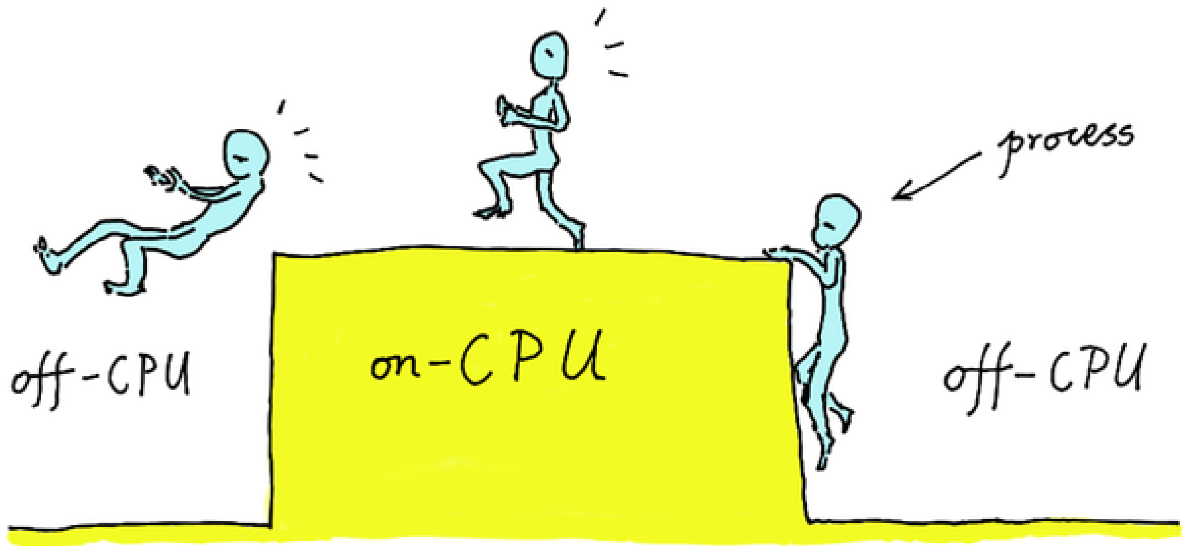
Introduction to off-CPU Time *Flame Graphs*

☺ *agentzh@cloudflare.com* ☺

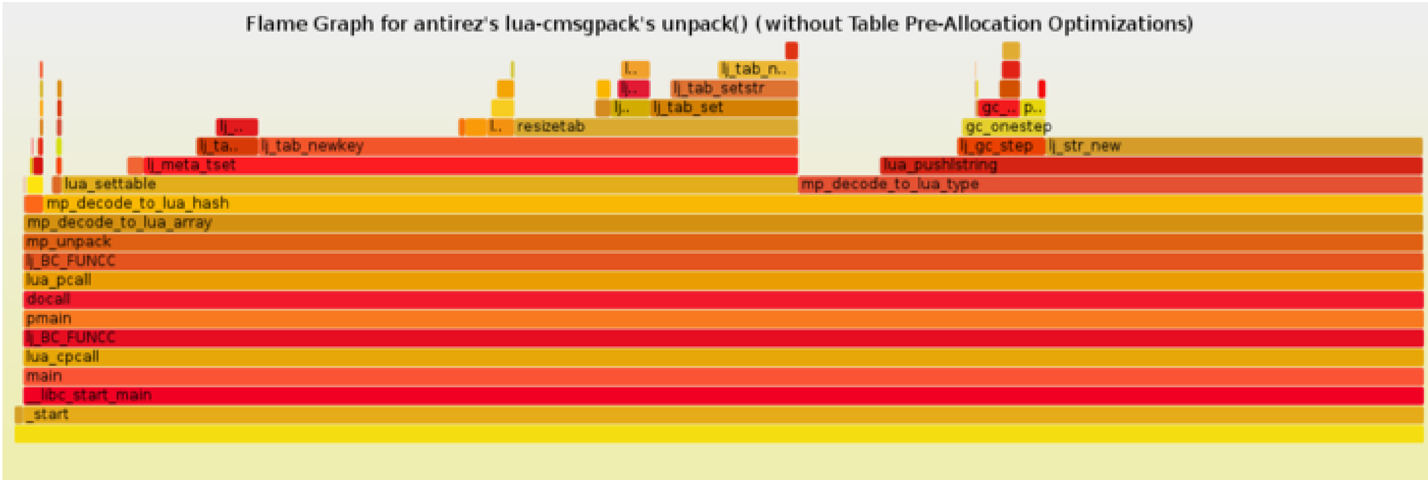
Yichun Zhang (agentzh)

2013.08.22

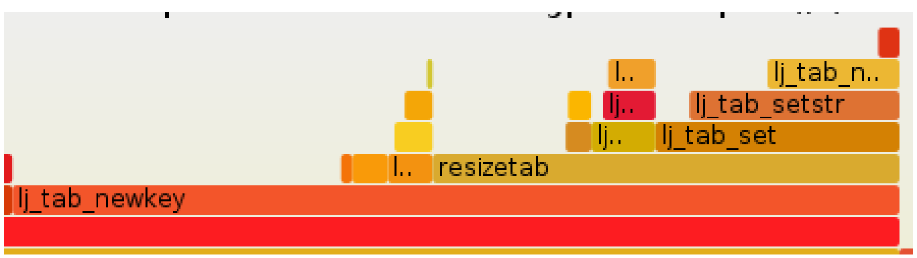
♡ Classic Flame Graphs are *on*-CPU time Flame Graphs per se.


