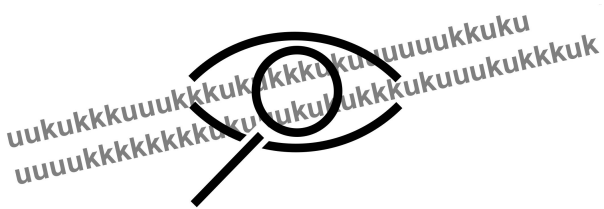
KUtrace 2020

Richard L. Sites
February 2020



dick.sites@gmail.com

Talk outline

The problem

Interesting solution

Example 1 hello world

Example 2 schedulers

Example 3 client-server database

Example 4 interference

Experimental examples

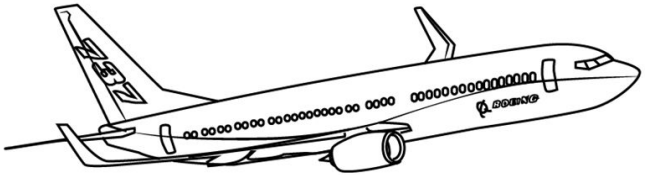
Contributions

The problem

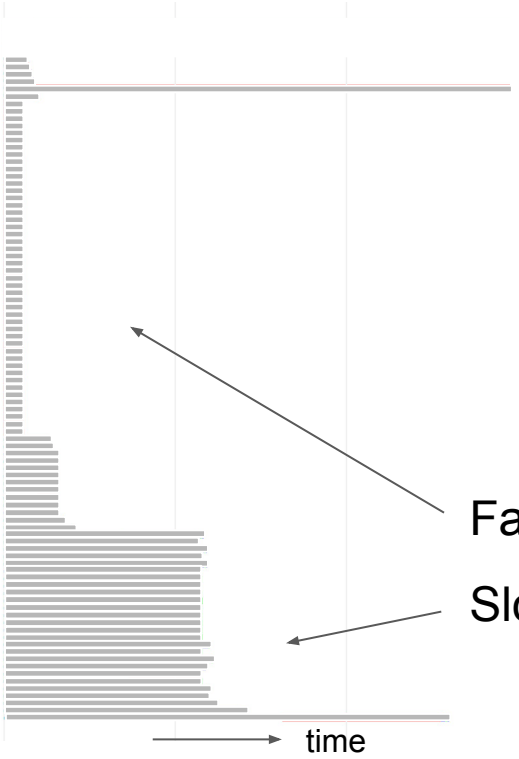
The problem

Understand **why** complex real-time software is slow

- datacenter response times
- database accesses
- real-time controls

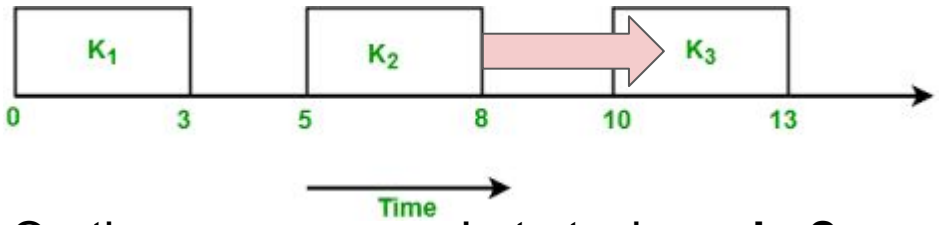


Transactions and deadlines



Fast/normal Transactions

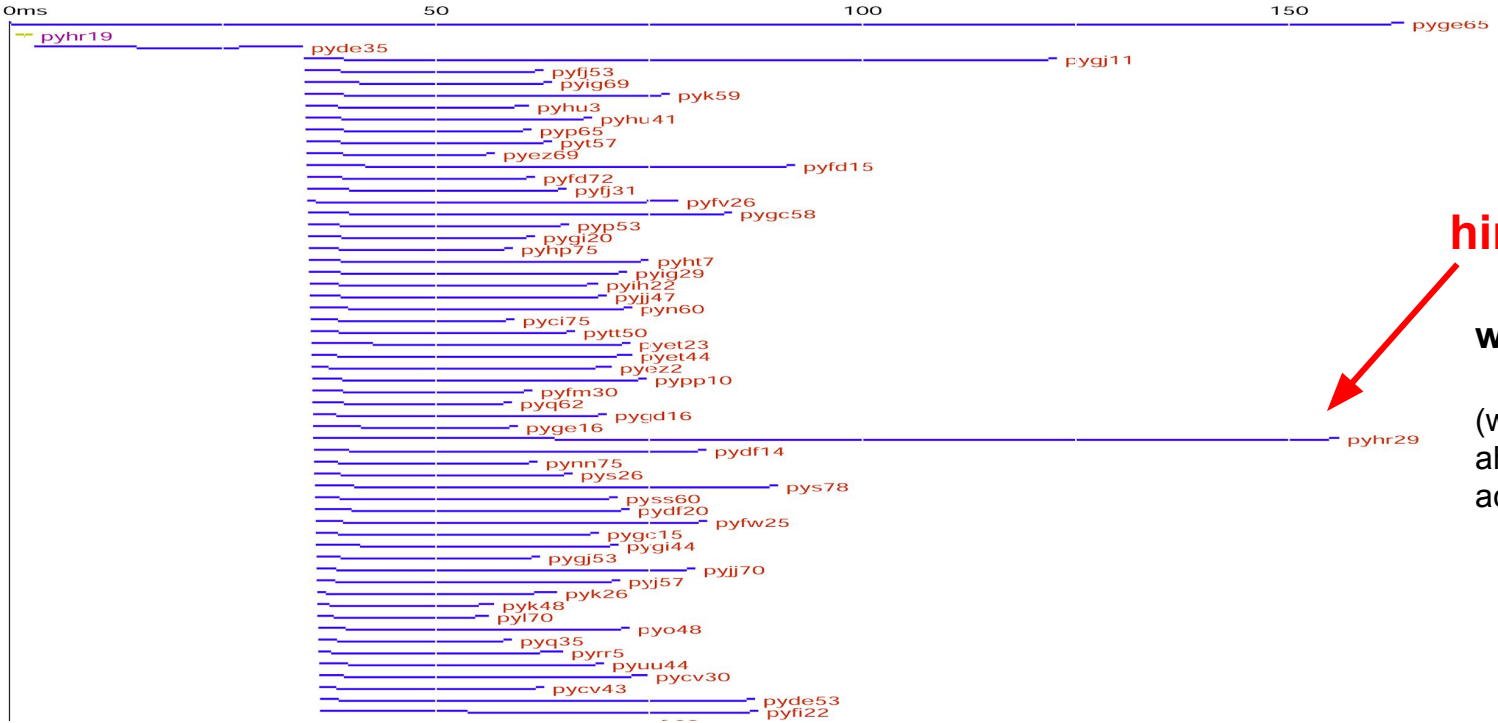
Slow transactions -- **why?**



On-time
real-time task

Late task -- **why?**

Slow transactions -- latency of nested search RPCs

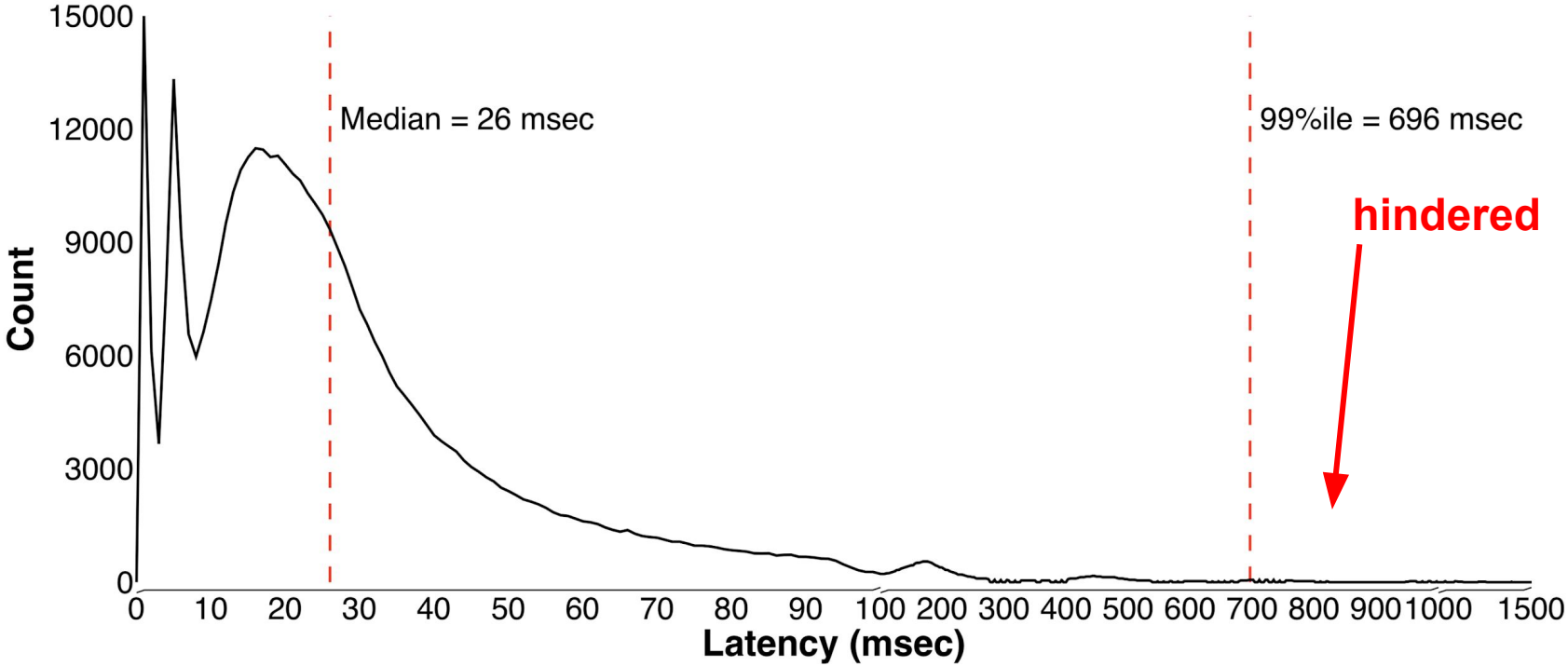


hindered

why?

(we can't predict ahead of time which actions will be slow)

Slow transactions -- histogram with long tail latency



Traditional performance tools don't help with **why**

- Counters (e.g. transactions/second) only tell you average behavior
- Profiles (sampled CPU time by procedure name) merge together 99 normal transactions with the 1% slow ones, obscuring them entirely
- CPU profiles are **blind** to non-execution (i.e. waiting)
- Traditional traces are much too slow and distorting for use on live traffic *in situ*
- The most common strategy: **Guessing** why something is slow

Programmers are singularly inept at guessing how the picture in their head differs from reality

Interesting solution

A way forward

Only **tracing** for several minutes of everything that is happening on a computer can catch unpredictable slow requests. If you know ahead of time which requests will be slow, simpler methods could suffice. But you don't.

