Unikernel Profiling: Flame Graphs from dom0

27 Jan 2016

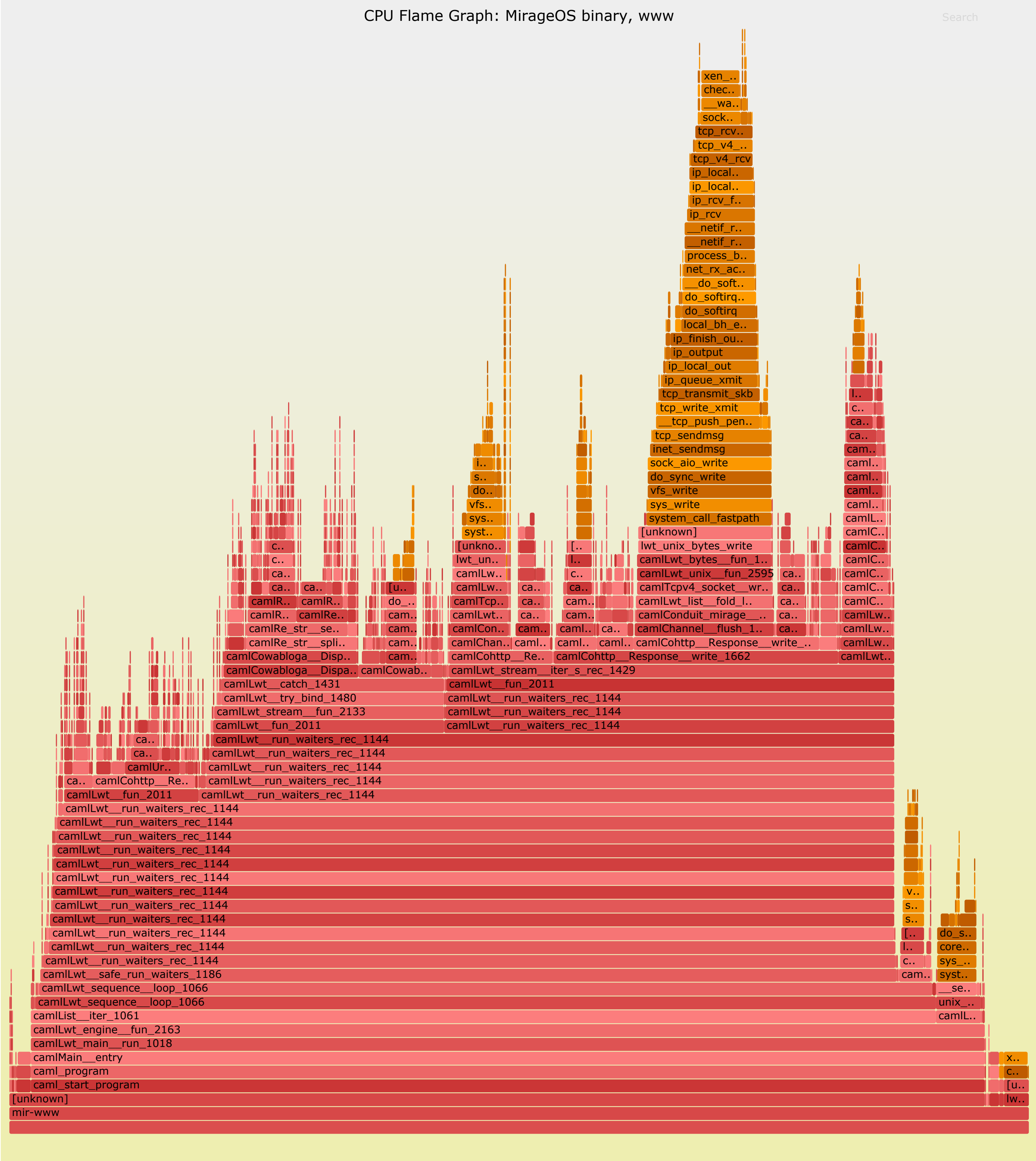
Is a unikernel an impenetrable black box, resistant to profilers, tracers, fire, and poison? At SCaLE14x this weekend there was a full day track on unikernels, after which I was asked about unikernel profiling and tracing. I'm not an expert on the topic, and wasn't able to answer these questions at the time, however, I've since taken a quick look using MirageOS and Xen.