process
off-CPU
on-CPU
off-CPU

♡ We are already relying on them to *optimize* our Lua WAF & Lua CDN Brain (cfcheck)

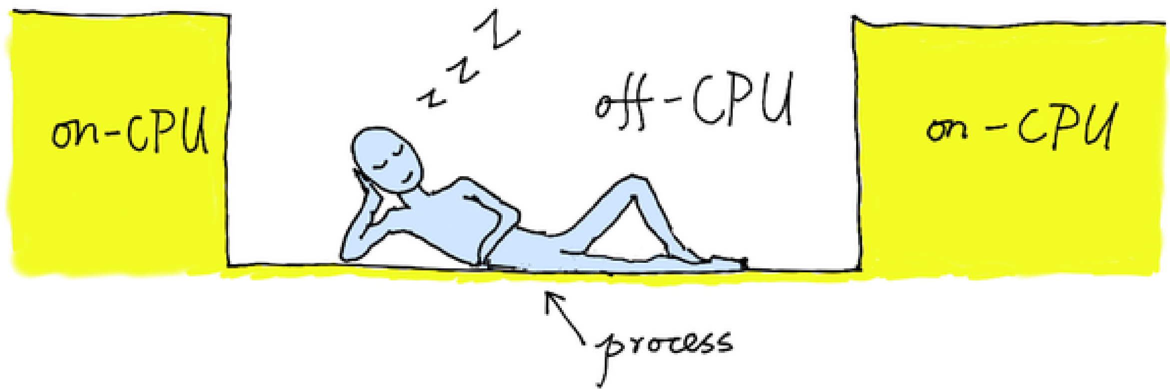


Flame Graph for antirez's lua-cmsgpack's unpack() (without Table Pre-Allocation Optimizations)
lj_tab_n..
lj_tab_setstr
lj_tab_set
resizetab
gc_onestep
lj_ta..
lj_tab_newkey
lj_gc_step
lj_str_new
lj_meta_tset
lua_pushlstring
lua_settable
mp_decode_to_lua_type
mp_decode_to_lua_hash
mp_decode_to_lua_array
mp_unpack
lj_BC_FUNCC
lua_pcall
docall
pmain
lj_BC_FUNCC
lua_cpcall
main
__libc_start_main
_start



l..
lj_tab_n..
lj..
lj_tab_setstr
lj..
lj_tab_set
l..
resizetab
lj_tab_newkey

♡ I invented *off*-CPU time Flame Graphs somewhere near Lake Tahoe 3 months ago.


on-CPU
zzZ
off-CPU
on-CPU
process

♡ I got the *inspiration*
from Brendan Gregg's blog post
"Off-CPU Performance Analysis"

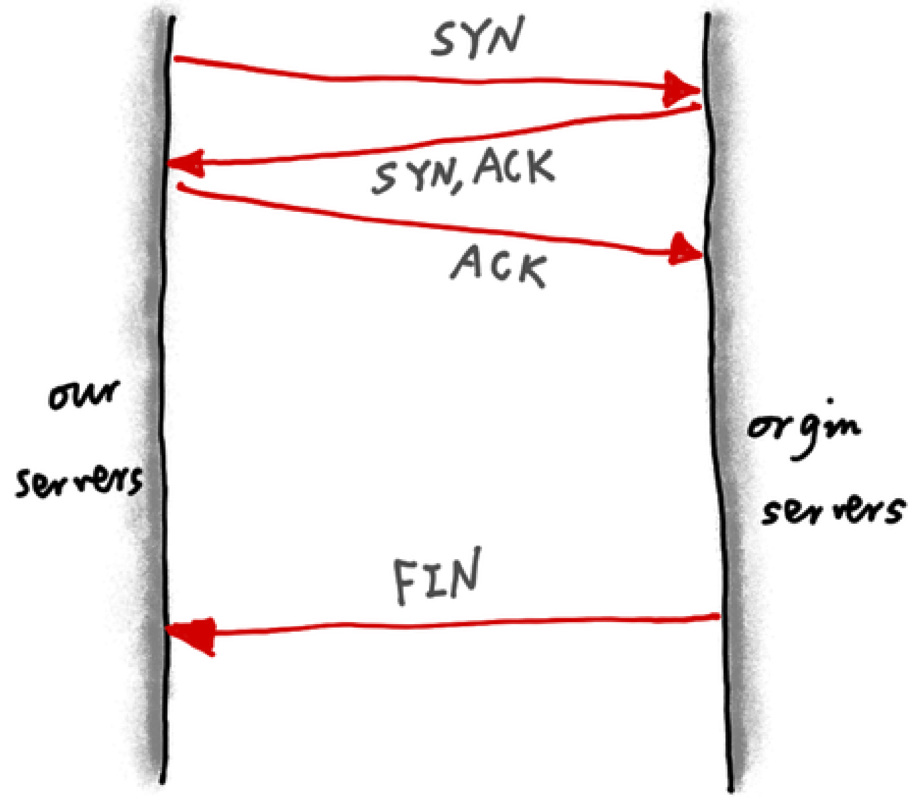
http://dtrace.org/blogs/brendan/2011/07/08/off-cpu-performance-analysis/

♡ Joshua Dankbaar *grabbed* me for an online issue right after the company Kitchen Adventure.

♡ Time to cast a spell over our Linux boxes by *systemtap*!

