Engenharia de Sistemas de Computação
Performance Optimisation

Class 3 – Profiling (contd)

Vítor Oliveira (vitor.s.p.oliveira@gmail.com)

13 Abril 2021

1 Profiling (contd)

Profiling consists on analyzing the behavior of the systems and applications providing views of the way resources are being used.

CPU Profiling

- With a sampling profiler one can track the functions that took the longest to execute by checking how many samples contains that functions in the top of the call stack. The parent/child execution time is taken as those functions are on the call stack of any sample.
- While there are profilers that support wall-clock time profiling, that is not the common, the samples are taken at regular intervals only while they are running on the CPU, so they don’t account for time spent being scheduled out or waiting on storage or network.
- Most importantly, they don’t consider the time waiting on synchronization primitives, so they won’t help when we have a slow system due to ineffective coordination or even contention on the locks.

CPU Profiling

Sampling:

- Interrupt the CPU regularly to check what is the code currently being executed;
- Store the process id and the instruction pointer in the tracing information;
- If the intermediate functions are needed, save all the pointers in the call-stack;
- Self time it the ratio between the number of samples where this function appears as terminal and the total number of samples;
- Time of the childs is the ratio between the number of samples where the function appears in the call stack and the total number of sample;
- Only processes that are running are sampled, if they are waiting they are skipped.

CPU Profiling

Events:

- Usually the sampling event is cpu time
  - After a few miliseconds running in the cpu a signal is sent to interrupt the process;
  - SIGPROF if running a profiler in user mode, kernel mode does not need to interrupt the process;

- In perf other events can trigger the sample, like hardware counters;

Function Profiling

- Code instrumentation;
- Dynamic instrumentation;
- Kernel probes
- User-mode probes

5

Profiling – Linux Perf

The perf profiler has been gaining a lot of features and is the reference for Linux profiling.

Profiling – Linux Perf

Perf

- perf profiling counts the time a process spends in the cpu
  - It does not count waits on IO or on locks after the process is scheduled out;
  - Threads waiting will be mostly ignored by the profiler, as it runs at a specified rate on each cpu.
- perf has many many options
  - One can select many options for profiling, including hardware counters, system probes

Profiling – Linux Perf

Several interesting features:

- perf top –g (callstacks with -g)
  Similar to perf top with profiling information
- perf stat
  Executes an application and shows basic information
- perf record –g
  Records the profile of an application for later treatment
- perf report -g –G
  Reports the execution previously recorded
- perf script
  Allows profiling data to be exported;
- ...

Profiling – Linux Perf

Perf

```
sudo perf stat ./messages -2 -c0 -c1
synctest: running 500000000 iteractions with 2 threads, using test2 on 2 cpus
waiting for 2 threads: Iter [1/15] took 434002 ns, 434002 cycles
…
Iter [15/15] took 429428 ns, 429428 cycles
Average time 0.432505 us, median time 0.432205 us
Test ended after 6.5s.

 Performance counter stats for './messages -2 -c0 -c1':

     12885,760924      task-clock (msec)         #    1,986 CPUs utilized
             1049      context-switches          #    0,081 K/sec
                3      cpu-migrations            #    0,000 K/sec
               83      page-faults               #    0,006 K/sec
      37737586848      cycles                    #    2,929 GHz
      30521095840      instructions              #    0,81  insn per cycle
       4528614326      branches                  #  351,443 M/sec
         43203491      branch-misses             #    0,95% of all branches

      6,489106135 seconds time elapsed
```

Profiling – Linux Perf

Perf

$ sudo perf record -g ./messages -2 -c0 -c1
synctest: running 500000000 iteractions with 2 threads, using test2 on 2 cpus
waiting for 2 threads: Iter [1/15] took 434002 ns, 434002 cycles
…
Iter [15/15] took 429428 ns, 429428 cycles
Average time 0.432505 us, median time 0.432205 us
Test ended after 6.5s.

