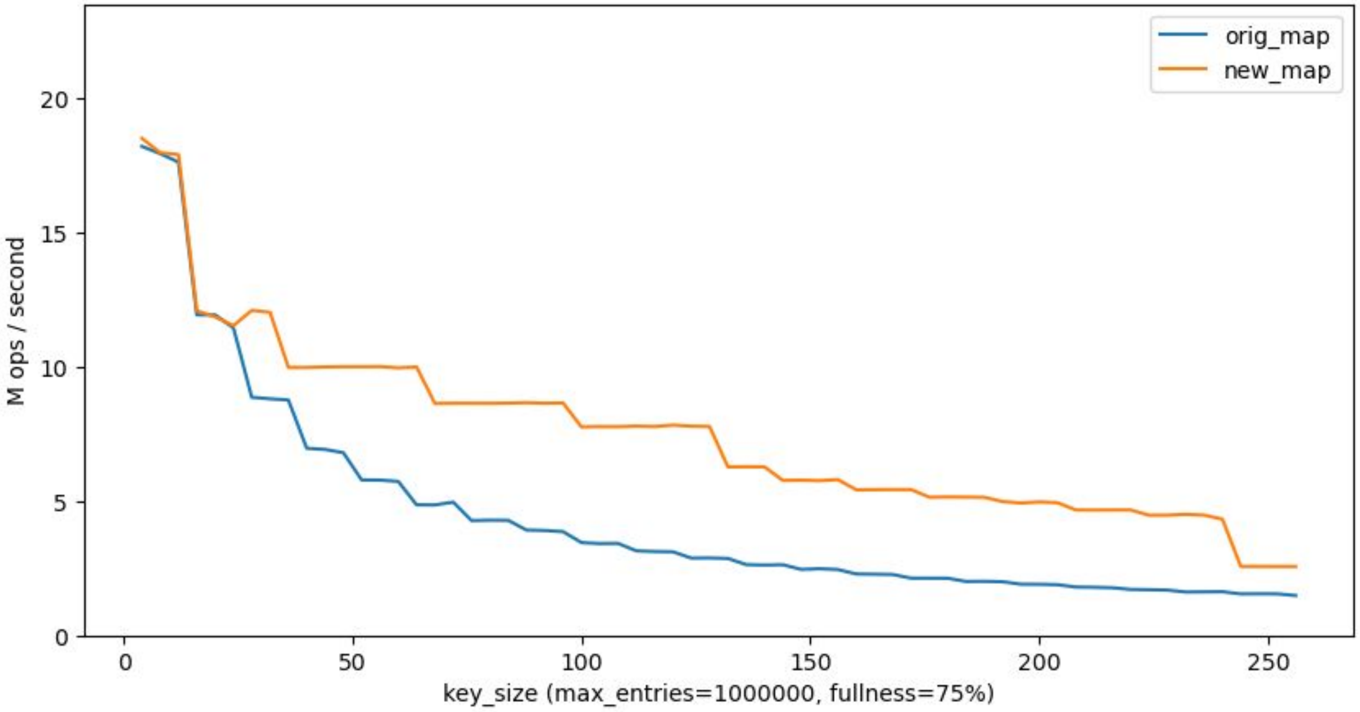
Hashmap: 100K, key_size=128



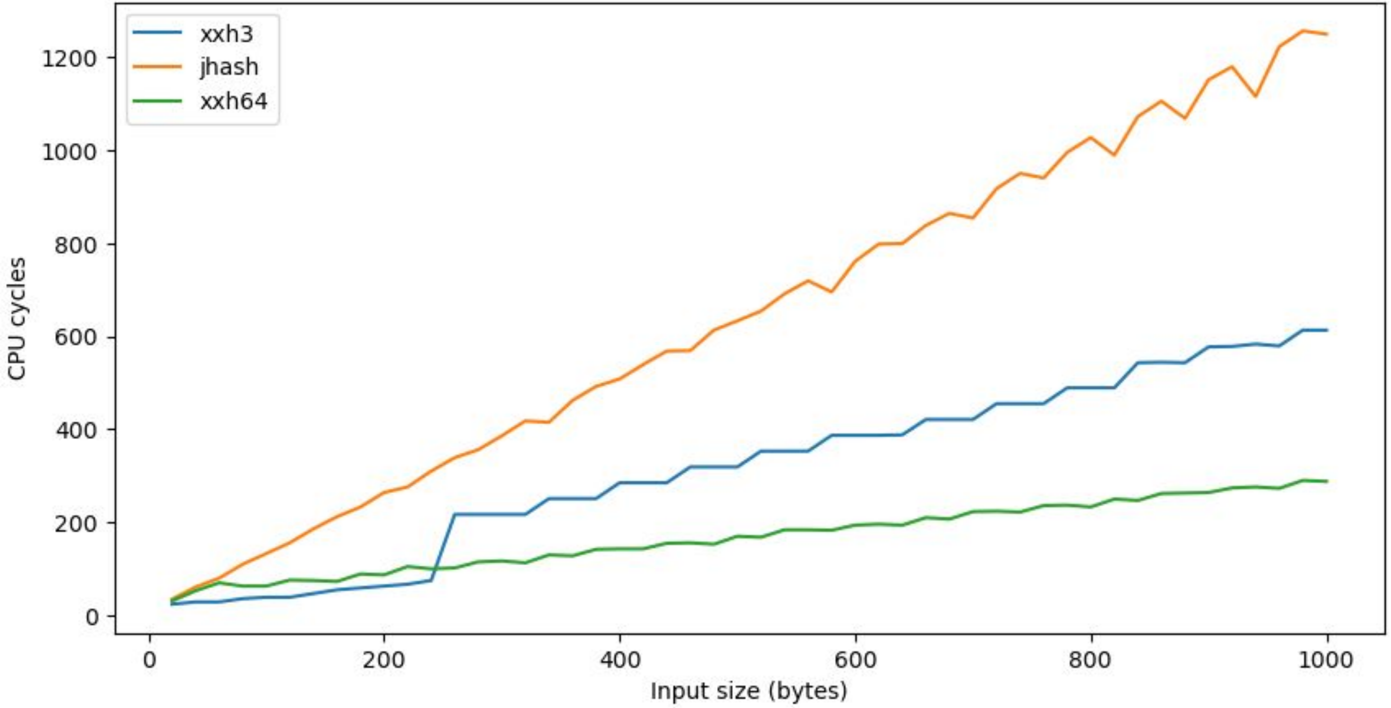
Bloom filters

- At the moment bloom filters use jhash2() for key sizes which are divisible by 4, and