Developing the mctop tool with eBPF

Dale Hamel

06/27/20 10:07:31 PM UTC

Contents

The estimated read time for this document is approximately one hour.

This document [1] is also available in epub and pdf format if you prefer.

You can contribute to this document on github by submitting a pull request, or filing an issue [2].

hot keys and mctop

The topic of hot keys in Memcached has been well-studied, and tools have existed to support this ecosystem since long before eBPF was on the scene.

An investigation into a cache hot-spotting problem lead to a eBPF-based redevelopment of the original `libpcap`-based `mctop` tool.

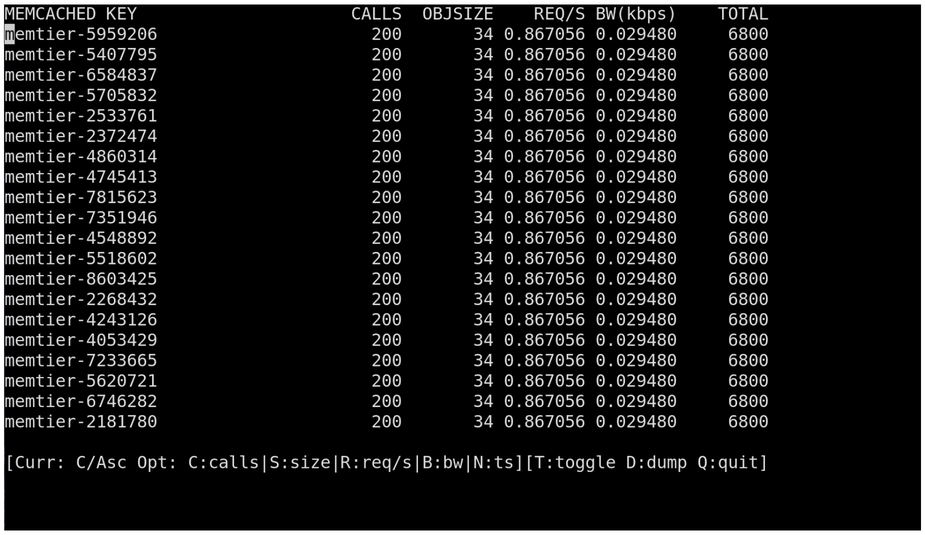
This report is verbose, and attempts to assume no advanced knowledge of eBPF, the `ELF` format, or Memcached itself. The referenced works can hopefully fill what gaps this report leaves.

mctop

The `mctop` tool was originally developed by etsy [3], and the author wrote an informative blog post [4] on the topic that motivated the development of the original tool. This concept was developed further by Tumblr in a similar tool, `memkeys` [5].

These tools both produced a top-like interface focussing on Memcached key access, with basic abilities to sort the data by column. Awareness of hot keys can inform application decisions of how best to utilize caching patterns under heavy load.

This is a screen capture of the redeveloped `mctop` tool built with eBPF and USDT tracing:



```
MEMCACHED KEY                      CALLS  OBJSIZE    REQ/S BW(kbps)    TOTAL
memtier-5959206                      200       34 0.867056 0.029480     6800
memtier-5407795                      200       34 0.867056 0.029480     6800
memtier-6584837                      200       34 0.867056 0.029480     6800
memtier-5705832                      200       34 0.867056 0.029480     6800
memtier-2533761                      200       34 0.867056 0.029480     6800
memtier-2372474                      200       34 0.867056 0.029480     6800
memtier-4860314                      200       34 0.867056 0.029480     6800
memtier-4745413                      200       34 0.867056 0.029480     6800
memtier-7815623                      200       34 0.867056 0.029480     6800
memtier-7351946                      200       34 0.867056 0.029480     6800
memtier-4548892                      200       34 0.867056 0.029480     6800
memtier-5518602                      200       34 0.867056 0.029480     6800
memtier-8603425                      200       34 0.867056 0.029480     6800
memtier-2268432                      200       34 0.867056 0.029480     6800
memtier-4243126                      200       34 0.867056 0.029480     6800
memtier-4053429                      200       34 0.867056 0.029480     6800
memtier-7233665                      200       34 0.867056 0.029480     6800
memtier-5620721                      200       34 0.867056 0.029480     6800
memtier-6746282                      200       34 0.867056 0.029480     6800
memtier-2181780                      200       34 0.867056 0.029480     6800

[Curr: C/Asc Opt: C:calls|S:size|R:req/s|B:bw|N:ts][T:toggle D:dump Q:quit]
```

Where other tools in this area use `libpcap`, the theory is[1] that using eBPF should offer performance advantages, as neither full or partial packet captures are necessary. Beyond this, the eBPF approach also has the advantage of inherently working both with the text-based and binary-based protocols, as no protocol interpretation is required.

[1] While this makes sense rationally, until it has been proven through a rigorous and scientific series of tests, measuring the overhead of both approaches under various conditions, it may not be the case. One possible drawback of the eBPF based approach is it causes some overhead as probes fire software-interrupts when triggered, which may not be the case with tcpdump, even if it is doing more processing.

Flash Sales

In the commerce-hosting business, there is a special class of merchants that run “flash-sales”. This is characterized by a huge number of visitors to a web storefront, followed (hopefully) by a lot of transactions to purchase whatever the newly-released or on-sale item is. These sorts of issues are especially notable for the employer of the author of this report, Shopify.

Success in a flash sale, unsurprisingly, depends heavily on being able to efficiently serve cached data. If a cache isn’t performing well, the sale won’t go well. Much of the contention in a flash sale is on the database. There are several caching strategies in place that protect requests from hammering the MySQL database instance for a Shopify Pod[2] of shops. By sharing access to a cache across a pool of web workers, all web workers within a Shopify Pod benefit from this large pool of surge capacity.

Despite optimization efforts, in some sales, there can be performance issues. Following on an investigation of a sale that didn’t go so well, we decided to perform some hot-key analysis on a (not real) test shop using a load testing tool. During these load tests, we developed some instrumentation with `bpftrace` to gain insight into the cache access patterns.

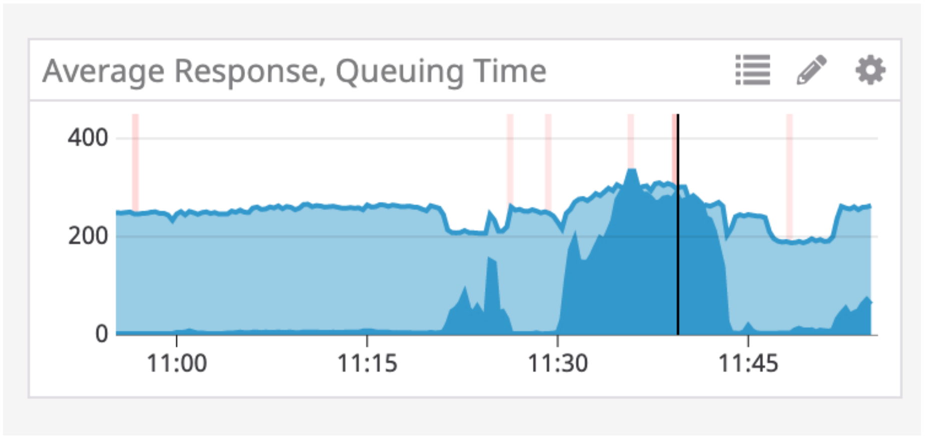
War Games

To make sure that we are testing our systems at scale, platform engineering teams at Shopify set up “Red team / Blue team” exercises, where the “Red team” tries to devise pathological scenarios using our internal load-testing tools, used to simulate flash-sale application flows against the platform.

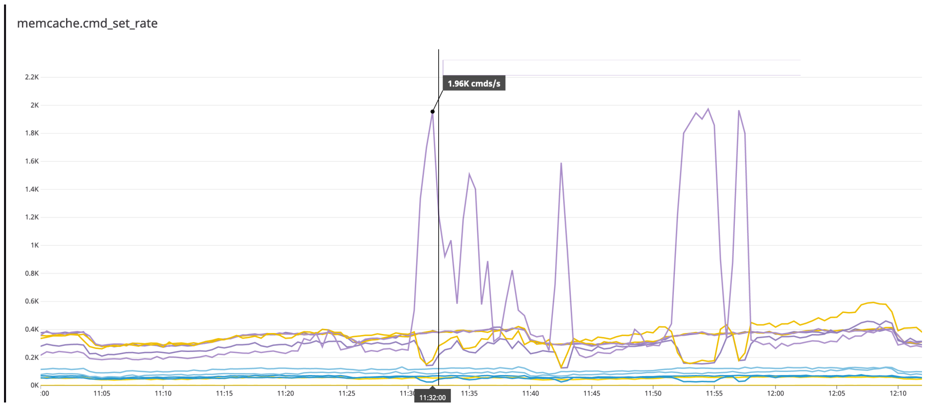
Meanwhile, the other “Blue team” monitors the system to investigate and mitigate any issues that may arise.

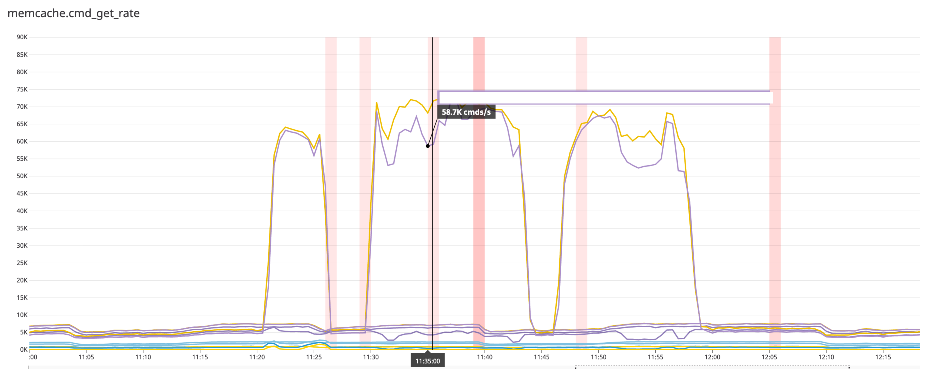
[2]a “Shopify Pod” is a distinct concept from a Kubernetes Pod, and it is an unfortunate and confusing naming collision. A Shopify Pod is a contained set of resources, built around the concept of MySQL sharding.