♡ I quickly wrote a *macro-style* language extension named stap++ for systemtap with a little bit of Perl.

https://github.com/agentzh/stapxx



SYN
SYN, ACK
ACK
our servers
orgin servers
FIN

♡ Nginx workers were badly *blocking* by something in a production box in Ashburn

```
   /* pseudo-
code for the nginx event loop */
   for (;;) {
       ret = epoll_wait(...);
       /* process new events
          and expired timers here... */
   }
```

♡ Let's write a simple tool to trace the long *blocking* latencies in the Nginx *event loop*!

```
$ vim epoll-looop-blocking.sxx
```

```
#!/usr/bin/env stap++
global begin
probe syscall.epoll_wait.return {
    if (target() == pid()) { begin = gettimeofday_ms() }
}
probe syscall.epoll_wait {
    if (target() == pid() && begin > 0) {
        elapsed = gettimeofday_ms() - begin
        if (elapsed >= $^arg_limit :default(200)) {
            printf("[%d] epoll loop blocked for %dms\n",
                   gettimeofday_s(), elapsed)
        }
    }
}
```

```
$ ./epoll-loop-blocking.sxx -x 22845 --arg limit=200
Start tracing 22845...
[1376595038] epoll loop blocked for 208ms
[1376595040] epoll loop blocked for 485ms
[1376595044] epoll loop blocked for 336ms
[1376595049] epoll loop blocked for 734ms
[1376595057] epoll loop blocked for 379ms
[1376595061] epoll loop blocked for 227ms
[1376595062] epoll loop blocked for 212ms
[1376595066] epoll loop blocked for 390ms
```

♡ Is it *file IO* blocking here?

```
# add some code to trace file IO latency at the same time...
global vfs_begin
global vfs_latency
probe syscall.rename, syscall.open, syscall.sendfile*,
      vfs.read, vfs.write
{
    if (target() == pid()) { vfs_begin = gettimeofday_us() }
}
probe syscall.rename.return, syscall.open.return,
      syscall.sendfile*.return, vfs.read.return, vfs.write.return
{
    if (target() == pid()) {
        vfs_latency += gettimeofday_us() - vfs_begin
    }
}
```

```
$ ./epoll-loop-blocking-vfs.sxx -x 22845 --arg limit=200
Start tracing 22845...
[1376596251] epoll loop blocked for 364ms (file IO: 19ms)
[1376596266] epoll loop blocked for 288ms (file IO: 0ms)
[1376596270] epoll loop blocked for 1002ms (file IO: 0ms)
[1376596272] epoll loop blocked for 206ms (file IO: 5ms)
[1376596280] epoll loop blocked for 218ms (file IO: 211ms)
[1376596283] epoll loop blocked for 396ms (file IO: 9ms)
```

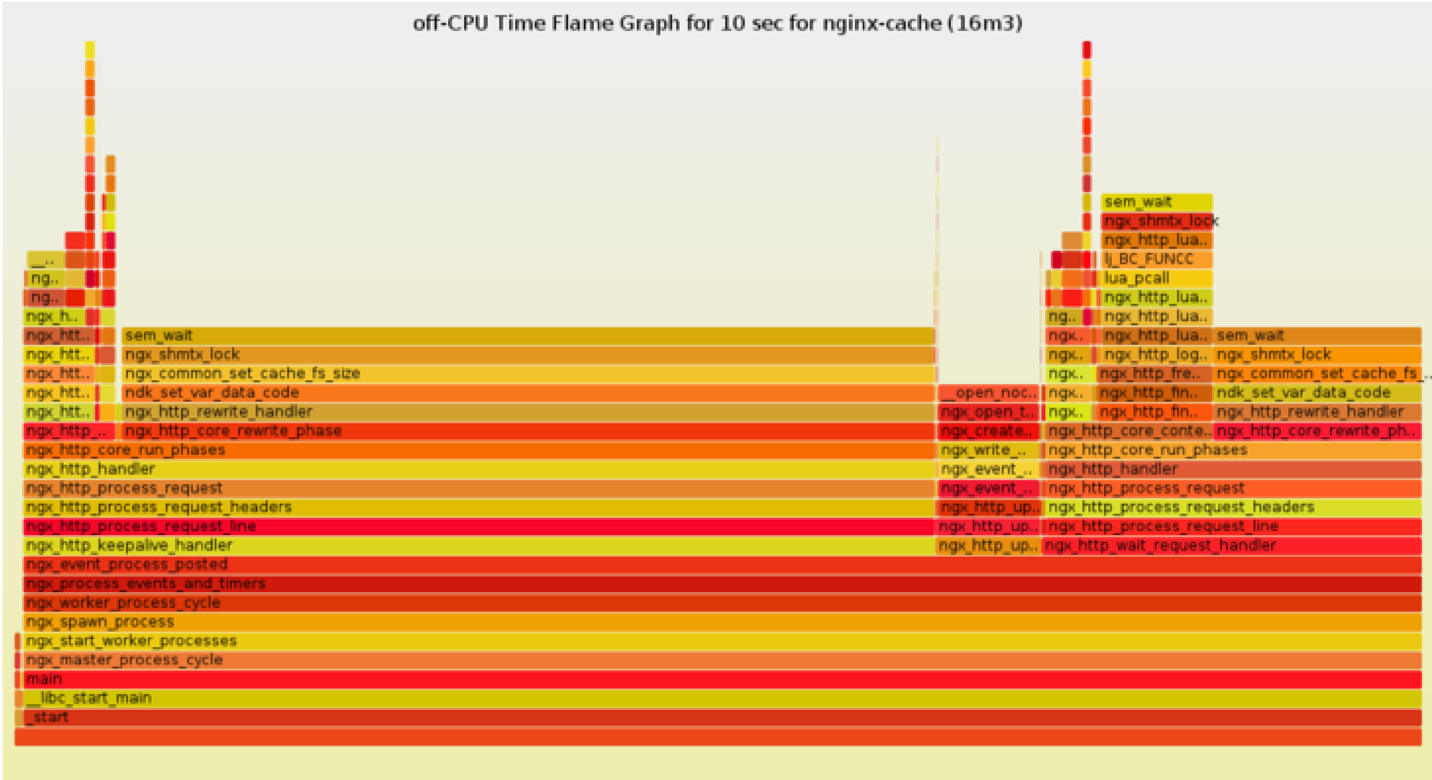
- Hmm...seems like file IO is *not* the major factor here...