jhash() otherwise, so speed gain for small keys is not expected
- Anyway, let’s try to use the new hash function and see what happens

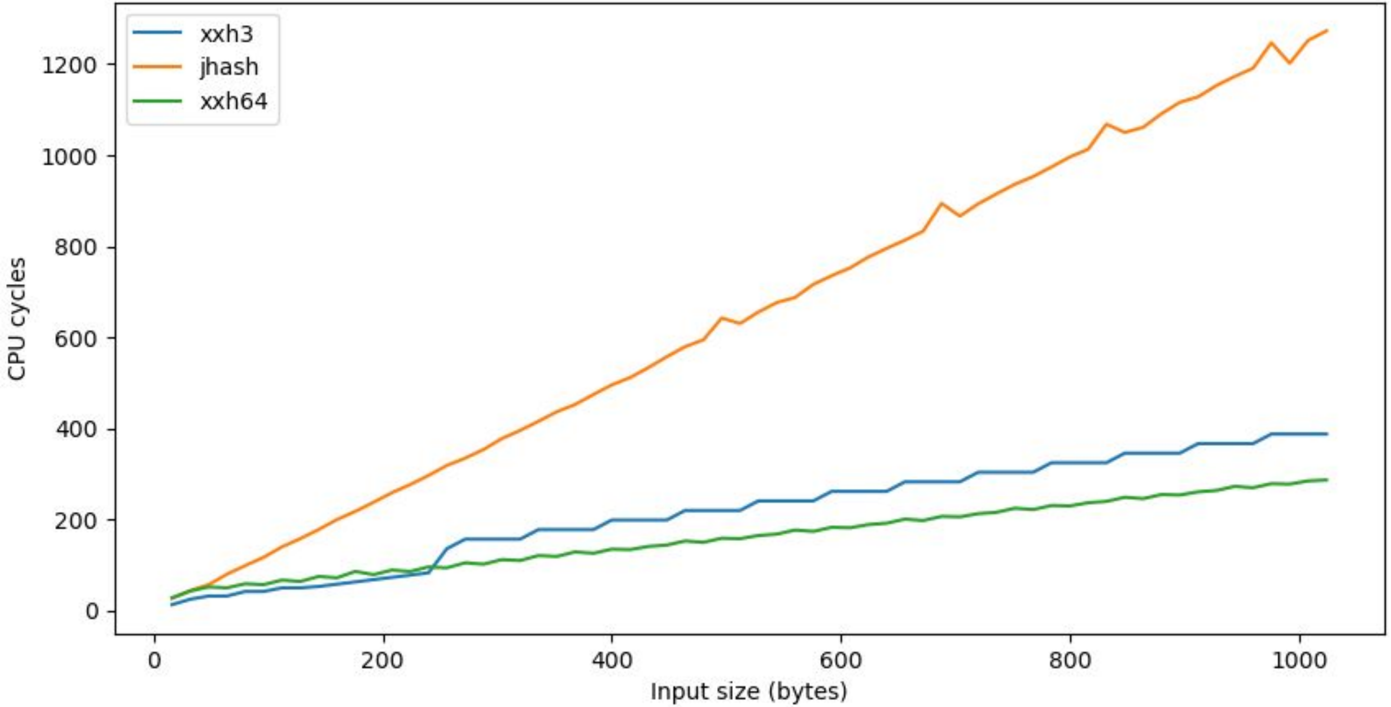
Bloom filter: 9 hashes, 1M elements, 75% full