KUtrace is an extremely low-overhead tracing tool

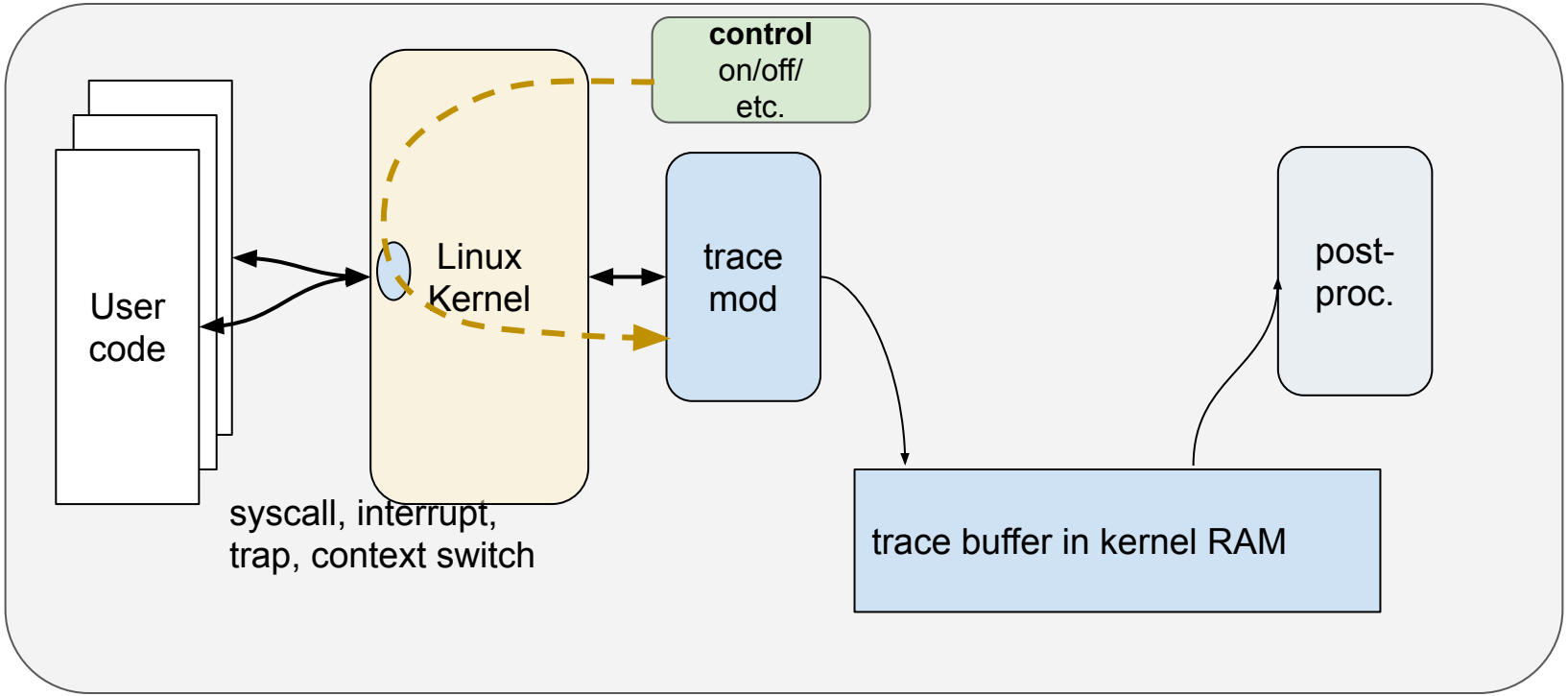
- Reveals the true execution and non-execution (wait) dynamics of complex software
- Runs *in situ* with live traffic
- Based on small Linux kernel patches
- Records/timestamps every *transition* between kernel- and user-mode execution, across all CPUs
- Overhead is **well under 1%** at a datacenter's 200,000 transitions per second per CPU core
- Display shows exactly what each CPU is doing every nanosecond, nothing missing
- Trace data shows exactly **why** unpredictable requests are slow

Why trace kernel-user transitions?

Goldilocks...

- Tracing more events, such as every procedure entry/exit, is too slow and a *subset* of a more-events trace gives an incomplete picture
- Tracing fewer events, such as just context switches, is not sufficient to understand slowness
- Tracing kernel-user transitions captures everything all CPUs are doing *every nanosecond* with no subsetting -- just right

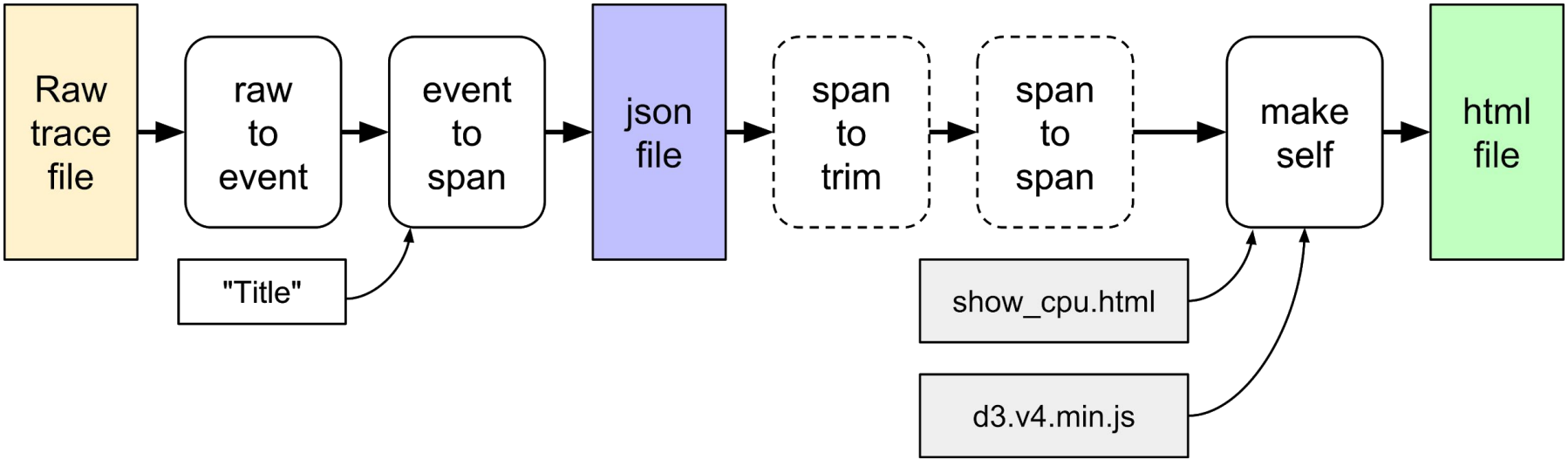
KUtrace implementation, via small Linux kernel patches



Syscall Linux code, patched

```
__visible void do_syscall_64(unsigned long nr, struct pt_regs *regs)
        ...
      if (likely(nr < NR_syscalls)) {
            nr = array_index_nospec(nr, NR_syscalls);
            kutrace1(KUTRACE_SYSCALL64 | kutrace_map_nr(nr), regs->di);
                                                              // arg0
            regs->ax = sys_call_table[nr](regs);
            kutrace1(KUTRACE_SYSRET64 | kutrace_map_nr(nr), regs->ax);
                                                              //ret
      }
        ...
```

KUtrace Postprocessing

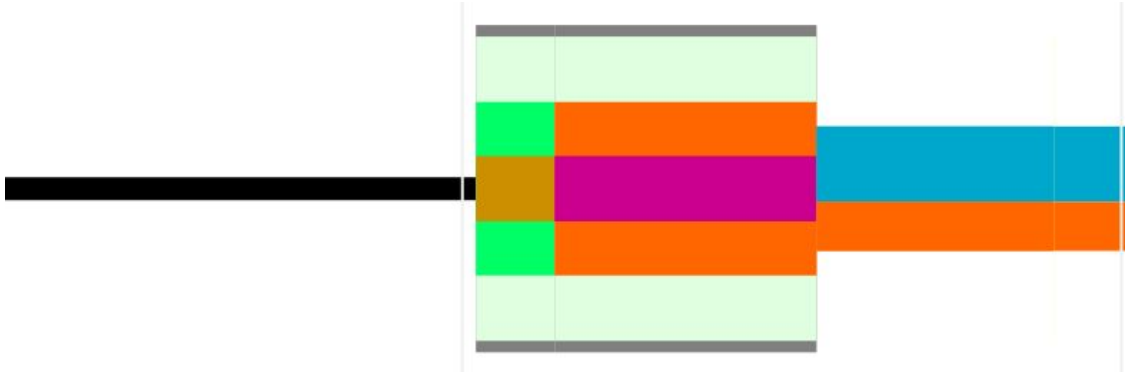


Example 1 hello world

Hello world

```
int main(int argc, const char** argv) {
  kutrace::mark_a("Hello");
  printf("hello world\n");
  kutrace::mark_a("/Hello");
  return 0;
}
```

Notation



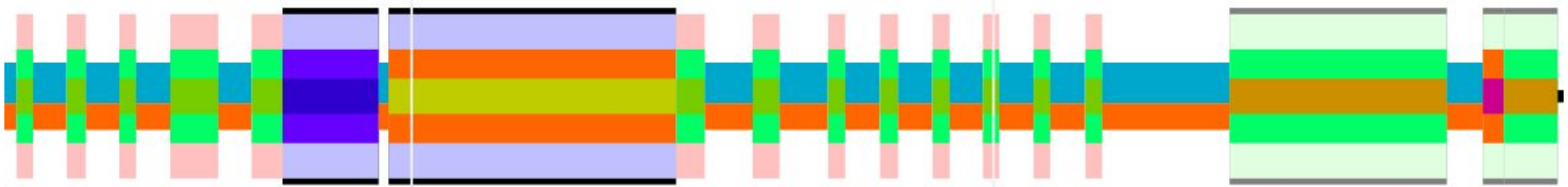
idle task
executing

kernel-mode

user-mode

center colors distinguish different process IDs, syscall #, etc.