During one such exercise, my colleague Bassam Mansoob [6] detected that there were a few instances where a specific Rails cache-ring would be overloaded under very high request rates. This reflected conditions we had seen in real production incidents. Problems were first detected with our higher-level statsd application monitoring:



We could also see a large spike in the rate of GET/SET operations in this span:





To pinpoint the problem, we looked to eBPF tools for detecting the hot keys on the production Memcached instance we were examining in our Red/Blue exercise.

Hot key detection with bpftrace

We used `bpftrace` to probe the Memcached process that was targeted by our load-testing tool. For one cache we quickly found one extremely hot key using our first uprobe-based prototype[^3]:

```
@command[gets podYYYrails:NN::feature_rollout:percentages]:
↪  6579978
@command[delete podYYY:rails:NN::jobs-KEY ...]: 2854
@command[delete podYYY:rails:NN::jobs-KEY ...]: 3572
@command[gets podYYY:rails:NN::shop-KEY ...]: 5638
@command[set podYYY:rails:NN::KEY 1 30 13961]: 9266
```

It seemed like the cache entry used to determine the ratio of for a particular feature that should be enabled was a very hot key, as the same command was being hit at dramatically higher rates than other keys.

In our identity cache, used here for checking if feature flags for new code are enabled, we found keys that were being hit very frequently:

```
@command[gets podXXX::M:blob:Feature::FEATURE_KEY:SHOP_KEY_1]:
↪  67772
@command[gets podXXX::M:blob:Feature::FEATURE_KEY:SHOP_KEY_N]:
↪  67777
@command[gets podXXX::M:blob:Feature::FEATURE_KEY:SHOP_KEY_M]:
↪  6779
```

Having gained a quick view into what keys were especially hot, we could direct our mitigation efforts towards investigating the code-paths that were interacting with these keys.

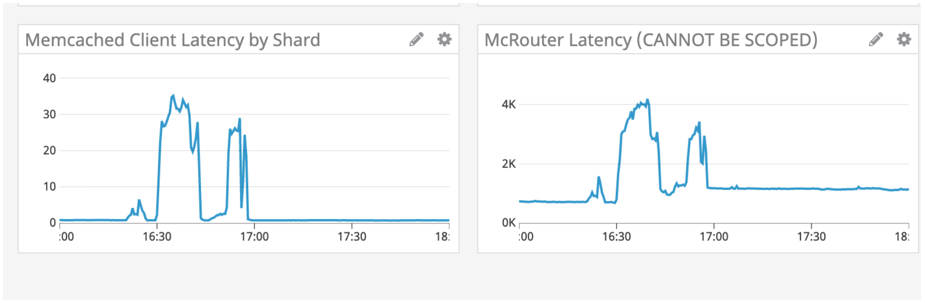
Hot key mitigation

Since these keys do not change very frequently, we decided to introduce an in-memory cache at the application layer inside of Rails itself. With a TTL of a full minute, it would hit Memcached much less frequently.

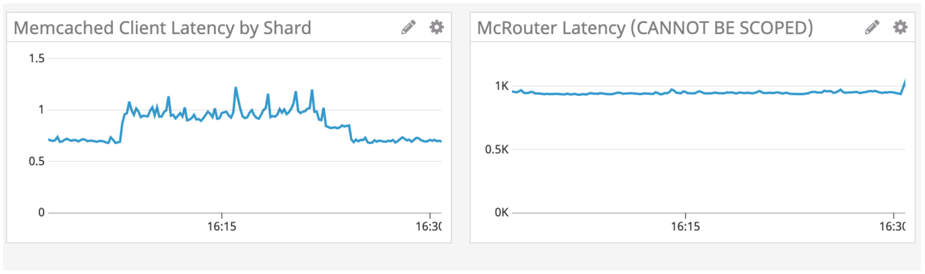
The change was simple, but the results were remarkable. Without the in-memory cache, there were large spikes on both Memcached, and the Mcrouter proxy.

Performance Results

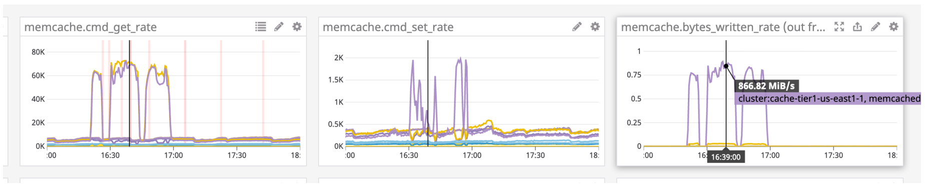
During these hot-spotting events from real or simulated flash sales, the impact without the cache is easy to spot:



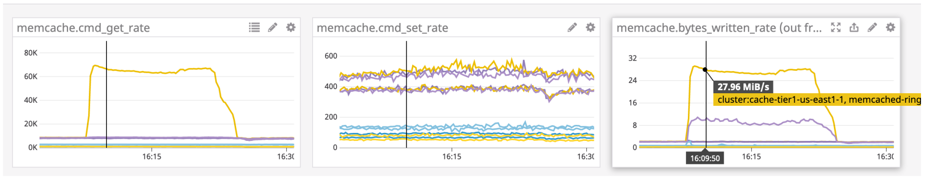
And with the in-memory cache, there was a substantial reduction in latency:



As for throughput, without the extra caching layer throughput to Memcached spiked:



And with the improvements from the in-memory cache, throughput was much lower as the new cache was not busted very frequently:



So a quick-and simple `bpftrace` one-liner was able to get pretty far towards resolving this problem!

Following this incident, the idea of making it easier to perform this type of investigation with a bespoke tool came about[3], and it was suggested to try and re-implement `mctop` in eBPF. This is what the remainder of this report will focus on.

[3] Jason Hiltz-Laforge and Scott Francis, put the idea in my head. Jason had suggested it to Scott, attempting to "nerd-snipe"[7] him, but Scott successfully deflected that onto me.

Probing memcached with bpftrace uprobes

One of the reasons we were able to deploy bpftrace so quickly to solve this issue was because we have distributed eBPF tools in production via a custom toolbox image since autumn of 2018, and have had `bpftrace` deployed to production along with the standard bcc tools.

At Shopify, `kubectl-trace` is standard issue to anyone with production access. This makes it easy for developers to probe their applications, and the system faculties that support them. Developing this sort of tool library allows for easily applying purpose-built analysis tools to investigate production issues.

This brings into reach tools that would otherwise be too scary or inaccessible, like kernel `kprobes` and `uprobes`. `bpftrace`, in particular, allows for simple and concise probe definitions, and is great for prototyping more complex tools, and poking around to find useful data sources.

For this issue, `bpftrace` has the ability to target any ELF binary with `uprobes` and read method calls and returns for an application like Memcached. This was the first entry-point into investigating the Memcached key access patterns.

memcached sources

Camilo Lopez [8] came up with the idea to attach a uprobe to the `process_command` function in Memcached. In the Memcached source code, the signature in `memcached.c` shows the argument types and order:

```
static void process_command(conn *c, char *command) {
```

This shows us that the second argument (`arg1` if indexed from 0) is the command string, which contains the key.

Now that we know the signature, we can verify that we can find this symbol in the Memcached binary:

`objdump-tT /proc/PID/root/usr/local/bin/memcached | grep process_command -B2 -A2`

4

Which shows us that it is indeed a symbol we can access:

```
...
00000000000155a5 l     F .text  0000000000000337
↪  process_lru_command
00000000000158dc l     F .text  00000000000003b8
↪  process_extstore_command
0000000000015c94 l     F .text  00000000000012ac process_command
↪  <--- Target
000000000001799a l     F .text  000000000000019d try_read_udp
0000000000017b37 l     F .text  00000000000002c7 try_read_network
...
```

This is how `bpftrace` will target the probe, by resolving the address of this symbol in the code region of the target process' memory space.

uprobe prototype

To probe read the commands issued to Memcached, we can target the binary directly[5], and insert a breakpoint at this address. When the breakpoint is hit, our eBPF probe is fired, and bpftrace can read the data from it.

The simplest solution and first step towards a more sophisticated tool is to just read the command and print it which can easily be done as a `bpftrace` one-liner:

```
bpftrace -e
↪  'uprobe:/proc/PID/root/usr/local/bin/memcached:process_command
↪  { printf("%s\n", str(arg1)) }'
```

6

Then running a test command on Memcached generates probe output! This shows that `bpftrace` can read data from user-space using the kernel's `uprobe`

[4]Using docker or crictl, we can find the container process and inspect its children to find the memcached process. This method then uses the `/root` handle to access the process's mount namespace, and read the exact instance of the memcached binary we want to probe.

[5]The initial prototype of the uprobe tool targeted the memcached binary directly, as while we were using a recent version of bpftrace (0.9.2), which ships with Ubuntu Eoan, it was linked with libbcc 0.8.0, which didn't have all of the USDT functionality and namespace support to read containerized processes correctly. For this reason

[6]This is not the ideal syntax and is a regression, container tracing is a bit working with USDT probes, as are uprobes. Specifying the full path from the /proc hierarchy seems to work well enough though.

faculties.

```
Attaching 1 probe...
set memtier-3652115 0 60 4
```

Now, to quickly turn this into a relatively useful tool, each time a key is hit it can be incremented in a dictionary. Using the `++` operator to count each time the command is called:

```
uprobe:/proc/896719/root/usr/local/bin/memcached:process_command
↪  /pid == 896719/
{
  @command[str(arg1)]++
}
```

When this exits, it will print a sorted map of the commands, which should correspond to the most frequently accessed keys.

With a working uprobe prototype, attention now turns to getting better quality data. `bpftrace` doesn't really have the faculty to parse strings at the moment and this is inherently pretty inefficient, and thus not something ideal to do each time a probe is called, so it is better if arguments are passed of a known type.

The problem of building more flexible tools is better solved by the use of the USDT tracepoint protocol for defining static tracepoints. Fortunately, this has already been established in many packages by the popular use of Dtrace on other Unix platforms like Solaris, BSD, and their derivatives, such as Darwin. Systemtap has provided Linux compatibility, which is what `bpftrace` and `bcc` are able to leverage.

Memcached Static Tracepoints

If an application supports USDT tracepoints already, then no modification of the source code is necessary. Fortunately, Memcached already includes Dtrace probes and strategic spots within the codebase.