$ sudo perf report -G

6 Visualizations

Visualization

Goals

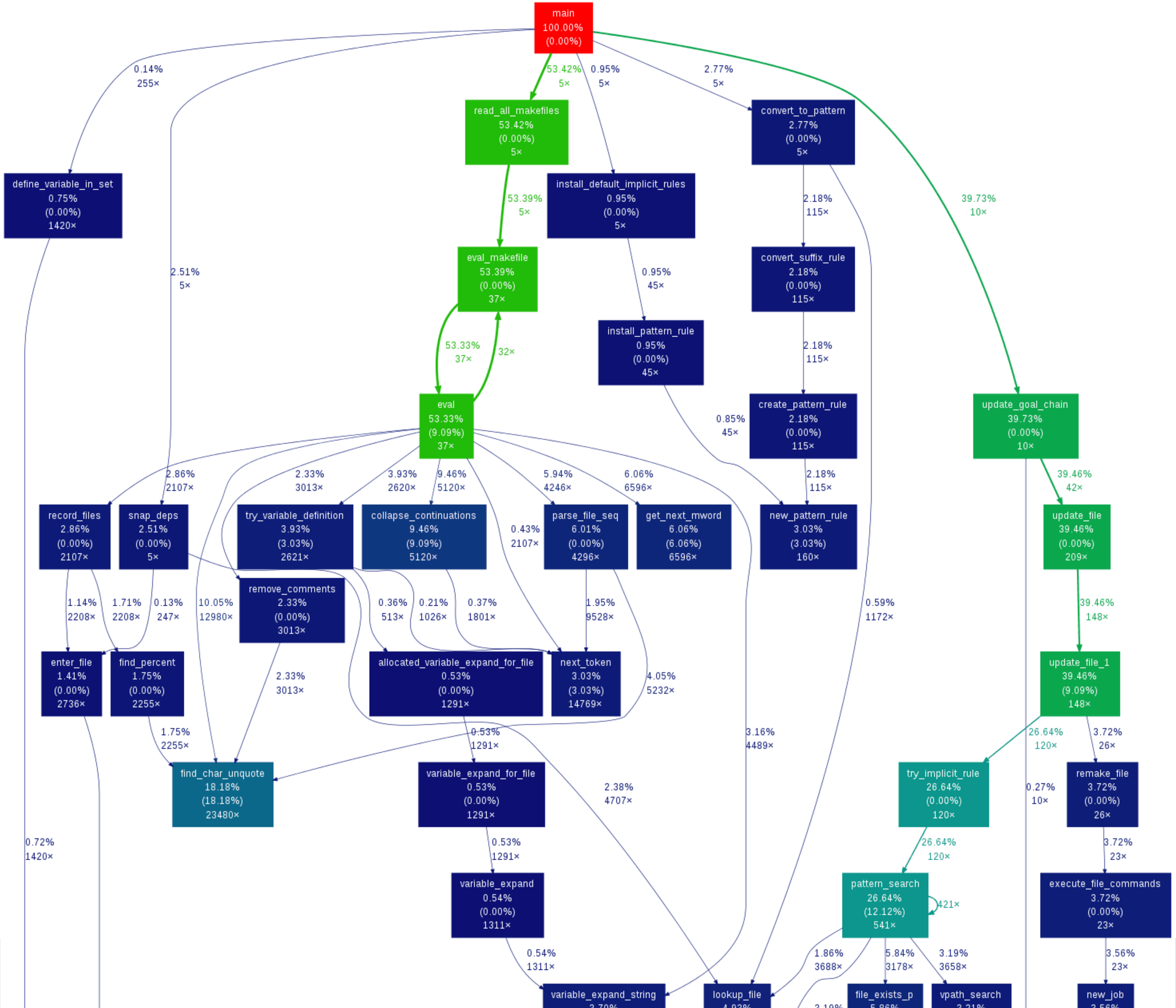
- Profiles can be examined in many diferent ways, depending on the type of information being analised;
- For sampling profiles it is common to use call-stack graphs, which shows the functions that are most used by the application, according to some selected metric;
- More recently, Flame graphs have become more used; it shows very quickly which is the usage of the main branchs, but it hides functions that are used in many places in the code;
- The next flamegraphs show the profile of the full mysqld process executing a Sysbench Update Non-Index with 32 threads, focusing on some of the main functions related to the binlog.

Visualization

For in depth analysis of the CPU profile the best tool is usually a call stack graph, which connects nodes using arcs that show the CPU utilization of each function and the functions that called.

Callgraph

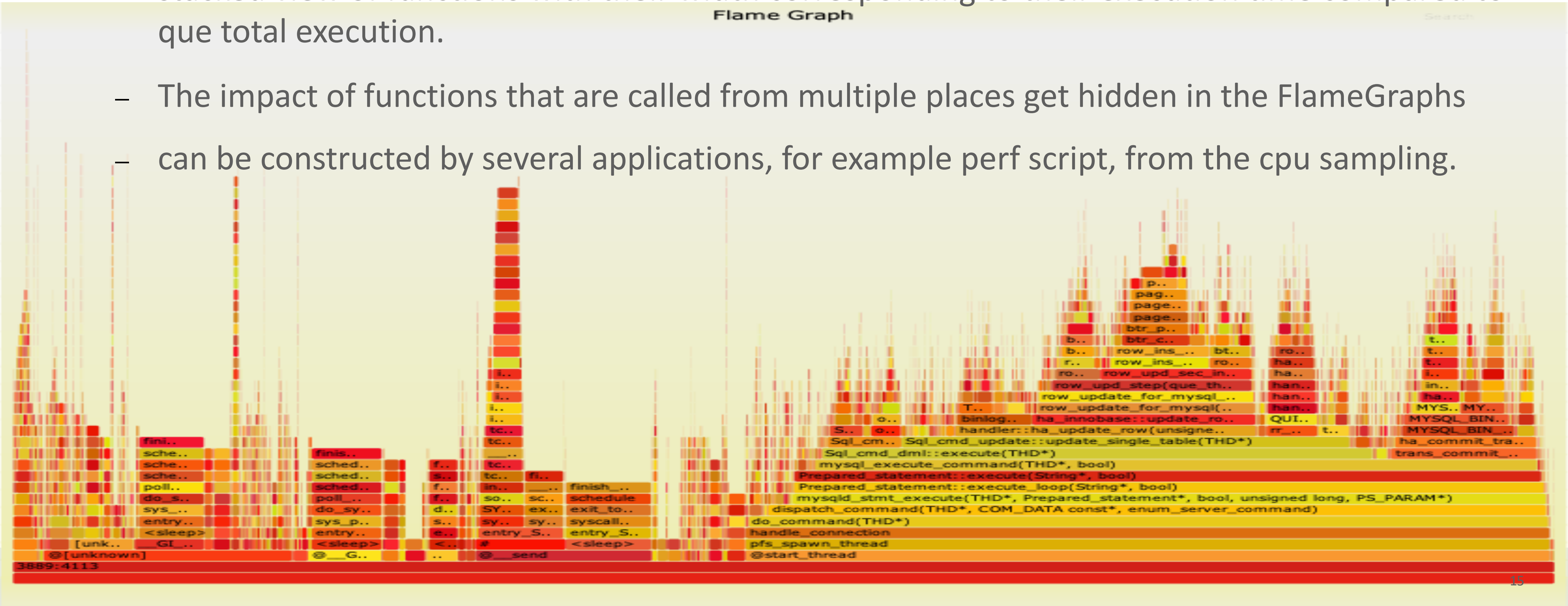
Connected graph with the the functions that call each other, arcs usually contain the usage of the target by the source.



