**Introduction to performance tuning using perf\_events and Tracing**

Tracing is a very powerful technique that allows us to get information about the functions executions. This can be for debugging, optimization purpose or even security purpose. By tracing we can log the operation, the developer can use this information to optimize their apps and extract extra performance out of the available hardware.

Tracing is difference than debugging, where in debugging we attach the debugger to the process and use Linux signals call to pause the execution of a running program or kill it.

Before we start with system tracing any developer should start with the default options available in the Linux toolbox, such *as* ***top****,* ***Htop****,* ***vmstat****,* ***iostat****.* These quick build in tools are allows you to measure the utilization of system parameters. In what is called the USE method (Utilization, Saturation and Error). The following link is a good way to start.

<http://www.brendangregg.com/USEmethod/use-unix7th.html>

After identifying what the performance bottleneck (CPU, RAM, Disk, Network …) is we can start to dig deeper into the cause of the problem.

Then we can use profilers and debuggers such gprof to see how many times userspace functions are called and the time spent in these calls, we can also use it to identify the cause which lines is causing the performance problems. If the problem is within your code than it is possible to identify the line causing the problem and fixing possible performance issues.

Beside debugging and profiling tools there are also some tracing tool that allow us to dynamically collect data about a running program without the need to recompile it, or libraries we are using that we don’t need to recompile or we don’t have access for recompilation.

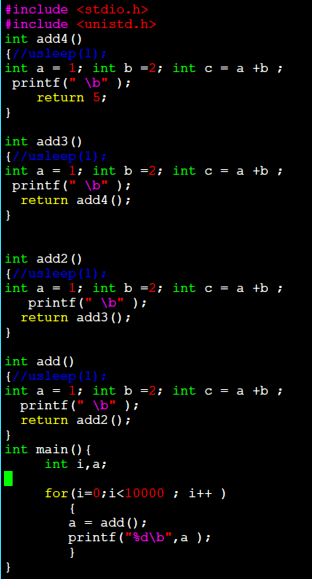
This document will first discuss the old tracing tool using ptrace, and then will discuss perf\_event, and finally the newer tracing tools built using ftrace, and then bcc tools.

1. **Old tracing tool**

There are some slower tracing tool that allow us to map the call stack of functions. These tools are build on top of Linux ptrace interface. This interface uses Linux signals to pause the execution of code on library or system calls to dump the information and extract relevant information from stack memory.

Strace (kernel space tracing) and ltrace (user space tracing) are build using ptrace interface, it is considered an invasive since it stops the execution of the code to get the trace. By doing so it will greatly affect speed. This should never be ran on production code. In our tests we found that strace was around tens of time slower than with no tracing while ltrace was a hundreds of times slower with no tracing

The following code is a test example hello.c. The goal of this code is to see how to track the call stack of the function.



To compile the code we use

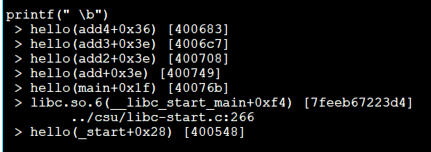
*gcc hello.c -g -o hello*

This will output a hello binary with call stack and symbols enabled. We can use the time function to get the runtime of this function

*time ./hello*



*ltrace –w 200 ./hello* # this will generate the call stack with 200 maximum character for every function call. This is one example of the output (this will repeat for every function call)



To time the ltrace :

*time ltrace –w 200 ./hello*



The slowdown when using ltrace : 24.595 / 0.003 = 8198.33

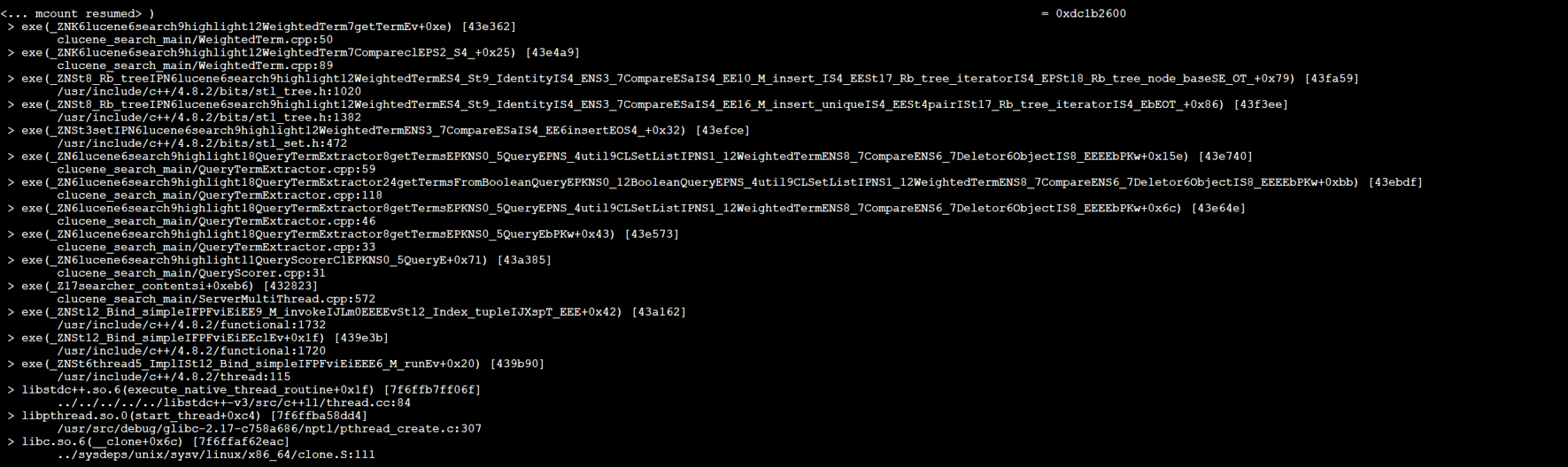
This huge slowdown is the reason why this should not be used in production.