The presence of this established pattern for comprehensive Dtrace tracepoints significantly simplified building a lightweight and simple tool. This section of the Memcached source, shows how these probes are invoked:

```
#ifdef ENABLE_DTRACE
      uint64_t cas = ITEM_get_cas(it);
      switch (c->cmd) {
      case NREAD_ADD:
          MEMCACHED_COMMAND_ADD(c->sfd, ITEM_key(it), it->nkey,
                                (ret == 1) ? it->nbytes : -1,
↪   cas);
          break;
      case NREAD_REPLACE:
          MEMCACHED_COMMAND_REPLACE(c->sfd, ITEM_key(it),
↪   it->nkey,
                                    (ret == 1) ? it->nbytes : -1,
↪   cas);
          break;
      case NREAD_APPEND:
          MEMCACHED_COMMAND_APPEND(c->sfd, ITEM_key(it),
↪   it->nkey,
                                   (ret == 1) ? it->nbytes : -1,
↪   cas);
          break;
      case NREAD_PREPEND:
          MEMCACHED_COMMAND_PREPEND(c->sfd, ITEM_key(it),
↪   it->nkey,
                                    (ret == 1) ? it->nbytes : -1,
↪   cas);
```

```
            break;
        case NREAD_SET:
            MEMCACHED_COMMAND_SET(c->sfd, ITEM_key(it), it->nkey,
                                  (ret == 1) ? it->nbytes : -1,
↪   cas);
            break;
        case NREAD_CAS:
            MEMCACHED_COMMAND_CAS(c->sfd, ITEM_key(it), it->nkey,
↪   it->nbytes,
                                  cas);
            break;
        }
#endif
```

Unfortunately, when a production Memcached instance was checked with `tplist` or `bpftrace -l 'usdt:* -p $(pidof memcached)`, no probes were shown. This meant there would be a need to modify the Memcached image to add Dtrace probes.

The `Dockerfile` [9] that was used is based on a production configuration which has been simplified for this analysis. The relevant addition to add Dtrace probes was this snippet:

```
# Get dtrace dependencies for alpine in a kinda hacky way
RUN mkdir /build && cd /build && wget
↪   http://launchpadlibrarian.net/251391227/systemtap-sdt-dev_2.9-2ubuntu2_amd64.deb
↪   && \
    ar x systemtap-sdt-dev_2.9-2ubuntu2_amd64.deb && \
    tar -xpf data.tar.xz && \
    mkdir -p /usr/include/sys && \
    mv usr/include/x86_64-linux-gnu/sys/* /usr/include/sys && rm
     ↪   -rf /build
```

Though the package is being pulled from Ubuntu, only a few text files are needed from it. This package just installs the `sys/sdt.h` header, and a stub command of Dtrace that can be used to convert a dtrace file into a generated header, providing the necessary macros to add tracepoints. The Debian archive is extracted, and the `/usr/bin/dtrace` shell stub and headers are copied into the docker image at standard paths.

Then on the configure line for Memcached, just adding `--enable-dtrace` was sufficient:

```
        && ./configure \
        CFLAGS="-ggdb3" \
        CXXFLAGS="-ggdb3" \
        LDFLAGS="-ggdb3" \
                --build="$gnuArch" \
                --enable-sasl \
                --enable-sasl-pwdb \
                --enable-dtrace \
```

The image can be built with `Docker build . -t memcached-dtrace` in this directory, producing a Memcached image with dtrace probes.

During the configure process, this output indicates it finds the Dtrace stub:

```
...
checking for dtrace... /usr/bin/dtrace
...
```

Later on it generates a header `memcached_dtrace.h`, which is conditionally included when Dtrace probes are enabled:[7]

```
/usr/bin/dtrace -h -s memcached_dtrace.d
sed -e 's,void \*,const void \*,g' memcached_dtrace.h | \
            sed -e 's,char \*,const char \*,g' | tr '\t' ' ' >
            ↪  mmc_dtrace.tmp
mv mmc_dtrace.tmp memcached_dtrace.h
```

This generated header defines the macros which are already called in the source code of Memcached:

```
/* MEMCACHED_COMMAND_SET ( int connid, const char *key, int
↪  keylen, int size, int64_t casid ) */
#define MEMCACHED_COMMAND_SET(arg1, arg2, arg3, arg4, arg5) \
DTRACE_PROBE5 (memcached, command__set, arg1, arg2, arg3, arg4,
↪  arg5)
```

So it seems like the dtrace support has been built into Memcached. Now that the image has been built, this can be verified against a running process instance. To start a test instance the docker commands to bind to localhost on the standard Memcached port are:

```
docker run --name memcached-dtrace -p 11211:11211 memcached-dtrace
```

Or, alternatively, use an image already built:

```
docker run --name memcached-dtrace -p 11211:11211 quay.io/dalehamel/memcached-dtrace:latest
```

To probe it, we'll need to get the process ID of Memcached:

```
MEMCACHED_PID=$(docker inspect --format '{{.State.Pid}}' memcached-dtrace)
```

[7] on a production instance, I had to further modify the dtrace setup in order to disable semaphores, see https://github.com/iovisor/bcc/issues/2230

Now I can run `tplist` from bcc, or use bpftrace[8] to list the USDT tracepoints:

```
tplist -p ${MEMCACHED_PID}
```

Shows These tracepoints[9]:

```
/usr/local/bin/memcached memcached:conn__create
/usr/local/bin/memcached memcached:conn__allocate
/usr/local/bin/memcached memcached:conn__destroy
/usr/local/bin/memcached memcached:conn__release
/usr/local/bin/memcached memcached:process__command__end
/usr/local/bin/memcached memcached:command__add
/usr/local/bin/memcached memcached:command__replace
/usr/local/bin/memcached memcached:command__append
/usr/local/bin/memcached memcached:command__prepend
/usr/local/bin/memcached memcached:command__set
/usr/local/bin/memcached memcached:command__cas
/usr/local/bin/memcached memcached:command__touch
/usr/local/bin/memcached memcached:command__get
/usr/local/bin/memcached memcached:process__command__start
/usr/local/bin/memcached memcached:command__delete
/usr/local/bin/memcached memcached:command__incr
/usr/local/bin/memcached memcached:command__decr
/usr/local/bin/memcached
↪  memcached:slabs__slabclass__allocate__failed
/usr/local/bin/memcached memcached:slabs__slabclass__allocate
/usr/local/bin/memcached memcached:slabs__allocate__failed
/usr/local/bin/memcached memcached:slabs__allocate
/usr/local/bin/memcached memcached:slabs__free
/usr/local/bin/memcached memcached:item__link
/usr/local/bin/memcached memcached:item__unlink
/usr/local/bin/memcached memcached:item__remove
/usr/local/bin/memcached memcached:item__update
/usr/local/bin/memcached memcached:item__replace
/usr/local/bin/memcached memcached:assoc__find
/usr/local/bin/memcached memcached:assoc__insert
/usr/local/bin/memcached memcached:assoc__delete
/usr/local/bin/memcached memcached:conn__dispatch
```

This showed that probes had been recognized on the ELF binary, and so had been compiled-in successfully, even though there was no available OS package. This shows the ease with which these probes can be applied to existing application suites.

[8] there is a bug right now where this isn't working for containerized processes, this will be fixed in a future bpftrace / bcc release. // FIXME file bug

[9] These entries correspond to the data read from `readelf --notes` elsewhere in this report, as that is where these entries are read from.

With USDT support now confirmed, a probe can be built that targets the `process__command` probe, reading arguments based on the probe signature.

```
/**
 * Fired when the processing of a command starts.
 * @param connid the connection id
 * @param request the incoming request
 * @param size the size of the request
 */
probe process__command__start(int connid, const void *request,
↪  int size);
```

The uprobe tool from earlier can be rewritten to target this static tracepoint:

```
usdt::memcached:process__command,
{
  @calls[str(arg1)]++;
}
```

This serves as a minimal proof of concept that the same tool can be built with a USDT probe, but is nowhere near parity for the data that the original `mctop` tool could provide.

bpftrace script for mcsnoop

With a basic test lab now set up to try out USDT probes on Memcached, it would now be easier to investigate how to pull out cleaner and more complete data from the Dtrace probes.

In the Dtrace probe definitions file:

```
   /**
    * Fired for a set-command.
    * @param connid connection id
    * @param key requested key
    * @param keylen length of the key
    * @param size the new size of the key's data (or signed int
↪   -1 if
    *             not found)
    * @param casid the casid for the item
    */
   probe command__set(int connid, const char *key, int keylen,
↪   int size, int64_t casid);
```

This `command__set` probe negates the need to parse the command string to get the values from it, and it looks like all of the other commands also have Dtrace probes with similar signatures.

These definitions are generated into header macros that are callable from the Memcached source code. This is what the calls to emit data to a probe look like in Memcached:

```
#ifdef ENABLE_DTRACE
      uint64_t cas = ITEM_get_cas(it);
      switch (c->cmd) {
      case NREAD_ADD:
          MEMCACHED_COMMAND_ADD(c->sfd, ITEM_key(it), it->nkey,
                                (ret == 1) ? it->nbytes : -1,
↪   cas);
          break;
      case NREAD_REPLACE:
          MEMCACHED_COMMAND_REPLACE(c->sfd, ITEM_key(it),
↪  it->nkey,
                                    (ret == 1) ? it->nbytes : -1,
↪  cas);
          break;
      case NREAD_APPEND:
          MEMCACHED_COMMAND_APPEND(c->sfd, ITEM_key(it),
↪  it->nkey,
                                   (ret == 1) ? it->nbytes : -1,
↪  cas);
          break;
      case NREAD_PREPEND:
          MEMCACHED_COMMAND_PREPEND(c->sfd, ITEM_key(it),
↪  it->nkey,
                                    (ret == 1) ? it->nbytes : -1,
↪  cas);
          break;
      case NREAD_SET:
          MEMCACHED_COMMAND_SET(c->sfd, ITEM_key(it), it->nkey,
                                (ret == 1) ? it->nbytes : -1,
↪  cas);
          break;
      case NREAD_CAS:
          MEMCACHED_COMMAND_CAS(c->sfd, ITEM_key(it), it->nkey,
↪  it->nbytes,
                                cas);
          break;
      }
#endif
```

This can be tested by sending a test ‘SET’ command to a Memcached instance. By piping printf into netcat, [10] standard shell tools can send test commands in the Memcached string protocol:

```
printf "set memtier-3652115 0 60 4\r\ndata\r\n" | nc localhost
↪  11211
```

By reading arg3 to get the probe size, a bpftrace script could be written that provided similar output to `mctop`, at least for the SET command:

```
BEGIN
{
  printf("%-20s %10s %10s %10s\n", "MEMCACHED KEY", "CALLS",
   ↪  "OBJSIZE", "REQ/s");
  @start = nsecs;
}

// NOTE - this presently omits incr, decr, and delete because
↪  they have a
// different signature
usdt::memcached:command__set,
{
  @calls[str(arg1)]++;

  $objsize = arg3;
  $interval = (nsecs - @start) / 1000000000;
  $cps = @calls[str(arg1)] / $interval;
  printf("%-20s %10d %10d %10d\n", str(arg1), @calls[str(arg1)],
                                 $objsize, $cps)
}

END
{
  clear(@start);
  clear(@calls);
}
```

But this wasn't really a top-like tool, it just prints results as it gets data. To see how this might be done, Brendan Gregg's examples from his new book's [11] git repository [12], has slabratetop.bt:

```
#include <linux/mm.h>
#include <linux/slab.h>
#ifdef CONFIG_SLUB
#include <linux/slub_def.h>
#else
#include <linux/slab_def.h>
#endif

kprobe:kmem_cache_alloc
{
        $cachep = (struct kmem_cache *)arg0;
        @[str($cachep->name)] = count();
}

interval:s:1
{
        time();
        print(@);
        clear(@);
}
```

This showed how to build a top-like tool in `bpftrace`, but also the limitations of doing so. You can basically just print the map data out on a recurring interval.