Notation

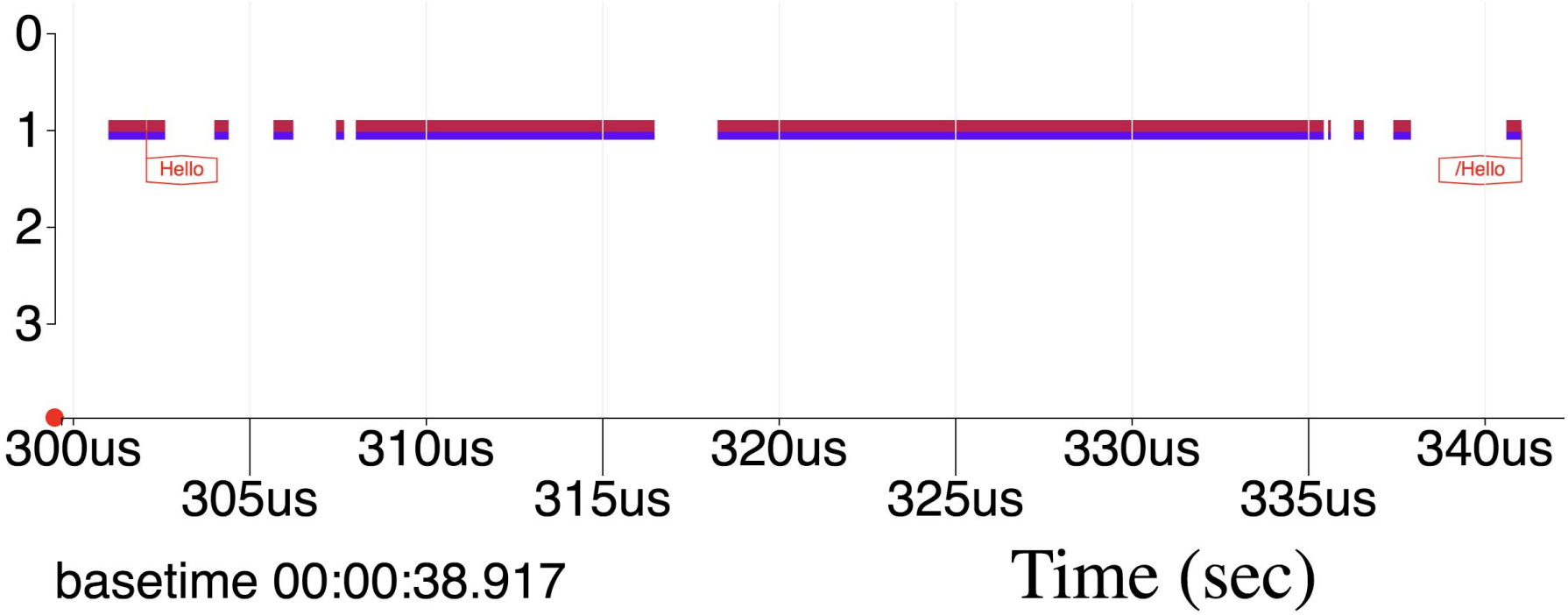


kernel: pink edge = faults

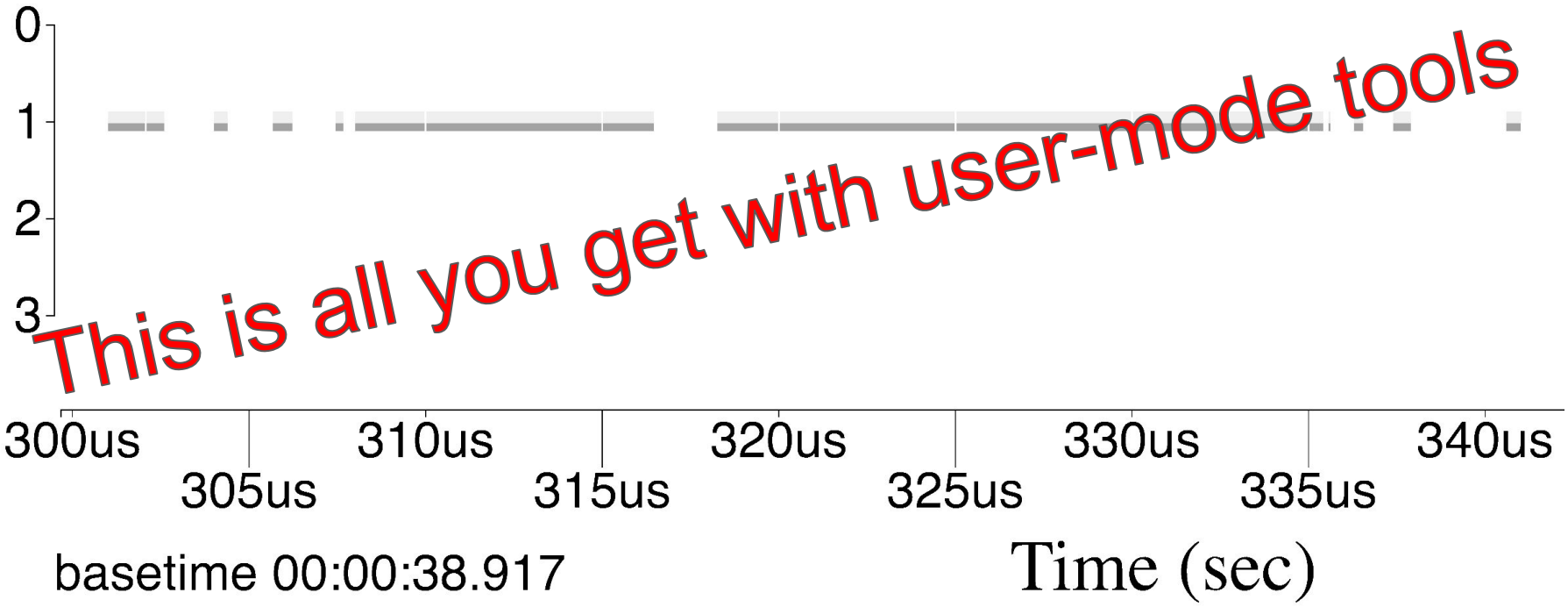
blue edge = interrupts

green edge = system calls

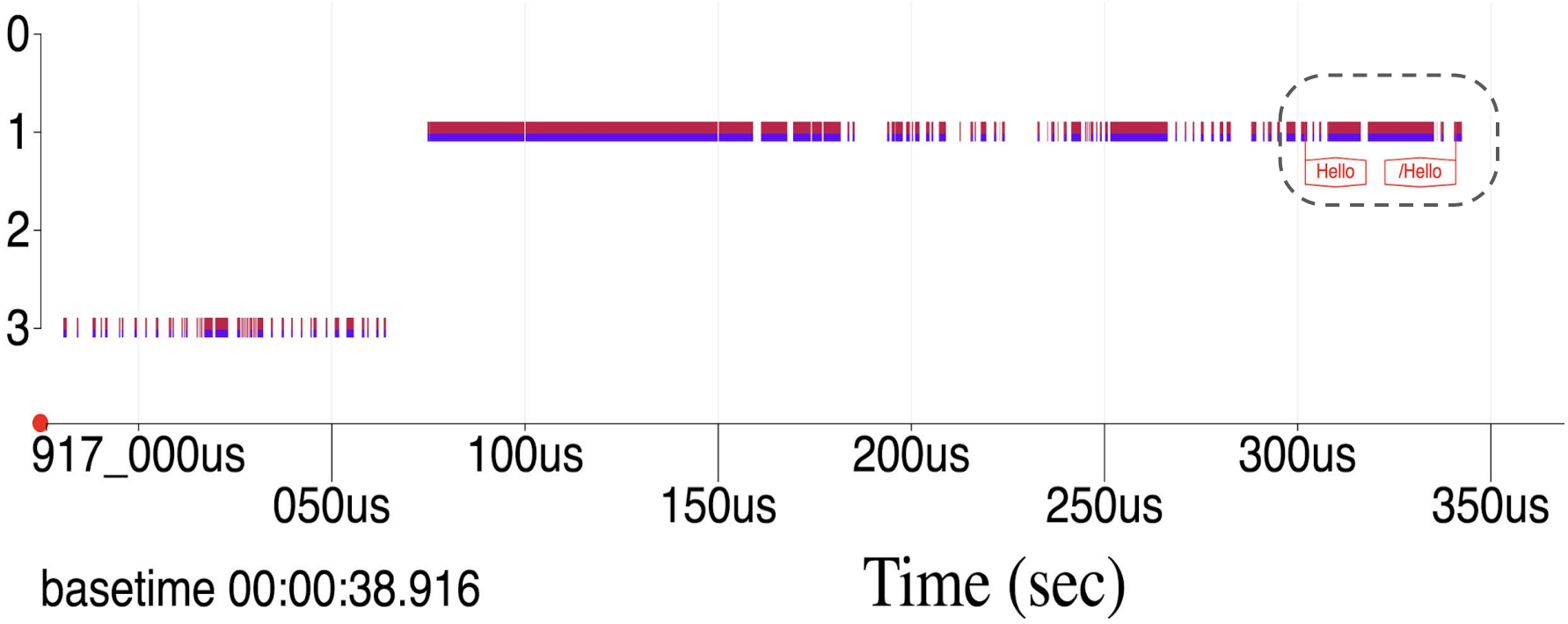
Hello world user-mode main program, 40 usec



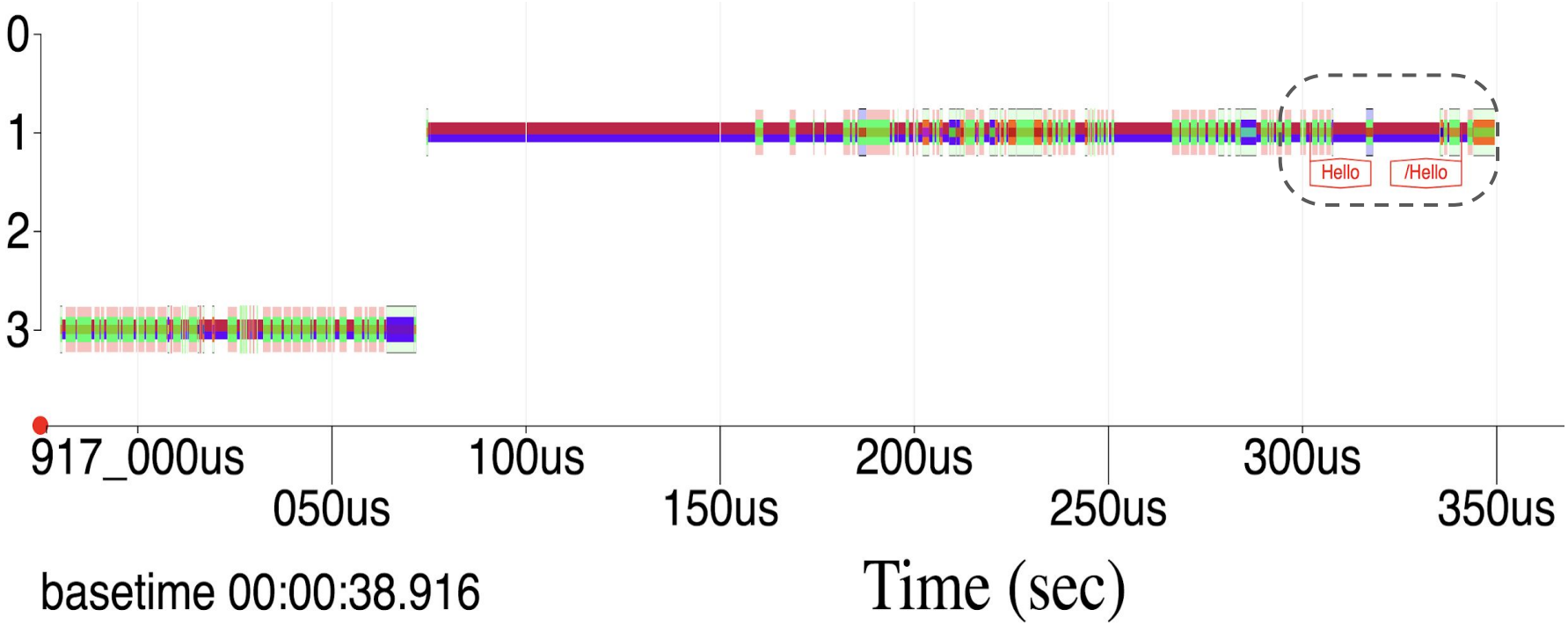
Hello world user-mode main program, 40 usec



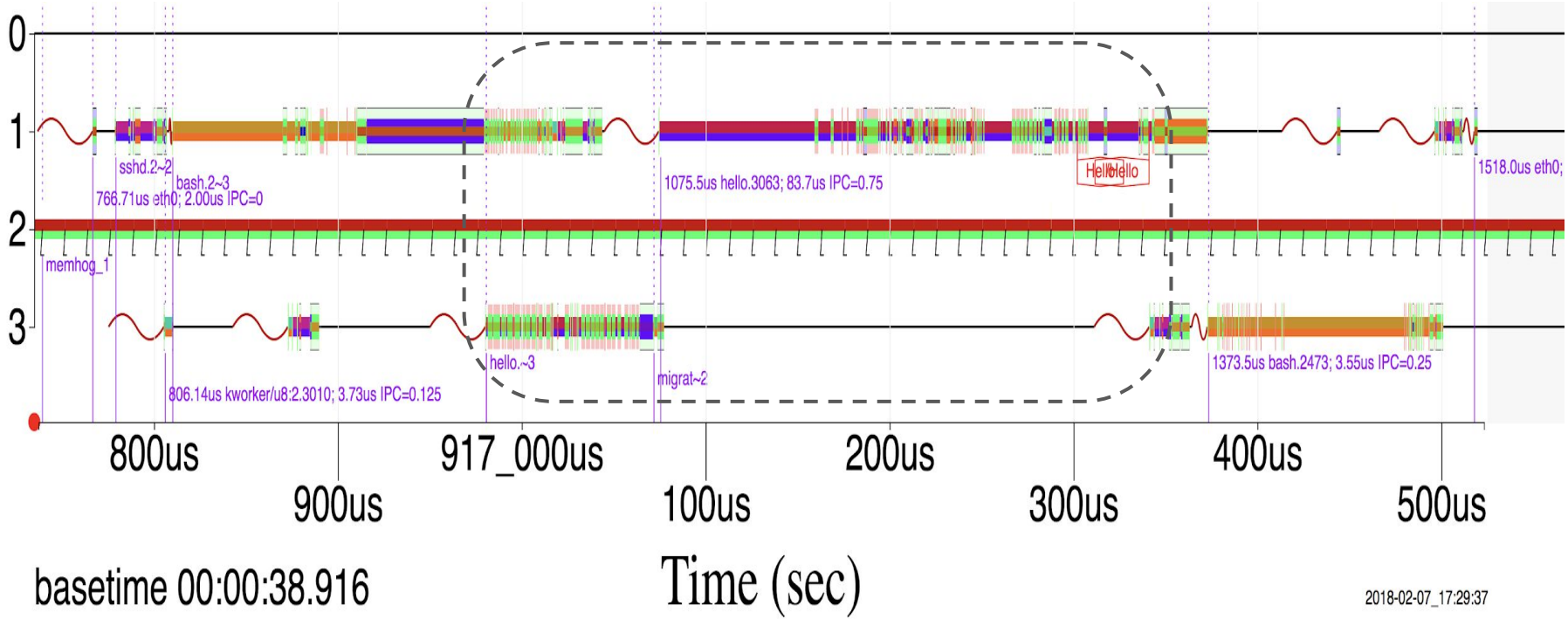
Hello world user-mode **all**, 350 usec



Hello world user-mode **plus kernel-mode**



Hello world **plus other programs**, 1300 usec



Example 2 schedulers

Linux CPU Scheduler Dynamics

CFS completely fair scheduler
  Run each task at equal speed each getting num_CPUs / #num_ running speed

FIFO real-time first-in first out
  Run each task in FIFO order to completion or until it blocks

RR round-robin
  Run like FIFO but with maximum time quantum; round-robin at quantum boundaries

Program: On a four-core machine, **run 1..12 CPU-bound threads** that checksum 240KB (~L2 cache) repeatedly