Another approach to see the call stack of user function is to use gdb. Which is the gcc debugger. In order to use the debugger correctly we have first compile with the correct gdb debugging information that allows the

Testing ltrace on the clucene code we can use ltrace, but since ltrace kills performance

Our clucene test includes two binaries a client and server binary. We launched one client and 1 server with 4 threads. With no ltrace running the server is stable and can run till completion or time out. We can either launch the server with ltrace or we can attach the server to ltrace by using –p PID with ltrace.

The result are shown in the image below. Note that the server performance dropped just after attaching ltrace and eventually the server crashed. But it did allow to take a look at the call stack from the server side. This can helpful to check how functions are being called.



Another approach to see the call stack is to use the backtrace command from gdb.

In order to use a gdb on a code it must be compiled with the –pg flag otherwise the gdb will fail to read the symbols.

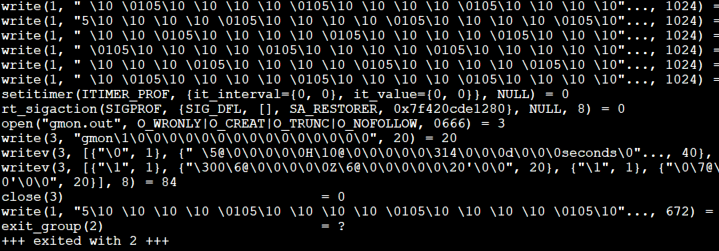
In case some symbol where missing the gdb will suggest some commands to download the debug library of the required library so the debugger can run well, the command used is ***debuginfo-install*** along with the name of debug library. This will suggested by the gdb, simply copy and past the suggested command to the terminal and run it.

strace allow us to check kernel calls

strace testing for hello.c

*strace ./hello #* allow us to see all system call issues by the program.

*strace -f ./hello*# allow us to see all children process system call also



Timing strace gave us 0.017 second, which is around 6 slower than running without tracing.

As we can see both strace and ltrace slows down the speed of execution of the system. This is why modern Linux system started to include the more modern ftrace that has significantly lower performance overhead. But these tools can still be helpful since they are easy to use, and usually available on most linux distribution.

1. **Perf event tools**

Perf\_events is a wide collection of performance tools that allows us to monitor performance profile. It comes with most distribution of Linux. If it is was missing from the distribution we can install it with apt-get or yum “linux-tools-common”.

Perf\_events first started as a tool to program and performance counters inside modern microprocessors based on events such as interrupts, overflows or preemption. Over the years it gain the ability to read software events, kernel events, tracepoints and many other events.

*perf –h*# this command will allow us to see all available tools inside perf\_events . This might every based on your system and

*perf list*# this will allow us to see what events can we diagnose

*perf stat* is probably the most popular tool and the simplest to use. This tool simply counts the number of time an event happens , we can use *perf stat –h* or *man perf stat* to get more information about this tool

perf stat will print out the values of counted events, we can use than number to diagnose the performance problem. A high L1 miss and branch miss might indicate unpredictable loops or if statement, a high L3 miss might indicate unoptimized data usage or memory bandwidth issues. We can use perf events with the following example:

*perf list* # to list all possible events that we can use

*perf stat –a sleep 5#* monitor default performance index for 5 seconds on all CPUs. The following is an example of the output for a system with 56 cores and 112 threads

Performance counter stats for 'system wide':

560,413.40 msec cpu-clock # 110.906 CPUs utilized

106,339 context-switches # 0.190 K/sec

1 cpu-migrations # 0.000 K/sec

3,294 page-faults # 0.006 K/sec

6,225,406,272 cycles # 0.011 GHz

6,837,012,117 instructions # 1.10 insn per cycle

1,626,254,812 branches # 2.902 M/sec

16,828,800 branch-misses # 1.03% of all branches

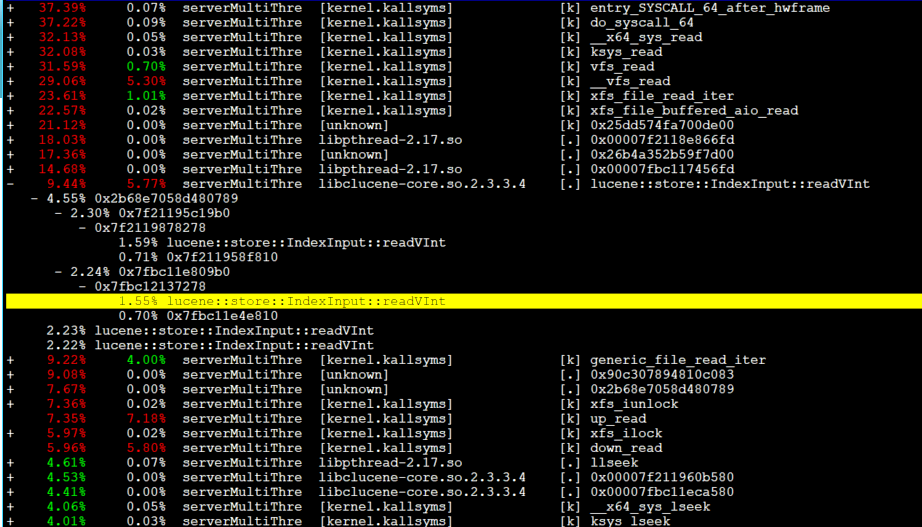
5.053071214 seconds time elapsed

*perf stat –e cycles,instructions ./my\_binary* # allow use to us counters to get cycles and instructions from our binary.