So for a UI, this was about the limit of what `bpftrace` could easily provide. It is great for analyzing map data, but not so great at producing interactive top-like UIs yet, as that involves some sophisticated post-processing of the map data.

Ultimately, the most complete working version of this `bpftrace` prototype is something more like a sniffer, so a name like `mcsnoop`, is more appropriate.

A full version of the latest source for `mcsnoop`[10] is available in the repository for this report [2]:

```
BEGIN
{
  printf("%-20s %10s %10s %10s\n", "MEMCACHED KEY", "CALLS",
   ↪  "OBJSIZE", "REQ/s");

  @start = nsecs;
}

// NOTE - this presently omits incr, decr, and delete because
 ↪  they have a
// different signature
usdt::memcached:command__get,
usdt::memcached:command__set,
usdt::memcached:command__add,
usdt::memcached:command__append,
usdt::memcached:command__prepend,
usdt::memcached:command__touch,
usdt::memcached:command__cas,
usdt::memcached:command__replace
{
  $key = str(arg1, arg2)
  @calls[$key]++;

  $objsize = arg3;
  $interval = (nsecs - @start) / 1000000000;
  $cps = @calls[$key] / $interval;
  printf("%-20s %10d %10d %10d\n", $key, @calls[$key],
                               $objsize, $cps)
}
```

[10]this solves the problem I would later have in my `bcc` version of treating `void *` byte buffers properly, which I'll be covering in more detail later.

```
END
{
  clear(@start);
  clear(@calls);
}
```

Getting started on a bcc tool

To make a fully-featured port of `mctop`, `bpftrace` wouldn't quite fit the bill, as it doesn't have quite the same flexibility as Python when it comes to post-processing data.

USDT example

From the `bcc` reference guide [13], an example program snippet is provided showing how to read data from a USDT argument:

```c
int do_trace(struct pt_regs *ctx) {
    uint64_t addr;
    char path[128];
    bpf_usdt_readarg(6, ctx, &addr);
    bpf_probe_read(&path, sizeof(path), (void *)addr);
    bpf_trace_printk("path:%s\\n", path);
    return 0;
};
```

It just declares a 64 bit integer to store an address, and 128-byte character array to store a path, presumably a string.

`bpf_usdt_readarg` is used to read the argument, and is called to store the literal value of an integer for `addr`, and this value happens to be a pointer to where the string for `path` is stored in the memory space. This is handled by the next call.

`bpf_probe_read` reads a fixed number of bytes, starting from the pointer address.

With these basics, the tool could be translated to C (for the probes) and Python (for the UI and post-processing / deriving second-order values).

Examining some real tools

“When in Rome, do as the Romans do”

To get an idea of how USDT probes were used in real-world scripts, existing bcc tools are a good source inspiration and to gain better understanding of how to port the `bpftrace` script to ‘bcc.

mysqld_qslower.py

The first example I looked at was one for instrumenting MySQL. This goes to show just how much of a swiss-army-knife USDT can be - the same tools can be used to debug Memcached and MySQL!

The C code segment of this script showed a real invocation of the methods to read USDT argument data, and how to set up a map to store structured data:

```c
struct start_t {
    u64 ts;
    char *query;
};

struct data_t {
    u64 pid;
    u64 ts;
    u64 delta;
    char query[QUERY_MAX];
};

BPF_HASH(start_tmp, u32, struct start_t);
BPF_PERF_OUTPUT(events);

int do_start(struct pt_regs *ctx) {
    u32 pid = bpf_get_current_pid_tgid();
    struct start_t start = {};
    start.ts = bpf_ktime_get_ns();
    bpf_usdt_readarg(1, ctx, &start.query);
    start_tmp.update(&pid, &start);
    return 0;
};
```

ucalls.py

Another great example that I spent a lot of time dissecting was ucalls.py, which is the script that powers `rubycalls` and other language-specific USDT tools in bcc. It does a little bit of meta-programming, in that it swaps out function calls and arguments to match that of the runtime for the target language. This allows for it to share the same control logic, regardless of which language is being traced. For instance, for Ruby it sets the probe points at one location:

```
elif language == "ruby":
    # TODO Also probe cmethod__entry and cmethod__return with
    ↪  same arguments
    entry_probe = "method__entry"
    return_probe = "method__return"
    read_class = "bpf_usdt_readarg(1, ctx, &clazz);"
    read_method = "bpf_usdt_readarg(2, ctx, &method);"
```

Then later, in the C code, it uses these to replace `READ_CLASS` and `READ_METHOD` when it is building out the probe function:

```
int trace_entry(struct pt_regs *ctx) {
    u64 clazz = 0, method = 0, val = 0;
    u64 *valp;
    struct entry_t data = {0};
#ifdef LATENCY
    u64 timestamp = bpf_ktime_get_ns();
    data.pid = bpf_get_current_pid_tgid();
#endif
    READ_CLASS
    READ_METHOD
    bpf_probe_read(&data.method.clazz, sizeof(data.method.clazz),
                   (void *)clazz);
    bpf_probe_read(&data.method.method,
↪   sizeof(data.method.method),
```

There are several other tools in this suite, targeting Ruby, Python, Java, PhP, tcl, and Perl. Some tools are specific to a given language, as support does vary somewhat depending on what probes the runtime maintainers choose to expose.

These scripts provided a wealth of examples for how USDT tracing was already being done in `bcc`, and a jumping off point for a new tool.

slabratetop.py

UI / UX design isn't my forte, and apparently imitation is the sincerest form of flattery. To start with, I looked through the `\*top.py` `top`-like tools for one to base the structure of my program on. A fine example was `slabratetop.py`, which happens to be the Python version of the `bpftrace` script that was showed earlier. The design of its main control loop and argument parsing were the main concepts borrowed:

```
exiting = 0
while 1:
    try:
        sleep(interval)
    except KeyboardInterrupt:
        exiting = 1

    # header
    if clear:
        call("clear")
    else:
        print()
    with open(loadavg) as stats:
        print("%-8s loadavg: %s" % (strftime("%H:%M:%S"),
         ↪  stats.read()))
    print("%-32s %6s %10s" % ("CACHE", "ALLOCS", "BYTES"))

    # by-TID output
    counts = b.get_table("counts")
    line = 0
    for k, v in reversed(sorted(counts.items(),
                                key=lambda counts:
↪   counts[1].size)):
        printb(b"%-32s %6d %10d" % (k.name, v.count, v.size))

        line += 1
        if line >= maxrows:
            break
    counts.clear()

    countdown -= 1
    if exiting or countdown == 0:
        print("Detaching...")
        exit()
```

This was then blended with the `select` approach used by the Ruby `mctop` in order to receive keyboard input, which will be covered in more detail in the UI section of this document.

Issues porting to bcc

In moving from the `bpftrace` prototype to a fully-fledged `bcc`-based Python tool, inevitably there were some issues. `bpftrace` does a lot of smart stuff under the hood, and basically does the equivalent of writing these C-based segments of the eBPF probes for you, using LLVM IR (intermediate representation).

In moving to writing the raw C to generate the eBPF code, there were a couple of hiccups in the form of rough edges that aren't as friendly as the faculties which `bpftrace` provides in its higher-level tracing language.

Debugging

To start off, to be able to print data in a way that can be readily used in debugging scenarios, the built-in `bpf_trace_printk` can be used, which is a printf-like interface. To read these values out of the kernel:

```
sudo cat /sys/kernel/debug/tracing/trace_pipe
```

Being able to read the data

The original eBPF trace function was based on the sample code from `bcc`. The Dtrace probe spec for `command__set`, can be used to determine the argument ordering and type information:

```
struct value_t {
    u64 count;
    u64 bytecount;
};

BPF_HASH(keyhits, struct keyhit_t, struct value_t);


int trace_entry(struct pt_regs *ctx) {
    u64 keystr = 0, bytecount = 0;
    struct keyhit_t keyhit = {0};
    struct value_t *valp, zero = {};

    bpf_usdt_readarg(2, ctx, &keystr);
    bpf_usdt_readarg(4, ctx, &bytecount);

    bpf_probe_read(&keyhit.keystr, sizeof(keyhit.keystr), (void
↪   *)keystr);

    valp = keyhits.lookup_or_init(&keyhit, &zero);
    valp->count += 1;
    valp->bytecount = bytecount;

    return 0;
}
```

This basic probe was printing data for the key! But it wasn't reading anything for the size parameter, which was needed in order to replicate the key size feature of the original `mctop`.

The calls to `bpf_usdt_readarg` are reading the parameter into a 64 bit container. Sometimes this is for literal values, and sometimes it is for addresses. Reading literal values is easy and efficient, they are simply copied into the address passed in as the third argument, as the bitwise AND operator is used for. This is why `u64 keystr = 0, bytecount = 0;` is in the code, to declare the sizes of these storage containers as 64 bits, unsigned.

In `bpftrace`, almost all storage is done in 64 bit unsigned integers like this, and it is a pretty normal standard to just use a container that is the size of a machine word on modern microprocessors. This is because type information is handled differently in `bpftrace`, and reads are cast to the appropriate storage class for their type before they occur.

As it turns out, for reading USDT args properly, it is best with `bcc` to match the storage class to the argument type being read, otherwise the result of a type mismatch on the probe read may result in a 0 value.