In this post I'll share a proof of concept for profiling a Xen domU unikernel from dom0. I didn't find any recent examples of doing this (there are some 2006 oprofile patches). This is just a PoC, not yet a polished product. Although, by the time you read this, there may be something better based on this work (check comments).

I'll begin by running a unikernel as a normal process for profiling, and then as a Xen guest.

Normal Process Profiling

Voila, a CPU flame graph (SVG)!



This is the MirageOS [www example](), compiled and executed in "Unix binary" mode, where it runs as a normal process under Linux -- no hypervisor:

```
# mirage configure --unix
# make
# ./mir-www &
```

I then profiled the process using Linux [perf_events](). See the OCaml docs [Using perf on Linux](), and my docs on [CPU flame graphs](). The steps were:

```
# perf record -F 99 -a --call-graph=dwarf -- sleep 10
# git clone https://github.com/brendangregg/FlameGraph
# perf script | ./FlameGraph/stackcollapse-perf.pl --kernel | \
    ./FlameGraph/flamegraph.pl --color=java > out.svg
```

The `--call-graph=dwarf` only works on newish Linux. On older Linux you'll need to use a `+fp` (frame pointer) version of the ocaml compiler. I also used the java palette here, because it highlights kernel-mode as orange and user-mode as red. These will not be separate modes once I start running this under Xen.

Does Unix binary profiling matter? A few scenarios come to mind:

- The developer writes code that is picked up by a build system and compiled straight to Xen. In this scenario, there is no chance for this type of profiling.
- Same as above, but with an extra build step added for the Unix binary, for running with a test suite and profiler before Xen compilation.
- A developer may use Unix binary builds as a normal step for testing.

In the latter two cases, this type of profiling can be used to catch a variety of issues, before running under Xen. You can also use any other tools to debug the binary unikernel (strace, etc).

This Unix binary build will execute differently to a Xen build for a number of reasons. A Xen unikernel will use a single address space without context switching, hypercalls for various resources, and will run a different body of kernel code (TCP/IP, etc). It's very possible that some performance issues may only manifest when running under Xen, so you will want to be able to profile it there as well.

xenctl stack walking

As I found no examples of stack trace profiling, I did start wondering if this was even possible. The following shows one approach I've found that works; I now suspect other approaches are also possible.

1. Compile unikernel with frame pointers

There's more than one way to walk a stack. The easiest is usually via frame pointers, provided they are preserved by the compiler.

For MirageOS, the default ocaml compiler omits the frame pointer, but it can be returned by switching to a `+fp` ocaml compiler (this is like needing `-fno-omit-frame-pointer` for gcc). The steps I used were (also follow the mirage install docs):

```
# opam switch 4.02.3+fp
# eval `opam config env`
# opam install mirage
```

Then I compiled and ran the unikernel as a Xen guest, going back to the console example (which I hacked to make it hotter on-CPU):

```
# mirage configure --xen
# make
# xl create console.xl
```

2. Create a symbol file

This can be anything that has memory addresses and function names, and will be used to translate the instruction pointer addresses in stack traces. If you don't have this working, you'll get hexadecimal numbers.

I found an `objdump` of the MirageOS compiled object to be suitable (didn't even need transposing, although it may for other unikernels/builds), and used some awk process the output:

```
dom0# objdump -x mir-console.xen | awk '{ print $1, $2, $NF }' > /tmp/xen.map
dom0# more /tmp/xen.map
[...]
00000000000d04c0 l caml_negf_mask
00000000000d04d0 l caml_absf_mask
00000000000f4378 l camlConsole_xen__370
00000000000f4398 l camlConsole_xen__371
00000000000f43b8 l camlConsole_xen__372
[...]
```

3. Take a stack trace snapshot

The xenctx command can be executed from dom0 to dump registers and the call stack of a domU. You might find it under /usr/lib/xen-4.4/bin/xenctx, or in the Xen source under tools/xentrace.