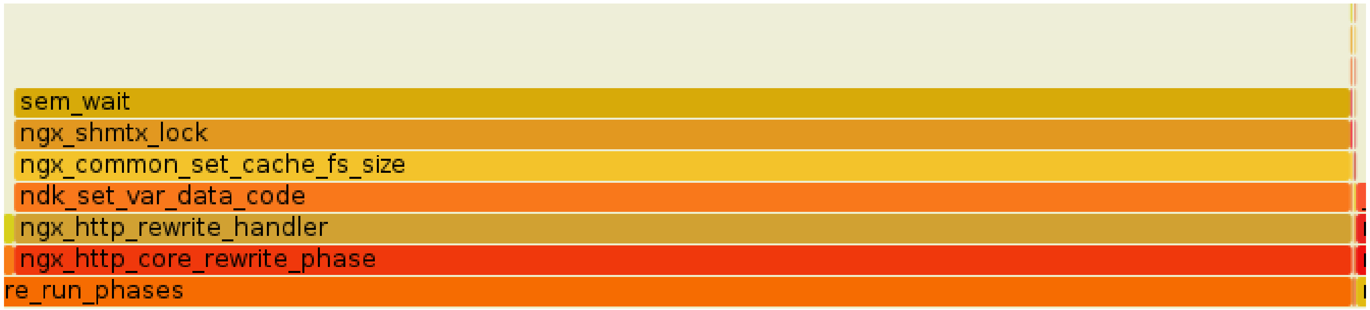
♡ I suddenly *remember* my off-CPU time Flame Graph tool created 3 months ago...

https://github.com/agentzh/nginx-systemtap-toolkit#ngx-sample-bt-off-cpu

```
$ ./ngx-sample-bt-off-cpu -t 10 -x 16782 > a.bt

$ stackcollapse-stap.pl a.bt > a.cbt
$ flamegraph.pl a.cbt > a.svg
```


off-CPU Time Flame Graph for 10 sec for nginx-cache (16m3)
sem_wait
ngx_shmtx_lock
ngx_common_set_cache_fs_size
ndk_set_var_data_code
ngx_http_rewrite_handler
ngx_http_core_rewrite_phase
ngx_http_core_run_phases
ngx_http_handler
ngx_http_process_request
ngx_http_process_request_headers
ngx_http_process_request_line
ngx_http_keepalive_handler
ngx_event_process_posted
ngx_process_events_and_timers
ngx_worker_process_cycle
ngx_spawn_process
ngx_start_worker_processes
ngx_master_process_cycle
main
__libc_start_main
_start
ngx_http_wait_request_handler
ngx_http_core_conte..
lua_pcall
lj_BC_FUNCC
ngx_http_lua..
ngx_shmtx_lock
sem_wait

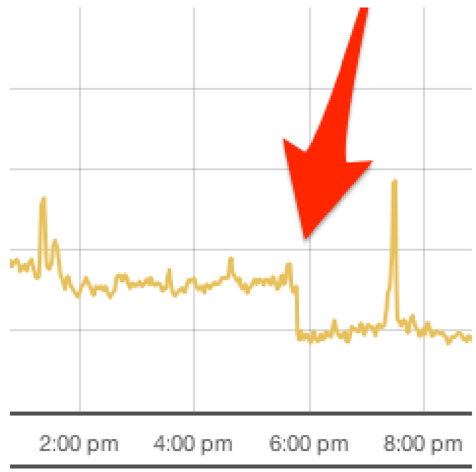


sem_wait
ngx_shmtx_lock
ngx_common_set_cache_fs_size
ndk_set_var_data_code
ngx_http_rewrite_handler
ngx_http_core_rewrite_phase
re_run_phases

♡ Okay, Nginx was mainly waiting on a *lock*
in an obsolete code path which was added to Nginx
by one of us (long time ago?)

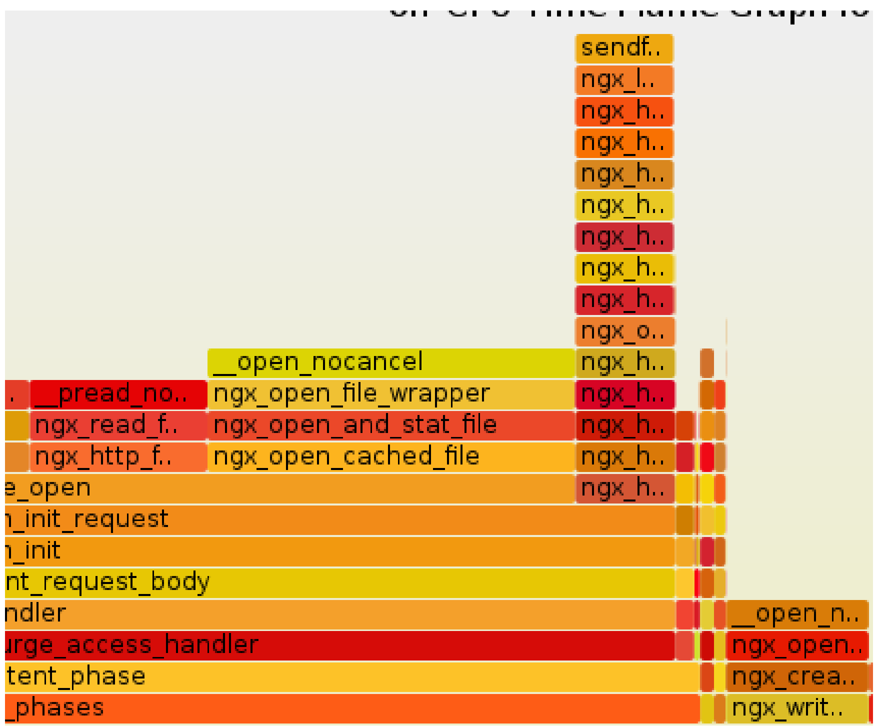
♡ Let's just *remove* the guilty code path from our production system!