To fix this problem, which is something also encountered in a separate report on Ruby USDT tracing [14], the Systemtap wiki page [15] has an explanation on the ELF note format, which is also used by `libstapsdt` when generating type signatures for probe arguments.

The command `readelf --notes` can be used to show the probe addresses that are added by the `systemtap` dtrace compatibility headers, supplying `sys/sdt.h` to Linux. The output in this case shows:

```
 stapsdt                0x00000058        NT_STAPSDT (SystemTap
↪  probe descriptors)
   Provider: memcached
   Name: command__set
   Location: 0x0000000000007a66, Base: 0x0000000000042a60,
    ↪  Semaphore: 0x00000000000497ec
   Arguments: -4@%edx 8@%rsi 1@%cl -4@%eax 8@-24(%rbp)
```

The argument signature token to the left[11] of the @ symbol is what can be used to decode the type.

`Arguments: -4@... 8@... 1@... -4@... 8@...`

Using the table from the Systemtap wiki:

Arg code	Description	Storage Class
1 ...	8 bits unsigned.	`uint8_t` ....
-1 ...	8 bits signed.	`int8_t` ....
2 ...	16 bits unsigned.	`uint16_t`....
-2 ...	16 bits signed.	`int16_t` ....
4 ...	32 bits unsigned.	`uint32_t`....
-4 ...	32 bits signed.	`int32_t` ....
8 ...	64 bits unsigned.	`uint64_t`....
-8 ...	64 bits signed.	`int64_t` ....

[15]

We can decode this as:

Arg index	Args from ELF notes	Storage Class
0	-4@...	`int32_t` ...
1	8@...	`uint64_t` ...
2	1@...	`uint8_t` ...
3	-4@...	`int32_t` ...
4	8@...	`uint64_t` ...

So, we can take it that the 4th argument to `command__set`'s probe call is actually meant to be stored in a signed 32-bit int!

Adjusting the code accordingly to match the proper storage class, data can now be read by the probe:

[11]The bit after the @ symbol seems to be the register to read this from. It also looks like it is able to specify offsets relative to the frame pointer, so this probably is based on the platform calling convention, denoting the offset in the stack and size to read.

```
int trace_entry(struct pt_regs *ctx) {
    u64 keystr = 0;
    int32_t bytecount = 0; // type is -4@%eax in stap notes,
    ↪  which is int32
    struct keyhit_t keyhit = {0};
    struct value_t *valp, zero = {};

    bpf_usdt_readarg(2, ctx, &keystr);
    bpf_usdt_readarg(4, ctx, &bytecount);

    bpf_probe_read_str(&keyhit.keystr, sizeof(keyhit.keystr),
↪   (void *)keystr);

    valp = keyhits.lookup_or_init(&keyhit, &zero);
    valp->count += 1;
    valp->bytecount = bytecount;
    valp->totalbytes += bytecount;
    valp->timestamp = bpf_ktime_get_ns();


    return 0;
}
```

Note that only the `bytecount` variable needed to be changed to `int32_t`, as it only matters for the read - the struct member used to store this can remain `uint64_t`, as the copy operation will pull this smaller type into a larger storage class without truncation.

Duplicate keys?

Now that probe data could be read, the UI could be replicated.

In initial testing, there was a confusing bug where the same key was printed multiple times. It was iterating over a map where these apparently identical keys were expected to be hashed to the same slot.

After finding no bug in the display code, it seemed that the keys must actually not be the same, even though they looked to be identical when they were printed to the screen. Some other unintended data must have been making it into the string buffer, and "garbling the keys".

Early tests were mostly with one request at a time, but once this was scripted to increase the call rate and vary the keys, the pattern became much more obvious.

In the case of the string, its `const char *` signature, which would hint at a string though is possibly a byte array. The earlier use of `process__command`, had `const void *` in its signature, which would be standard for indicating arbitrary binary data / down-casting to this could be "pretty much anything".

Both signatures often refer to byte arrays of either binary or string data though, so the context of the data received here depends on the context that the probe is calling it.

In either case, it is necessary to read this data into a buffer. For this a buffer is declared inside of a struct for `keystr`:

```
struct keyhit_t {
    char keystr[MAX_STRING_LENGTH];
};
```

This allows for the members of this struct to be easily mapped as members of Python objects. Inside the probe, the struct is initialized to 0:

```
    struct keyhit_t keyhit = {0};
```

This is used to received the data is copied into the character buffer `keystr` on the `keyhit_t` struct:

```
    bpf_probe_read(&keyhit.keystr, sizeof(keyhit.keystr), (void
↪   *)keystr);
```

An attempt was made to use the `bcc` function `bpf_probe_read_str` the documentation indicates is able to read a buffer of a fixed size until it finds a null-byte, which seemed to be a fitting solution to the problem.[12]

This worked more reliably and there were fewer errors, but when benchmarking was done at much higher call rates, it became clear that some of the payload must be making it into the buffer for the key read. This indicated that it was reading too many bytes, and collecting data from the adjacent memory space.

Memcached key read quirks

Originally I thought this might be a bug, so I filed an upstream issue [17]. Despite the argument being of type `const char *`, in this instance, it wasn't guaranteed to be a null terminated string.

Where the probe is called, it is using the macro `ITEM_key` to get the value that is passed to the Dtrace macro:

```
            MEMCACHED_COMMAND_SET(c->sfd, ITEM_key(it), it->nkey,
```

[12] Dormando [16] mentioned in [17]

This macro is just getting the address of the start of the data segment, and clearly isn't copying a string into a null-terminated buffer:

```
#define ITEM_key(item) (((char*)&((item)->data)) \
```

So this meant that the `bpf_probe_read_str` from `bcc`, will read the full size of the buffer object for its read, and can blow past the actual length of the key data! It turns out that if using `bpf_probe_read_str`, it never finds a null byte, and so will also just read the whole buffer.

This taught that, Memcached doesn't necessarily store keys as null terminated strings, or even string data at all - it is arbitrary binary bytes. This is why it passes the argument `keylen` in the USDT probe, so that the correct size of the key can be read. Using the same process as above, it was determined that the `keylen` argument was actually stored as a `uint8_t`, and was able to get the key length easily enough. This was stored as `keysize`.

De-garbling in Userspace

Unfortunately, using this read `keysize` wasn't trivial, as it produced a verifier error if it was passed as an argument to the probe read function. This seemed to be because it was not a const or provably safe value.

To prevent this from blocking the development of the rest of the tool, the `keysize` value was passed into userspace by adding it as a field to the value data struct. This would enable de-garbling this data in Python.

This meant that the same key could be hashed to multiple slots, as they would include whatever arbitrary data is after the key in the buffer that is read.

In hindsight, this behavior of passing a buffer and a length to read seems to have been intentional for Memcached. Not performing a string copy is more efficient, which is why the probe just submits the buffer and the length of the data to read, leaving it up to the kernel handler to copy the data.

To resolve this, a Python workaround was used to combine the keys in userspace:

```python
def reconcile_keys(bpf_map):
  new_map = {}

  for k,v in bpf_map.items():
      shortkey = k.keystr[:v.keysize].decode('utf-8', 'replace')
      if shortkey in new_map:

          # Sum counts on key collision
          new_map[shortkey]['count'] += v.count
          new_map[shortkey]['totalbytes'] += v.totalbytes
```

```
            # If there is a key collision, take the data for the
            ↪  latest one
            if v.timestamp > new_map[shortkey]['timestamp']:
                new_map[shortkey]['bytecount'] = v.bytecount
                new_map[shortkey]['timestamp'] = v.timestamp
        else:
            new_map[shortkey] = {
                "count": v.count,
                "bytecount": v.bytecount,
                "totalbytes": v.totalbytes,
                "timestamp": v.timestamp,
            }
    return new_map
```

This just added to or replaced values as necessary, using a timestamp to take the more recent between the two if values were being replaced.

This was sufficient to finish the prototype, while a solution to the verifier issue could be worked on.

Different signatures for USDT args

Most of the development was done by testing only with the `SET` command, because this was the most convenient. Once there was a more fully-featured tool, attention turned to tracing the other commands. As they all had the same signatures according to their Dtrace probe definitions, it was assumed to be an easy task to iterate over all of the commands to be traced.

There was a nasty little surprise right away when implementing `GET`. As it turns out, it can have **more than one argument signature**, depending where it is called from:

```
    Name: command__get
    Arguments: -4@%edx 8@%rdi      1@%cl       -4@%esi 8@%rax
    Arguments: -4@%eax 8@-32(%rbp) 8@-24(%rbp) -4@$-1 -4@$0
    Arguments: -4@%edx 8@%rdi      1@%cl       -4@%esi 8@%rax
    Arguments: -4@%eax 8@-40(%rbp) 8@-32(%rbp) -4@$-1 -4@$0
    Arguments: -4@%eax 8@-24(%rbp) 8@-16(%rbp) -4@$-1 -4@$0
```

This manifested as the `GET` requests frequently returning 0 for `keylen`, as it could be stored in either a `uint8_t` or a `uint64_t`.

To get around this, checking if the value was 0 and trying to read again with a different (larger) storage class resulted in actually reading a value correctly.

eBPF deep dive

This section gets into the eBPF code disassembly in order to explain how to structure probes to ensure they will be accepted by the kernel's BPF verifier.

Verifier error with variable read

With a working replacement of all of the basic `mctop` functionality, the priority became to try and fix the garbled keys at the right layer: in the eBPF probe. Bas Smit [18] pointed out on IRC that non-const probe reads for string data already a problem `bpftrace` had solved.

This gave some renewed hope that there **must** be a way to get the eBPF verifier to accept a non-const length read.