Visualization

Flamegraph

- FlameGraphs have lately been used to allow a simpler first look at the profile. They allow a stacked view of functions with their width corresponding to their execution time compared to que total execution.
- The impact of functions that are called from multiple places get hidden in the FlameGraphs
- can be constructed by several applications, for example perf script, from the cpu sampling.



7

Profiling with eBPF

Profiling with eBPF

eBPF

- The Linux kernel has had several ways to allow introspection, with limited success
- The DTrace packages were partially migrated to Linux, but never had the capabilities they had on Solaris
- Recently the infrastructure developed for BPF was extended to support tracing at the kernel level
- Recent kernels (4.9+) support kernel level profiling, user-defined kernel (kprobes) and user level probes (probes), performance counters, etc.

Profiling with eBPF

Profiling with eBPF allows:

- call-stack collecting to be performed at the kernel level, avoiding expensive context switching to user level;
- It requires that the kernel has enough information to rebuild the call-stack, so -fno-omit-frame-pointers is essential;
- The interception of kernel functions that are relevant for any particular use;
- presently not all functions can be intercepted, trial and error is necessary;
- The dynamic instrumentation of user functions without changes in the code;
- Several ways to use with different levels of difficulty, bcc makes a lot of the work easier.

Profiling with eBPF

eBPF

- Using BPF:
- Several ways to use with different levels of difficulty, bcc makes a lot of the work easier
- Brendan Gregg has a lot of great work in this area, please check out his site: brendangregg.com
- BPF is expected be the basis for future profiles in Linux

Profiling with eBPF

eBPF

- Main limitations
- Call-stack tracking on the kernel is not correct sometimes
- Frame-pointers are required, which is a problem for libc and libstdc++
- LBR is in the kernel, but cannot be used to track it
- Event counting is not deterministic
- It makes extensive use of kernel level hashes to track counters, but it seems some events get lost and not counted
- Requires kernel 4.4 (4.9 for the in-kernel profiling)

Profiling with eBPF

eBPF

- The potential is large, but bcc has many quirks that need to be consider when developing profiling tools.
- My current utilities include a dynamic function tracer, a profiler with on/off cpu time and mutex usage
- Working on a storage access profiler and ways to present the scheduling in/out of threads with their causes

Profiling with eBPF

argdist

- argdist probes functions you specify and collects parameter values into a histogram or a frequency count. This can be used to understand the distribution of values a certain parameter takes, filter and print interesting parameters without attaching a debugger, and obtain general execution statistics on various functions.
- Examples:

```
argdist -p 1005 -C 'p:c:malloc(size_t size):size_t:size:size==16'
Print a frequency count of how many times process 1005 called malloc with an allocation size of 16 bytes

argdist -C 'r:c:gets():char*:(char*)$retval#snooped strings’
Snoop on all strings returned by gets()

argdist -p 1005 -C 'p:c:write(int fd):int:fd' -T 5
Print frequency counts of how many times writes were issued to a particular file descriptor number, in process
1005, but only show the top 5 busiest fds

argdist -p 1005 -H 'r:c:read()'
Print a histogram of results (sizes) returned by read() in process 1005

argdist -C 'r::__vfs_read():u32:$PID:$latency > 100000'
Print frequency of reads by process where the latency was >0.1ms

argdist -H 'r::__vfs_read(void *file, void *buf, size_t count):size_t:$entry(count):$latency > 1000000'
        Print a histogram of read sizes that were longer than 1ms
argdist -H \\
        'p:c:write(int fd, const void *buf, size_t count):size_t:count:fd==1'
```

Profiling with eBPF

biolatency

- Summarize block device I/O latency as a histogram.
- Examples
  - `./biolatency                    # summarize block I/O latency as a histogram`
  - `./biolatency 1 10               # print 1 second summaries, 10 times`
  - `./biolatency -mT 1              # 1s summaries, milliseconds, and timestamps`
  - `./biolatency -Q                 # include OS queued time in I/O time`
  - `./biolatency -D                 # show each disk device separately`
  - `./biolatency -F                 # show I/O flags separately`
  - `./biolatency -j                 # print a dictionary`

