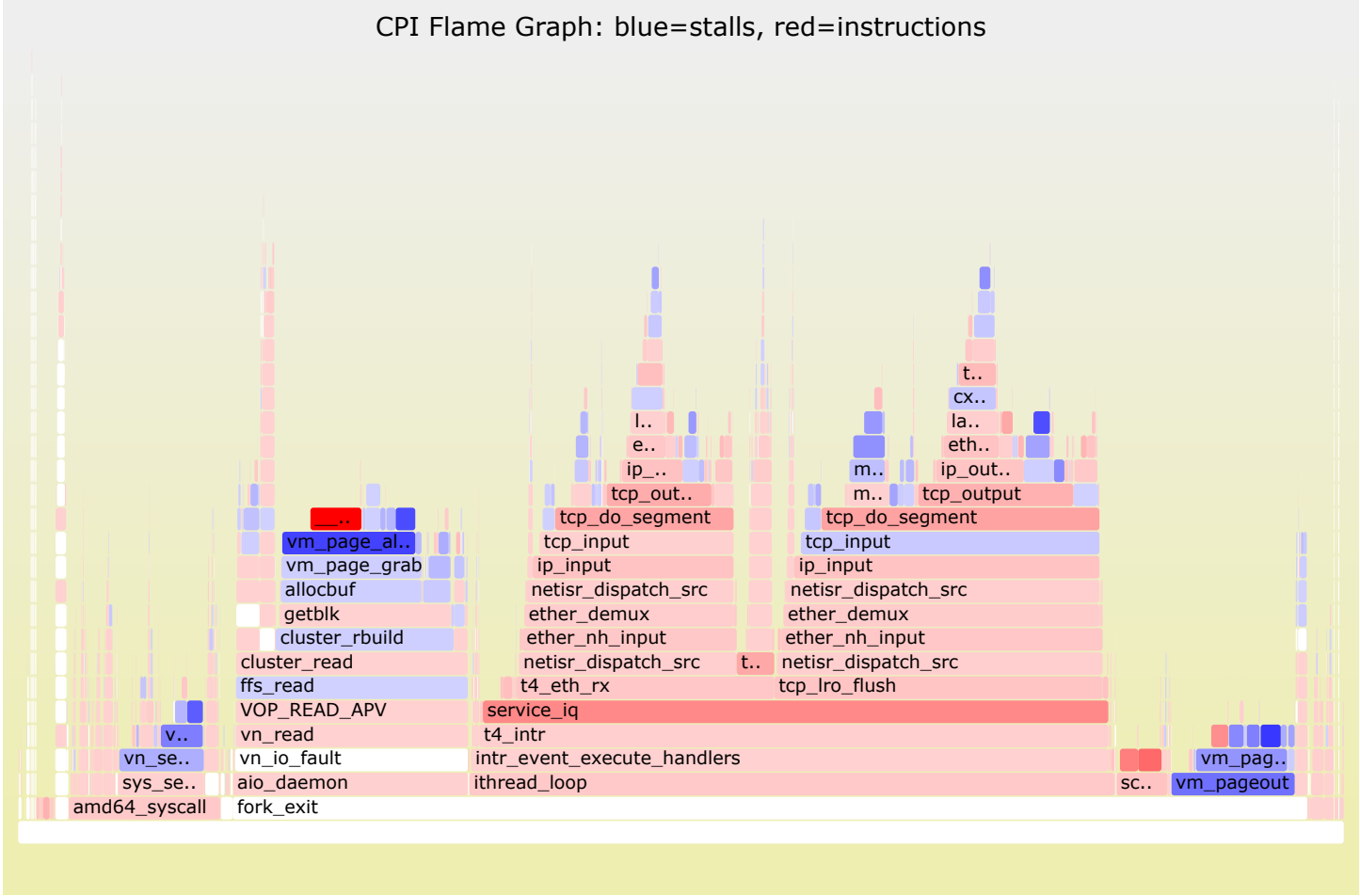
CPI Flame Graphs: Catching Your CPUs Napping

31 Oct 2014

When you see high CPU usage, you may be forgiven for believing that your CPUs must be furiously executing software, like obedient robots that never rest. In reality, you don't know what's happening from the CPU utilization alone. Your CPUs might be napping, during an instruction, while they wait for some resource.