*perf record -F 99 –e cycles,instructions ./my\_binary* # record cycles and instructions and store output 99 times every second

*perf record –F 99 -g –e cycles,instructions ./my\_binary*# same as above but also record –g with call stack of functions

*perf report –g* # allows us to see the functions that spend of the time executing.



If the symbols are correctly displayed you can track the call stack and see which function took the most time.

The library should be compiled with –g option to make sure that these function are correctly labeled, otherwise it might result in broken symbols.

We can also use the command ***perf top -g*** to get a real-time report every 4 seconds (configurable) to detect call traces.

*perf trace* allows to see registered system calls and prints out the time stamp.

For example we can use ***perf trace*** to see when sending and receiving tcp messages.

*perf trace --call-graph fp*

The command above allow you to trace the call graph of system calls, if available.

To add an event to perf trace we must first register it with perf event, for example:

*perf probe --add tcp\_sendmsg*

Perf\_events can also use ftrace to add custom events and record. This is explained in the ftrace section since perf trace uses ftrace interface to get its output

1. **Ftrace (built in kernel tracer)**

Ftrace is a modern tracing interface that uses in program hooks to read the stack and redirect the result to a fast ring buffer to give more information about the execution habit of the program. The built in ftrace can use sysfs interface to provide the tracing information.

The location of the tracing sysfs directory is in:

*/sys/kernel/tracing/* or */sys/kernel/debug/tracing/*

In our machine the location is the latter:

*cat /sys/kernel/debug/tracing/available\_events* # to see all available kernel events.

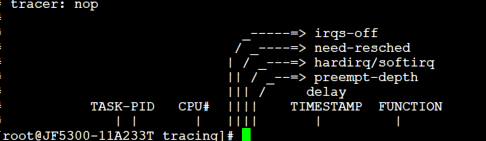
**all the next steps requires us to be the trace diretory:**

at default the trace is empty.

*cat current\_tracer*

this command will show the current kernel function being traced. In this case it is **nop** , which indicate no function is being traced

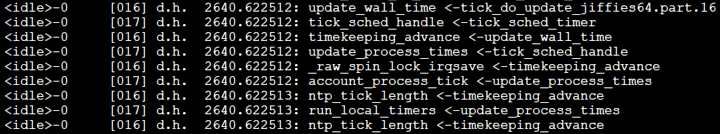
*cat trace* # will show you the output of traced function, but in order to display anything we need first to enable tracers. In this case the tracer is empty



To enable something to be traced we can use echo command

*echo function > current\_tracer**#* this will allow use to trace most system calls

we can then see the output using *cat trace* , the output looks something like this:

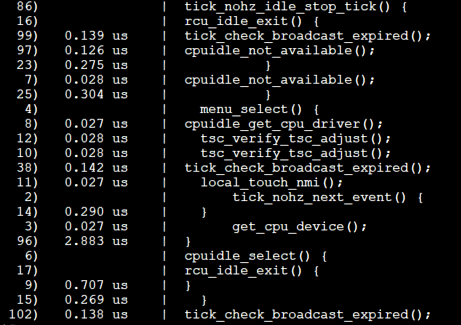


This output will fill the screen immediately. To disable all tracer we can do:

*echo nop > current\_tracer*

we can also have a c like call graph of function by using:

*echo function\_graph > current\_tracer #*which will print the output of these command in a c-code like origination



Once you do *echo nop > current\_tracer* the tracing is disabled and ring buffer is cleared

Alternatively we can pause or disable ftrace using ***echo 0 > tracing\_on*** . This will pause the output to the ring buffer, but it not disable tracing an its overhead, to re-enable output we can use:

*echo 1 >**tracing\_on*

Also make sure that we have at least one space between on and the output otherwise Linux bash won’t accept the command.

We can also enable a specific function tracing

*echo schedule > set\_ftrace\_filter*

we can also use the glob or wildcard symbols such as *echo ‘xen\*’ > set\_ftrace\_filter* we can aslo append a value by using >> instead of > . We can also add *?* instead of letter for example

*echo ‘x?n\*’ >> set\_ftrace\_filter*

*echo > set\_ftrace\_filter*# this remove all traced functions

Some kernel functions will share the same name. like SyS\_read and sys\_read. Ideally we want to use sys\_read since it is the actual implementation while SyS\_read is a function that will call sys\_read

There is a folder called options. Inside we can enable func\_stack\_trace which issue a stack dump on system call. Note that this will kill the performance of the machine. It is very important to filter the command first otherwise this might result in system become unresponsive as every kernel call is doing a stack dump. The option to do so is with:

*echo 1 > options/func\_stack\_trace* ***#*** we disable it with 0 instead of 1

To trace a specific pid:

*sh –c ‘echo $$’ > set\_ftrace\_pid; echo 1 > tracing\_on ; exec ./path/to/binary*

To enable stack tracing:

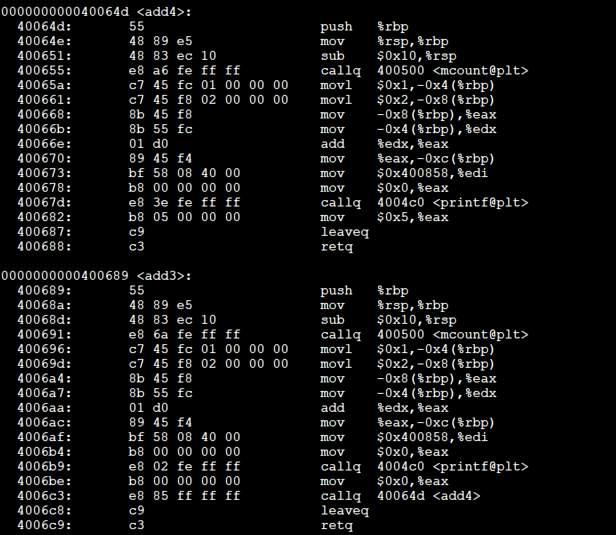
*echo 1 > max\_graph\_depth*

There is also a tool called trace-cmd which allow us to do the same command but from anywhere in system (substitute trace with trace-cmd) but we need to download and install it first.