Profiling with eBPF

biolatency

```
./biolatency
Tracing block device I/O... Hit Ctrl-C to end.
^C
     usecs               : count     distribution
       0 -> 1          : 0        |                                      |
       2 -> 3          : 0        |                                      |
       4 -> 7          : 0        |                                      |
       8 -> 15         : 0        |                                      |
      16 -> 31         : 0        |                                      |
      32 -> 63         : 0        |                                      |
      64 -> 127        : 1        |                                      |
     128 -> 255        : 12       |********                              |
     256 -> 511        : 15       |**********                            |
     512 -> 1023       : 43       |*******************************       |
    1024 -> 2047       : 52       |**************************************|
    2048 -> 4095       : 47       |**********************************    |
    4096 -> 8191       : 52       |**************************************|
    8192 -> 16383      : 36       |**************************            |
   16384 -> 32767      : 15       |**********                            |
   32768 -> 65535      : 2        |*                                     |
   65536 -> 131071     : 2        |*                                     |
```

Profiling with eBPF

biosnoop

- traces block device I/O (disk I/O), and prints a line of output.
- Example:

```
./biosnoop
TIME(s)      COMM           PID     DISK     T SECTOR      BYTES    LAT(ms)
0.000004     supervise      1950    xvda1    W 13092560    4096        0.74
0.000178     supervise      1950    xvda1    W 13092432    4096        0.61
0.001469     supervise      1956    xvda1    W 13092440    4096        1.24
0.001588     supervise      1956    xvda1    W 13115128    4096        1.09
1.022346     supervise      1950    xvda1    W 13115272    4096        0.98
1.022568     supervise      1950    xvda1    W 13188496    4096        0.93
1.023534     supervise      1956    xvda1    W 13188520    4096        0.79
1.023585     supervise      1956    xvda1    W 13189512    4096        0.60
2.003920     xfsaild/md0    456     xvdc     W 62901512    8192        0.23
```

Profiling with eBPF

biolatpcts

- Monitor IO latency distribution of a block device.
- Examples

```
./biolatpcts.py /dev/nvme0n1
#  nvme0n1    p1    p5   p10   p16   p25   p50   p75   p84   p90   p95   p99  p100
#  read     95us 175us 305us 515us 895us 985us 995us 1.5ms 2.5ms 3.5ms 4.5ms  10ms
#  write     5us   5us   5us  15us  25us 135us 765us 855us 885us 895us 965us 1.5ms
#  discard   5us   5us   5us   5us 135us 145us 165us 205us 385us 875us 1.5ms 2.5ms
#  flush     5us   5us   5us   5us   5us   5us   5us   5us   5us 1.5ms 4.5ms 5.5ms
```

Profiling with eBPF

biostop

- block device (disk) I/O by process.
- Examples

```
./biotop
Tracing... Output every 1 secs. Hit Ctrl-C to end
08:04:11 loadavg: 1.48 0.87 0.45 1/287 14547
PID    COMM             D MAJ MIN DISK       I/O  Kbytes  AVGms
14501  cksum            R 202 1   xvda1      361   28832   3.39
6961   dd               R 202 1   xvda1     1628   13024   0.59
13855  dd               R 202 1   xvda1     1627   13016   0.59
326    jbd2/xvda1-8     W 202 1   xvda1        3     168   3.00
1880   supervise        W 202 1   xvda1        2       8   6.71
1873   supervise        W 202 1   xvda1        2       8   2.51
1871   supervise        W 202 1   xvda1        2       8   1.57
```

Profiling with eBPF

cachestat

- shows hits and misses to the file system page cache.
- Examples

```
cachestat

    HITS   MISSES  DIRTIES HITRATIO   BUFFERS_MB  CACHED_MB

    1132        0        4  100.00%          277       4367

     161        0       36  100.00%          277       4372

      16        0       28  100.00%          277       4372

   17154    13750       15   55.51%          277       4422

      19        0        1  100.00%          277       4422

      83        0       83  100.00%          277       4421

      16        0        1  100.00%          277       4423

^C       0      -19      360    0.00%          277       4423

Detaching...
```

Profiling with eBPF

cachestat

- shows hits and misses to the file system page cache.
- Examples

```
cachestat
    HITS   MISSES  DIRTIES HITRATIO   BUFFERS_MB  CACHED_MB
    1132        0        4  100.00%          277       4367
     161        0       36  100.00%          277       4372
      16        0       28  100.00%          277       4372
   17154    13750       15   55.51%          277       4422
      19        0        1  100.00%          277       4422
      83        0       83  100.00%          277       4421
      16        0        1  100.00%          277       4423
^C      0      -19      360    0.00%          277       4423
Detaching...
```

Profiling with eBPF

cachetop

- Linux page cache hit/miss statistics including read and write hit % per processes in a UI like top.
- Examples

```
./cachetop 5
13:01:01 Buffers MB: 76 / Cached MB: 114 / Sort: HITS / Order: ascending
PID      UID      CMD              HITS     MISSES   DIRTIES  READ_HIT%  WRITE_HIT%
       1 root     systemd                 2        0        0     100.0%       0.0%
     680 root     vminfo                  3        4        2      14.3%      42.9%
     567 syslog   rs:main Q:Reg          10        4        2      57.1%      21.4%
     986 root     kworker/u2:2           10     2457        4       0.2%      99.5%
     988 root     kworker/u2:2           10        9        4      31.6%      36.8%
     877 vagrant  systemd                18        4        2      72.7%      13.6%
     983 root     python                148        3      143       3.3%       1.3%
```

