Making the Invisible Visible

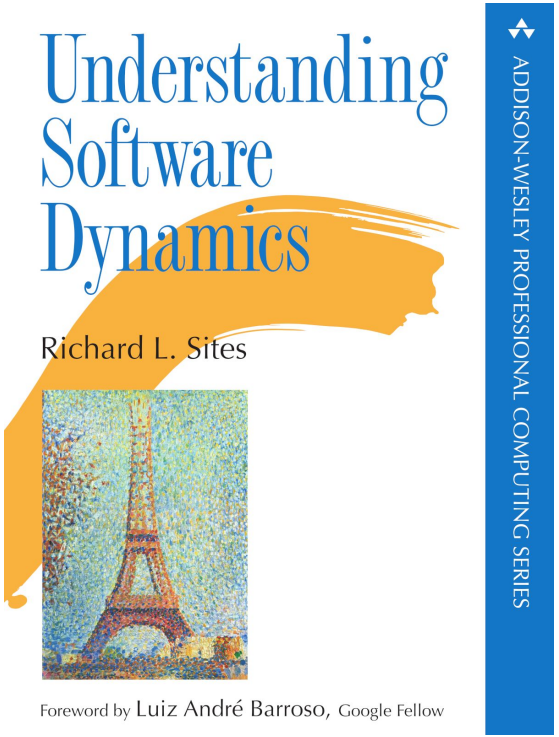
Observing Complex Software Dynamics

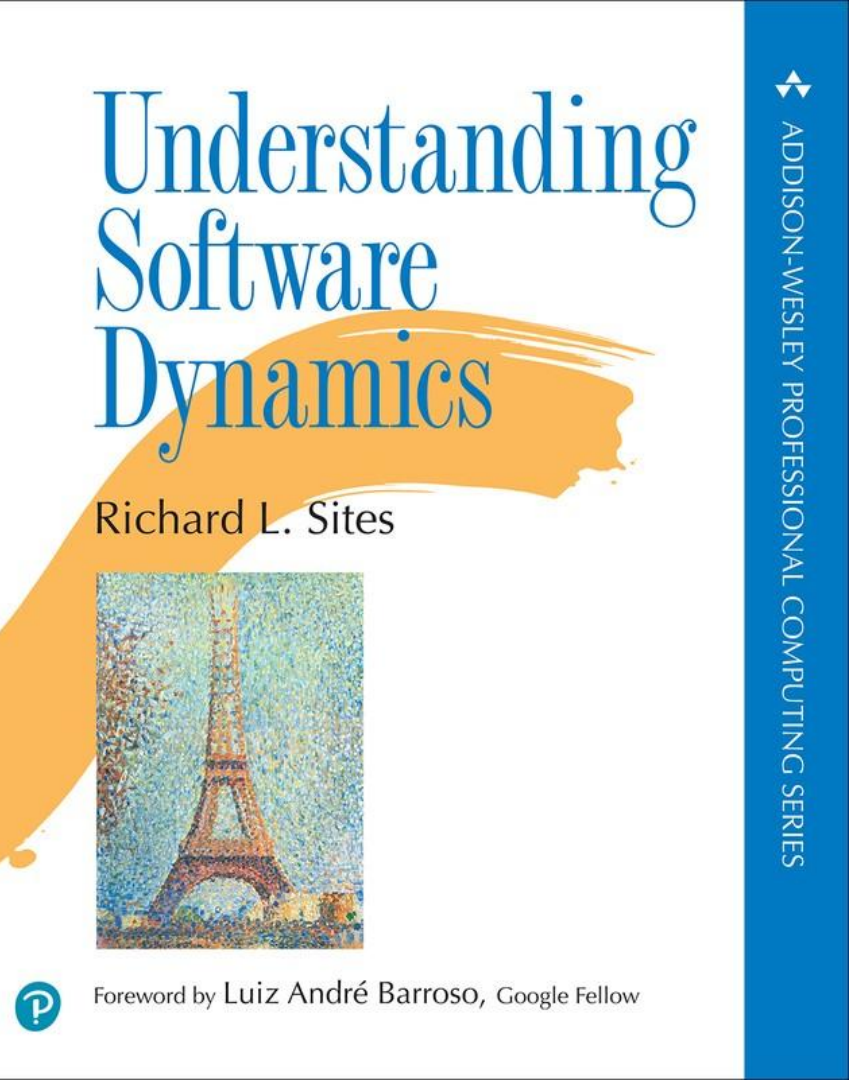
Dick Sites

April 2022

Talk Outline

- Complex Software Dynamics
  - Executing Too Much Code
  - Executing Too Slowly
  - Waiting for CPU
  - Waiting for Memory
  - Waiting for Disk
  - Waiting for Network
  - Waiting for Locks
- Summary