Knowing that this works in `bpftrace`, it would make sense to take a look at how this is handled there. This is the relevant LLVM IR generation procedure from `bpftrace`:

```
else if (call.func == "str")
{
  AllocaInst *strlen = b_.CreateAllocaBPF(b_.getInt64Ty(),
↪ "strlen");
  b_.CreateMemSet(strlen, b_.getInt8(0), sizeof(uint64_t), 1);
  if (call.vargs->size() > 1) {
    call.vargs->at(1)->accept(*this);
    Value *proposed_strlen = b_.CreateAdd(expr_,
↪ b_.getInt64(1)); // add 1 to accommodate probe_read_str's
↪ null byte

    // largest read we'll allow = our global string buffer size
    Value *max = b_.getInt64(bpftrace_.strlen_);
    // integer comparison: unsigned less-than-or-equal-to
    CmpInst::Predicate P = CmpInst::ICMP_ULE;
    // check whether proposed_strlen is less-than-or-equal-to
     ↪ maximum
```

```
      Value *Cmp = b_.CreateICmp(P, proposed_strlen, max,
↪   "str.min.cmp");
      // select proposed_strlen if it's sufficiently low,
       ↪  otherwise choose maximum
      Value *Select = b_.CreateSelect(Cmp, proposed_strlen, max,
↪   "str.min.select");
      b_.CreateStore(Select, strlen);
    } else {
      b_.CreateStore(b_.getInt64(bpftrace_.strlen_), strlen);
    }
    AllocaInst *buf = b_.CreateAllocaBPF(bpftrace_.strlen_,
↪   "str");
    b_.CreateMemSet(buf, b_.getInt8(0), bpftrace_.strlen_, 1);
    call.vargs->front()->accept(*this);
    b_.CreateProbeReadStr(buf, b_.CreateLoad(strlen), expr_);
    b_.CreateLifetimeEnd(strlen);

    expr_ = buf;
    expr_deleter_ = [this,buf]() { b_.CreateLifetimeEnd(buf); };
  }
```

This generates the LLVM IR for doing a comparison between the size parameter given, and the maximum size. This is sufficient for it to pass the eBPF verification that this is a safe read and can run inside the in-kernel BPF virtual machine.

Taking inspiration from an existing issue for this in `bcc`, the probe definition, as described in iovisor/bcc#1260 [19] to include a logical assertion that the `keysize` must be smaller than the buffer size via a ternary.

This didn't work unfortunately, and it threw this eBPF verifier error:

```
54: (57) r2 &= 255
55: (bf) r6 = r10
56: (07) r6 += -80
57: (bf) r1 = r6
58: (85) call bpf_probe_read#4
invalid stack type R1 off=-80 access_size=255
processed 103 insns (limit 1000000) max_states_per_insn 0
↪  total_states 7 peak_states 7 mark_read 4
```

As will be shown later, this message is more helpful than it initially seems, but at the time these values of -80 and 255 didn't seem significant, and it wasn't clear what was meant by an invalid stack offset, as this code was generated and difficult to associate back to the C code which resulted in it.

Safe Code Generation

A comment[20] on iovisor/bcc#1260, provided a hint towards a mechanism which could be used to demonstrate safety for passing a non-const length value to the probe read. In the commit message, this C snippet is used:

```
int len;
char buf[BUFSIZE]; /* BUFSIZE is 128 */

if (some_condition)
  len = 42;
else
  len = 84;

some_helper(..., buf, len & (BUFSIZE - 1));
```

That showed that a bitwise AND with a const value was enough to convince the verifier that this was safe! Of course, this only really be easy if the const value as a hex mask with all bits set, like `0xFF`.

In the Memcached source, we can see that `KEY_MAX_LENGTH` is `250`. This is close enough to 255 that a mask of `0xFF` could be applied:

```
/** Maximum length of a key. */
#define KEY_MAX_LENGTH 250
```

By just setting the buffer size to 255, the maximum that will fit in a single byte, the verifier is now able to determine that no matter what value is read from `keylen` into `keysize`, it will be safe, and that a buffer overflow cannot be possible.

The binary representation of 0xFF (255 decimal) is `1111 1111`. To test this theory, the most significant bit can be flipped to 0, to get `0111 1111`. Back to hexadecimal, this is 0x7F, and in decimal this is 127. By manually comparing the `keysize` with this mask via a bitwise AND, it works and is accepted by the verifier! If, however, the size of the buffer is dropped to just 126, there is the familiar verifier error once again.

The reason why this happens is visible in the disassembly of the generated eBPF program:

```
; bpf_probe_read(&keyhit.keystr, keysize & READ_MASK, (void
↪   *)keystr); // Line  97
  56:   57 02 00 00 7f 00 00 00         r2 &= 127
  57:   bf a6 00 00 00 00 00 00         r6 = r10
  58:   07 06 00 00 80 ff ff ff         r6 += -128
  59:   bf 61 00 00 00 00 00 00         r1 = r6
  60:   85 00 00 00 04 00 00 00         call 4
```

By convention [21], `R1` is used for the first argument to the call of `bpf_probe_read` (built-in function “4”), and `R2` is used for the second argument. `R6` is used as a temporary register, to store the value of `R10`, which is the frame pointer.

Register	x86 reg	Description
R0	rax	return value from function
R1	rdi	1st argument
R2	rsi	2nd argument
R3	rdx	3rd argument
R4	rcx	4th argument
R5	r8	5th argument
R6	rbx	callee saved
R7	r13	callee saved
R8	r14	callee saved
R9	r15	callee saved
R10	rbp	frame pointer

The disassembly shows the buffer is initialized right at the start, putting the struct initialization at the bottom of the stack. In the crashing version there is a `uint16_t` and a `uint32_t` near the start of the stack:

```
; struct keyhit_t keyhit = {0}; // Line  89
   1:    6b 3a fc ff 00 00 00 00         *(u16 *)(r10 - 4) = r3
   2:    63 3a f8 ff 00 00 00 00         *(u32 *)(r10 - 8) = r3
   3:    7b 3a f0 ff 00 00 00 00         *(u64 *)(r10 - 16) = r3
   4:    7b 3a e8 ff 00 00 00 00         *(u64 *)(r10 - 24) = r3
   5:    7b 3a e0 ff 00 00 00 00         *(u64 *)(r10 - 32) = r3
   6:    7b 3a d8 ff 00 00 00 00         *(u64 *)(r10 - 40) = r3
   7:    7b 3a d0 ff 00 00 00 00         *(u64 *)(r10 - 48) = r3
   8:    7b 3a c8 ff 00 00 00 00         *(u64 *)(r10 - 56) = r3
   9:    7b 3a c0 ff 00 00 00 00         *(u64 *)(r10 - 64) = r3
  10:    7b 3a b8 ff 00 00 00 00         *(u64 *)(r10 - 72) = r3
  11:    7b 3a b0 ff 00 00 00 00         *(u64 *)(r10 - 80) = r3
  12:    7b 3a a8 ff 00 00 00 00         *(u64 *)(r10 - 88) = r3
  13:    7b 3a a0 ff 00 00 00 00         *(u64 *)(r10 - 96) = r3
  14:    7b 3a 98 ff 00 00 00 00         *(u64 *)(r10 - 104) = r3
  15:    7b 3a 90 ff 00 00 00 00         *(u64 *)(r10 - 112) = r3
  16:    7b 3a 88 ff 00 00 00 00         *(u64 *)(r10 - 120) = r3
  17:    7b 3a 80 ff 00 00 00 00         *(u64 *)(r10 - 128) = r3
```

But in the non-crashing version, there is also a `uint8_t`:

```
; struct keyhit_t keyhit = {0}; // Line  89
   1:    73 3a fe ff 00 00 00 00         *(u8 *)(r10 - 2) = r3
   2:   6b 3a fc ff 00 00 00 00          *(u16 *)(r10 - 4) = r3
   3:   63 3a f8 ff 00 00 00 00          *(u32 *)(r10 - 8) = r3
   4:   7b 3a f0 ff 00 00 00 00          *(u64 *)(r10 - 16) = r3
   5:   7b 3a e8 ff 00 00 00 00          *(u64 *)(r10 - 24) = r3
   6:   7b 3a e0 ff 00 00 00 00          *(u64 *)(r10 - 32) = r3
   7:   7b 3a d8 ff 00 00 00 00          *(u64 *)(r10 - 40) = r3
   8:   7b 3a d0 ff 00 00 00 00          *(u64 *)(r10 - 48) = r3
   9:   7b 3a c8 ff 00 00 00 00          *(u64 *)(r10 - 56) = r3
  10:   7b 3a c0 ff 00 00 00 00          *(u64 *)(r10 - 64) = r3
  11:   7b 3a b8 ff 00 00 00 00          *(u64 *)(r10 - 72) = r3
  12:   7b 3a b0 ff 00 00 00 00          *(u64 *)(r10 - 80) = r3
  13:   7b 3a a8 ff 00 00 00 00          *(u64 *)(r10 - 88) = r3
  14:   7b 3a a0 ff 00 00 00 00          *(u64 *)(r10 - 96) = r3
  15:   7b 3a 98 ff 00 00 00 00          *(u64 *)(r10 - 104) = r3
  16:   7b 3a 90 ff 00 00 00 00          *(u64 *)(r10 - 112) = r3
  17:   7b 3a 88 ff 00 00 00 00          *(u64 *)(r10 - 120) = r3
  18:   7b 3a 80 ff 00 00 00 00          *(u64 *)(r10 - 128) = r3
```

The difference is subtle, but comparing the space allocated on the stack, the crashing version allocates 15 `uint64_t` + 1 `uint32_t` + 1 `uint16_t`. Converting this to bytes, this becomes (15 * 8 + 1 * 4 + 1 * 2) = 126 bytes allocated.

In the non-crashing version, it is 15 `uint64_t` + 1 `uint32_t` + 1 `uint16_t` + 1 `uint8_t`. This works out to 127 bytes. So that verifier message for the crashing program:

```
60: (85) call bpf_probe_read#4
invalid indirect read from stack off -128+126 size 127
```

Is complaining that the first argument, `R1`, which is set relative to the frame pointer, is not of sufficient size to be certain that the value read in `R2` (guaranteed by the bitwise AND operation to be no more than 127).

To summarize, there were two ways to solve this issue - either increase the buffer size to 255 so that there was no way that the `uint8_t` container used by `keysize` could possibly overflow it, or a bitwise AND the `keysize` value with a hex-mask that is sufficient to prove it cannot be a buffer overflow.

This might seem like a pain, but this extra logic is the cost of safety. This code will be running within the kernel context, and needs to pass the verifier's pat-down. In the meantime, `libbpf` continues to improve to make this sort of explicit proof of safety less necessary.

Final bcc tool

Now that all of the data reading was fixed up, and there was no more need to de-garble the keys in userspace, the final version of this tool could be put together.

DISCLAIMER

This tool has been designed primarily against benchmark workloads, but has not seen extensive production testing outside of basic testing. In order to run `mctop`, Linux Kernel v4.20 or later is needed, but 5.3 or later is recommended.

UI Re-Design

This probably took most of the time. The other `*top.py` tools I saw didn't really offer the interactive experience that the original `mctop` in Ruby did.

Most of the time here was spent reacquainting myself with TTY concepts, and getting the `select` statement set up properly for receiving user input. I based the scaffold of this on the original `mctop` in Ruby, and copied its design patterns.