Scalar xxh3 vs xxh64 for inputs > 240 bytes, -O2



Important: -O3 makes it all different*!



* … but -O3 is no go at the moment, see this thread

** See also this thread at github for benchmarks on different architectures made by Yann Collet

What’s next?

- Looks like the composite variant of hash is a good candidate for hashmap/Bloom filters, however, need to run my benchmarks on more architectures first [e.g., didn’t run on aarch64]
- The xxh3 looks ready to use for the stacktrace map [maybe after someone will actually “benchmark” the collision rate; I couldn’t see much difference on random inputs, but stack traces aren’t random, so xxh3 is expected to work better]

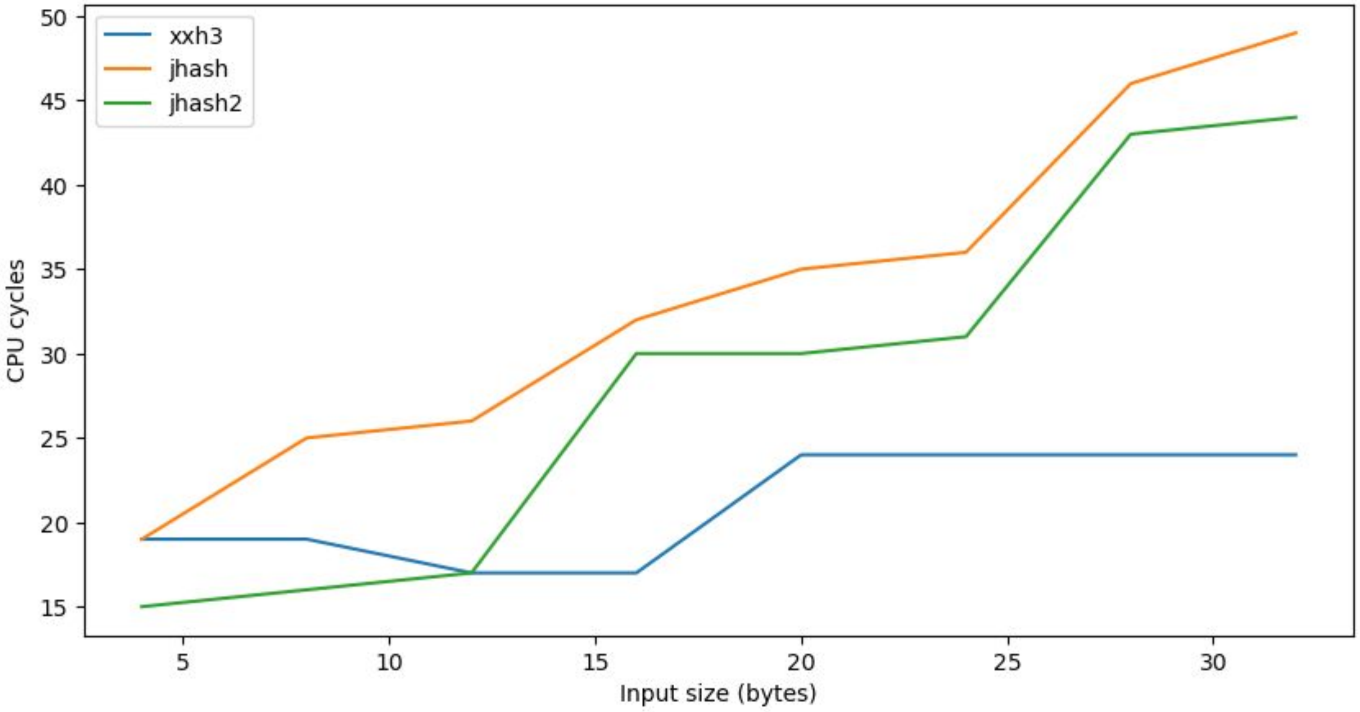
Links to some benchmarks

- The scripts I've used to benchmark and plot hash functions and hash maps are here
- The whitepaper from Intel is a good source on how to benchmark things which you can't execute in a loop
- [userspace] benchmarks from author of XX*hash
- Kernel: see the bench utility in tools/testing/selftests/bpf

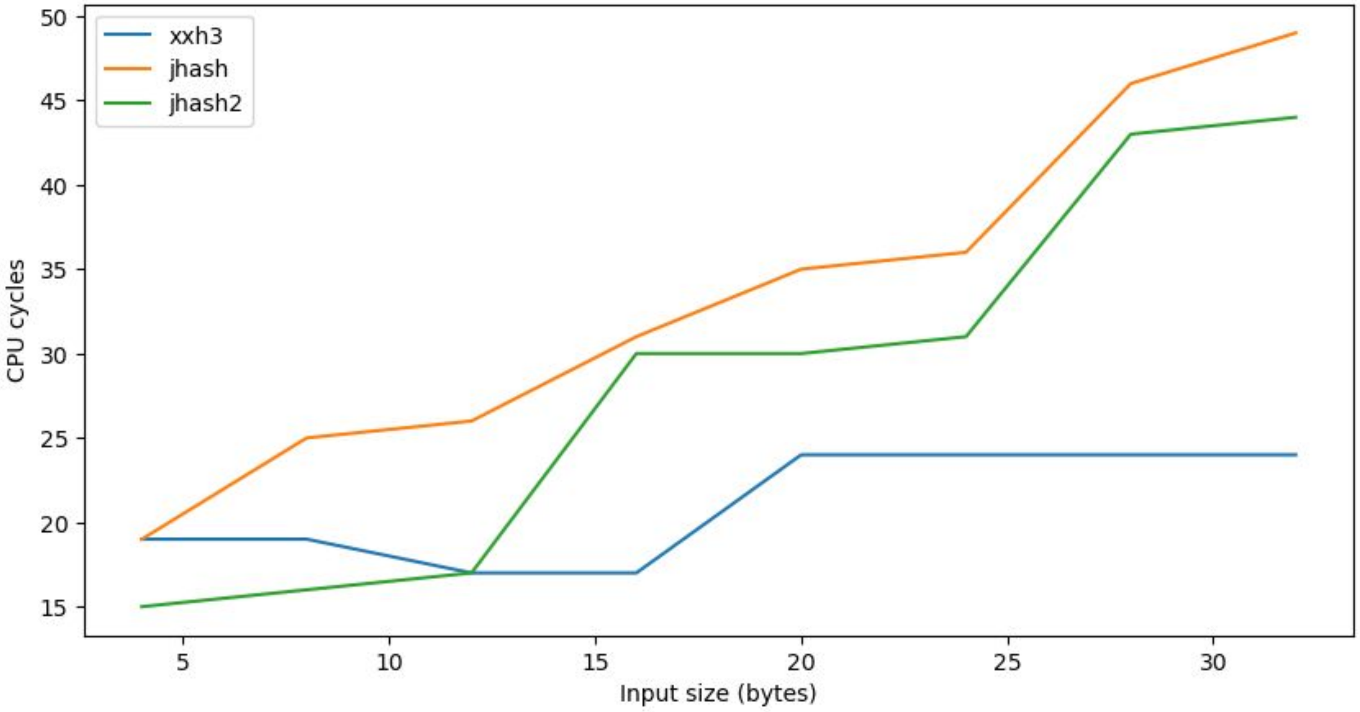
ISOVALENT

Thank you!