Understanding what the CPUs are really doing helps direct performance tuning. The simplest way is to measure the average cycles-per-instruction (CPI) ratio: higher values mean it takes more cycles to complete instructions (often "stalled cycles" waiting on memory I/O). This is usually studied as a system-wide metric.

The following new visualization takes a CPU flame graph and then assigns colors relative to CPI:



Wow! For the first time I can *see* where the stall cycles are, providing a clear visualization of CPU efficiency by function.

The width of each frame shows the time a function (or its children) was on-CPU, as with a normal CPU flame graph. The color now shows what that function was doing when it was on-CPU: running or stalled.

The color range in this example has been scaled to color the highest CPI blue (slowest instructions), and lowest CPI red (fastest instructions). Flame graphs also now do click to zoom (thanks Adrien Mahieux), as well as mouse-overs. (Direct SVG link.)

This example is showing a FreeBSD kernel. The vm_* (virtual memory) functions have the slowest instructions, which is expected as they involve memory I/O to walk page tables, which may not cache as well as other workloads. The fastest instructions were in __mtx_lock_sleep, which is also expected as it is a spin loop.

The most color saturated frames on the left (vm_page_alloc->__mtx_lock_sleep) are being worked on by a fellow engineer in our OCA development team. Had that not already been the case, this CPI flame graph should have prompted their study and development.

There are two parts that make this possible: differential flame graphs, and measuring stacks on instructions and stall cycles. I'll briefly summarize these, with the latter using pmcstat(8) on FreeBSD (this approach is also possible on Linux, provided you have PMC access, which my current cloud instances don't!).

Differential Flame Graphs

This is a new feature which I'll explain in a separate post (I also want to add some more options first). In summary, flamegraph.pl usually takes input like the following:

```
func_a;func_b;func_c 31
...
```

These are single line entries with a semi-colon delimited stack followed by a sample count.

A differential flame graph will be generated if you provide the following input:

```
func_a;func_b;func_c 31 35
...
```

This now has two value columns, which are intended to show before and after counts. The flame graph is sized using the second column (the "after", or "now"), and then colored using the delta of 2 - 1: positive is red, negative is blue. The difffolded.pl tool takes two folded-style profiles (generated using the stackcollapse scripts) and emits this two-column output.

For CPI flame graphs, the two value columns are:

- stall cycle count
- unhalted cycle count

The first is scaled so that its average is the same as the second column, and the full blue-red range is used. Otherwise, there's always more unhalted cycles, and the differential flame graph is just red.

FreeBSD pmcstat

The data in this example was collected using FreeBSD's pmcstat(8) using the following (on Intel):

```
# pmcstat -n 1000000 -S RESOURCE_STALLS.ANY -S CPU_CLK_UNHALTED.THREAD_P -O out.pmclog_cpi01 slee
# pmcstat -R out.pmclog_cpi01 -z 32 -G out.pmcstat_cpi01
CONVERSION STATISTICS:
 #exec/elf                              12
 #samples/total                         123464
 #samples/unknown-function              2059
 #callchain/dubious-frames              4723
# ls -lh *cpi01
-rw-r--r--  1 root  wheel    14M Oct 29 04:24 out.pmclog_cpi01
-rw-r--r--  1 root  wheel   7.2M Oct 29 04:24 out.pmcstat_cpi01
```

Be careful with pmcstat(8): you can generate a lot of output quickly, which in turn can perturb the performance of what you are measuring. I only measured for 10 seconds (sleep 10), and only one event every million (-n 1000000). Despite this, the raw output file is 14 Mbytes.

In this example I measured kernel stacks (-S), and up to 32 frames deep. Note that I did need to set kern.hwpmc.callchaindepth=32 in /boot/loader.conf and reboot, as my stacks were at first always being truncated to 8 frames.