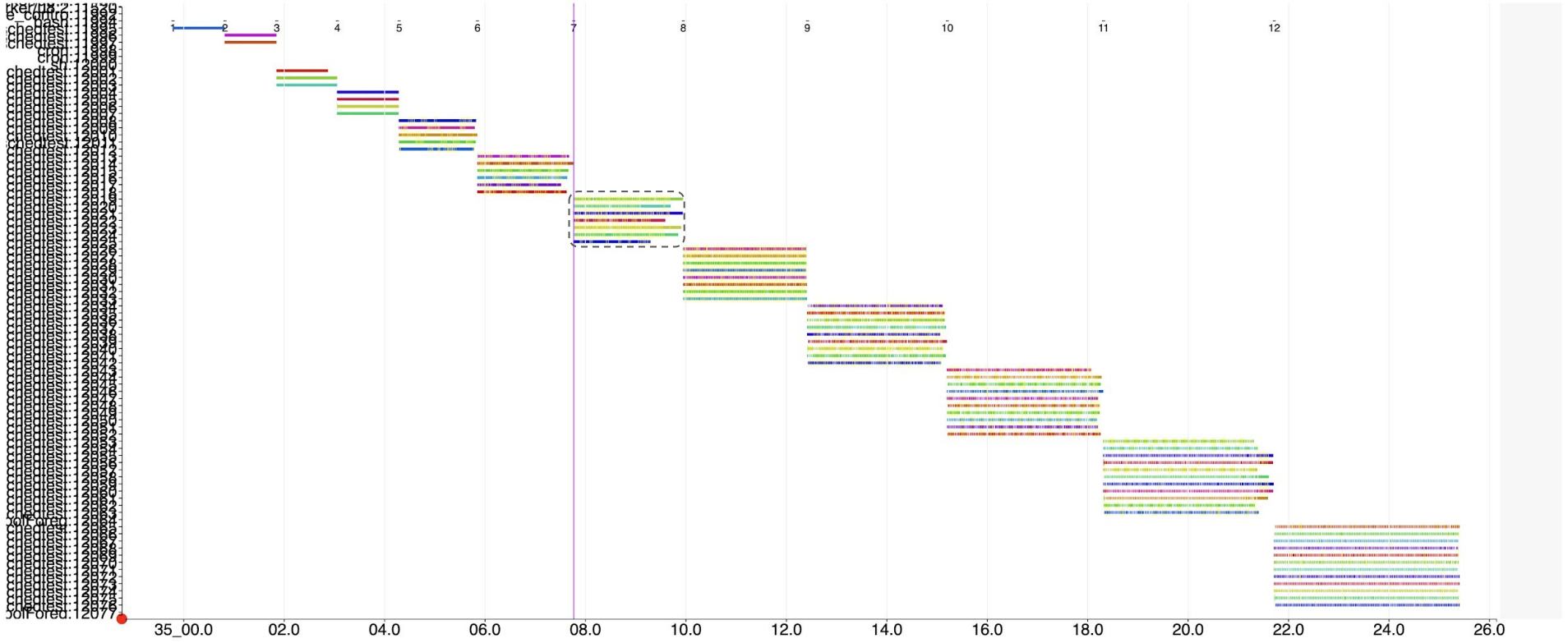
Inbox (15,147) - dick.sites@ | Google News - Search | Google Calendar - February | My Drive - Google Drive | Nvidia HelloTalk - Google Sli | schedtest_cfs_pc.html | schedtest_fifo_pc.html | schedtest_rr_pc.html | sched: overview of CPU sch

File | /Users/dicksites/schedtest_cfs_pc.html

Apps | SVG Crowbar 2 | Setup - Google Docs | cld2 - Compact La... | Sockets Tutorial | OpenSSH Server | Kernel modules - A... | www.tldp.org/LDP/... | Desktop 3rd Gen In... | Comics | ArcaMax... | Common Trace For... | Developing for Perf... | Custom Photo Mug... | try_cpu.html

Annot: User All Option: IPC Mark Arc Wait PC CB Search: ! clone usec: ..

Sched CFS



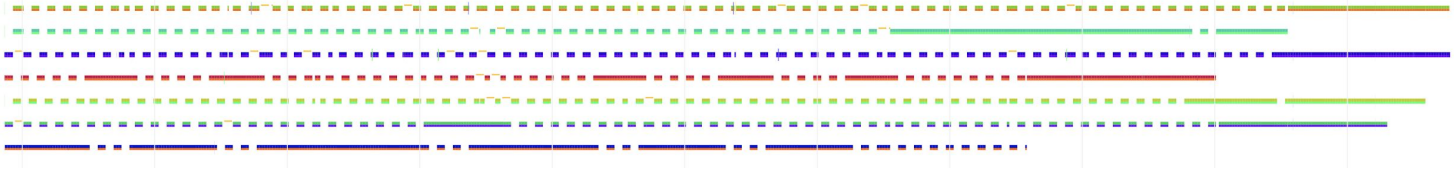
2020-02-04 11:34:00

Time (sec)

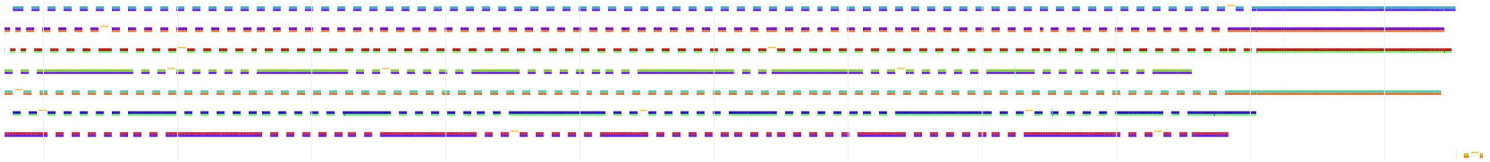
↕ ① ② ③ ④ ⑤ Shift-click 1-5 to save, click to restore. Axes: scroll wheel to zoom, drag to pan. Items: shift-click-unclick to annotate. Shift-click-unshift to keep, shift-drag to measure. Red dot resets. [more]