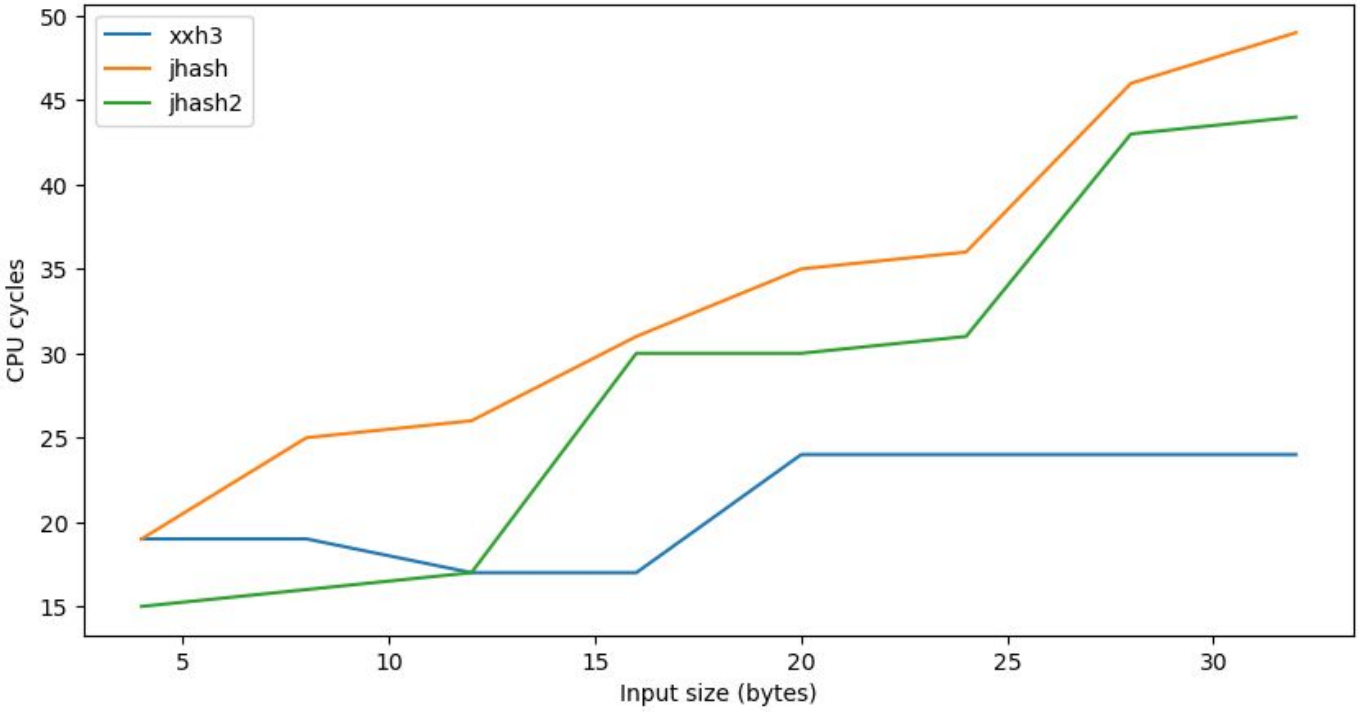
xxh3 vs jhash (how stable is our bench, part1)