1. **User level tracing**

To trace user level function we fist have to get the address of the function

*objdump –d hello* # this will give the output dump with symbols resolved. We can use either the symbol or the actual address for tracing. The

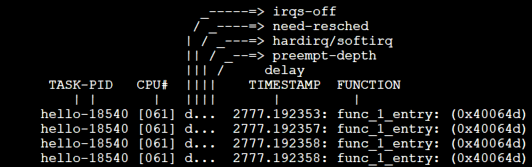


*echo 'p:func\_1\_entry hello:0x64d' > /sys/kernel/debug/tracing/uprobe\_events*# set up ftrace filter

*echo 1 > /sys/kernel/debug/tracing/events/uprobes/enable*# enable ftrace

*./hello run program*

*cat /sys/kernel/debug/tracing/trace*# sees the user trace



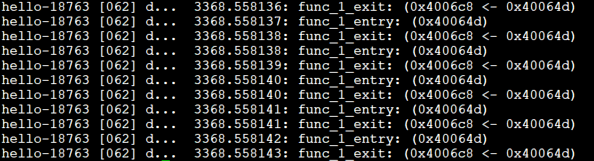
To add another probe:

*echo 0 > /sys/kernel/debug/tracing/events/uprobes/enable* # disable uprobe

*echo 'r:func\_1\_exit hello:0x64d' >> /sys/kernel/debug/tracing/uprobe\_events*# allow another user tracing with return probe (r) . >> will append the return probe. If we used > it will delete all previous entry and only trace the return probe.

*echo 1 > /sys/kernel/debug/tracing/events/uprobes/enable*# enable probes

running our *./hello* and using *cat /sys/kernel/debug/tracing/trace* will allow us to see the probe points.



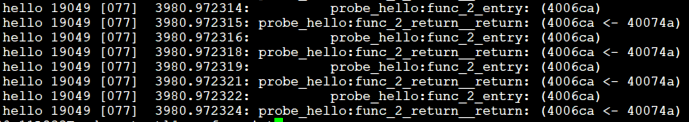
1. **Perf event tracing**

*perf probe -x ./hello func\_2\_entry=add2*# create trace point called func\_2\_entry for function add2

*perf probe -x ./hello func\_2\_return=add2%return*# create trace point called func\_2\_return for the return instruction in the function add2

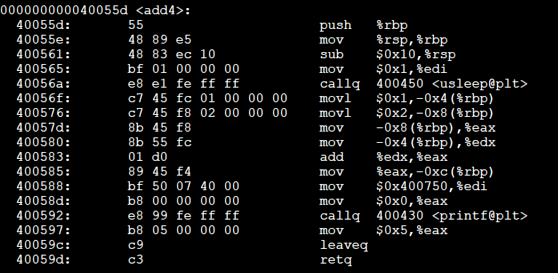
*perf record -e probe\_hello:func\_2\_entry -e probe\_hello:func\_2\_return\_\_return ./hello*# record information when hitting these tracepoints.

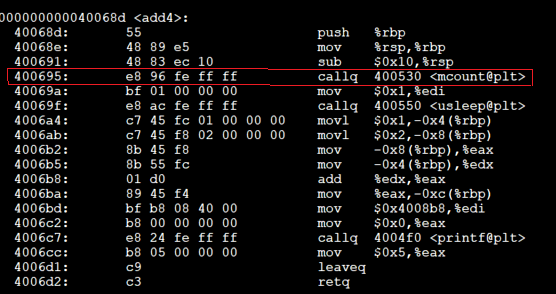
*perf script*# display the output of this perf record when using tracepoint



Notes

Probes and tracepoints are different. While using probes the kernel dynamically changes the assembly instructions and insert additional functionality with the probe. Tracepoints are static points compiled within the code that allows tracers to get additional information from the running program. Tracepoints are skipped if not used while running . In the example below we see the difference between a userspace compiled with and without –pg option.





Mcount is dynamic probe, in can be skipped while excecution if we desire, ot it can trigger a function call that pause the execution to allow for debugging or exection, note that the mcount is callq instruction which is basically a function call.

For more information on how ftrace works you can check this link:

<https://blog.linuxplumbersconf.org/2014/ocw/system/presentations/1773/original/ftrace-kernel-hooks-2014.pdf>

In summary, if the kernel is compiled with –pg option and we have defined and added the ftrace structure inside the kernel which will add mcount instruction inside the code, this function call is replaced with nop (no operation instruction) in link time for performance reasons, and the address of this nop instruction is kept in page file in memory. When running a tracer, or probe, the nop instruction is replaced with the address of a function call (which correspond with our probe or tracer). This function would save the state of registers and arguments, execute the probe and then reload the args and return to normal code execution of code.

If the kernel was compiled with –pg option, this will allow us to probe almost any function inside the kernel. This inserted nop operation are sometimes referred to as kernel hooks.

1. **BPF tracing**

BPF tracing uses ftrace to trace inside the kernel. Tracing uses probes on kernel or userspace hooks, once these get hit, it calls a function which pauses the execution of the running code and allows some operation on the registers and stack of the running programs. However this might be dangerous since it can cause application or even system crashing. There are multiple ways to create probes, with kprobe, dtrace and ftrace, but the best way is to use the BPF tools.

Modern version of Linux has a built in eBPF virtual machine (sometimes referred to as BPF), which allows the execution of kernel probes in a safe way inside the kernel. Once probes are hit it allows the execution of some prewritten eBPF functions. These function are already compiled and passed through a verifier and additional checks to make sure that the written functions are not executing an unsafe or non ending code.