Profiling with eBPF

cpudist

- Summarize on- and off-CPU time per task as a histogram.
- Examples

```
cpudist              # summarize on-CPU time as a histogram
cpudist -O           # summarize off-CPU time as a histogram
cpudist 1 10         # print 1 second summaries, 10 times
cpudist -mT 1        # 1s summaries, milliseconds, and timestamps
cpudist -P           # show each PID separately
cpudist -p 185       # trace PID 185 only
```

Profiling with eBPF

cpudist

```
./cpudist.py -p $(pidof parprimes)
Tracing on-CPU time... Hit Ctrl-C to end.
^C
     usecs               : count     distribution
         0 -> 1          : 3        |                                        |
         2 -> 3          : 17       |                                        |
         4 -> 7          : 39       |                                        |
         8 -> 15         : 52       |*                                       |
        16 -> 31         : 43       |                                        |
        32 -> 63         : 12       |                                        |
        64 -> 127        : 13       |                                        |
       128 -> 255        : 0        |                                        |
       256 -> 511        : 1        |                                        |
       512 -> 1023       : 11       |                                        |
      1024 -> 2047       : 15       |                                        |
      2048 -> 4095       : 41       |                                        |
      4096 -> 8191       : 1134     |************************                |
      8192 -> 16383      : 1883     |****************************************|
     16384 -> 32767      : 65       |*                                       |
```

Profiling with eBPF

dbslower

- dbslower traces queries served by a MySQL or PostgreSQL server, and prints those that exceed a latency (query time) threshold.
- Examples

```
dbslower mysql
Tracing database queries for pids 25776 slower than 1 ms...
TIME(s)        PID          MS QUERY
1.315800       25776  2000.999 call getproduct(97)
3.360380       25776     3.226 call getproduct(6)
^C
```

Profiling with eBPF

dcsnoop

- Trace directory entry cache (dcache) lookups.
- Examples

```
./dcsnoop           # trace failed dcache lookups
./dcsnoop -a        # trace all dcache lookups
```

Profiling with eBPF

deadlock

- Detects potential deadlocks (lock order inversions) on a running process..
- Examples

```
./deadlock.py 181
Tracing... Hit Ctrl-C to end.
----------------
Potential Deadlock Detected!
Cycle in lock order graph: Mutex M0 (main::static_mutex3 0x0000000000473c60) => Mutex M1 (0x00007fff6d738400)
=> Mutex M2 (global_mutex1 0x0000000000473be0) => Mutex M3 (global_mutex2 0x0000000000473c20) => Mutex M0
(main::static_mutex3 0x0000000000473c60)
Mutex M1 (0x00007fff6d738400) acquired here while holding Mutex M0 (main::static_mutex3 0x0000000000473c60) in
Thread 357250 (lockinversion):
@ 00000000004024d0 pthread_mutex_lock
@ 0000000000406dd0 std::mutex::lock()
@ 00000000004070d2 std::lock_guard<std::mutex>::lock_guard(std::mutex&)
@ 0000000000402e38 main::{lambda()#3}::operator()() const
@ 0000000000406ba8 void std::_Bind_simple<main::{lambda()#3} ()>::_M_invoke<>(std::_Index_tuple<>)
@ 0000000000406951 std::_Bind_simple<main::{lambda()#3} ()>::operator()()
@ 000000000040673a std::thread::_Impl<std::_Bind_simple<main::{lambda()#3} ()> >::_M_run()
@ 00007fd4496564e1 execute_native_thread_routine
@ 00007fd449dd57f1 start_thread
@ 00007fd44909746d __clone

Mutex M0 (main::static_mutex3 0x0000000000473c60) previously acquired by the same Thread 357250
(lockinversion) here:
@ 00000000004024d0 pthread_mutex_lock
```

Profiling with eBPF

dirtop

- shows reads and writes by directory.
- Examples

```
./dirtop -d '/hdfs/uuid/*/yarn' -Cr 5
Tracing... Output every 1 secs. Hit Ctrl-C to end

14:29:08 loadavg: 25.66 23.42 21.51 17/2850 67167

READS  WRITES R_Kb     W_Kb     PATH
100    8429   0        48243    /hdfs/uuid/b94cbf3f-76b1-4ced-9043-02d450b9887c/yarn
2066   4091   8176     26457    /hdfs/uuid/d04fccd8-bc72-4ed9-bda4-c5b6893f1405/yarn
10     2043   0        8172     /hdfs/uuid/b3b2a2ed-f6c1-4641-86bf-2989dd932411/yarn
38     1368   0        2652     /hdfs/uuid/a78f846a-58c4-4d10-a9f5-42f16a6134a0/yarn
86     19     0        123      /hdfs/uuid/c11da291-28de-4a77-873e-44bb452d238b/yarn
```