informIT
the trusted technology learning source

Available at Yale Bookstore

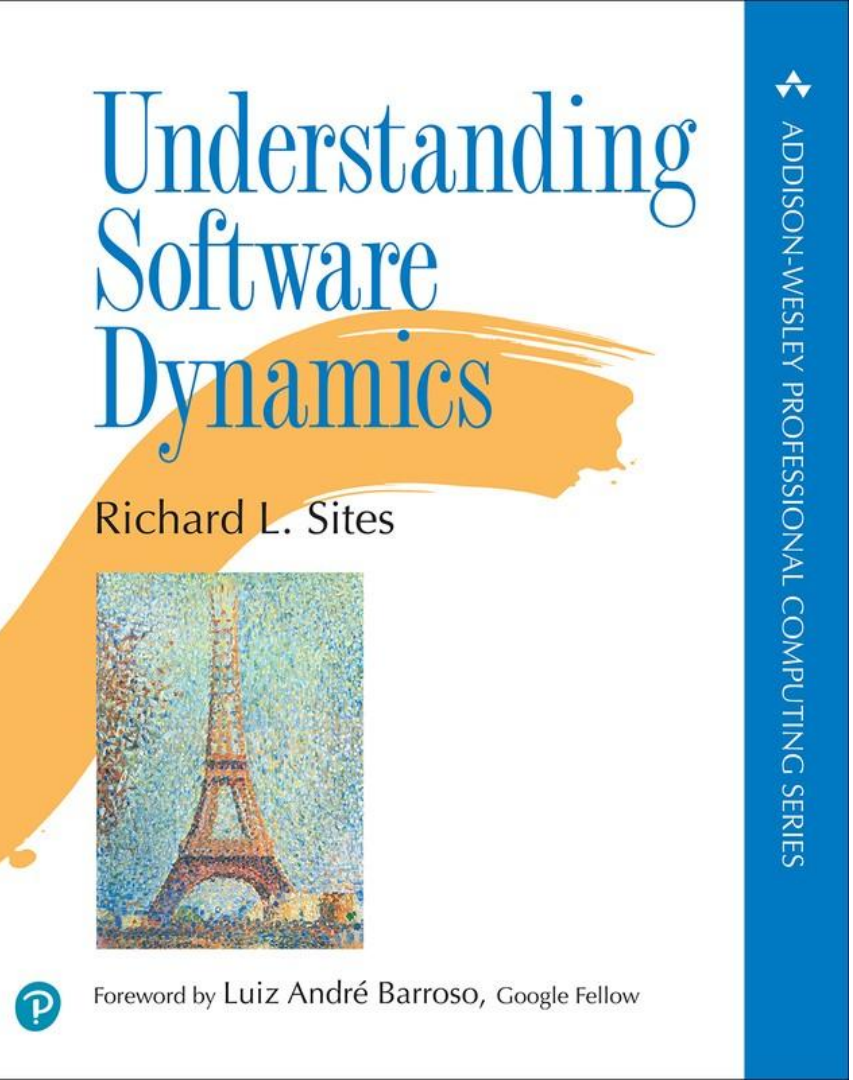
77 Broadway (run by Barnes & Noble)

DRM-Free PDF, EPUB, & MOBI **eBook** files available at **informit.com/dsites**. Use code DSITES to save 35%.

Also Available
Booksellers including Amazon and bn.com, and in O'Reilly's Online Learning subscription service (aka Safari).

*Discount code DSITES is only good at informit.com and cannot be used on the already discounted book + eBook bundle or combined with any other offer.

**Outside the U.S. print books. Please check your local or online store where you purchase technical related books. If your order print books from InformIT, your order is subject to import duties and taxes, which are levied once the package reaches the destination country.