Here I'm using the `-C` option prints all CPUs, `-s` to specify my symbol file, and a domU ID of 26. I'd guess for some unikernels you may need to use `-k` to change the kernel base address, although I didn't need to here:

```
dom0# xenctx -C -s /tmp/xen.map 26
vcpu0:
rip: 0000000000049ad7 camlLwt__return_1319+0x27
flags: 00000206 i nz p
rsp: 000000000019fe50
rax: 000000000014e198   rcx: 0000000000350020   rdx: 000000000004a464
rbx: 0000000000000001   rsi: 0000000000007270   rdi: 000000000044e9c8
rbp: 000000000019fe50    r8: 000000000000000d    r9: 000000000000000d
r10: 000000000015e5f0   r11: 0000000000000010   r12: 00000000000f1150
r13: 0000000000590c78   r14: 000000000019feb0   r15: 000000000044e9c0
 cs: e033    ss: e02b    ds: 0000    es: 0000
 fs: 0000 @ 0000000000000000 .comment
 gs: 0000 @ 000000000015f4b0/0000000000000000 cpu0_pda/ .comment
Code (instr addr 00049ad7)
d2 46 10 00 4c 3b 38 72 24 49 8d 7f 08 48 c7 47 f8 00 04 00 00 <48> 89 1f 48 8d 47 10 48 c7 40 f8

Stack:
 000000000019fe70 0000000000007312 000000000044e9e8 0000000000000003
 000000000019fe80 0000000000007154 000000000019fe90 000000000000725d
 000000000019fea0 000000000000358d 000000000019ff50 00000000000b07b2
 0000000000000000 00000000000b07e9 0000000000000000 0000000000000001
 0000000000000000 00000000000af871 00000008000f4000 0000000000000000

Call Trace:
                    [<0000000000049ad7>] camlLwt__return_1319+0x27 <--
000000000019fe58:   [<0000000000007312>] camlUnikernel____pa_lwt_loop_1376+0x72
000000000019fe68:   [<0000000000000003>] _start+0x3
000000000019fe78:   [<0000000000007154>] camlMain__fun_1404+0x14
000000000019fe88:   [<000000000000725d>] camlMain__entry+0xcd
000000000019fe98:   [<000000000000358d>] caml_program+0x58d
000000000019fea8:   [<00000000000b07b2>] caml_start_program+0x4a
000000000019feb0:   [<0000000000000000>] .comment
[...]
```

Excellent, we have a call trace and translated frames. If I can collect many of these, I'm profiling!