Profiling with eBPF

ext4dist

- traces ext4 reads, writes, opens, and fsyncs, and summarizes their latency as a power-of-2 histogram.
- Examples

```
./ext4dist
Tracing ext4 operation latency... Hit Ctrl-C to end.
^C
operation = 'read'
     usecs               : count     distribution
         0 -> 1          : 1210     |****************************************|
         2 -> 3          : 126      |****                                    |
         4 -> 7          : 376      |************                            |
         8 -> 15         : 86       |**                                      |
        16 -> 31         : 9        |                                        |
        32 -> 63         : 47       |*                                       |
        64 -> 127        : 6        |                                        |
       128 -> 255        : 24       |                                        |
       256 -> 511        : 137      |****                                    |
       512 -> 1023       : 66       |**                                      |
      1024 -> 2047       : 13       |                                        |
      2048 -> 4095       : 7        |                                        |
      4096 -> 8191       : 13       |                                        |
      8192 -> 16383      : 3        |                                        |
```

Profiling with eBPF

fileslower

- shows file-based synchronous reads and writes slower than a threshold.
- Examples

```
echo 3 > /proc/sys/vm/drop_caches; ./fileslower 1
Tracing sync read/writes slower than 1 ms
TIME(s)  COMM           PID    D BYTES   LAT(ms) FILENAME
0.000    bash           9647   R 128        5.83 man
0.050    man            9647   R 832       19.52 libmandb-2.6.7.1.so
0.066    man            9647   R 832       15.79 libman-2.6.7.1.so
0.123    man            9647   R 832       56.36 libpipeline.so.1.3.0
0.135    man            9647   R 832        9.79 libgdbm.so.3.0.0
0.323    man            9647   R 4096      59.52 locale.alias
0.540    man            9648   R 8192      11.11 ls.1.gz
0.558    man            9647   R 72         6.97 index.db
0.563    man            9647   R 4096       5.12 index.db
0.723    man            9658   R 128       12.06 less
```

Profiling with eBPF

fileslower

- shows file-based synchronous reads and writes slower than a threshold.
- Examples

```
echo 3 > /proc/sys/vm/drop_caches; ./fileslower 1
Tracing sync read/writes slower than 1 ms
TIME(s)  COMM           PID    D BYTES   LAT(ms) FILENAME
0.000    bash           9647   R 128        5.83 man
0.050    man            9647   R 832       19.52 libmandb-2.6.7.1.so
0.066    man            9647   R 832       15.79 libman-2.6.7.1.so
0.123    man            9647   R 832       56.36 libpipeline.so.1.3.0
0.135    man            9647   R 832        9.79 libgdbm.so.3.0.0
0.323    man            9647   R 4096      59.52 locale.alias
0.540    man            9648   R 8192      11.11 ls.1.gz
0.558    man            9647   R 72         6.97 index.db
0.563    man            9647   R 4096       5.12 index.db
0.723    man            9658   R 128       12.06 less
```

Profiling with eBPF

filetop

- shows reads and writes by file, with process details.
- Examples

```
./filetop -Cr 5
Tracing... Output every 1 secs. Hit Ctrl-C to end

08:05:11 loadavg: 0.75 0.35 0.25 3/285 822

PID    COMM             READS  WRITES R_Kb    W_Kb    T FILE
32672  cksum            5006   0      320384  0       R data1
809    run              2      0      8       0       R nsswitch.conf
811    run              2      0      8       0       R nsswitch.conf
804    chown            2      0      8       0       R nsswitch.conf
```

Profiling with eBPF

funccount

- Count functions, tracepoints, and USDT probes.
- Examples

```
./funccount u:pthread:*mutex* -p 1442
Tracing 7 functions for "u:pthread:*mutex*"... Hit Ctrl-C to end.
^C
FUNC                                          COUNT
mutex_init                                        1
mutex_entry                                  547122
mutex_acquired                               547175
mutex_release                                547185
Detaching...

# ./funccount -i 1 'vfs_*'
Tracing... Ctrl-C to end.

FUNC                          COUNT
vfs_fstatat                       1
vfs_fstat                        16
vfs_getattr_nosec                17
vfs_getattr                      17
vfs_write                        52
vfs_read                         79
vfs_open                         98
```

Profiling with eBPF

funclatency

- Time functions and print latency as a histogram.
- Examples:

```
./funclatency do_sys_open       # time the do_sys_open() kernel function
./funclatency c:read            # time the read() C library function
./funclatency -u vfs_read       # time vfs_read(), in microseconds
./funclatency -m do_nanosleep   # time do_nanosleep(), in milliseconds
./funclatency -i 2 -d 10 c:open # output every 2 seconds, for duration 10s
./funclatency -mTi 5 vfs_read   # output every 5 seconds, with timestamps
./funclatency -p 181 vfs_read   # time process 181 only
./funclatency 'vfs_fstat*'      # time both vfs_fstat() and vfs_fstatat()
./funclatency 'c:*printf'       # time the *printf family of functions
./funclatency -F 'vfs_r*'       # show one histogram per matched function
```

Profiling with eBPF

llcstat

- traces cache reference and cache miss events system-wide, and summarizes them by PID and CPU.
- Examples:

```
./llcstat.py 20 -c 5000
Running for 20 seconds or hit Ctrl-C to end.
PID      NAME             CPU     REFERENCE         MISS   HIT%
0        swapper/15       15        3515000       640000  81.79%
238      migration/38     38           5000            0 100.00%
4512     ntpd             11           5000            0 100.00%
150867   ipmitool         3           25000         5000  80.00%
150895   lscpu            17         280000        25000  91.07%
151807   ipmitool         15          15000         5000  66.67%
150757   awk              2           15000         5000  66.67%
151213   chef-client      5         1770000       240000  86.44%
151822   scribe-dispatch  12          15000            0 100.00%
123386   mysqld           5            5000            0 100.00%
[...]
Total References: 518920000 Total Misses: 90265000 Hit Rate: 82.61%
```