I decided to add a couple of fields, as I was capturing more data than the original, and I changed how tray bar of the tool works entirely. Beyond just sorting keys by various attributes, specific keys could be analyzed.

Feature Implementation

Key entry

The usage of select was based on the original Ruby:

```
  def input_handler
    # Curses.getch has a bug in 1.8.x causing non-blocking
    # calls to block reimplemented using IO.select
    if RUBY_VERSION =~ /^1.8/
           refresh_secs = @config[:refresh_rate].to_f / 1000

      if IO.select([STDIN], nil, nil, refresh_secs)
        c = getch
        c.chr
      else
        nil
      end
    else
      getch
    end
  end

  def done
```

In Python, without pulling in dependencies, the `termios` library along with `select` can be used to recreate the experience of using the original `mctop`:

```

    return list(output)

# Set stdin to non-blocking reads so we can poll for chars


```

And just as Ruby had a switch on the different inputs:

```
# main loop
until done do
  ui.header
  ui.footer
  ui.render_stats(sniffer, sort_mode, sort_order)
  refresh

  key = ui.input_handler
  case key
    when /[Qq]/
      done = true
    when /[Cc]/
      sort_mode = :calls
    when /[Ss]/
      sort_mode = :objsize
    when /[Rr]/
      sort_mode = :reqsec
    when /[Bb]/
      sort_mode = :bw
    when /[Tt]/
      if sort_order == :desc
        sort_order = :asc
      else
        sort_order = :desc
      end
  end
end
```

So too was this almost directly ported to Python:

```
def readKey(interval):
    new_settings = termios.tcgetattr(sys.stdin)
    new_settings[3] = new_settings[3] & ~(termios.ECHO |
↪   termios.ICANON)
    tty.setcbreak(sys.stdin.fileno())
    if select.select([sys.stdin], [], [], 5) == ([sys.stdin], [],
    ↪   []):
        key = sys.stdin.read(1).lower()
        global sort_mode

        if key == 't':
            global sort_ascending
            sort_ascending = not sort_ascending
        elif key == 'c':
            sort_mode = 'C'
        elif key == 's':
            sort_mode = 'S'
        elif key == 'r':
            sort_mode = 'R'
        elif key == 'b':
            sort_mode = 'B'
        elif key == 'n':
            sort_mode = 'N'
        elif key == 'd':
            global args
```

The concept is just a giant if-ladder, as Python has no case statements. This matches on the letters, and can run a function or update a global variable as the specific case requires. This got complicated as I added more keys to allow for navigation of the sorted key data.

Sorting

To sort the data, a lambda was defined for each sort mode:

```
}
"""


def sort_output(unsorted_map):
    global sort_mode
    global sort_ascending

    output = unsorted_map
    if sort_mode == "C":
        output = sorted(output.items(), key=lambda x:
↪   x[1]['count'])
    elif sort_mode == "S":
        output = sorted(output.items(), key=lambda x:
↪   x[1]['bytecount'])
    elif sort_mode == "R":
        output = sorted(output.items(), key=lambda x:
↪   x[1]['bandwidth'])
    elif sort_mode == "B":
        output = sorted(output.items(), key=lambda x:
↪   x[1]['cps'])
    elif sort_mode == "N":
        output = sorted(output.items(), key=lambda x:
↪   x[1]['timestamp'])
```

This is called on the map for each period of the refresh interval, so the ordering of keys displayed may change each second, should the rank of a key differ from the previous interval.

Dumping data

Since it would probably be useful to be able to analyze the collected data, the Python mapping of the original eBPF map can be saved to a JSON file for analysis when the map is cleared. This also allows for `mctop` to act as a sort of `memcached-dump` tool (à la `tcpdump`), saving the data for archival purposes or offline analysis.

```
                dump_map()
        elif key == 'q':
            print("QUITTING")
            global exiting
            exiting = 1


def dump_map():
    global outfile
    global bpf
    global sorted_output
```

This should allow for a simple pipeline of Memcached metrics into other centralized logging systems.

View Modes

The current/traditional UI for `mctop` was limited in that it couldn't drill down into patterns, and there was no way to navigate the data that was being selected aside from to sort it.

Streaming / NoClear

This design is important to maintain, as it allows for metrics to be collected from line-based logging systems that understand how to parse `mctop` output.

In this mode, `mctop` behaviors similar to `mcsnoop`.

Interactive

This is built around a TTY-interactive experience, using ANSI escape sequences to construct a basic UI. The UI uses vim-like bindings, and is meant for keyboard navigation that should feel natural to any vim user.

Outside of being interactive, `mctop` maintains the original sort-functionality of its namesake.

`mctop` has different visual modes, that correspond to different probes to collect data for a specific key and analyse it.

Navigation

To navigate, the `j` and `k` keys can be used to move up or down a keys, and the selected key is displayed in the footer bar.

The footer bar also now shows the number of pages, segmented by the `maxrows` argument. To easily navigate this buffer, `u` and `d` can be used to navigate up and down a page in this buffer.

Finally, to jump to the end of the buffer, `G`, and to the start of the sorted key list, `g`.

As this control sequence is extremely common in command line tools, the hope is that the navigation keys will feel natural to users of similar tools.

Command latency

To be able to add a new data source and expand on the functionality of the `mctop` predecessor, the latency commands hitting each key could be measured and displayed in aggregate.

This additional data could also be used to plug into `bcc`'s histogram map type and print function, showing an informative `lg2` representation of the latency for commands hitting the key.

Printing Histogram

Printing a histogram of latency data entails recompiling the `eBPF` source to have the static key to collect latency data embedded in the eBPF source.

An inline `match_key` function is used to iterate through the buffer to compare until it finds the key in full or finds a mismatching character and returns early. This bounded loop is permitted in eBPF, but may be wasteful processing at large key sizes.

When a trace on a Memcached command is executed, it stores the `lastkey` in a map.[13] In another probe on `process__command__end`, this is accessed and compared with the hard-coded and selected key from the UI. When there is a match, the computed latency data is added to the histogram.

Upon entering histogram mode, the selected data will be immediately displayed on the same refresh interval. This shows the real-time variations in Memcached latency, in buckets of doubling size.

Switching to histogram mode will detach and replace running probes, and discard the collected data, replacing the eBPF probes with a function that is targeted to a specific cache key.[14]

[13]this is indexed by connection ID right now, but I think that thread id or perhaps a composition of connection and thread id should be used, to ensure that this representation is compatible with memcached's threading model.

[14]this is due to the need to get a lock on the uprobe addresses, and it seems there is no way to hot-patch eBPF programs to encode the selected key.

Inspect Key

Finishing touches and final tool

Since the goal of the tool is to share it, especially so that fans of the original `mctop` or `memkeys` command could have access a light-weight eBPF option, it is definitely a goal to share this tool and get it into good enough shape for it to pass a pull request review [22].

For this reason, this report was prepared to supplement the material around the `mctop` tool included in the pull request.

