KUTrace: Where have all the nanoseconds gone?

Richard Sites, Invited Professor EPFL
2017.10.27

Tracing Summit 2017, Prague
Outline

Observability problem statement
KUTrace solution
Tracing design goals
Goals drive the design
Results
Display Design
Comparisons
Conclusions
Observability problem statement
Problem statement

This talk is about tail latency in real user-facing datacenter transactions. It is not about batch processing throughput, nor about benchmarks.

Context: A datacenter of perhaps 20,000 servers running software services that spread work for a user-facing transaction across a few hundred or thousand machines in parallel. Each server handles hundreds of transactions per second.

Some transactions are unusually slow, but not repeatably. Slow transactions occur unpredictably, but there are several per minute.

We wish to observe where all the time goes in such transactions, and observe why they are slow.
Problem statement

☞ Some transactions are unusually slow, but not repeatably.

∴ There is some source of interference just before or during a slow transaction.

Understanding tail latency requires **complete traces** of CPU events over a few minutes, with small enough CPU and memory overhead to be usable under busiest-hour live load.

Existing tracing tools have much-too-high overhead.

**Problem: build better tail-latency observation tools**
KUTrace solution
KUTrace solution

KUTrace uses minimal Linux kernel patches on a single server to trace every transition between kernel- and user-mode execution, on every CPU core, with small enough overhead to use routinely on live loads. Postprocessing does the rest.

KUTrace traces all executables unmodified

Why K/U transitions? The data gathered is sufficient to identify where 100% of the CPU time goes, and also to observe:

- All kernel, unrelated-process, and cross-thread interference
- All reasons for waiting: disk, network, CPU, timer, software locks
KUTrace solution

Note that KUTrace is a one-trick pony -- it does one thing well and does not do anything else. It shows the *entire* server CPU dynamics for a minute or so.

- No fine-grained user-mode per-routine timing
- No fine-grained kernel-mode per-routine timing
- No user-mode debugging
- No kernel-mode debugging
- No interpretive language
- No subsetting
- No sampling

Programs/transactions being observed are assumed to be normally fast but with unexplained long tail latency
KUTrace solution

For hundreds to thousands of transactions per second, each normally taking about 1 msec to 1 second on a server, KUTrace is able to observe the 99th percentile slow transactions and their surrounding context whenever they occur.

The "interesting" slow transactions are fast if run again, and typically only occur on live load during the busiest hour of the day.

Their slowness is entirely related to some unknown interference, which is why their dynamics can only be observed in live load.

Once the true dynamics are observed, fixes often take only 20 minutes.
Tracing design goals
Design goals

Record every kernel-user-mode transition, with nothing missing

Less than 1% CPU overhead

Less than 1% memory overhead

For 30-120 seconds

On user-facing live load during the busiest hour of the day -- about 200,000 transitions per CPU core per second
syscall, interrupt, trap, context switch

User code

Linux Kernel

trace mod

trace buffer in kernel RAM

post-proc.
Goals drive the design
Goals drive the design, CPU overhead

Less than 1% CPU overhead, about 200,000 transitions per CPU core per second
=> Record to RAM; nothing else is fast enough
=> Each CPU core must write to its own buffer block to avoid cache thrashing

200,000 transitions = one every 5 usec. 1% overhead = 50 nsec budget
50 nsec ~= one cache miss to RAM, so

=> Size of each trace entry must be much less than a cache line; 4, 8 and 16 bytes are the only realistic choices.
Goals drive the design, memory overhead

Less than 1% memory overhead, for 30-120 seconds
For a server with 256 GB of main memory, 1% is ~2.5 GB

For N bytes per trace entry and 24 CPU cores and 200K transitions per second, 60 seconds needs

\[ 24 \times 200K \times 60 \times N \text{ bytes} \]
\[ = 288M \times N \text{ bytes} \]

=> This implies that N <= 9. I chose four bytes.
Goals drive the design, memory overhead

At four bytes each, a trace entry has room for

20-bit timestamp
12-bit event number

<table>
<thead>
<tr>
<th></th>
<th>timestamp</th>
<th>event</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td></td>
<td>12</td>
</tr>
</tbody>
</table>

timestamp: cycle counter shifted 6 bits = ~20ns resolution, 20ms wrap
must guarantee at least one event per 20ms to reconstruct full time -- timer IRQ

event: ~300 system calls, 300 returns, 300 32-bit system calls, 300 returns,
12 interrupts, 12 faults, a few others
Ross Biro observed in 2006 that it is particularly useful to have some bits of the first parameter of any syscall, and to have the matching return value. To do so, we put call + return into a single 8-byte entry:

- 20-bit timestamp
- 12-bit event number
- 8-bit delta-time (call to return)
- 8-bit retval
- 16-bit arg0

As a useful side-effect, this makes trace-entry recording substantially faster. (Occasionally, if delta or retval does not fit in 8 bits, or if some event is inbetween, two entries are used.)
Goals drive the design

Modest kernel patches to capture each event. `apic.c entire patch`

```c
... 
if (dclab_tracing)
(*dclab_global_ops.dclab_trace_1)(DCLAB_TRACE_IRQ + kTimer, 0);

    local_apic_timer_interrupt();

if (dclab_tracing)
(*dclab_global_ops.dclab_trace_1)(DCLAB_TRACE_IRQRET + kTimer, 0);

    exiting_irq();
    set_irq_regs(old_regs);

... 
```
Goals drive the design -- speed even with preemptable kernel

Normal path for making one entry, \(\sim 40\text{cy}\)

```c
void trace_1(u64 num, u64 arg) {
    if (!dclab_tracing) {return;}
    Insert1((num << 32) | arg);
}

u64 Insert1(u64 arg1) {
    u64 now = get_cycles();
    u64* claim = GetClaim(1);
    if (claim != NULL) {
        claim[0] = arg1 | ((now >> RDTSC_SHIFT) << 44);
        return 1;
    }
}
```

```c
u64* GetClaim(int len) {
    tb = &get_cpu_var(dclab_traceblock_per_cpu);
    nexti = atomic64_read(&tb->next);
    limiti = tb->limit;
    if (nexti < limiti) {
        u64* myclaim = (atomic64_add_return(
            len * sizeof(u64), &tb->next)) - len;
        if (myclaim < limiti) {
            put_cpu_var(dclab_traceblock_per_cpu);
            return myclaim;
        }
    }
    ...
Tracing Design results
Tracing Design results

==> 50 nsec trace-entry budget

Actual is 4x better:
~12.5 nsec and four bytes per transition

So 25 nsec and 8 bytes per syscall/return or interrupt/return or fault/return pair

1/4 of 1% CPU overhead, 1/4 of 1% RAM overhead for 30-60 seconds of trace
Tracing Design results, full system

Linux loadable module, reserve kernel ram, patches to look at trace bit, call module routines.

Routines insert trace items, occasionally allocate new trace blocks

Control interface to start, stop, extract completed trace
User-mode library to optionally insert human-readable markers

Postprocessing to turn transition points into time-span durations
Postprocessing to turn into pictures
Tracing Design results

Postprocessing 1
- Takes raw binary trace file of transitions and creates time-spans
- Takes time-spans and names embedded in trace (process, syscall, etc.) and expands to have name for most spans
- Propagates current CPU#, process id#, RPCID# to every span
- Writes .json file

System sort of .json file by start times
Tracing Design results

Postprocessing 2
  Takes small HTML file plus wonderful d3 javascript library plus .json file
  and displays picture of every CPU core every nanosecond

Users can pan/zoom/label as desired

Shows all threads processing our slow transaction
Shows interference from other threads
Shows not-processing wait time

Can easily time-align across multiple machines
time(sec) C Ev name
28.9995927 0  801  syswrite
28.9996293 0  a01  return, 36.6us
28.9996302 1  0000  -idle-
28.9996333 0  80c  sysbrk
28.9996480 1  5d1  eth0
28.9996764 1  5d1  eth0
28.9997007 0  a0c  ret brk, 67.4us
28.9997015 0  823  nanosleep
28.9997038 0  0000  -idle-
28.9997504 0  5ef  local_timer_vector
28.9997534 0  59a5  bash
28.9997540 0  a23  ret nanosleep 52u

C: CPU#  Ev: event#

Postprocessing

Nagle's algorithm delays response by 40 msec

Note: gettimeofday() differs by 16ms
Display Design
Display Design

Goal: Turn time-spans into a complete picture of 30-120 seconds of all CPU cores and what they are doing every nanosecond. Allow pan and zoom.

Result: The .json file derived from each trace has all the time-spans sorted by start time. A modest amount of HTML/javascript plus Mike Bostock's excellent d3.js library provides the mechanism.

But more than ~25,000 spans on screen at once is slow. So the postprocessing allows combining spans to give a time granularity of 1us to 1ms+, and allows picking a subset of the trace time. Using these gives full-resolution display interactions of interest.
Display Design

Goal: Human-meaningful labels

Result: The .json file derived from each trace has human-readable names for all time-spans. These can optionally be displayed for any span or for on-screen groups of spans.

The raw traces have an initial bunch of entries for the names of every syscall, interrupt, fault. Whenever a context switch encounters a new process id, the name of that process is added to the raw trace.