Profiling with eBPF

memleak

- traces and matches memory allocation and deallocation requests, and collects call stacks for each allocation.
- Examples:

```
./memleak -p $(pidof allocs)
Attaching to pid 5193, Ctrl+C to quit.
[11:16:33] Top 2 stacks with outstanding allocations:
        80 bytes in 5 allocations from stack
                 main+0x6d [allocs]
                 __libc_start_main+0xf0 [libc-2.21.so]

[11:16:34] Top 2 stacks with outstanding allocations:
        160 bytes in 10 allocations from stack
                 main+0x6d [allocs]
                 __libc_start_main+0xf0 [libc-2.21.so]
```

Profiling with eBPF

netqtop

- traces the kernel functions performing packet transmit (xmit_one) and packet receive (__netif_receive_skb_core) on data link layer.
- Examples:

```
./netqtop.py -n lo -i 1
Thu Sep 10 11:28:39 2020
TX
 QueueID    avg_size   [0, 64)   [64, 512)  [512, 2K)  [2K, 16K)  [16K, 64K)
    0          88          0          9          0          0          0
 Total         88          0          9          0          0          0

RX
 QueueID    avg_size   [0, 64)   [64, 512)  [512, 2K)  [2K, 16K)  [16K, 64K)
    0          74          4          5          0          0          0
 Total         74          4          5          0          0          0
----------------------------------------------------------------------------
```

Profiling with eBPF

offcputime

- This program shows stack traces that were blocked, and the total duration they were blocked.
- Examples:

```
./offcputime -K
Tracing off-CPU time (us) of all threads by kernel stack... Hit Ctrl-C to end.
^C
    schedule
    schedule_timeout
    io_schedule_timeout
    bit_wait_io
    __wait_on_bit
    wait_on_page_bit_killable
    __lock_page_or_retry
    filemap_fault
    __do_fault
    handle_mm_fault
    __do_page_fault
    do_page_fault
    page_fault
    chmod
        13
```

Profiling with eBPF

offwaketime

- shows kernel stack traces and task names that were blocked and "off-CPU", along with the stack traces and task names for the threads that woke them, and the total elapsed time from when they blocked to when they were woken up.
- Examples:

```
./offwaketime 5
Tracing blocked time (us) by kernel off-CPU and waker stack for 5 secs.

[...]

    waker:           swapper/0
    ffffffff8137897c blk_mq_complete_request
    ffffffff81378930 __blk_mq_complete_request
    …
    target:          cksum
        56529
```

Profiling with eBPF

offwaketime

- shows kernel stack traces and task names that were blocked and "off-CPU", along with the stack traces and task names for the threads that woke them, and the total elapsed time from when they blocked to when they were woken up.
- Examples:

```
./offwaketime 5
Tracing blocked time (us) by kernel off-CPU and waker stack for 5 secs.

[...]

    waker:           swapper/0
    ffffffff8137897c blk_mq_complete_request
    ffffffff81378930 __blk_mq_complete_request
    …
    target:          cksum
        56529
```

Profiling with eBPF

profile

- CPU profiler, takes samples of stack traces at timed intervals, and frequency counting them in kernel context for efficiency.
- Examples:

```
./profile
Sampling at 49 Hertz of all threads by user + kernel stack... Hit Ctrl-C to end.
^C
    filemap_map_pages
    handle_mm_fault
    __do_page_fault
    do_page_fault
    page_fault
    [unknown]
    -                cp (9036)
        1

    [unknown]
    [unknown]
    -                sign-file (8877)
        1
```

Profiling with eBPF

readahead

- shows the performance of the read-ahead caching on the system under a given load to investigate any caching issues.
- Examples:

```
readahead -d 30
Tracing... Hit Ctrl-C to end.
^C
Read-ahead unused pages: 6765
Histogram of read-ahead used page age (ms):

     age (ms)            : count     distribution
         0 -> 1          : 4236     |****************************************|
         2 -> 3          : 394      |***                                     |
         4 -> 7          : 1670     |***************                         |
         8 -> 15         : 2132     |********************                    |
        16 -> 31         : 401      |***                                     |
        32 -> 63         : 1256     |***********                             |
        64 -> 127        : 2352     |**********************                  |
       128 -> 255        : 357      |***                                     |
       256 -> 511        : 369      |***                                     |
       512 -> 1023       : 366      |***                                     |
      1024 -> 2047       : 181      |*                                       |
```

Profiling with eBPF

readahead

- shows the performance of the read-ahead caching on the system under a given load to investigate any caching issues.
- Examples:

```
readahead -d 30
Tracing... Hit Ctrl-C to end.
^C
Read-ahead unused pages: 6765
Histogram of read-ahead used page age (ms):

     age (ms)               : count     distribution
         0 -> 1             : 4236     |****************************************|
         2 -> 3             : 394      |***                                     |
         4 -> 7             : 1670     |***************                         |
         8 -> 15            : 2132     |********************                    |
        16 -> 31            : 401      |***                                     |
        32 -> 63            : 1256     |***********                             |
        64 -> 127           : 2352     |**********************                  |
       128 -> 255           : 357      |***                                     |
       256 -> 511           : 369      |***                                     |
       512 -> 1023          : 366      |***                                     |
      1024 -> 2047          : 181      |*                                       |
```