pmstat -G was used to create a callchain text file. It looks like this:

```
# more out.pmcstat_cpi01
@ CPU_CLK_UNHALTED.THREAD_P [10867 samples]

07.86%  [854]       __mtx_lock_sleep @ /boot/kernel/kernel
 48.95%  [418]        vm_page_alloc
  99.76%  [417]         vm_page_grab
   99.04%  [413]          allocbuf
    100.0%  [413]           getblk
     95.16%  [393]            cluster_rbuild
      100.0%  [393]             cluster_read
       100.0%  [393]              ffs_read
        100.0%  [393]               VOP_READ_APV
         100.0%  [393]                vn_read
          100.0%  [393]                 vn_io_fault
           100.0%  [393]                  aio_daemon
            100.0%  [393]                   fork_exit
     04.84%  [20]             cluster_read
      100.0%  [20]              ffs_read
[...]
@ CPU_CLK_UNHALTED.THREAD_P [10856 samples]
[...]
@ RESOURCE_STALLS.ANY [3462 samples]

06.15%  [213]       sf_buf_mext @ /boot/kernel/kernel
 100.0%  [213]        mb_free_ext
  100.0%  [213]         m_freem
   54.46%  [116]          reclaim_tx_descs
    100.0%  [116]           t4_eth_tx
[...]
```

For each counter, and then each CPU, the samples are printed as call chains.

CPI Flame Graph

To make a CPI flame graph from this call chain output, I needed to separate out the counters into separate files. I did this in about 60 seconds using vi, creating:

- out.pmcstat_cycles: all the sections titled CPU_CLK_UNHALTED.THREAD_P
- out.pmcstat_stalls: all the sections titled RESOURCE_STALLS.ANY

Yes, I should just write some Perl/awk to do this. I will next time.

I then converted these to the folded format, using stackcollapse-pmc.pl (by Ed Maste):

```
$ cat out.pmcstat_cycles | ./stackcollapse-pmc.pl > out.pmcstat_cycles.folded
$ cat out.pmcstat_stalls | ./stackcollapse-pmc.pl > out.pmcstat_stalls.folded
```

Finally, making the CPI flame graph:

```
$ ./difffolded.pl -n out.pmcstat_stall.folded out.pmcstat_cycles.folded | \
    ./flamegraph.pl --title "CPI Flame Graph: blue=stalls, red=instructions" --width=900 > cpi.sv
```

That's the SVG at the top of this page.

All the flame graph software (flamegraph.pl, stackcollapse-pmc.pl, difffolded.pl) can be found in the Flame Graph collection on github.

Conclusion

There is much you can do as either a software developer or system administrator once you know that CPU is either memory-bound or instruction-bound, resulting in small to large performance improvements. CPU flame graphs are already a great way to visualize how your software is using the CPUs. CPI flame graphs use color to show what the CPUs are really doing: their efficiency by function.

PS. This weekend I'm speaking at MeetBSD on performance analysis. I will be sure to mention pmcstat(8), which can do a lot more than just stalls and cycles.

Randall Stewart • 4 years ago

Brendan

Nice use of PMC.. if your an old crufty like me you can use
http;//people.freebsd.org/~rrs/tra...
Instead of vi.. (us old crufty's use 'C' :-D)

3 ^ v • Share ›

hmijail • 3 years ago

Brendan,

if I am understanding correctly the FreeBSD pmcstat manpage and your command lines, you sampled once every 1M events – but did so independently on 2 different counters; and each sample was recorded linked to its triggering counter value, but NOT to the other counter value. Is that right?

If that is so, then the final connection between both counters is rather tenuous, and depends totally on the code running on a rather tight loop, doesn't it? Because if it's not looping enough, the events from different counters will happen in totally unrelated code sections. Which is OK if the goal is to "only" compare which parts of the code trigger more stall events than cycle events; but IMHO calling that "CPI" is a bit of a stretch.

I am asking this because I am trying to use CPI flamegraphs on Mac OS X, and having to use Instruments.app to do so. But Instruments.app samples on one kind of event, and captures any other counters together with the stack trace. So, given that both the cycles and stalls counters are captured at the same moment, it is reasonable to calculate an average CPI for the last sample period. However now I am struggling to map this style of counters into your "folded format"… I would propose generalising the "2-column" case used for differential flamegraphs: instead of needing to refer to both values, make them independent, and use one of the columns for width and the other for color. That way any variation on the data to be visualized could be implemented independently of flamegraph generation. I might try implementing it myself.

Regards, and thanks for a lot of interesting posts!

^ v • Share ›

suken woo • 4 years ago

I create a SVG file following your instruction and could you help me to figure out why my cpu usage > 200% . I just running Wildfly8.1 and my server has 40 CPUS ,256GB RAM

^ v • Share ›

Jaehyun Kim • 4 years ago

Is it possible to find the index of this article?^^ I think that it is more convenient if there is an index. Thank you Brendan for really useful article all the time.

^ v • Share ›