Looking just at 7 threads on 4 cores, 2.18 seconds

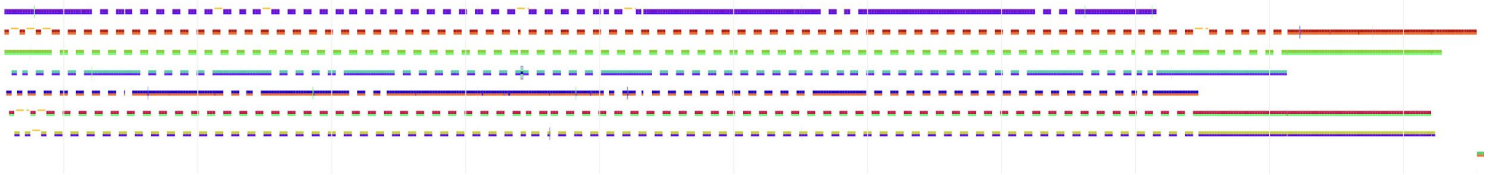
CFS scheduler variation: 30%



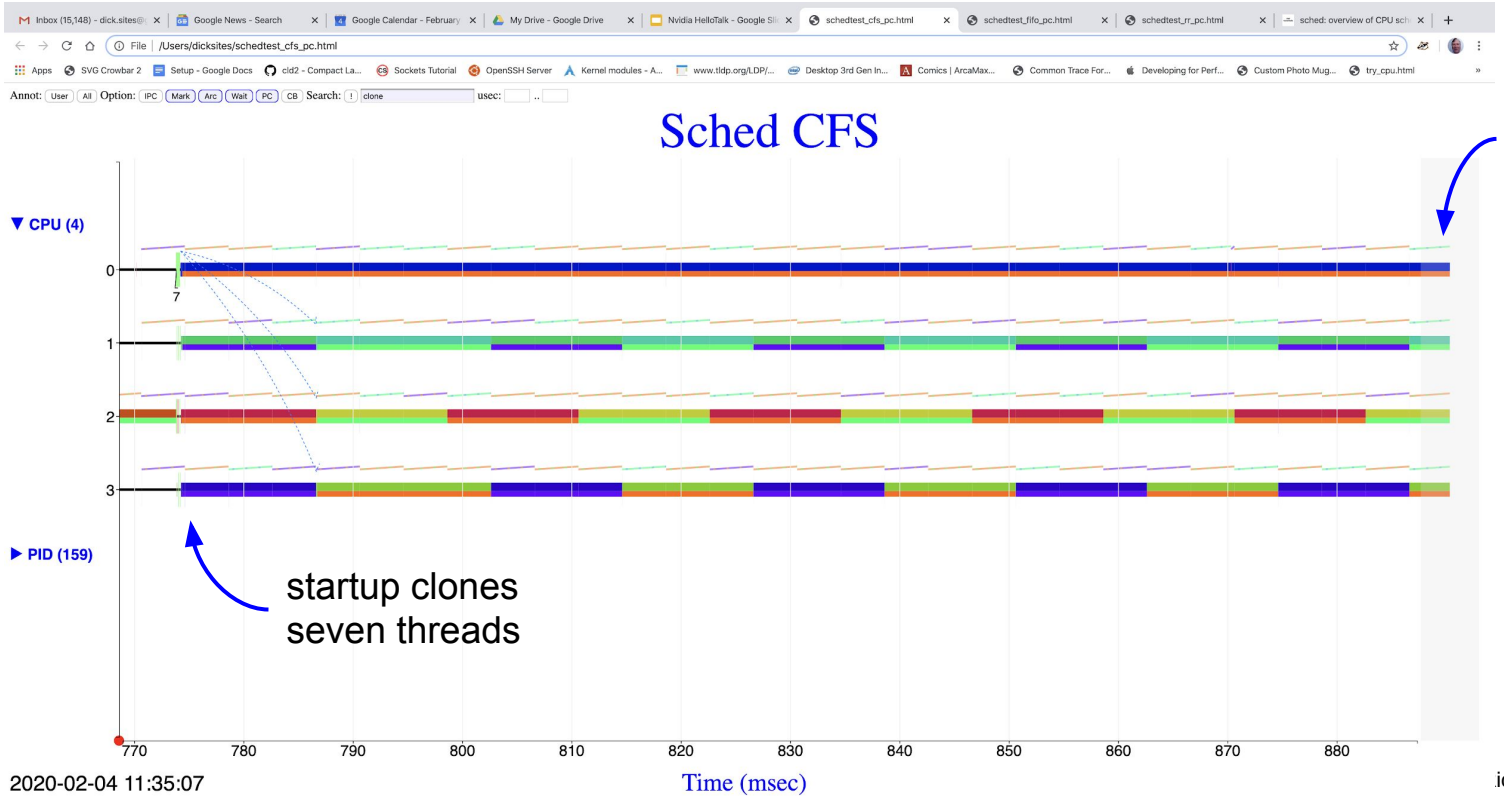
FIFO scheduler 18%



RR scheduler 22%

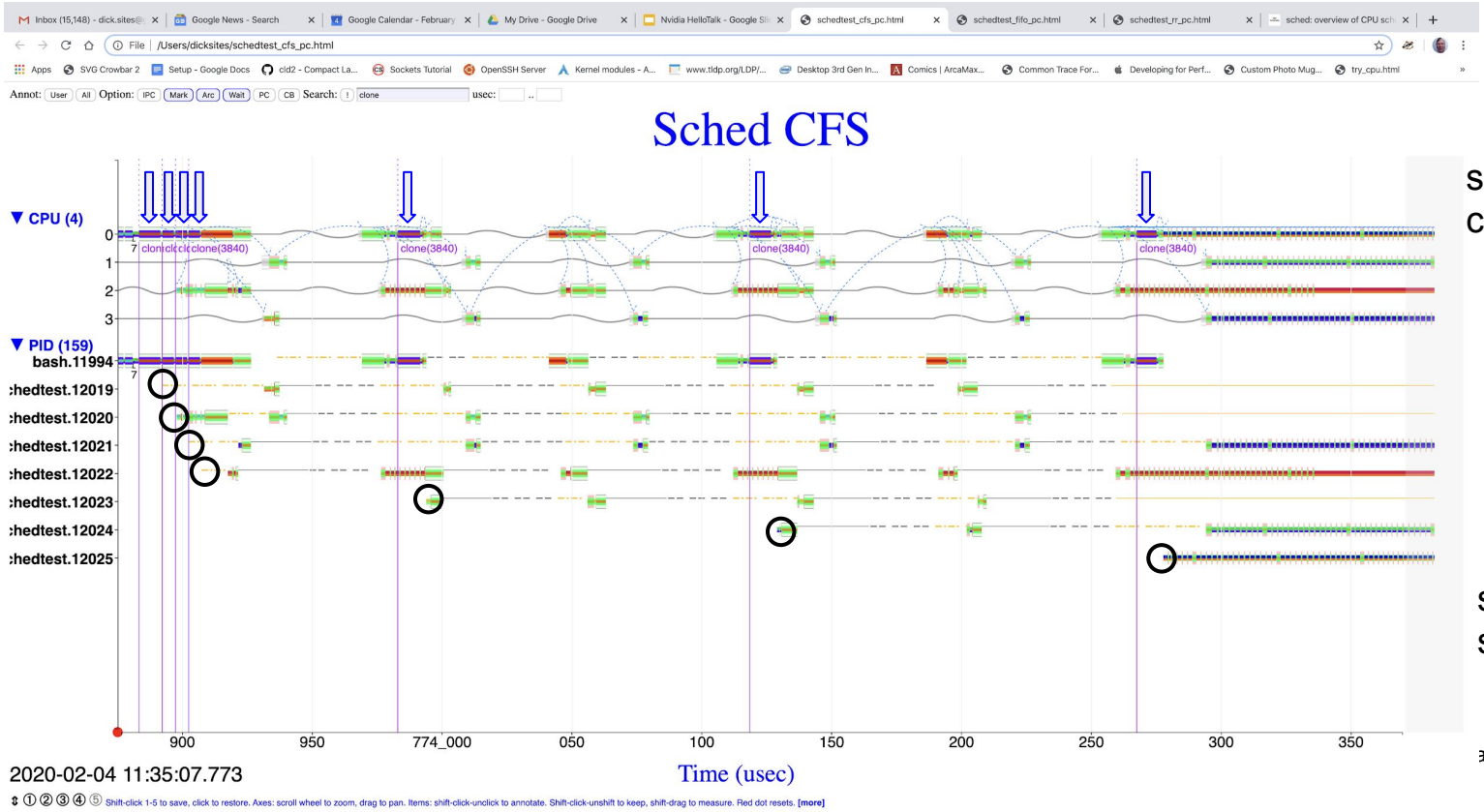


Startup cloning threads, 120 msec



Wavy lines are PC samples at every 4 msec timer interrupt

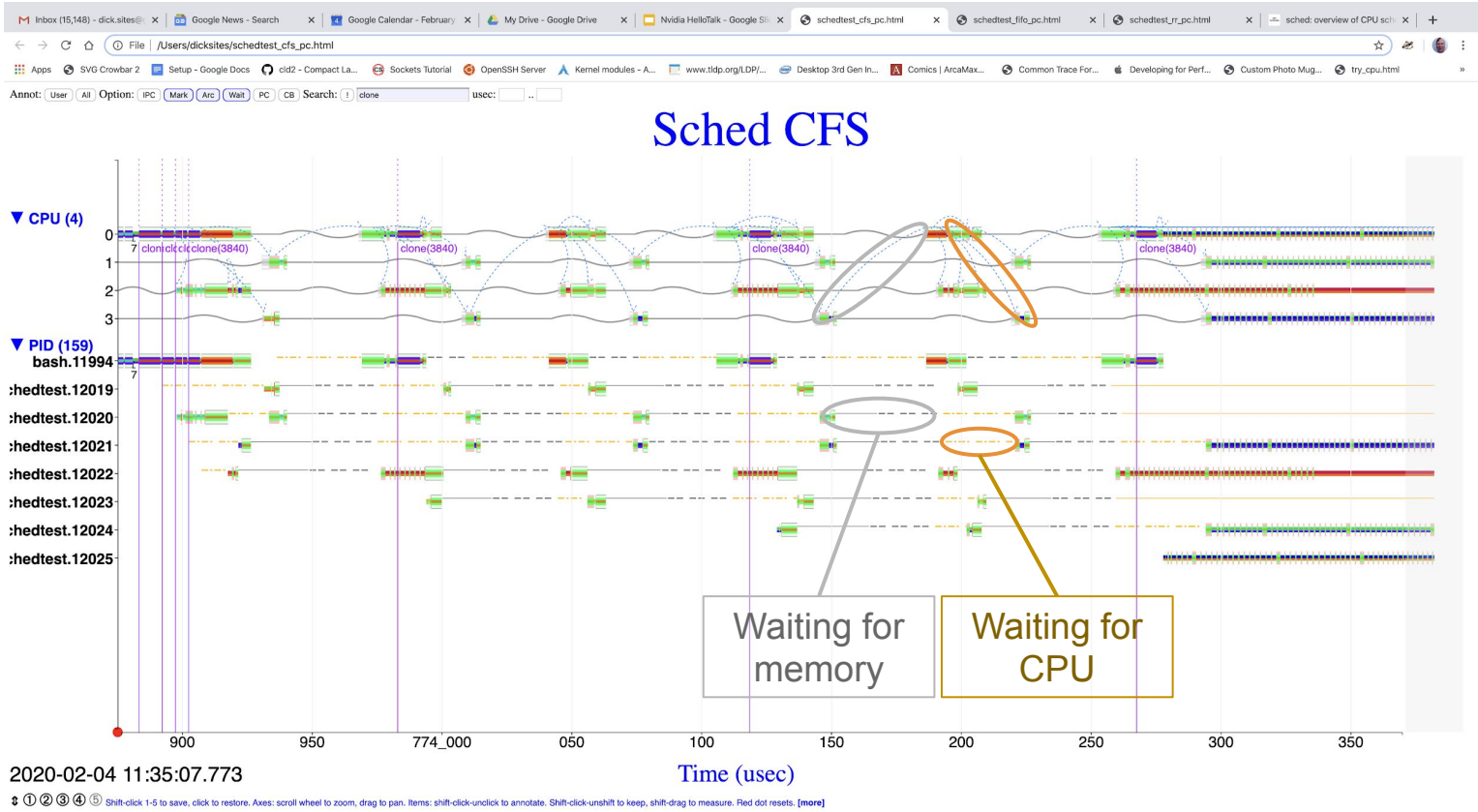
Startup cloning threads, 0.5 msec



seven clone() calls

seven thread starts

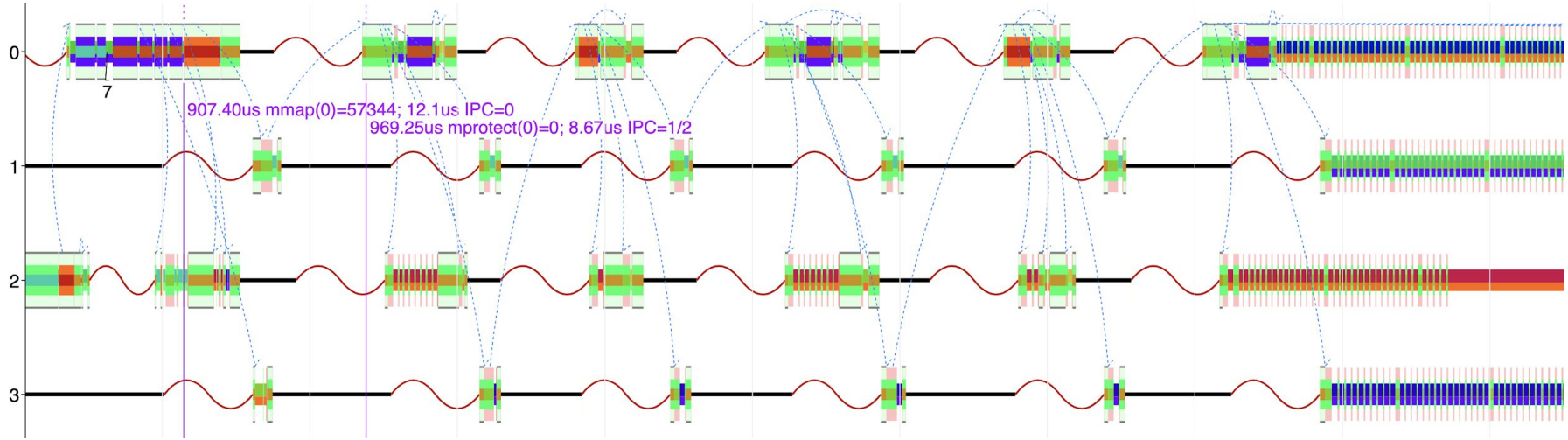
Startup cloning threads, 0.5 msec



What are those dashed lines?

Reason each thread is blocked

Startup cloning threads, 320 usec

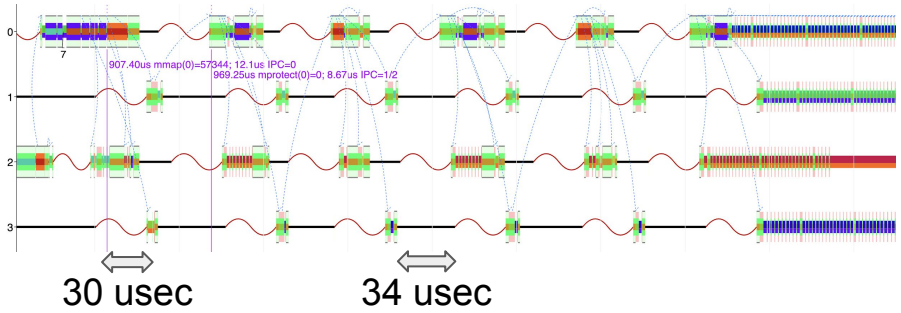


On CPU 0 top left, bash clones threads with **lazy sharing** of pages, then **copy-on-write** later. It does mmap() at the left, but started threads take c-o-w page faults and wait for bash to do mprotect, bouncing back and forth.

Startup cloning threads, 320 usec

Why so much idle time? (382us)

What are those sine waves?(540us)



If we got rid of those, **3x faster** startup

Time to understand the dynamics...

But wait! There's more

Notation: power saving deep sleep (C6 state)

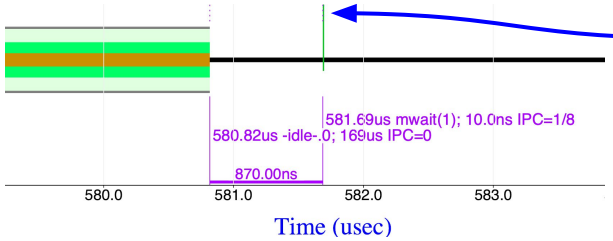
clock/voltage

cache

Notation: power saving deep sleep (C6 state)

clock/voltage

cache

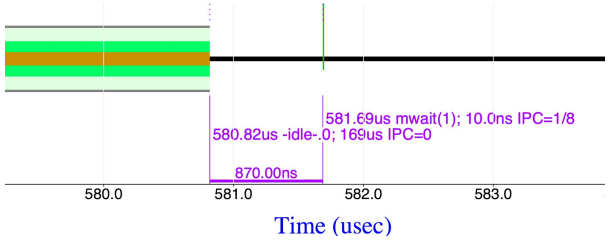


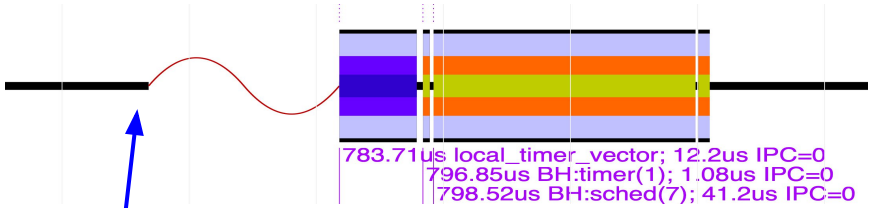
mwait instruction triggers sleep,
in this case, **issued 870 nsec (!) into idle**

Notation: power saving deep sleep (C6 state)

clock/voltage

cache





interrupt triggers wakeup;
sine wave shows wakeup time
in this case, **~30 usec (!)**

Pattern -- Lesson

1. Cloning spawns threads to run on different idle CPUs, **30 usec** to wake up
2. Threads block almost immediately in their first page fault: wait for bash.
3. **500-800 nsec** later, thread CPUs go into deep sleep
4. bash takes **30 usec** to wake up
5. repeat about every 50 usec across the four CPUs

Linux FLAW: If it takes time T to come out of some state, wait ~T before going into it. Then you or no worse than 50% of the optimal time if you knew the future.

Doing so would avoid deep sleep here and be be 3x faster.