This script is submitted in its entirety:

```
#!/usr/bin/python
# @lint-avoid-python-3-compatibility-imports
#
# mctop   Memcached key operation analysis tool
#         For Linux, uses BCC, eBPF.
#
# USAGE: mctop.py  -p PID
#
# This uses in-kernel eBPF maps to trace and analyze key access
↪  rates and
# objects. This can help to spot hot keys, and tune memcached
↪  usage for
# performance.
#
# Copyright 2019 Shopify, Inc.
# Licensed under the Apache License, Version 2.0 (the "License")
#
# 20-Nov-2019   Dale Hamel   Created this.
# Inspired by the ruby tool of the same name by Marcus Barczak in
↪  2012, see
# see also https://github.com/etsy/mctop
# see also https://github.com/tumblr/memkeys

from __future__ import print_function
from time import sleep, strftime, monotonic
from bcc import BPF, USDT, utils
from subprocess import call
import argparse
import sys
import select
import tty
import termios
import json

# FIXME better help
# arguments
examples = """examples:
    ./mctop -p PID          # memcached usage top, 1 second
↪  refresh
"""

parser = argparse.ArgumentParser(
    description="Memcached top key analysis",
    formatter_class=argparse.RawDescriptionHelpFormatter,
    epilog=examples)
parser.add_argument("-p", "--pid", type=int, help="process id to
↪  attach to")
parser.add_argument(
    "-o",
    "--output",
    action="store",
    help="save map data to /top/OUTPUT.json if 'D' is issued to
    ↪  dump the map")

parser.add_argument("-C", "--noclear", action="store_true",
                    help="don't clear the screen")
parser.add_argument("-r", "--maxrows", default=20,
                    help="maximum rows to print, default 20")
parser.add_argument("interval", nargs="?", default=1,
                    help="output interval, in seconds")
parser.add_argument("count", nargs="?", default=99999999,
                    help="number of outputs")
parser.add_argument("--ebpf", action="store_true",
                    help=argparse.SUPPRESS)

# FIXME clean this up
args = parser.parse_args()
interval = int(args.interval)
countdown = int(args.count)
maxrows = int(args.maxrows)
clear = not int(args.noclear)
outfile = args.output
pid = args.pid

# Globals
exiting = 0
sort_mode = "C"
```

```
sort_ascending = True
bpf = None
sorted_output = None


sort_modes = {
    "C": "calls",  # total calls to key
    "S": "size",  # latest size of key
    "R": "req/s",  # requests per second to this key
    "B": "bw",    # total bytes accesses on this key
    "N": "ts"     # timestamp of the latest access
}

commands = {
    "T": "toggle",  # sorting by ascending / descending order
    "D": "dump",   # clear eBPF maps and dump to disk (if set)
    "Q": "quit"    # exit mctop
}

# /typedef enum {START, END, GET, ADD, SET, REPLACE, PREPEND,
↪  APPEND,
#                        TOUCH, CAS, INCR, DECR, DELETE}
↪  memcached_op_t;

# FIXME have helper to generate per  type?
# load BPF program
bpf_text = """
#include <uapi/linux/ptrace.h>
#include <bcc/proto.h>


#define READ_MASK 0xff // allow buffer reads up to 256 bytes
struct keyhit_t {
    char keystr[READ_MASK];
};

struct value_t {
    u64 count;
    u64 bytecount;
    u64 totalbytes;
    u64 keysize;
    u64 timestamp;
};

BPF_HASH(keyhits, struct keyhit_t, struct value_t);
```

```
int trace_entry(struct pt_regs *ctx) {
    u64 keystr = 0;
    int32_t bytecount = 0; // type is -4@%eax in stap notes,
↪   which is int32
    uint8_t keysize = 0; // type is 1@%cl, which should be uint8
    struct keyhit_t keyhit = {0};
    struct value_t *valp, zero = {};

    bpf_usdt_readarg(2, ctx, &keystr);
    bpf_usdt_readarg(3, ctx, &keysize);
    bpf_usdt_readarg(4, ctx, &bytecount);

    // see https://github.com/memcached/memcached/issues/576
    // as well as https://github.com/iovisor/bcc/issues/1260
    // we can convince the verifier the arbitrary read is safe
↪   using this
    // bitwise &, but only because our max buffer size happens to
↪   be 0xff,
    // which corresponds roughly to the the maximum key size
    bpf_probe_read(&keyhit.keystr, keysize & READ_MASK, (void
↪   *)keystr);

    valp = keyhits.lookup_or_init(&keyhit, &zero);
    valp->count++;
    valp->bytecount = bytecount;
    valp->keysize = keysize;
    valp->totalbytes += bytecount;
    valp->timestamp = bpf_ktime_get_ns();


    return 0;
}
"""


def sort_output(unsorted_map):
    global sort_mode
    global sort_ascending

    output = unsorted_map
    if sort_mode == "C":
        output = sorted(output.items(), key=lambda x:
↪   x[1]['count'])
```

```
    elif sort_mode == "S":
        output = sorted(output.items(), key=lambda x:
↪   x[1]['bytecount'])
    elif sort_mode == "R":
        output = sorted(output.items(), key=lambda x:
↪   x[1]['bandwidth'])
    elif sort_mode == "B":
        output = sorted(output.items(), key=lambda x:
↪   x[1]['cps'])
    elif sort_mode == "N":
        output = sorted(output.items(), key=lambda x:
↪   x[1]['timestamp'])

    if sort_ascending:
        output = reversed(output)

    return list(output)

# Set stdin to non-blocking reads so we can poll for chars


def readKey(interval):
    new_settings = termios.tcgetattr(sys.stdin)
    new_settings[3] = new_settings[3] & ~(termios.ECHO |
↪   termios.ICANON)
    tty.setcbreak(sys.stdin.fileno())
    if select.select([sys.stdin], [], [], 5) == ([sys.stdin], [],
    ↪   []):
        key = sys.stdin.read(1).lower()
        global sort_mode

        if key == 't':
            global sort_ascending
            sort_ascending = not sort_ascending
        elif key == 'c':
            sort_mode = 'C'
        elif key == 's':
            sort_mode = 'S'
        elif key == 'r':
            sort_mode = 'R'
        elif key == 'b':
            sort_mode = 'B'
        elif key == 'n':
            sort_mode = 'N'
        elif key == 'd':
            global args
            if args.output is not None:
                dump_map()
        elif key == 'q':
            print("QUITTING")
            global exiting
            exiting = 1


def dump_map():
    global outfile
    global bpf
    global sorted_output

    keyhits = bpf.get_table("keyhits")
    out = open('/tmp/%s.json' % outfile, 'w')
    json_str = json.dumps(sorted_output)
    out.write(json_str)
    out.close
    keyhits.clear()


def run():
    global bpf
    global args
    global exiting
    global ebpf_text
    global sorted_output

    if args.ebpf:
        print(bpf_text)
        exit()

    usdt = USDT(pid=pid)
    # FIXME use fully specified version, port this to python
    usdt.enable_probe(probe="command__set",
↪   fn_name="trace_entry")
    bpf = BPF(text=bpf_text, usdt_contexts=[usdt])

    old_settings = termios.tcgetattr(sys.stdin)
    first_loop = True

    start = monotonic()  # FIXME would prefer monotonic_ns, if
↪   3.7+
    print("HERE")
    while True:
        try:
            if not first_loop:
                readKey(interval)
            else:
                first_loop = False
        except KeyboardInterrupt:
            exiting = 1

        # header
        if clear:
            print("\033c", end="")

        print("%-30s %8s %8s %8s %8s %8s" % ("MEMCACHED KEY",
         ↪   "CALLS",
                                              "OBJSIZE", "REQ/S",
                                              "BW(kbps)",
                                              ↪   "TOTAL"))
        keyhits = bpf.get_table("keyhits")
        line = 0
        interval = monotonic() - start

        data_map = {}
        for k, v in keyhits.items():
            shortkey = k.keystr[:v.keysize].decode('utf-8',
↪   'replace')
            data_map[shortkey] = {
                "count": v.count,
                "bytecount": v.bytecount,
                "totalbytes": v.totalbytes,
                "timestamp": v.timestamp,
                "cps": v.count / interval,
                "bandwidth": (v.totalbytes / 1000) / interval
            }

        sorted_output = sort_output(data_map)
        for i, tup in enumerate(sorted_output):  # FIXME sort
         ↪   this
            k = tup[0]
            v = tup[1]
            print("%-30s %8d %8d %8f %8f %8d" % (k, v['count'],
             ↪   v['bytecount'],
                                                  v['cps'],
↪   v['bandwidth'],
↪  v['totalbytes']))

            line += 1
            if line >= maxrows:
                break

        print((maxrows - line) * "\r\n")
        sys.stdout.write("[Curr: %s/%s Opt:
↪  %s:%s|%s:%s|%s:%s|%s:%s|%s:%s]" %
                          (sort_mode,
                           "Asc" if sort_ascending else "Dsc",
                           'C', sort_modes['C'],
                           'S', sort_modes['S'],
                           'R', sort_modes['R'],
                           'B', sort_modes['B'],
                           'N', sort_modes['N']
                           ))

        sys.stdout.write("[%s:%s %s:%s %s:%s]" % (
            'T', commands['T'],
            'D', commands['D'],
            'Q', commands['Q']
        ))
        print("\033[%d;%dH" % (0, 0))

        if exiting:
            termios.tcsetattr(sys.stdin, termios.TCSADRAIN,
↪  old_settings)
            print("\033c", end="")
            exit()


run()
```

Testing mctop tool

Initial basic testing of the `mctop` and `mcsnoop.bt` tools were made easier by `printf` to write commands to test tracing. At these lower call frequencies though, errors such as were encountered are not immediately obvious. It wasn't until `memtier_benchmark` was first used to generate load was it completely clear what the cause of the garbled key reads were.

Now that `mctop` has been cleaned up, and keys are stored properly this tool can be used to demonstrate how `mctop` works, and show that it can keep up with tracing requests to Memcached.

memtier benchmark

The `memtier_benchmark` tool can be used to generate load to the test Memcached instance that I built earlier, with dtrace probes enabled.

Rather than having to print to `nc`, this allows for rapidly firing off a large number of commands, showing that the tool is behaving as expected. This also gives a lot more data, for more interesting exploration of the tool, allowing for sorting on real data, and testing out dumping real data to a JSON file.

A simple invocation of the tool:

```
memtier_benchmark --server localhost --port 11211 -P memcache_text  --key-pattern=G:G
```

// FIXME dive into other options, show output in mctop

Final remarks

I hope that this has been an interesting, comprehensive, and comprehensible read. If you have any feedback on the content, please feel free to submit a pull request or contact me with your feedback. You can submit a pull request to the Github repository listed in the bibliography [2].

References

The following works were either cited directly or researched as part of the preparation of this report. They contain additional information for continued studies in this topic.

[1] D. Hamel, “Developing the mctop tool.” [Online]. Available: http://blog.srvthe.net/mctop-tool-example/

[2] D. Hamel, “Github Repository for this document.” [Online]. Available: http://github.com/dalehamel/mctop-tool-example/

[3] M. Barczak, “original mctop.” [Online]. Available: https://github.com/etsy/mctop

[4] M. Barczak, “mctop - a tool for analyzing memcache get traffic.” [Online]. Available: https://codeascraft.com/2012/12/13/mctop-a-tool-for-analyzing-memcache-get-traffic/

[5] B. Matheny, “memkeys tool, improvements on mctop.” [Online]. Available: https://github.com/tumblr/memkeys

[6] B. Mansoob, “Bassam Mansoob Github.” [Online]. Available: https://github.com/bmansoob

[7] R. Munroe, “XKCD Comic explaining Nerd-Sniping.” [Online]. Available: https://xkcd.com/356/

[8] C. Lopez, “Camilo Lopez github.” [Online]. Available: https://github.com/camilo

[9] “Dockerfile for memcached with dtrace.” [Online]. Available: https://github.com/dalehamel/mctop-tool-example/blob/master/src/docker/Dockerfile

[10] L. Windolf and M. Panji, “Memcached Cheatsheet.” [Online]. Available: https://lzone.de/cheat-sheet/memcached

[11] B. Gregg, “BPF Performance tools.” [Online]. Available: http://www.brendangregg.com/bpf-performance-tools-book.html

[12] B. Gregg, “BPF Performance tools book repo.” [Online]. Available: https://github.com/brendangregg/bpf-perf-tools-book/tree/master/originals

[13] “BCC USDT reference guide.” [Online]. Available: https://github.com/iovisor/bcc/blob/master/docs/reference_guide.md#6-usdt-probes

[14] D. Hamel, “USDT Report Doc.” [Online]. Available: https://blog.srvthe.net/usdt-report-doc/

[15] M. Mark Wielaard and F. Eigler, “UserSpaceProbeImplementation - Systemtap Wiki.” [Online]. Available: https://sourceware.org/systemtap/wiki/UserSpaceProbeImplementation

[16], “Dormando’s Home Page.” [Online]. Available: http://www.dormando.me/

[17] D. Hamel, “dtrace probes emit byte arrays for keys, not null terminated strings.” [Online]. Available: https://github.com/memcached/memcached/issues/576

[18] B. Smit, “Bas Smit github.” [Online]. Available: https://github.com/fbs

[19] Y. Song, “bpf_probe_read requires const len.” [Online]. Available: https://github.com/iovisor/bcc/issues/1260#issuecomment-406365168

[20] G. Borello and D. Miller, “bpf: allow helpers access to variable memory.” [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/davem/net-next.git/commit/kernel/bpf/verifier.c?id=06c1c049721a995dee2829ad13b24aaf5d7c5cce

[21] S. Sharma, “BPF internals - architecture.” [Online]. Available: https://github.com/iovisor/bpf-docs/blob/master/bpf-internals-2.md#architecture

[22] B. Gregg, “BCC contribution guidelines.” [Online]. Available: https://github.com/iovisor/bcc/blob/master/CONTRIBUTING-SCRIPTS.md#tools