informIT
the trusted technology learning source

Available at the MIT COOP

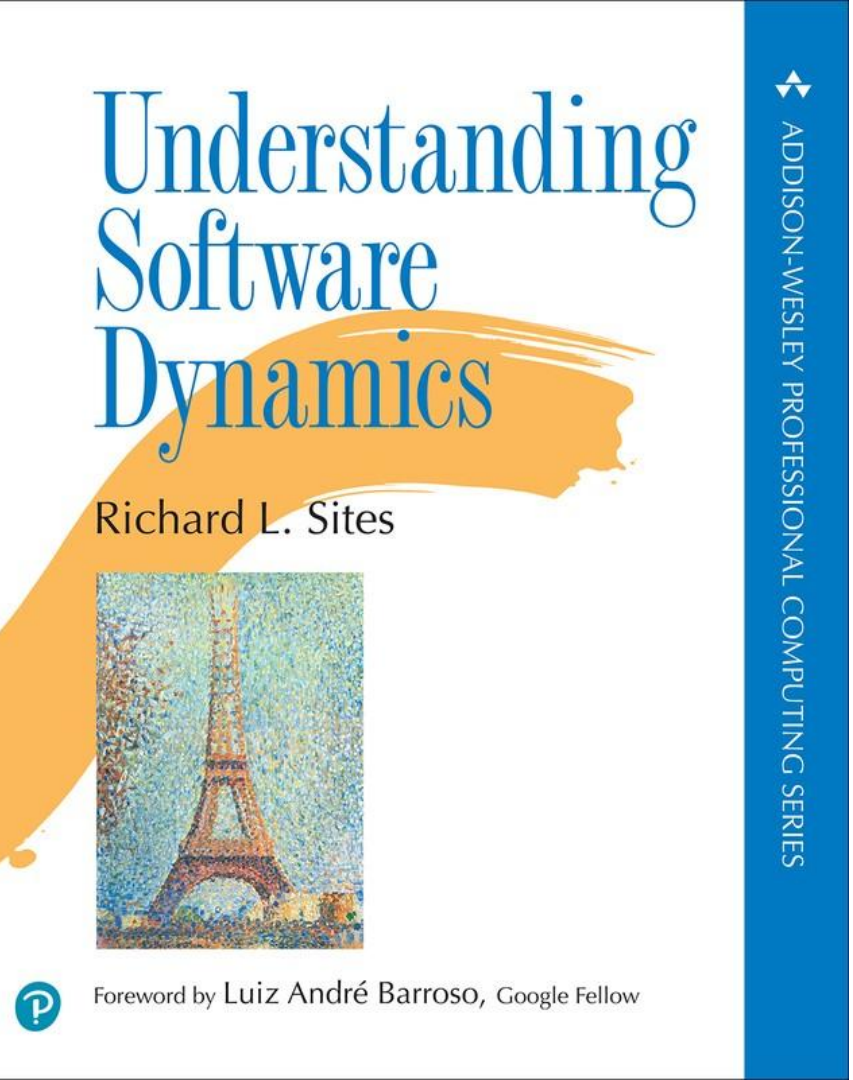
80 Broadway, Kendall Square

DRM-Free PDF, EPUB, & MOBI **eBook** files available at **informit.com/dsites**. Use code DSITES to save 35%.

Also Available
Booksellers including Amazon and bn.com, and in O'Reilly's Online Learning subscription service (aka Safari).

*Discount code DSITES is only good at informit.com and cannot be used on the already discounted book + eBook bundle or combined with any other offer.

**Outside the U.S. print books. Please check your local or online store where you purchase technical related books. If your order print books from InformIT, your order is subject to import duties and taxes, which are levied once the package reaches the destination country.



informIT
the trusted technology learning source

Available at University Co-op

2246 Guadalupe St

DRM-Free PDF, EPUB, & MOBI **eBook** files available at **informit.com/dsites**. Use code DSITES to save 35%.

Also Available
Booksellers including Amazon and bn.com, and in O'Reilly's Online Learning subscription service (aka Safari).

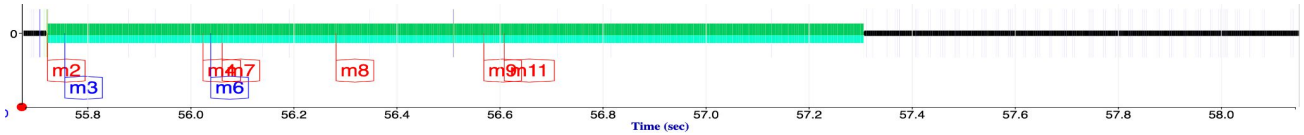
*Discount code DSITES is only good at informit.com and cannot be used on the already discounted book + eBook bundle or combined with any other offer.

**Outside the U.S. print books. Please check your local or online store where you purchase technical related books. If your order print books from InformIT, your order is subject to import duties and taxes, which are levied once the package reaches the destination country.