♡ Yay! The number of long-running requests (longer than 1 second) is almost halved!



```
$ ./epoll-loop-blocking-vfs.sxx -x 16738 --arg limit=200
Start tracing 16738...
[1376626387] epoll loop blocked for 456ms (file IO: 455ms)
[1376626388] epoll loop blocked for 207ms (file IO: 206ms)
[1376626396] epoll loop blocked for 364ms (file IO: 363ms)
[1376626402] epoll loop blocked for 350ms (file IO: 349ms)
[1376626414] epoll loop blocked for 309ms (file IO: 309ms)
```


off-CPU Time Flame Graph for 10 sec for nginx-cache (16m3) - V2
sendf..
sen..
__open_nocancel
sem_..
__pread_no..
ngx_open_file_wrapper
ngx_read_f..
ngx_open_and_stat_file
ngx_http_file_ca..
ngx_http_f..
ngx_open_cached_file
ngx_http_file_cache_open
ngx_http_upstream_init_request
ngx_http_upstream_init
ngx_http_read_client_request_body
ngx_http_proxy_handler
ngx_http_cache_purge_access_handler
ngx_http_core_content_phase
ngx_http_core_run_phases
ngx_http_handler
ngx_http_process_request
ngx_http_process_request_headers
ngx_http_process_request_line
ngx_http_keepalive_handler
__open_n..
ngx_open..
ngx_crea..
ngx_writ..
ngx_even..
ngx_http_ups..
ngx_http_upstream..
ngx_http_upstream_hand..
sem_wait
__pread..
ngx_shmt..
ngx_rea..
ngx_open_and_stat_f..
ngx_http_file_cache..
ngx_htt..
rename
ngx_ext_ren..
ngx_http_fi..
ngx_http_up..
ngx_http_upstr..
ngx_event_process_posted
ngx_epoll_process_events
ngx_process_events_and_timers
ngx_worker_process_cycle
ngx_spawn_process
ngx_start_worker_processes
ngx_master_process_cycle
main
__libc_start_main
_start



sendf..
ngx_l..
ngx_h..
ngx_o..
__open_nocancel
__pread_no..
ngx_open_file_wrapper
ngx_read_f..
ngx_open_and_stat_file
ngx_http_f..
ngx_open_cached_file
e_open
_init_request
_init
nt_request_body
ndler
__open_n..
urge_access_handler
ngx_open..
tent_phase
ngx_crea..
_phases
ngx_writ..

♡ Okay, now it is *file IO* that's killing us!

♡ Let's tune Nginx's *open_file_cache* configurations to save the open() system calls.

- ♡ But...wait...we have *not* even enabled it yet in production...

```
# 2520 is the nginx worker process's pid
$ stap++ -x 2520 \
    -e 'probe @pfunc(ngx_open_cached_file)
{printf("%p\n",$cache);exit()}'
0x0
```

- ♡ It is *faster* and more *accurate* than asking Dane to check nginx.conf.

♡ Let's start by using the *sample* configuration in Nginx's official documentation.

```
# file nginx.conf
open_file_cache  max=1000 inactive=20s;
```

♡ Yay! Our online metrics immediately showed even *better* numbers!

♡ What is the cache *hit rate* then?

Can we improve the cache configurations even further?

```
#!/usr/bin/env stap++
global misses, total, in_ctx
probe @pfunc(ngx_open_cached_file) {
    if (pid() == target()) { in_ctx = 1 total++ }
}
probe @pfunc(ngx_open_cached_file).return {
    if (pid() == target()) { in_ctx = 0 }
}
probe @pfunc(ngx_open_and_stat_file) {
    if (pid() == target() && in_ctx) { misses++ }
}
probe end {
    printf("nginx open file cache miss rate: %d%%\n", misses * 100 / total)
}
```

```
$ ./ngx-open-file-cache-misses.sxx -x 19642
WARNING: Start tracing process 19642...
Hit Ctrl-C to end.
^C
nginx open file cache miss rate: 91%
```

♡ So only *9% ~ 10%* cache hit rate
for open_file_cache in our production systems.