User code may optionally add marker entries to the trace, highlighting areas of interest.
hello world example trace (hello, /hello annotation added for talk)
hello world example trace (hello, /hello annotation added for talk)
hello world example trace (hello, /hello annotation added for talk)
Hello World example trace, main() 80 usec across
hello world example trace, main() 80 usec across

No C6 wakeup delay CPU 1 to 3 because they are two hyperthreads of the same physical core; 3 woke up back when 1 did
SOMETHING YOU HAVE NEVER OBSERVED BEFORE
IPC, instructions per cycle at nsec scale

memhog_1_2_clients IPC

CPU Number

Time (sec)
IPC, instructions per cycle **at nsec scale**

![Graph showing memhog_1_2_clients IPC with instructions per cycle (IPC) levels of ~0.25, ~0.5, ~1, and ~2.](image)

- ~0.25 IPC
- ~0.5 IPC
- ~1 IPC
- ~2 IPC
IPC, instructions per cycle at nsec scale

After syswrite() call, user-mode program IPC slow then increases

~0.25 IPC  ~0.5 IPC  ~1 IPC  ~2 IPC
IPC, instructions per cycle **at nsec scale**

- Page-fault code gets faster at reuse
  - ~0.25 IPC
  - ~0.5 IPC
  - ~1 IPC
  - ~2 IPC
Direct observation of cross-CPU communication

IPI send to resched interrupt 1.2 usec, 90ns IRQ execution

sched code, rcu callback, sched 2nd part faster
Direct observation of cross-thread interference

Top/bot L3 cache sweeps slow each other down

Top L3 cache sweep + bot L1 cache sweep, no slowdown
Comparisons
Comparisons

Yet another tracing facility?
   All 26 exist: atrace, btrace, ctrace, ... ztrace

But few with the global scope and efficiency to address the datacenter environment.

For example, strace[6] has a tracing overhead of about 350x KUtrace
Even tcpdump's [10] best-case CPU overhead of about 7% makes it too slow for the datacenter environment
<table>
<thead>
<tr>
<th>Tool</th>
<th>All programs</th>
<th>System calls</th>
<th>Interrupts &amp; traps</th>
<th>Scripts, or ASCII</th>
<th>getuid() trace ovhd</th>
<th>Time overhead</th>
<th>Space overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>KUtrace</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>25ns</td>
<td>1x</td>
<td>1x</td>
</tr>
<tr>
<td>G.ktrace[14]</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>200ns</td>
<td>8x</td>
<td>12x</td>
</tr>
<tr>
<td>dtrace[1]</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>x</td>
<td></td>
<td>30x</td>
<td></td>
</tr>
<tr>
<td>ftrace[2]</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>x</td>
<td>587ns</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ktrace[3]</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ltrace[4]</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LTTng[5]</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td>449ns</td>
<td>16x</td>
<td>8x</td>
</tr>
<tr>
<td>strace[6]</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td>x</td>
<td></td>
<td>350x</td>
<td></td>
</tr>
<tr>
<td>straceNT[7]</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SystemTap[8]</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>truss[9]</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Comparison notes

An earlier version of KUtrace was built at Google circa 2006 by Ross Biro and the author. David Sharp took over that work and created the second more easily-maintained but slower version. It is in the table above as G.ktrace, and under the name ktrace in [12] and [14], not to be confused with the FreeBSD ktrace[3]. Sharp gives some competitive performance numbers in [13].

Trace systems with scripts or ASCII output have high tracing overhead, so can't be used on live datacenter traffic.

Some facilities compensate for high tracing overhead by selectively disabling of trace points, but that destroys the ability to observe interference mechanisms.
Conclusions
Conclusions

There is a need for something like KUtrace in datacenters
Kernel-user transitions are a good cutpoint: not too much, not too little data
Careful engineering is necessary to make tracing fast/small
A little extra information, especially human-meaningful names, is important/cheap
Having a good display mechanism makes traces useful
References

References


Additional References

Dapper, a Large-Scale Distributed Systems Tracing Infrastructure, Benjamin H. Sigelman, et. al., 2010
https://research.google.com/pubs/pub36356.html

Reduce tracing payload size, David Sharp <dhsharp@google.com>, 2010
https://lwn.net/Articles/418709/


http://www.morganclaypool.com/doi/pdf/10.2200/S00516ED2V01Y201306CAC024
Questions?

Enjoy.