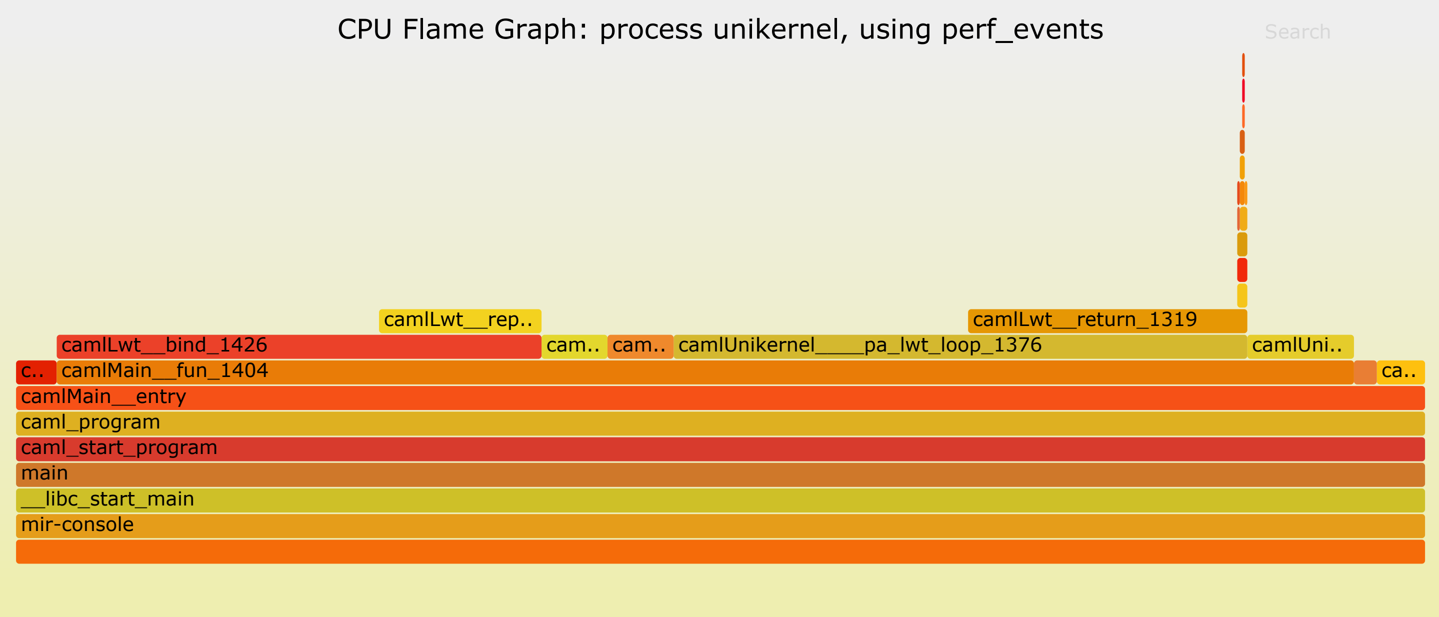
Unikernels make profiling a bit easier, for a couple of reasons:

- No separate kernel-/user-mode stacks. We only need to walk one stack for everything.
- No separate processes. We only need one symbol map for translating addresses.

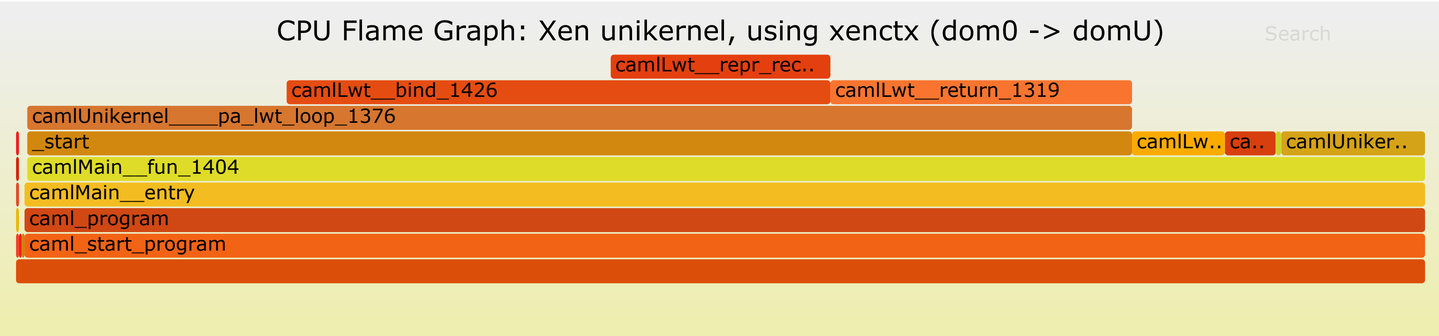
I thought I might have to write my own stack walker, but xenctx can already do this. If I execute xenctx many times, I'll have a rough PoC.

Dom0 to DomU: Proof of Concept

Here's a CPU flame graph profile of my simple test program, compiled as a Unix binary and profiled using Linux perf_events (SVG):



Now the same program as a Xen domU unikernel, profiled using xenctx in a loop (SVG):



I used a xenctx wrapper (xenctx_prof.sh) which uses awk to scrape the output. I'd executed it, and made the flame graph, in the following vastly inefficient way:

```
dom0# (for i in `seq 1 1000`; do
    ./xenctx_prof.sh -s /tmp/xen.map 26; echo 1; echo; sleep 0.01; done) > out.stacks
dom0# ./stackcollapse.pl < out.stacks | ./flamegraph.pl > out.svg
...
```

Please don't actually run this. See the next section.