♡ Let's *double* the cache size!

```
# file nginx.conf
open_file_cache  max=2000 inactive=180s;
```

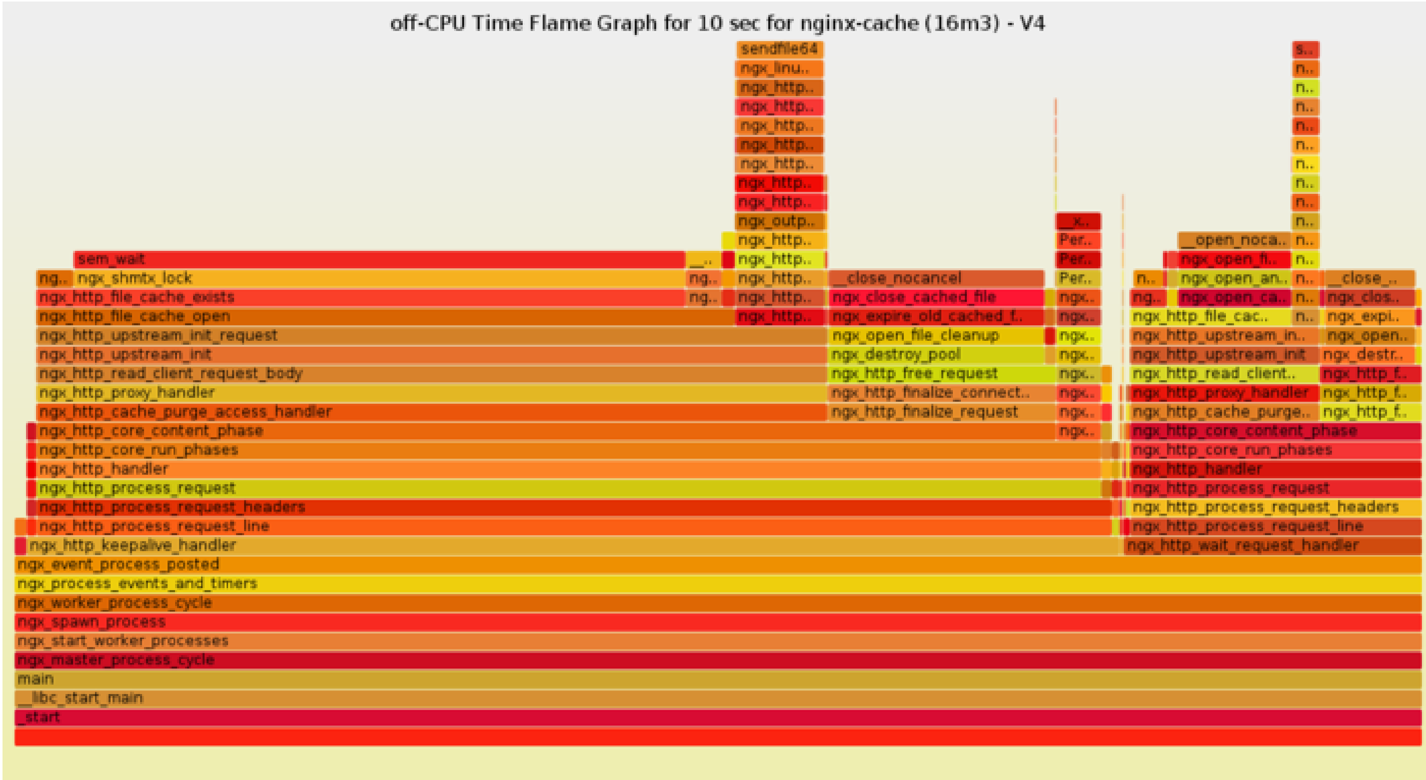
```
$ ./ngx-open-file-cache-misses.sxx -x 7818
WARNING: Start tracing process 7818...
Hit Ctrl-C to end.
^C
nginx open file cache miss rate: 79%
```

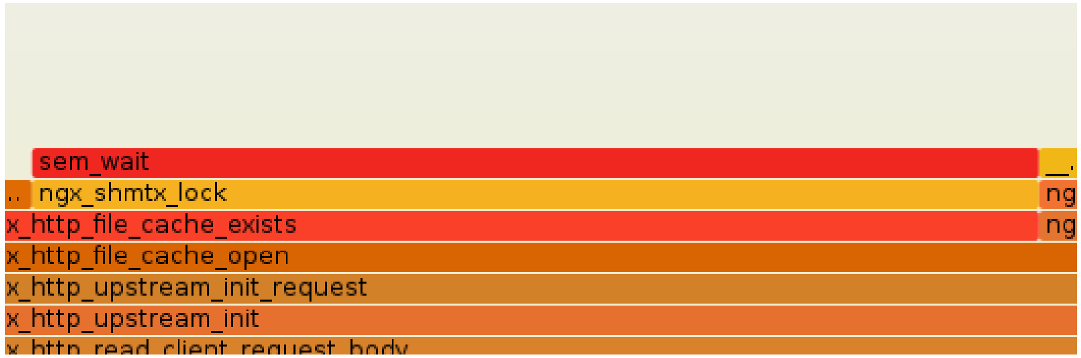
♡ Yay! The cache *hit rate* is also doubled!

21% Now!

♡ Lee said, "try *50k*!"

♡ Even a cache size of 20k did *not* fly.
The over-all performance was dropping!


off-CPU Time Flame Graph for 10 sec for nginx-cache (16m3) - V4
sendfile64
ngx_linu..
ngx_outp..
sem_wait
ngx_shmtx_lock
__close_nocancel
ngx_http_file_cache_exists
ngx_close_cached_file
ngx_http_file_cache_open
ngx_expire_old_cached_f..
ngx_http_upstream_init_request
ngx_open_file_cleanup
ngx_http_upstream_init
ngx_destroy_pool
ngx_http_read_client_request_body
ngx_http_free_request
ngx_http_proxy_handler
ngx_http_finalize_connect..
ngx_http_cache_purge_access_handler
ngx_http_finalize_request
ngx_http_core_content_phase
ngx_http_core_run_phases
ngx_http_handler
ngx_http_process_request
ngx_http_process_request_headers
ngx_http_process_request_line
ngx_http_keepalive_handler
ngx_http_wait_request_handler
ngx_event_process_posted
ngx_process_events_and_timers
ngx_worker_process_cycle
ngx_spawn_process
ngx_start_worker_processes
ngx_master_process_cycle
main
__libc_start_main
_start
__open_noca..
ngx_open_fi..
ngx_open_an..
ngx_open_ca..
__close_..
ngx_clos..
ngx_http_file_cac..
ngx_expi..
ngx_http_upstream_in..
ngx_open..
ngx_destr..
ngx_http_read_client..
ngx_http_cache_purge..