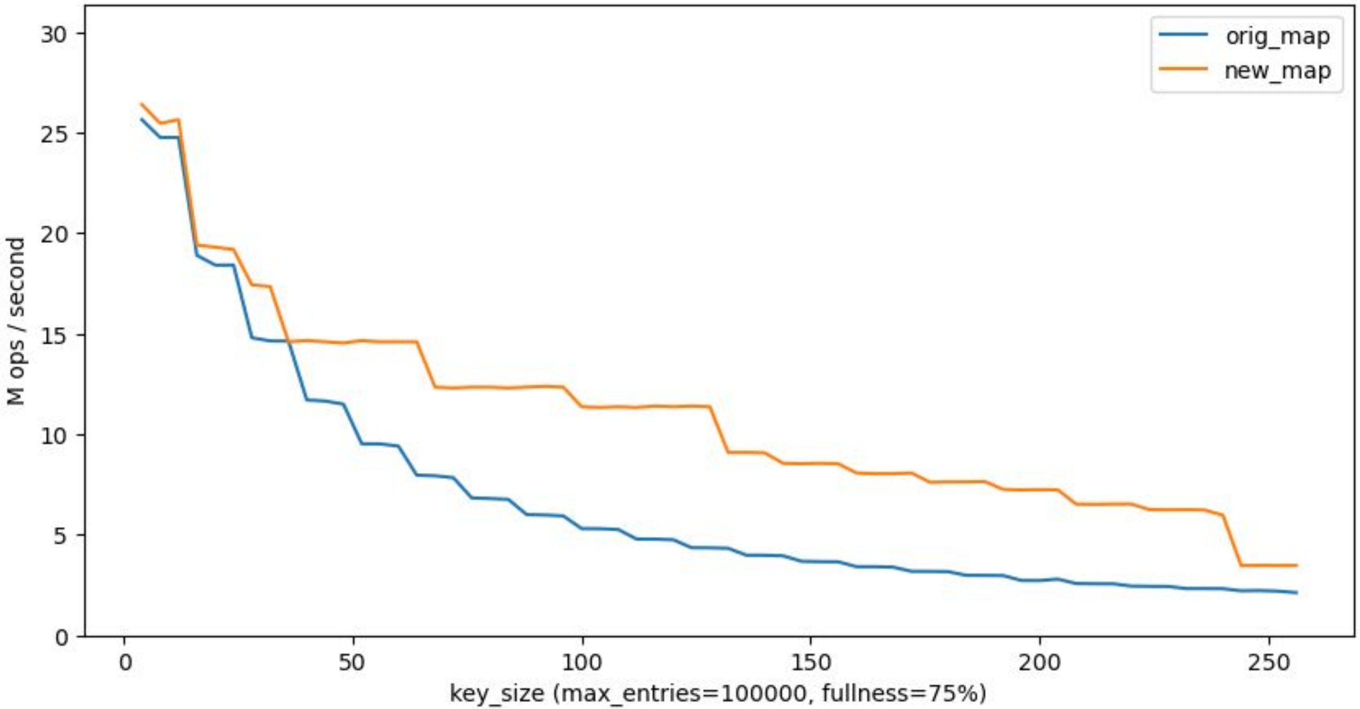
xxh3 vs jhash (how stable is our bench, part2)



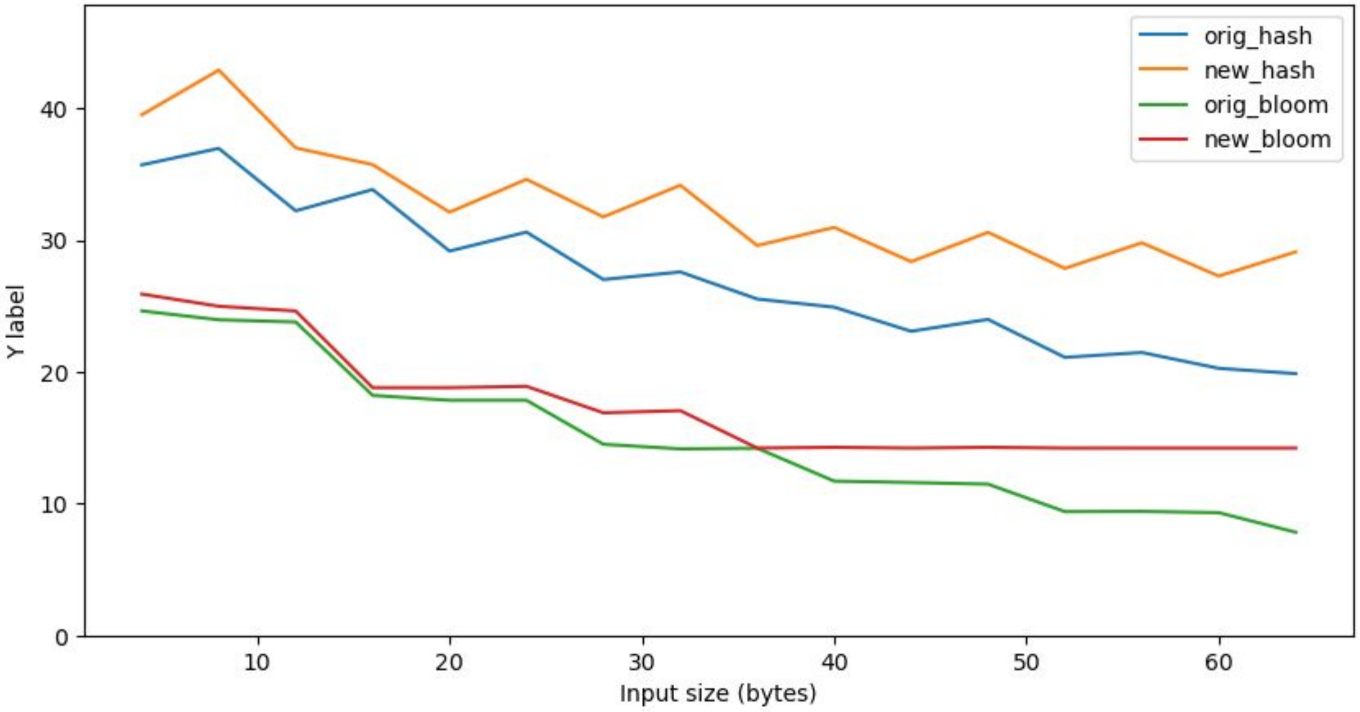
xxh3 vs jhash (how stable is our bench, part3)