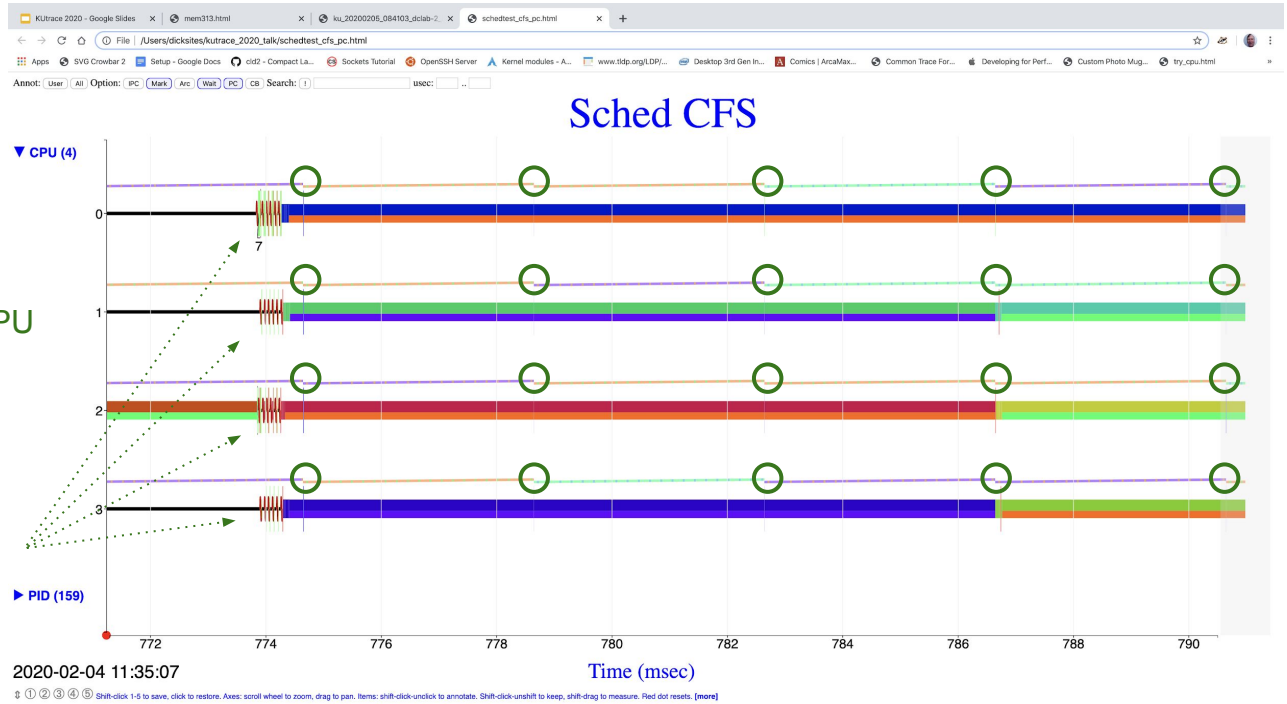
Paucity of PC samples vs. KUtrace spans

20 msec across

only five timer interrupts per CPU (250 Hz)

PC samples: 20 total

(startup cloning missing entirely)



20 msec across

KUtrace spans: 2133 total

(no nanosecond missing)

Example 3 client-server database

Client A sends database RPCS to server B

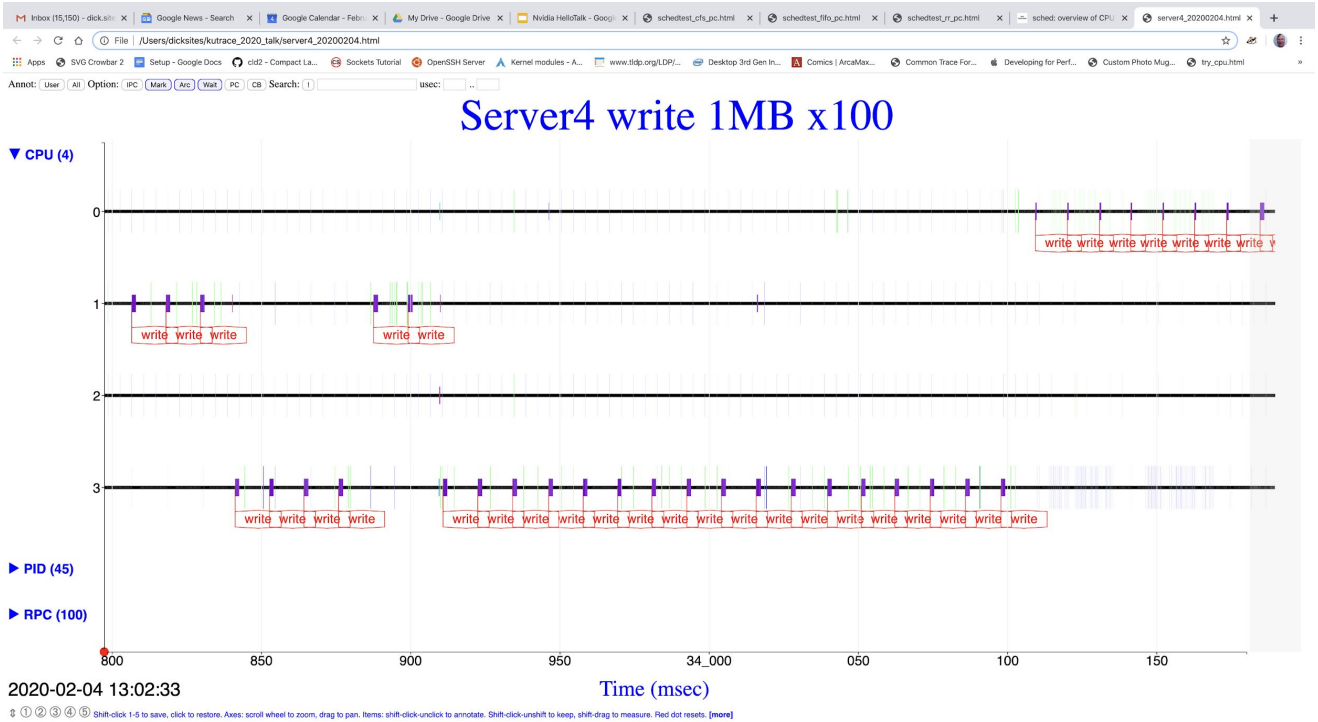
A sends 100 writes of 1,000,000 bytes each

Observations:

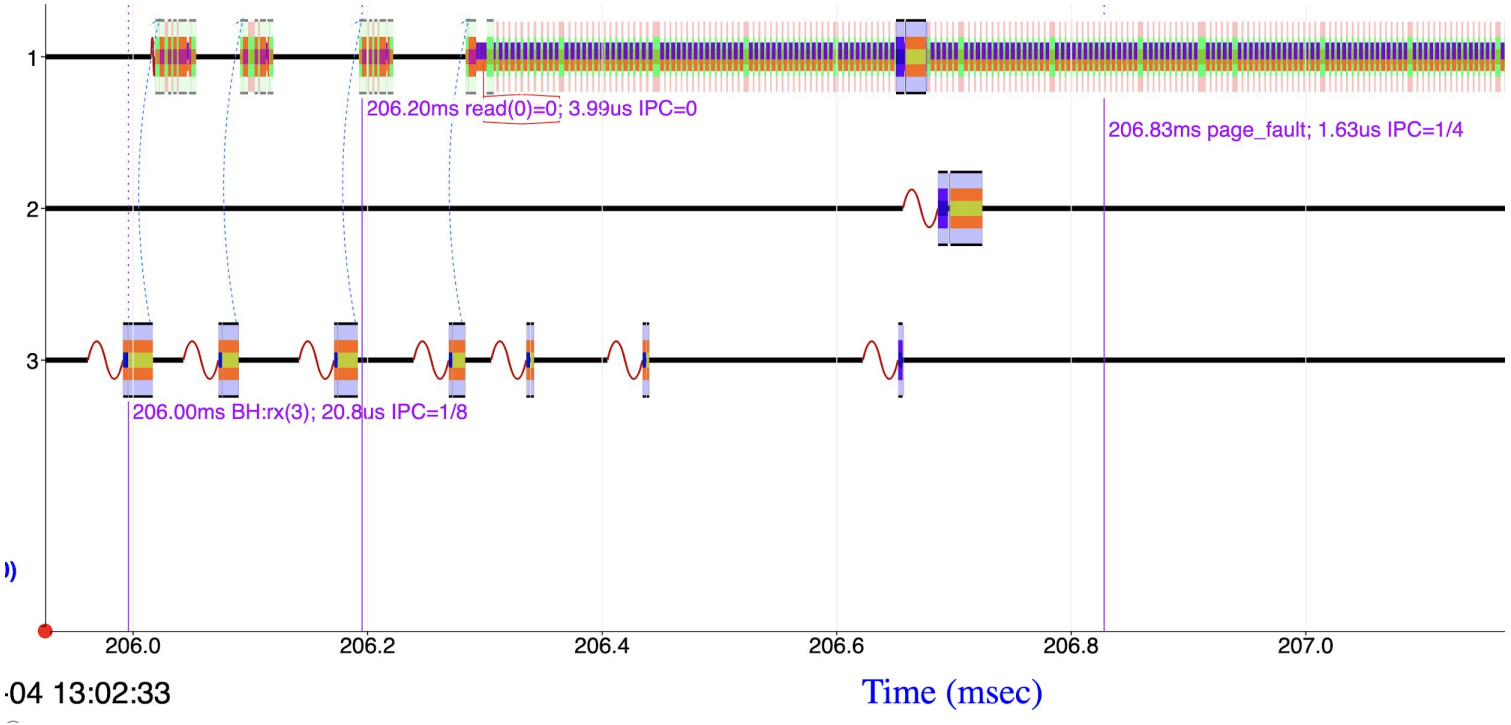
- Client and server both report about 85 transactions per second, about **11.5 msec** each
- The server reports that each transaction takes **1.5 msec user** CPU time, **2.8 msec elapsed** each
- The server is **97% idle**

What is going on?

A few transactions on the server, ~11.5 msec spacing



Ah, network interrupts on left, page faults on right



Ah, network interrupts on left, page faults on right

Transmitting 1,000,000 bytes on the 1 Gb/s network takes about 8 msec of the 11.5.

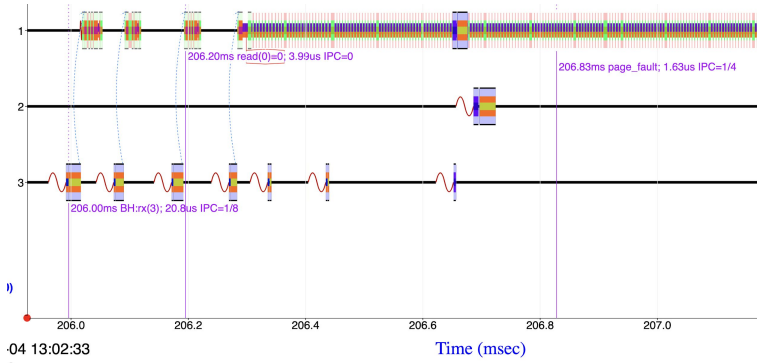
- ⇨ Nothing to change here

The client takes about 1.5 msec after finishing one write before starting the second one.

- ⇨ Client could be improved

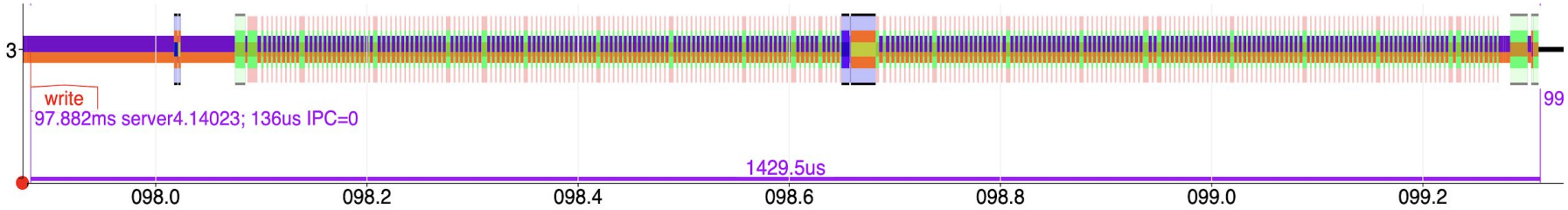
The database program on B reads a few header bytes of the RPC message to find the data length, mallocs 1,000,000 bytes of buffer, reads of the network into it, taking 245 page faults to zero the buffer pages before using them.