sem_wait
ngx_shmtx_lock
x_http_file_cache_exists
x_http_file_cache_open
x_http_upstream_init_request
x_http_upstream_init
x_http_read_client_request_body

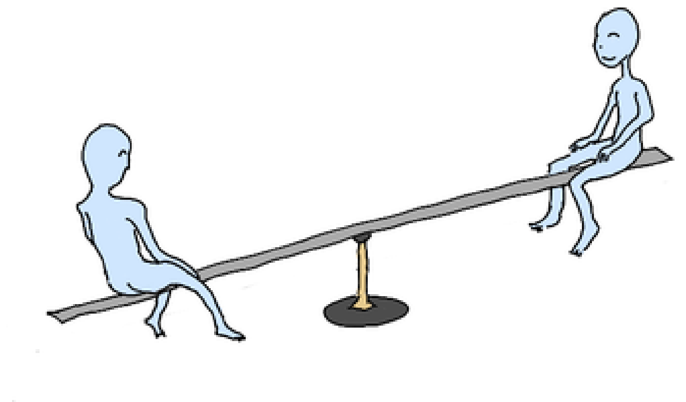
♡ So Nginx's open_file_cache is hopelessly waiting on shm *locks* when the cache size is large.

♡ So Flame Graphs *saved* us again ☺

♡ When we are focusing on optimizing *one metric*, we might *introduce* new bigger bottleneck by accident.

♡ Flame Graphs can always give us the *whole picture*.

♡ Optimizations are also all about *balance*.



♡ Nginx's open_file_cache is already a *dead end*.

Let's focus on file IO itself instead.

```
$ ./func-latency-distr.sxx -x 18243 --arg func=syscall.open --arg time=20
Start tracing 18243...
Please wait for 20 seconds.
```

```
Distribution of sys_open latencies (in microseconds)
max/avg/min: 565270/2225/5
  value |-------------------------------------------------- count
      8 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@   731
     16 |@@@@@@@@@@@@@@                                     211
     32 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@                 510
     64 |@@@@                                                65
    128 |                                                     2
    256 |@@@@@@@@@@                                         150
    512 |@@@@@@@                                            119
   1024 |@                                                   21
   2048 |                                                    14
   4096 |                                                     9
   8192 |                                                    10
  16384 |                                                     3
  32768 |                                                     9
  65536 |                                                     4
 131072 |                                                     3
 262144 |                                                     5
 524288 |                                                     1
```

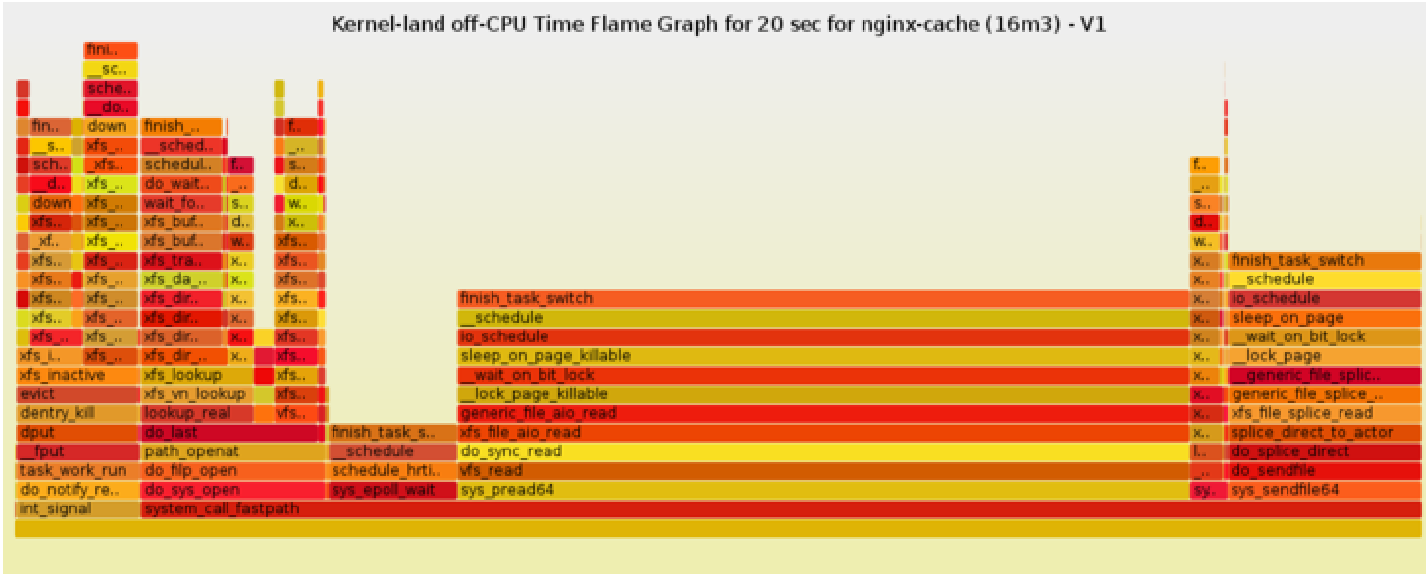
♡ Knowing how the latency of individual file IO operations is *distributed*, we can trace the details of those "slow samples".