Dom0 to DomU: Profiler

The xenctx source can be made into a real profiler by adding something like:

> for (int sample = 0; sample < max; sample++) { ... print stack only ...; usleep(10000); }

Command line options can also be added: I'd suggest "-F frequency", and a "-T duration". I'd also test it to get a handle on its overhead and safety, and then document expectations clearly. It does do a xc_domain_pause(), which I'm guessing introduces latency. This may also need some optimization to reduce that latency.

Unfortunately, developing a unikernel profiler isn't a priority for me (my company is not currently using unikernels), so won't be doing this right now. You are welcome to do this yourself, if this interest you!

DomU profiler

What if you don't have access to dom0? Ideally, we have a domU-from-domU profiler.

For MirageOS, the ocaml compiler source includes stack trace routines for printing exception stacks. This can serve as example code for walking stack traces, and could be made into a profiler interface that worked similar to tools like Java Flight Recorder / Java Mission Control. These attach to a port to enable and disable profiling, and to collect profile data.

Summary

I've spent a few hours with MirageOS, and was able to profile a unikernel as a Unix binary with traditional tools (Linux perf_events), and then develop a proof of concept dom0 to domU unikernel profiler, for profiling a Xen guest unikernel. For both approaches, I was able to create CPU flame graphs.

I'm tempted to spend a bit more time and polish this dom0 profiler, rewriting it entirely in C. However, it may be more practical to write a domU-from-domU profiler instead. I wouldn't be surprised if one already existed (I'd been exploring the dom0 approach).

For more reading about unikernel observability, check out Thomas Leonard's [trace visualizations].

Comments for this thread are now closed ×

7 Comments **Brendan Gregg's Blog** 1 Login

Recommend 1 Tweet Share Sort by Best

tal195 • 3 years ago

Great stuff! Would be good to get it added to the domU tracing stuff here: https://mirage.io/wiki/prof...

2 ^ | v • Share ›

brendangregg Mod → tal195 • 3 years ago

Thanks, good idea, and I'd just added a link to your trace visualizations too!

1 ^ | v • Share ›

flosch • 2 years ago

Very nice! We actually tried something almost exactly the same, but we realized that the xenctx tool just has a lot of overhead for every stack trace. (That's not really a slight at the xenctx, since I wasn't designed for high-performance stack tracing in the first place). In the end, we did indeed develop our own tool for the purpose. We published it here: https://github.com/cnplab/u... We don't have numbers published on the overhead at this point in time, but we will get around to it soon(tm). Bottom line is: the pausing/unpausing of domains is actually quite cheap, by far the most overhead is from translating memory addresses and mapping the memory so that dom0 can access the domU's memory.

1 ^ | v • Share ›

brendangregg Mod → flosch • 2 years ago

Looks promising!

^ | v • Share ›

brendangregg Mod • 3 years ago

There's also been work to make Linux perf_events be able to profile guests, via "perf kvm --guest record".

1 ^ | v • Share ›

brendangregg Mod • 3 years ago

Realized I should have added a third reason unikernels are a bit easier to profile: one compiler to configure to use frame pointers. On a normal system, you may have different apps and libraries with different compilers (including runtimes using JIT) that need different configuration settings to use frame pointers. With a unikernel, you only need to customize the compiler in one place.

^ | v • Share ›

anonymous2047 • 3 years ago

http://libvmi.com/ and https://dl.acm.org/citation... might also be of interest

^ | v • Share ›

You can comment here, but I can't guarantee your comment will remain here forever: I might switch comment systems at some point (eg, if disqus add advertisements).

About this blog

Blog
Full Site Map
Sys Perf book
Linux Perf
Perf Methods
USE Method
TSA Method
Off-CPU Analysis
Active Bench.
Flame Graphs
Heat Maps
Frequency Trails
Colony Graphs
perf Examples
eBPF Tools
DTrace Tools
DTraceToolkit
DtkshDemos
Guessing Game
Specials
Books
Other Sites