The default way is to create BPF kernel by writing assembly or bytecode code, but it is preferable write the BPF code in the new bcc compiler, note that there are some restrictions on what is allowed to be done inside the BPF kprobes to ensure there stability like small size (used to be 4KB and now expanded up to 196 KB). Also bcc code cannot include loop or going back in the code (Loops has to be unrolled which will expand the code).

eBPF needs at least 4.0 kernel , but it is preferable to get 4.9 or newer as some tools requires new kernel.

To download the bcc use the following github link, this link also contains the description for lots of built-in tools written using eBPF

<https://github.com/iovisor/bcc>

To install bcc-compiler and tools use the following link (make sure to follow instructions that correlate to your OS):

<https://github.com/iovisor/bcc/blob/master/INSTALL.md#centos---source>

Once installed the collection of tools is availed at ..~INSTALL\_DIR/bcc/tools

Try to use available tools first to see if they match your needs

Example :

*./trace.py –t –K tcp\_recvmsg* #allows you to call tp tcp\_recvmsg kernel fuction call along with the call stack

*./funclatency.py tcp\_recvmsg* # print out in kernel histogram of latency of specific function

.

With each tools there is an example.txt that demonstrate some examples with how the program works

To see all kernel functions that can be traced we can use :

***$*** *cat /proc/kallsyms*

*./trace.py –t –K [kernel\_function\_name]*

In this list we can find all functions that are capable of being traced

Since there are so many kernel registered function we can direct the output to ***less*** command or ***grep*** command

What happen behind the scene in these tools in that there is a BPF code written in C, each tool bundled with the bcc compile the code and uses python to attach the code as kprobe or uprobes. The compiler has some predefined maps and arrays and that allows copying information directly for kernel space to user space. The recommend approach is to use the ***perf\_events*** fast ring buffer to copy the data.

Some data can be processed within the kernel such as a built histograms which reduces the amount of copies to user space, or we can dump the data from kernel space to user space and process them there.

.

.

You can create your own bcc code using the bcc tool,

It is typical to use python to set argument and process data in user space and BPF C code to write probe in kernel space.

If you need to write you own probe you can use:

<https://github.com/iovisor/bcc/blob/master/docs/tutorial_bcc_python_developer.md>

This includes entry level tutorial and examples to write simple probes and build from there.

Additional resources and BPF functions can found at the following page:

<https://github.com/iovisor/bcc/blob/master/docs/reference_guide.md>

While eBPF is a good way to trace kernel functions, it might behave strangely with user level functions and libraries

The trace.py tool from bcc can trace user level function can trace USDT written libraries .

to trace a specific function call we have to use ***objdump*** or ***readelf*** on the so binary and extract the function name before using it.

We can check the functionality of the command with man command

*man nm*

*man readelf*

*man objdump*

What these function allow us to do is read the symbol file and get the symbols of specific call. We can use these symbols to trace these libraries

For example the next function allow us to read the symbol file for c library

*readelf –a /usr/lib64/libc-2.17.so*

While the next line allow use to trace the usage of *printf* in library

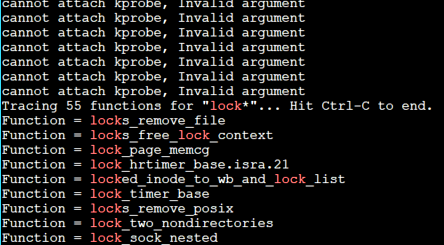
*./trace.py /usr/lib64/libc-2.17.so:printf*

To make sure that libraries wok well they have to be compiled with correct flags that allows them to properly show symbols

Like *–rdynamic*

To check what function can be traced we can use the funclatency tool (from bcc). The funclatency traces the in kernel latency, and prints in kernel histogram, behind the scene, the funclatency tool also tries to match all functions to the pattern (if we give the wildcard symbol \*).

*./funclatency.py -d 5 -F “\*lock\*” | grep lock*# The output of this command is the following :



Everything in red can be traced. For lock we can trace only these 9 function.

The funclatency.py tries to find every the “lock\*” pattern to /proc/kallsyms , There are lots of matches, it then tries to attach a probe for 5 seconds with args “-d 5”. The “-F” args tells the function to have separate output for every funclatency, then it will pipe the output to grep which will parse the output to find which functions did produce an output. Without grep it will print out a separate histogram for each successful function probing.

Another way to get the call stack of a kernel and userspace function is to use *offcputime.py* form the bcc

*offcputime.py*# this tool run until ctrl+c and prints out the trace of every function within that period.

*offcputime.py –U –p 34504 5* ***#*** this function check the userspace call (-U) stack of functions in process PID of 34504 (-P) for 5 seconds.

Behind the scene bcc tool are based on ftrace

**Bcc code example:**

A typical bcc program is written in python. At first it will import the necessary libraries where BPF is the most important module since it is one that will handle compilation of the code BPF code and the insertion of kprobe and uprobe.

Then it can followed by an argument parser if needed, which handles some additional options if available.

It will be then followed by a BPF code. This BPF code is written between triple quotations as a string. Within in it we should import the necessary libraries, define our types and write some probe handlers. The BPF context have some additional data type and function to make some operation easier, like maps, histograms, timing functions and PID parser.

We then compile the code with python function b =BPF(BPF\_string\_code). In some advanced BPF program they use python to modify the BPF code before it complies, for example we can divide the time by 1000,000 to convert from nanosecond to microsecond by inserting a line to divide the timing 1000000 in the BPF code There are multiple examples of this in the tools such as funclatency.py.

The final step is get the information from BPF kernel space to user space.