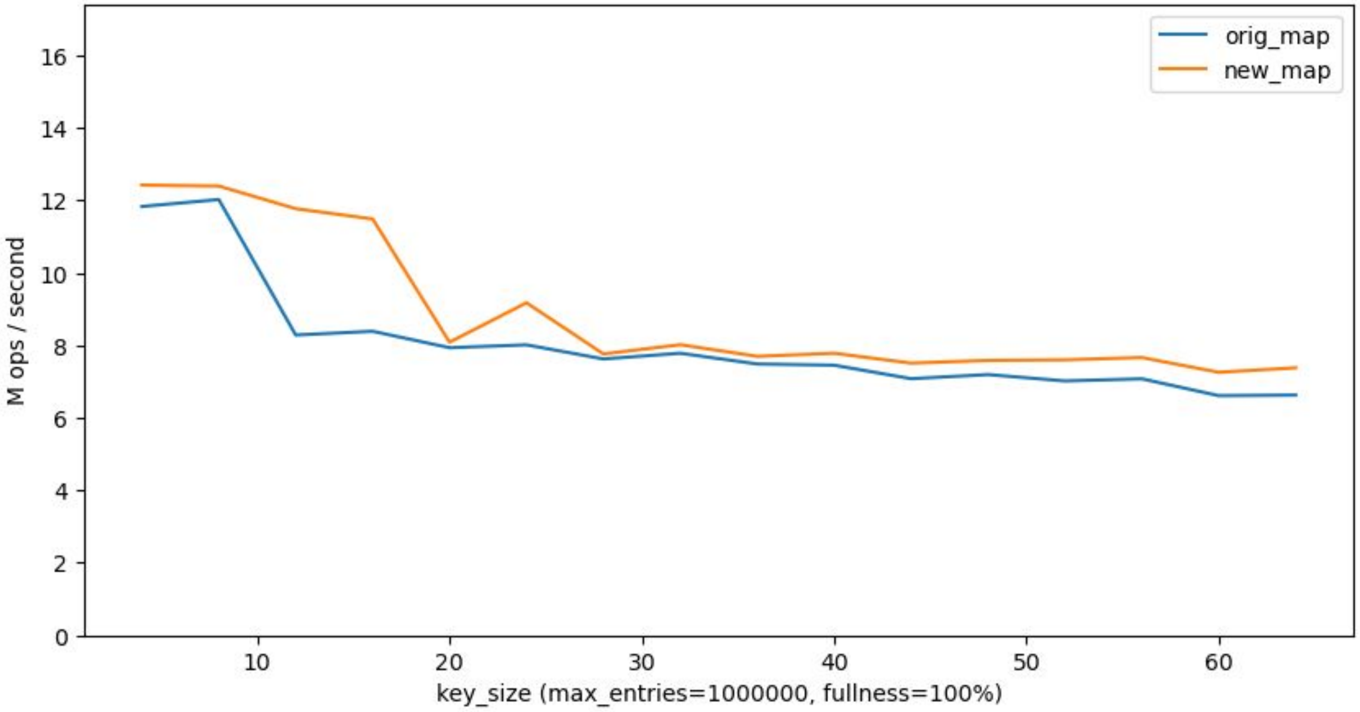
Bloom filter: 5 hashes, 100K elements, 75% full



Bloom filter 5 hashes vs. hashmap (10K, 100% full)



Hashmap: 1M, 100% full, see the next slide



The previous benchmark correlates to this one