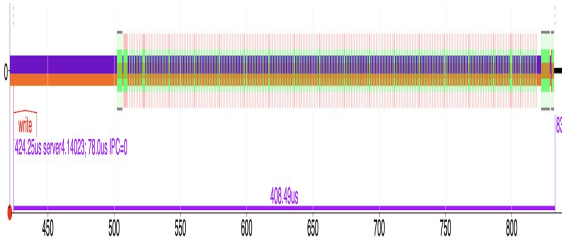
- ⇨ Static buffer alloc fixes



But wait! There's more

Some transactions are 3x faster than others

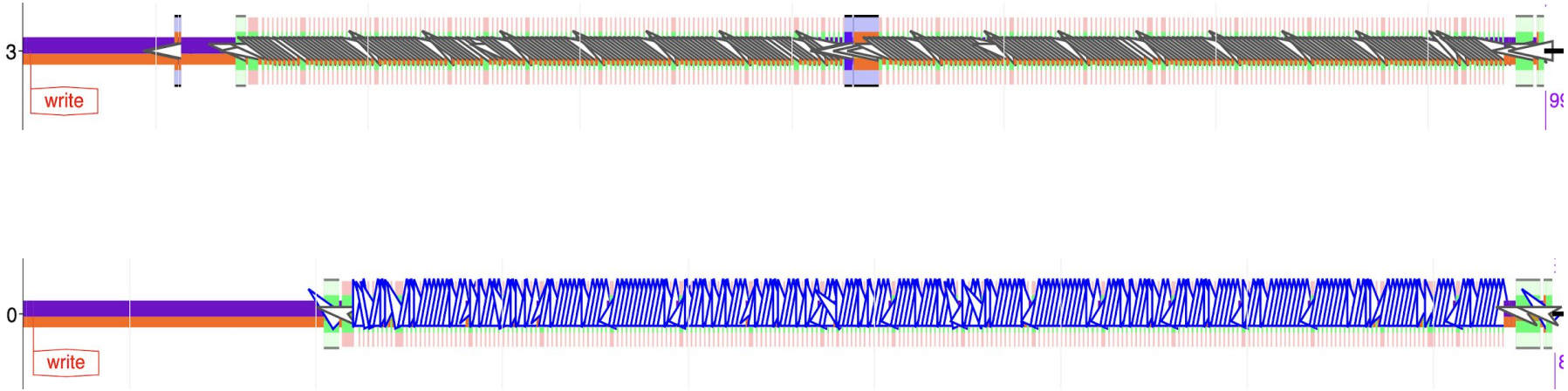




Top: 1429 usec. Bottom: 408 usec

why?

3x difference in IPC --



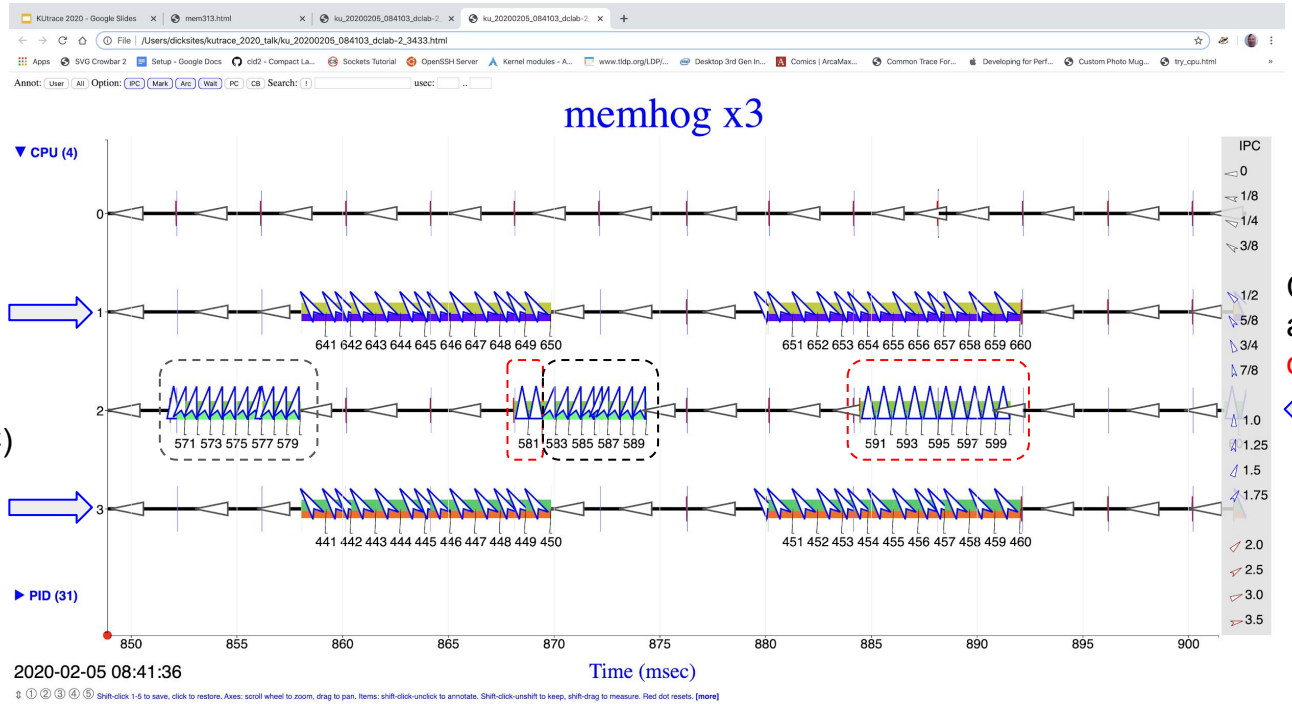
Top: 1/4 to 1/2 instructions/cycle. Bottom: 7/8 to 1 instructions/cycle
(I don't know **why)**

Example 4 interference

Three copies of L3 cache sweep memory hog

CPU 1 & 3 are hyperthreads

overlap: 5/8 IPC
(= total 1.25 IPC)

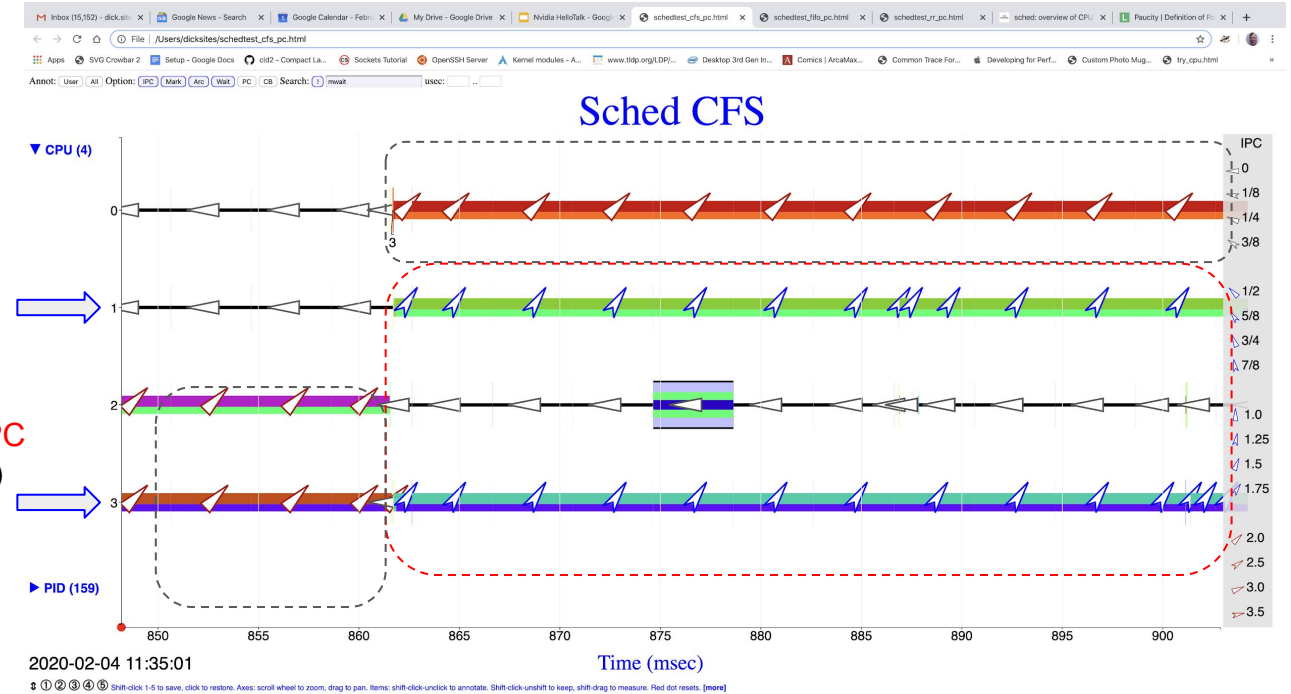


CPU 2
alone: 1.25 IPC
overlap: 1.0 IPC

Three L2-sweep threads (hyperthreads share an L2)

CPU 1 & 3 are hyperthreads

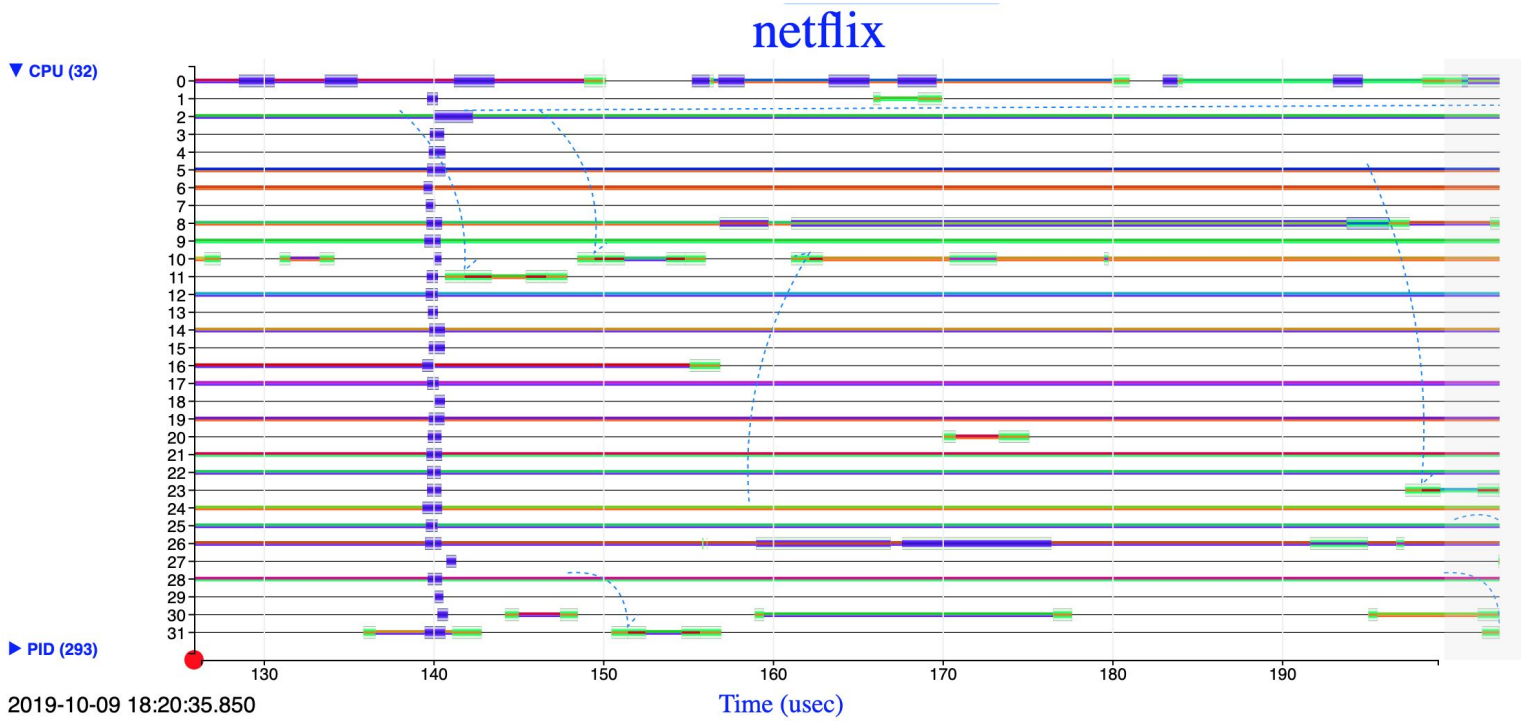
alone: 2.0 IPC
overlap: 1.75 IPC
(= total 3.5 IPC)