There are multiple ways to do this, such as bpf\_trace\_printk function (like example below), but the recommending one is BPF\_PERF\_OUTPUT, which uses perf\_events ring buffer for fast copies from kernel space to user space. Note that perf\_events does not need to be running. We are just using the same way that perf is copying the data from kernel space to user space since it is fast.

The bcc program below is a modified example form the python bcc beginner guide. The example simply get the delay between two consecutive rsync calls.

When we call this program, the code gets compiled with bcc Clang compiler and then put into a verifier to make sure that this code is safe to run. After the verifier makes sure that it is safe it will be inserted into the kernel. The nop instruction gets replaced with a function call that points to the location in memory where this code is compiled.

The code defines a hash array in memory to keep information between multiple call. At each call the current date is obtained, it is then subtracted from the previous stored value. If the value is greater than one second it is printed into a kernel buffer.

Now we have attached a kprobe with its own function handler. We now wait for a rsync call to happen

when an rsync system call happens the rsync will run normally run until it reach to the kprobe (previously a Nop instruction which was previously a mcount function compiled from gcc –pg). The probe will cause the kernel function to call out BPF program, which will do its work. This will happen as long as the probe is attached.

Now the BPF program only print inside the kernel this is won’t be visible directly to the user. To get it to user space we have another function that copies the information form kernel space and prints it out in user space. We use this function in a loop to read consciously read out form the print buffer from the kernel. We should also mention that the buffer is small in size hence we should read the information as fast as possible so it wouldn’t overflow.

Just like BPF can attach kprobe for a function, it can also attach a kretprobe, which it is triggered on the return of kernel function. We can then attach the same BPF function handler or a different function to handle kretprobe.

Similarly, we can use BPF to attach uprobes and uretprobes which are user space probe to trace user level libraries, this can be helpful if we are probing a precompiled library. Probing a user library is a bit different since we need to get the address of the library.so file and attempt to find first the address or symbol of the function call we want to trace.

If there are missing symbols in the libraries the output of the stack would be unreadable.

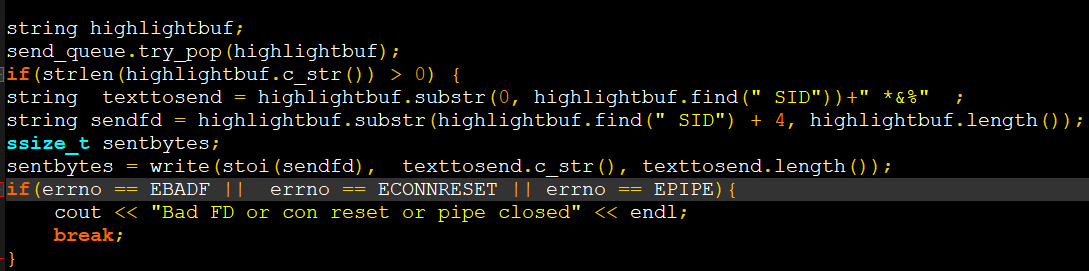


**Examples of userspace tracing:**

**Using perf:**

We already talked about tracing using perf event. Let’s see an example in action

We are no trying to trace the function try\_pop from serverMultithread code (2nd line):



This code is written within a while loop (not visible in this snippet). It will constantly look at the concurrent buffer and attempt to pop out the result and then send the result to connected client on the correct port.

To trace this function we can use the probe event form perf\_event tools.

First we need to add two tracepoints. The first is the entry of try\_pop queue:

*perf probe -x ./serverMultiThread try\_pop\_entry=try\_pop*

and the second is the return of this function:

*perf probe -x ./serverMultiThread try\_pop\_return=try\_pop%return*

once an event name is added perf tool give us an example how it was named :

try\_pop\_entry=try\_pop => probe\_serverMultiThread:try\_pop\_entry

try\_pop\_return=try\_pop%return => probe\_serverMultiThread:try\_pop\_return\_\_return

To run these trace point :

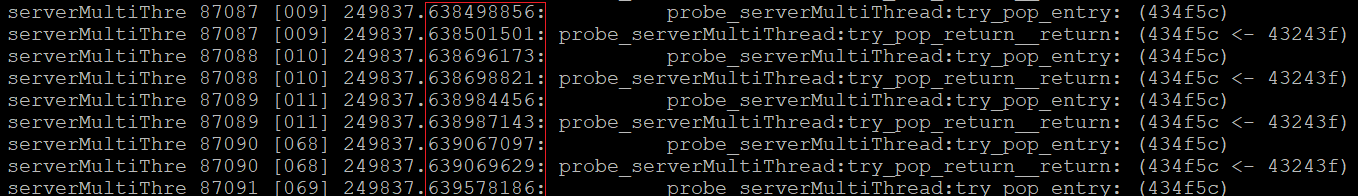
*perf record -e probe\_serverMultiThread:try\_pop\_entry -e probe\_serverMultiThread:try\_pop\_return\_\_return ./B\_new\_script.sh*

Note that once we type in the command line *perf probe –e probe\_* and we then press the **tab** button it will auto complete with the available trace points, or we can double tap on the **tab** button, it will list all available tracepoints.

