Introduction to

{red}off-CPU {white}Time {red}Flame Graphs

😊 agentzh@cloudflare.com 😊
Yichun Zhang (agentzh)

2013.08.22
Classic Flame Graphs are on-CPU time Flame Graphs per se.
off-CPU → on-CPU → off-CPU

process
We are already relying on them to optimize our Lua **WAF** & Lua **CDN Brain** (cfcheck)
I invented off-CPU time Flame Graphs somewhere near Lake Tahoe 3 months ago.
on-CPU  

zzz  

off-CPU  

process  

on-CPU
 Hearts I got the *inspiration* from Brendan Gregg's blog post "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
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  
    }  
}
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
sem_wait
ngx_shmfx_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)
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 @pfnc(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.

```plaintext
# 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*!

```conf
# 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!
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

<table>
<thead>
<tr>
<th>value</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>731</td>
</tr>
<tr>
<td>16</td>
<td>211</td>
</tr>
<tr>
<td>32</td>
<td>510</td>
</tr>
<tr>
<td>64</td>
<td>65</td>
</tr>
<tr>
<td>128</td>
<td>2</td>
</tr>
<tr>
<td>256</td>
<td>150</td>
</tr>
<tr>
<td>512</td>
<td>119</td>
</tr>
<tr>
<td>1024</td>
<td>21</td>
</tr>
<tr>
<td>2048</td>
<td>14</td>
</tr>
<tr>
<td>4096</td>
<td>9</td>
</tr>
<tr>
<td>8192</td>
<td>10</td>
</tr>
<tr>
<td>16384</td>
<td></td>
</tr>
<tr>
<td>32768</td>
<td></td>
</tr>
<tr>
<td>65536</td>
<td></td>
</tr>
<tr>
<td>131072</td>
<td></td>
</tr>
<tr>
<td>262144</td>
<td></td>
</tr>
<tr>
<td>524288</td>
<td></td>
</tr>
</tbody>
</table>
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 volume 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
<table>
<thead>
<tr>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>finish_task_switch</td>
</tr>
</tbody>
</table>
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?