CPU 0 & 2 are hyperthreads

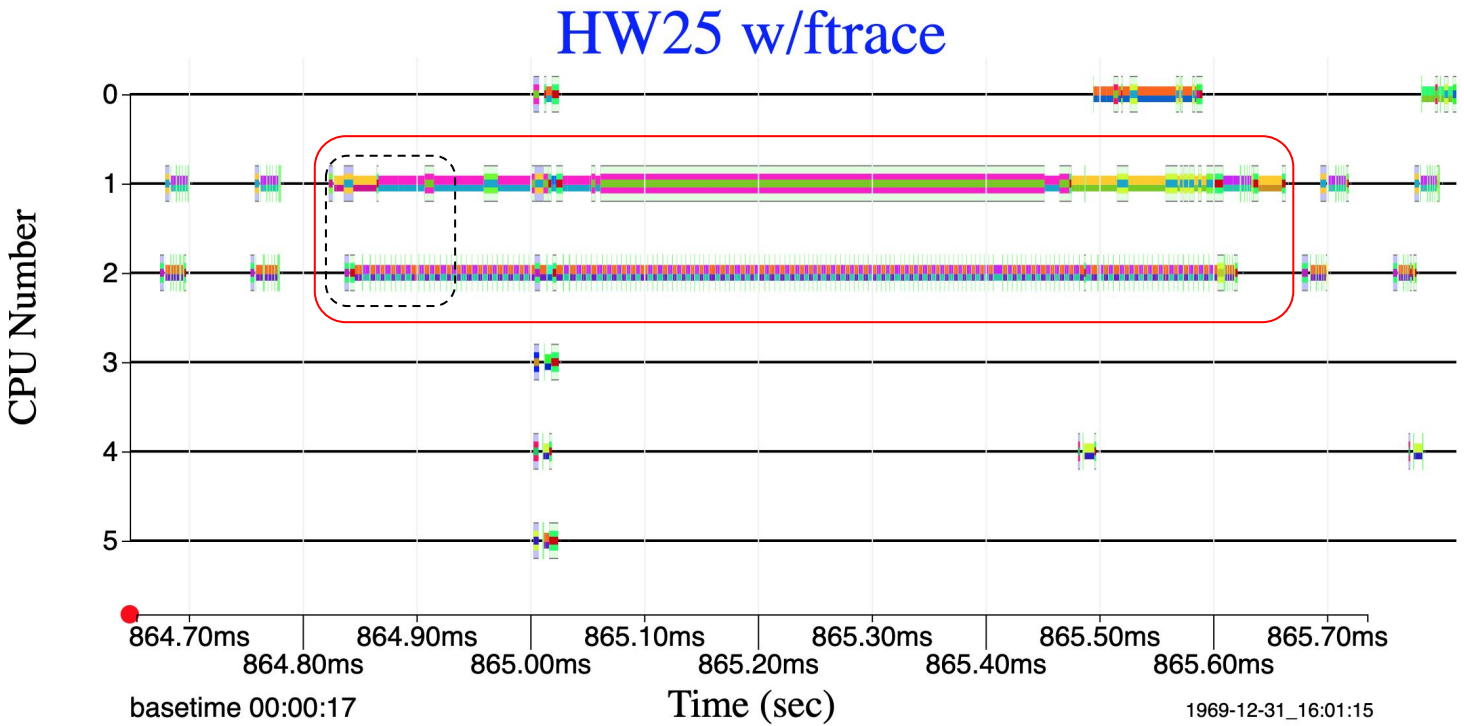
Experimental examples

Early FreeBSD on 32 cores, 75 usec across, blue timer interrupts



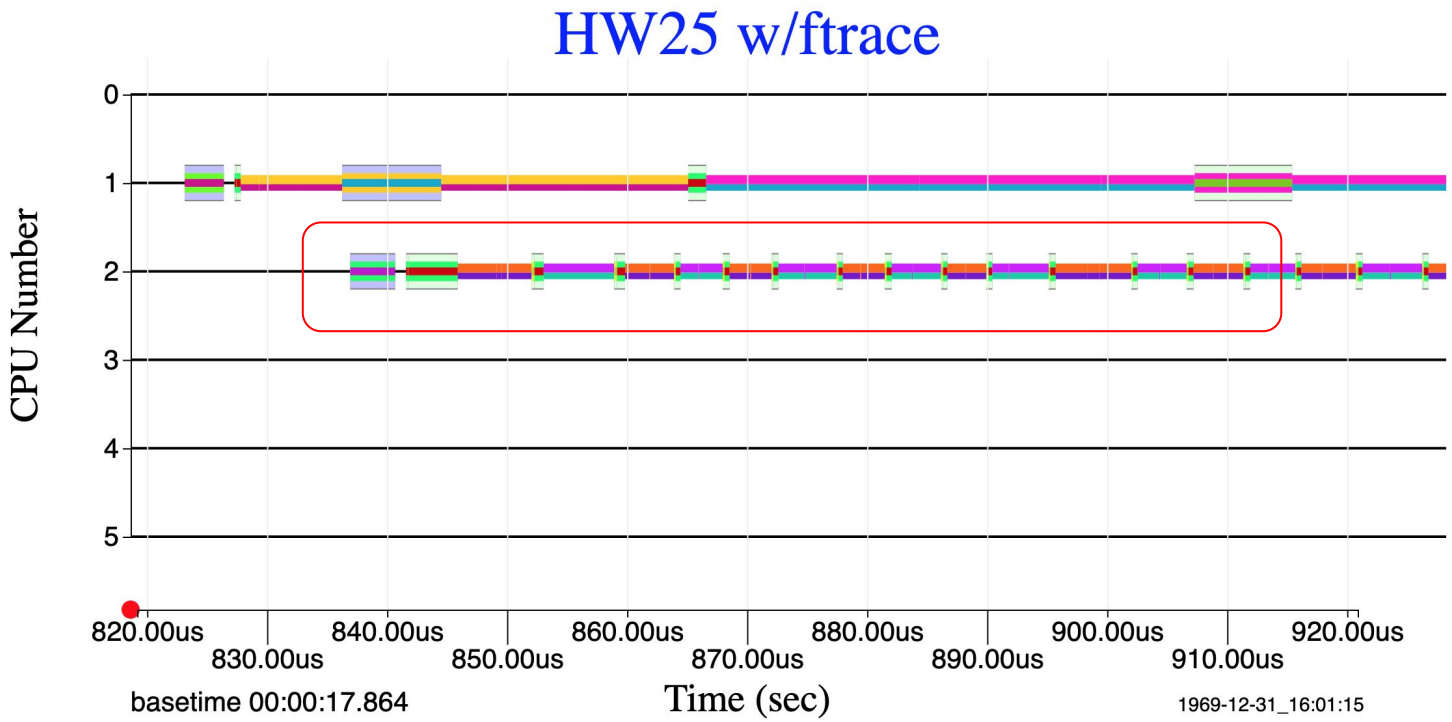
credit: Drew Gallatin, Netflix

Real-time ARM port, 1 msec across, unexpected dynamics



credit: Dick Sites, Tesla

ARM 100 usec across; constant sched_yield, no progress



Contributions

Only **tracing** for several minutes of everything that is happening on a computer can catch *unpredictable* slow requests. If you know ahead of time which requests will be slow, simpler methods could suffice. But you don't.

Tracing only works *in situ* for real-time live traffic when its overhead is extremely small

The tool: KUtrace observes and shows **why**

Execution

- All CPU cores
- All processes
- All user-mode execution
- All kernel-mode execution
- All programs, unmodified, any language

Waiting

- All reasons for **not** executing
- All interactions between processes: block/wakeup dynamics

Execution, but slowly

- Cross-program interference via IPC

The tool: KUtrace is practical

Overhead

- Kernel patched but not tracing: unmeasurable overhead
- Active tracing: 0.25% CPU overhead; ~1% memory overhead for trace buffer
- Active tracing with IPC: 0.75% CPU overhead; ~1.12% memory overhead

Nothing-missing philosophy

Limitations

- Only addresses performance issues
- Not useful for kernel- or user-mode program debugging
- Only implemented for Linux

Linux 4.19 and postprocessing sources are in github

The observations

A drill-down of hello world:
   user main program, + all user, +kernel, +other programs

Linux scheduler dynamics -- not "completely fair"
Thread-spawning dynamics with idle/sleep delays [Linux sleeps too soon]

Transaction dynamics: network time, server user and kernel time, client next-request time; together they explain the total time [buffer malloc hurts]

Cross-program interference *directly* observed

What is different

KUtrace is the **only** tracing tool that is fast enough to run ***in situ*** with live traffic -- about 40 CPU cycles (12.5 nsec) per event

Traces process wakeups: the kernel routine doing so = reason for **each wait,** including disk, network, software locks, and interprocess-signals

Traces instructions per cycle IPC at microsecond scale: shows instruction-slowdown **interference** between CPUs/programs

The **total** trace of any request, executing & waiting & executing slowly, shows **exactly why** it is slow

What is different

A simple library can add user-supplied markers to traces, to identify major code chunks; these are so cheap they can be left in production code

Tracing includes PC samples at timer interrupts, so it profiles at millisecond scale long execution stretches that have no kernel/user transitions

Postprocessing produces dynamic HTML/SVG displays of everything that happened; you can pan and zoom from minutes to nanosecond scale

Futures (helpers welcome)

Complete: trace RPC message header arrivals (to disambiguate client-server "transmission" delays)A port to ARM (Raspberry Pi 4)

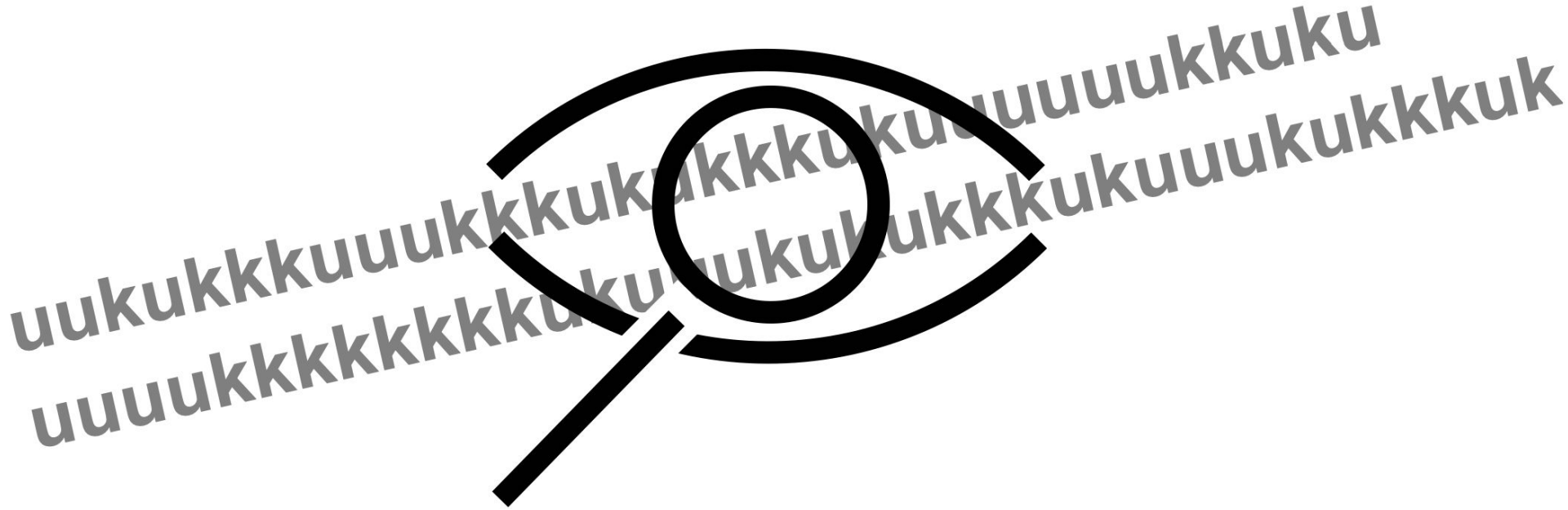
Complete: finish tracing contended lock names and source line numbers

Complete: turn PC samples into proper profiles with source names/lines

Other Linux versions than 4.19

Other operating systems

Questions?


uukukkkuuukkkukukkkkukuuuuukkkuku
uuuukkkkkkkkkukuuukukkkkukuuuukukkkuk

References

References

[1] ACM AQueue Magazine article, Oct 2018
https://queue.acm.org/detail.cfm?id=3291278

[2] Tracing Summit talk in Prague, Oct 2017
https://tracingsummit.org/wiki/TracingSummit2017 the 11:00am talk

[3] Open-source code
https://github.com/dicksites/KUtrace