Finally after recording we can see the result with two approach

*Perf report* or *perf script*

The perf script looks similar to the output of regular ftrace

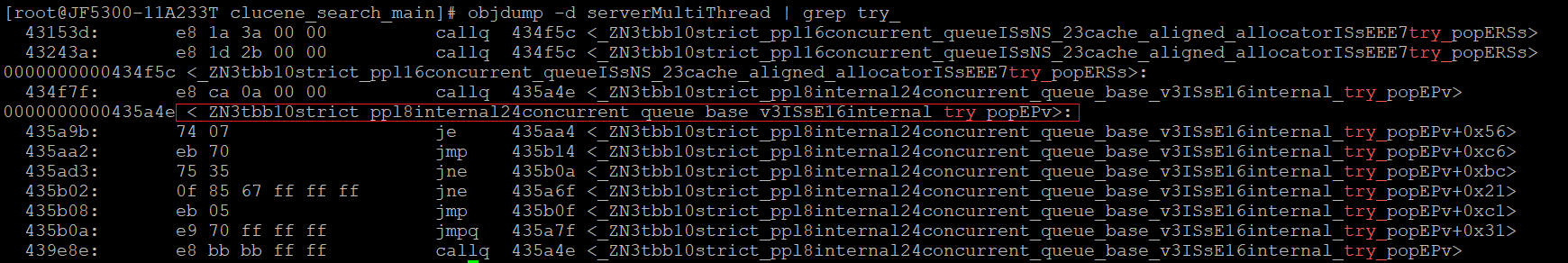


The image above shows a snippet from the output of *perf script –ns.* From this sample we can see the output different as 2500 ns or 2.5 us.

We can trace the same function using ebpf tools. But first we need to get the symbol

So we do:

*objdump -d serverMultiThread | grep try\_*

****

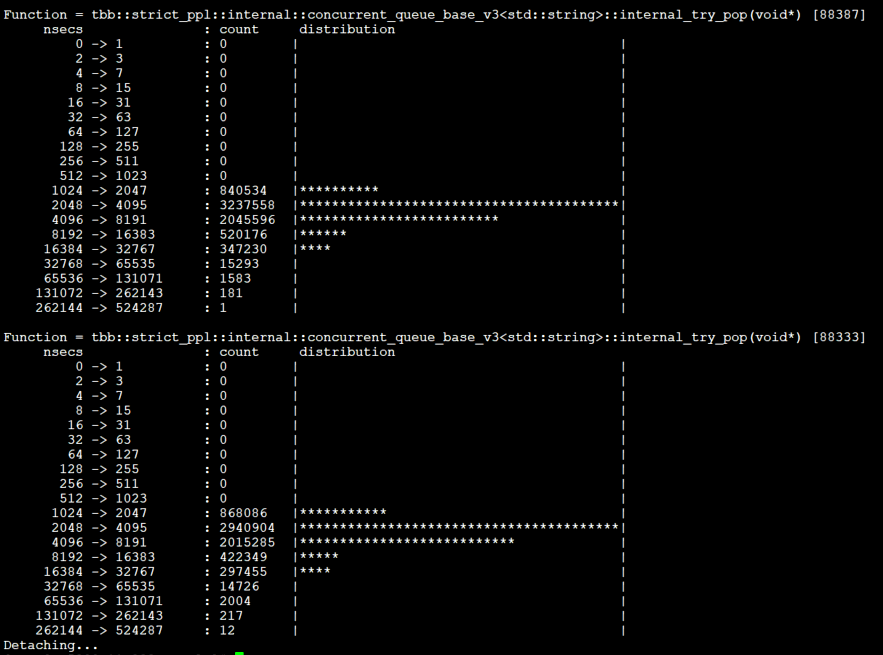
The raw internal symbol is:

*:\_ZN3tbb10strict\_ppl8internal24concurrent\_queue\_base\_v3ISsE16internal\_try\_popEPv*

we use this symbol with the bcc tool funclatency.py to get the histogram if latency of executing this function.

*./funclatency.py /pnpdata/clucene\_benchmark\_new/src/benchmark-dev/clucene\_search\_main/serverMultiThread:\_ZN3tbb10strict\_ppl8internal24concurrent\_queue\_base\_v3ISsE16internal\_try\_popEPv*

This bcc command will attach a uprobe and uretprobe when this function is called and once it return and measure the difference, and use the difference to build a histogram of latency. The output of this histogram is shown in the image below

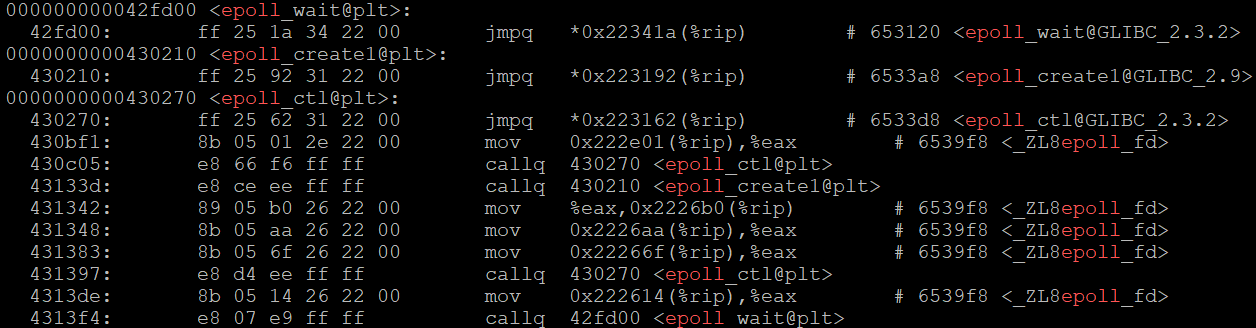


Form this image we found the latencies are most common between 2000-4000 ns (2-4 us) which goes nicely with the sample from perf\_event tool that got us 2.5 us delay.

Example 2 :

Lets attempt to trace the latency of epoll\_wait.