Profiling with eBPF

runqlat

- Run queue (scheduler) latency as a histogram.
- Examples:

```
./runqlat
Tracing run queue latency... Hit Ctrl-C to end.
^C
     usecs               : count     distribution
         0 -> 1          : 233      |***********                             |
         2 -> 3          : 742      |************************************    |
         4 -> 7          : 203      |**********                              |
         8 -> 15         : 173      |********                                |
        16 -> 31         : 24       |*                                       |
        32 -> 63         : 0        |                                        |
        64 -> 127        : 30       |*                                       |
       128 -> 255        : 6        |                                        |
       256 -> 511        : 3        |                                        |
       512 -> 1023       : 5        |                                        |
      1024 -> 2047       : 27       |*                                       |
      2048 -> 4095       : 30       |*                                       |
      4096 -> 8191       : 20       |                                        |
      8192 -> 16383      : 29       |*                                       |
     16384 -> 32767      : 809      |****************************************|
     32768 -> 65535      : 64       |***                                     |
```

Profiling with eBPF

syncsnoop

- traces calls to the kernel sync() routine, with basic timestamps.
- Examples:

```
./syncsnoop
TIME(s)             CALL
16458148.611952     sync()
16458151.533709     sync()
^C
```

Profiling with eBPF

syscount

- summarizes syscall counts across the system or a specific process, with optional latency information.
- Examples:

```
syscount
Tracing syscalls, printing top 10... Ctrl+C to quit.
[09:39:04]
SYSCALL                   COUNT
write                     10739
read                      10584
wait4                      1460
nanosleep                  1457
select                      795
rt_sigprocmask              689
clock_gettime               653
rt_sigaction                128
futex                        86
ioctl                        83
^C
```

Profiling with eBPF

trace

- probes functions you specify and displays trace messages if a particular condition is met.
- Examples:

```
trace.py -U -a 'r::sys_futex "%d", retval'
PID     TID     COMM            FUNC             -
793922  793951  poller          sys_futex        0
        7f6c72b6497a __lll_unlock_wake+0x1a [libpthread-2.23.so]
              627fef folly::FunctionScheduler::run()+0x46f [router]
        7f6c7345f171 execute_native_thread_routine+0x21 [libstdc++.so.6.0.21]
        7f6c72b5b7a9 start_thread+0xd9 [libpthread-2.23.so]
        7f

trace 'r:c:read ((int)retval < 0) "read failed: %d", retval' \
        'r:c:write ((int)retval < 0) "write failed: %d", retval' -T
TIME     PID    COMM         FUNC             -
05:31:57 3388   bash         write            write failed: -1
05:32:00 3388   bash         write            write failed: -1
```

Profiling with eBPF

vfscount

- counts VFS calls during time, by tracing all kernel functions beginning with "vfs_".
- Examples:

```
./vfscount
Tracing... Ctrl-C to end.
^C
ADDR             FUNC                          COUNT
ffffffff811f3c01 vfs_create                        1
ffffffff8120be71 vfs_getxattr                      2
ffffffff811f5f61 vfs_unlink                        2
ffffffff81236ca1 vfs_lock_file                     6
ffffffff81218fb1 vfs_fsync_range                   6
ffffffff811ecaf1 vfs_fstat                       319
ffffffff811e6f01 vfs_open                        475
ffffffff811ecb51 vfs_fstatat                     488
ffffffff811ecac1 vfs_getattr                     704
ffffffff811ec9f1 vfs_getattr_nosec               704
ffffffff811e80a1 vfs_write                      1764
ffffffff811e7f71 vfs_read                       2283
```

Profiling with eBPF

vfsstat

- traces some common VFS calls and prints per-second summaries.
- Examples:

```
./vfsstat
TIME          READ/s  WRITE/s CREATE/s   OPEN/s  FSYNC/s
18:35:32:        231       12        4       98        0
18:35:33:        274       13        4      106        0
18:35:34:        586       86        4      251        0
18:35:35:        241       15        4       99        0
18:35:36:        232       10        4       98        0
18:35:37:        244       10        4      107        0
18:35:38:        235       13        4       97        0
18:35:39:       6749     2633        4     1446        0
18:35:40:        277       31        4      115        0
18:35:41:        238       16        6      102        0
18:35:42:        284       50        8      114        0
^C
```

Profiling with eBPF

wakeuptime

- measures when threads block, and shows the stack traces for the threads that performed the wakeup, along with the process names of the waker and target processes, and the total blocked time.
- Examples:

```
./wakeuptime
Tracing blocked time (us) by kernel stack... Hit Ctrl-C to end.
^C
[...truncated...]

    target:          vmstat
    ffffffff810df082 hrtimer_wakeup
    …
    ffffffff81473e83 __xen_evtchn_do_upcall
    ffffffff81475cf0 xen_evtchn_do_upcall
    ffffffff8178adee xen_do_hypervisor_callback
    waker:           swapper/1
        4000415
```