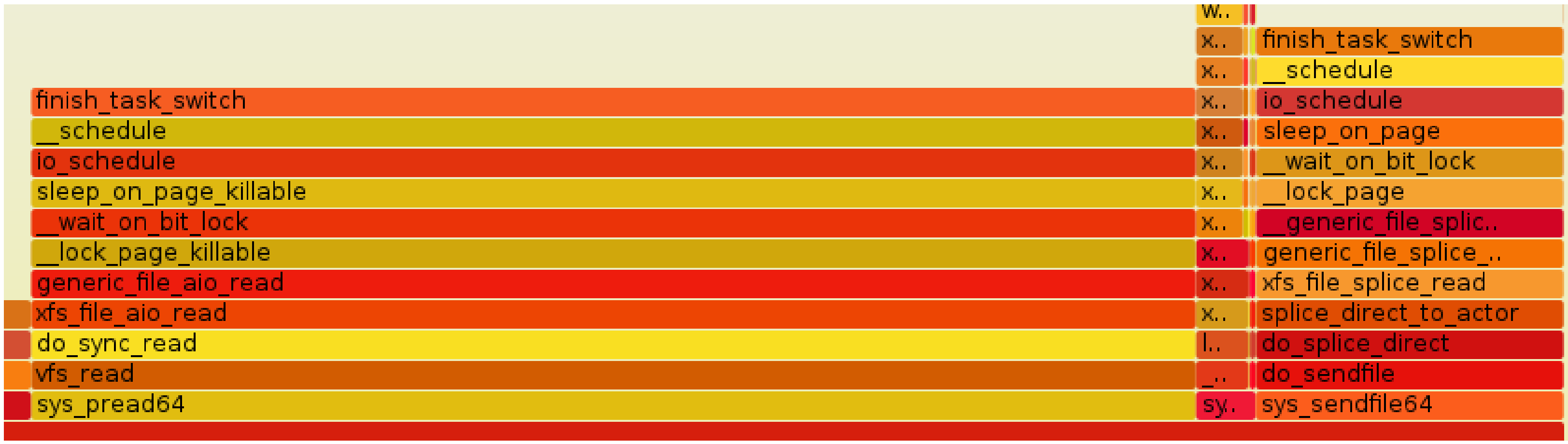
```
$ ./slow-vfs-reads.sxx -x 6954 --arg limit=100
Start tracing 6954...
Hit Ctrl-C to end.
[1377049930] latency=481ms dev=sde1 bytes_read=350 err=0 errstr=
[1377049934] latency=497ms dev=sdc1 bytes_read=426 err=0 errstr=
[1377049945] latency=234ms dev=sdf1 bytes_read=519 err=0 errstr=
[1377049947] latency=995ms dev=sdb1 bytes_read=311 err=0 errstr=
[1377049949] latency=208ms dev=sde1 bytes_read=594 err=0 errstr=
[1377049949] latency=430ms dev=sde1 bytes_read=4096 err=0 errstr=
[1377049949] latency=338ms dev=sdd1 bytes_read=402 err=0 errstr=
[1377049950] latency=511ms dev=sdc1 bytes_read=5799 err=0 errstr=
```

♡ So the slow samples are distributed *evenly* among all the disk drives, and the data volumn involved in each call is also quite small.

♡ *Kernel*-level off-CPU Flame Graphs

```
$ ./ngx-sample-bt-off-cpu -p 7635 -k -t 10 > a.bt
```


Kernel-land off-CPU Time Flame Graph for 20 sec for nginx-cache (16m3) - V1
int_signal
system_call_fastpath
do_notify_re..
do_sys_open
sys_epoll_wait
sys_pread64
sys_sendfile64
task_work_run
do_filp_open
schedule_hrti..
vfs_read
do_sendfile
__fput
path_openat
__schedule
do_sync_read
do_splice_direct
dput
do_last
finish_task_s..
xfs_file_aio_read
splice_direct_to_actor
dentry_kill
lookup_real
generic_file_aio_read
xfs_file_splice_read
evict
xfs_vn_lookup
__lock_page_killable
generic_file_splice_..
xfs_inactive
xfs_lookup
__wait_on_bit_lock
__generic_file_splic..
sleep_on_page_killable
__lock_page
io_schedule
__wait_on_bit_lock
__schedule
sleep_on_page
finish_task_switch
io_schedule
__schedule
finish_task_switch



w..
x..
finish_task_switch
x..
__schedule
finish_task_switch
x..
io_schedule
__schedule
x..
sleep_on_page
io_schedule
x..
__wait_on_bit_lock
sleep_on_page_killable
x..
__lock_page
__wait_on_bit_lock
x..
__generic_file_splic..
__lock_page_killable
x..
generic_file_splice_..
generic_file_aio_read
x..
xfs_file_splice_read
xfs_file_aio_read
x..
splice_direct_to_actor
do_sync_read
l..
do_splice_direct
vfs_read
_..
do_sendfile
sys_pread64
sy..
sys_sendfile64

♡ I love Flame Graphs because they are one kind of *visualizations* that are truly actionable.

Credits

Thanks Brendan Gregg for *inventing* Flame Graphs.

Thanks *systemtap* which was created after dtrace.

Thanks Joshua Dankbaar for *walking* me through our production environment.

Thanks Ian Applegate for *supporting* use of systemtap in production.

Thanks Dane for *pushing* everyone onto the same page.

Systems and systems' laws lay hid in night.
God said, "let dtrace be!" and all was light.

☺ *Any questions*? ☺