*objdump -d serverMultiThread | grep epoll*

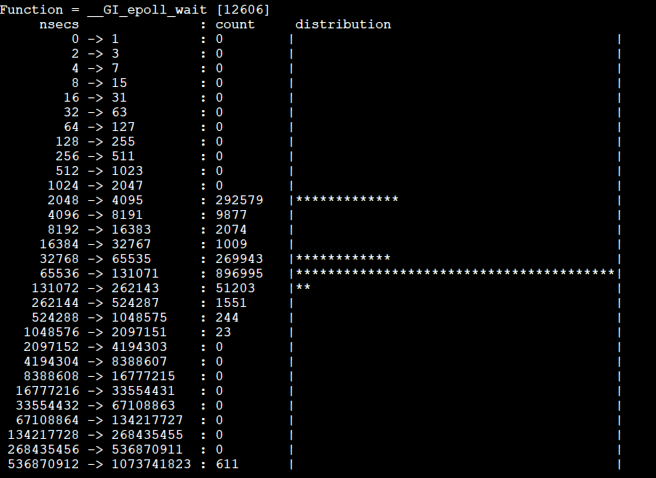


In this example we can see that **epoll\_wait@glib\_c2.3.2** indicates that this is part of glibc library.

Tracing this would be to find the glibc library and initiate a trace on that function.

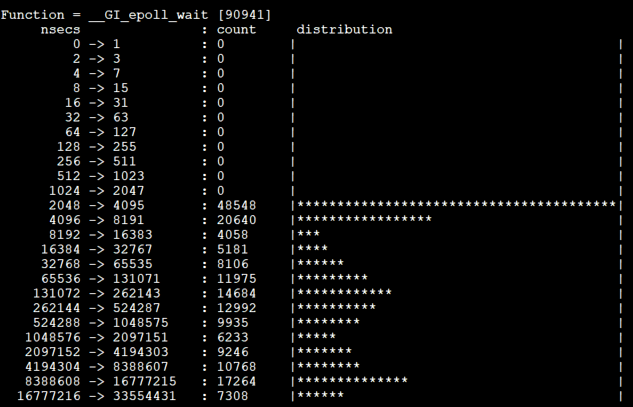
*./funclatency.py c:epoll\_wait* or *./funclatency.py /usr/lib64/libc-2.17.so:epoll\_wait*

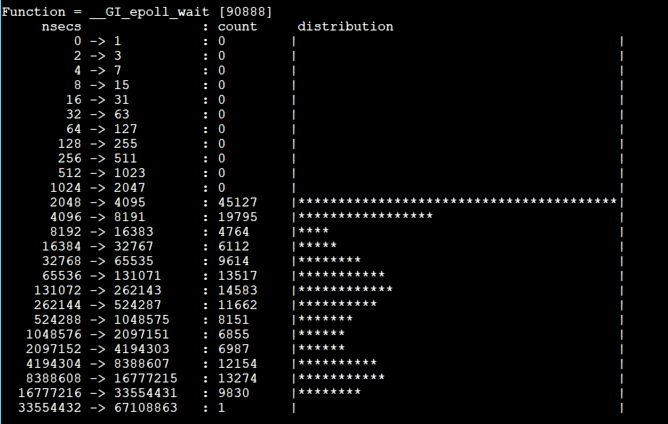
Doing so resulted in a 13 different histogram, the value printed next to the histogram represent the process ID, hence to get the correct latency for our trace we should only look at the histogram with the desired PID, for example this might be deceptive since this is no the PID of the process that we are interested in. We can then look manually for the latency with our desired PID, or use the –p argument to restrict the tracing to the desired PID.



Hence to get the correct latency we should use the correct pid ,

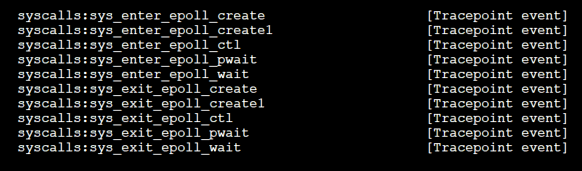
In our case we launched two clucene server each with a different pid we used the funclatency to get the average delay of epoll wait. We are only showing the PID that correspond with our server threads. The most common latency were between 2-4 µs we can see that latency can go up to 16,000 µs or 16 ms.





We can also use perf event to record and output the latency of epoll.

*perf list epoll #* allow us to see the list event with “epoll” in the the name



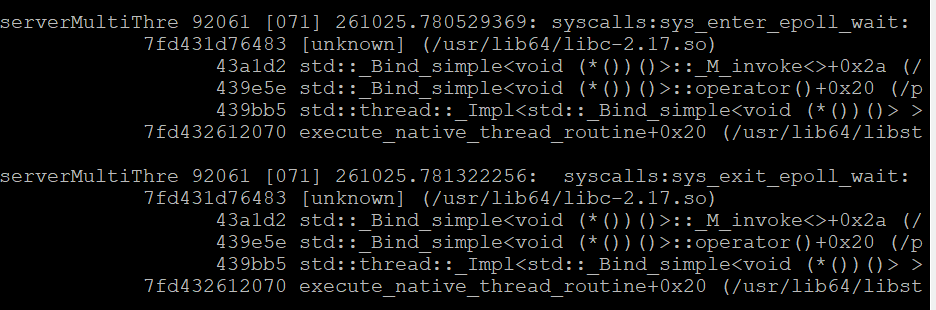
We don’t have to create our own event since these are predefined static trace point inside the kernel. we can directly record the corresponding event with:

*perf record -e syscalls:sys\_enter\_epoll\_wait -e syscalls:sys\_exit\_epoll\_wait -a –g*

what the command above will do that It will record all entries and exist of system call epoll\_wait both on entry and exit.

To visualize the output we can use:

*perf script --ns --pid 92059*



In this example we can see the timestamp of the entry and exit of this function call in nanosecond for PID *92059* which is one of the two server instance running. In this example we find the latency to be ~0.7 ms or 700 µs. this fall within the upper range of histogram latency.

**Additional References**

For additional references about tracing :

<https://jvns.ca/blog/2017/07/05/linux-tracing-systems/>