Complex Software Dynamics

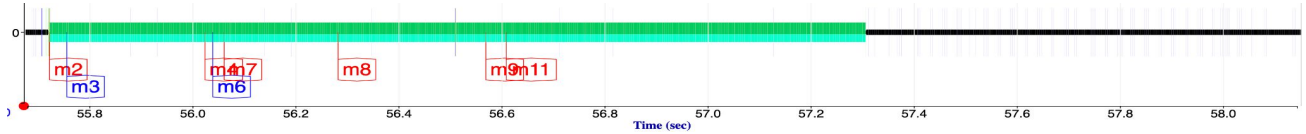
Complex Software Dynamics

Simple software: Single thread, CPU bound (e.g. benchmarks)

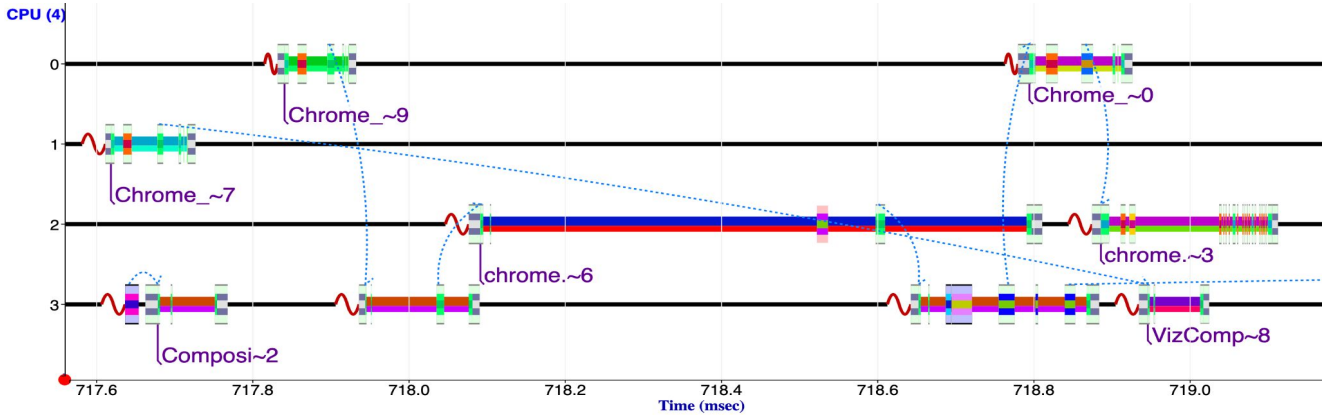


Complex Software Dynamics

Simple software: Single thread, CPU bound (e.g. benchmarks)



Complex software: Multiple threads blocking and waking each other up, interrupts, system calls, page faults



Our focus

Time-sensitive application code that is
- debugged
- has good design and algorithms
- normally has the desired performance

BUT has sporadic unexpected serious **delays** in a production environment

When repetitive code sometimes runs too slowly

There are **only three** possibilities:

1. Executing more code
2. Executing same code more slowly
3. Not executing -- waiting for something

When repetitive code sometimes runs too slowly

There are **only three** possibilities:

1. Executing more code
2. Executing same code more slowly
3. Not executing -- waiting for something:
    a. CPU
    b. Memory
    c. Disk
    d. Network
    e. Lock

When repetitive code sometimes runs too slowly

There are **only three** possibilities:

1. Executing more code
2. Executing same code more slowly
3. Not executing -- waiting for something:
    a. CPU
    b. Memory
    c. Disk
    d. Network
    e. Lock

As an industry, we generally have poor tools for observing what is really happening in time-sensitive production code.

When repetitive code sometimes runs too slowly

We use **KUtrace** to show the invisible

KUtrace is a small set of Linux kernel patches that record every *transition* between kernel-mode execution and user-mode execution, on every CPU core.

Its overhead is less than 1% in a busy production datacenter environment, and much less than that in simpler environments.

Postprocessing produces the dynamic HTML behind the pictures here.

KUtrace: visibility

Running: All execution time of all threads is captured

Not-running: Threads do not spontaneously stop or start.

There is always a reason; there is no magic.

What are some reasons?
How do we find out which one?

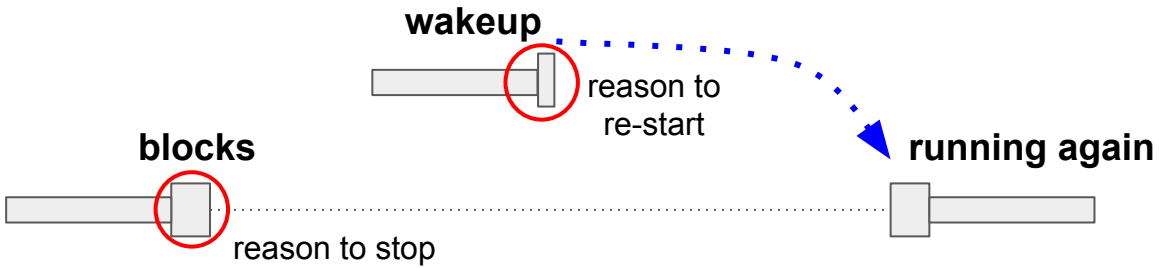
KUtrace: visibility

Not-running: Threads do not spontaneously stop or start.

There is always a reason; there is no magic.

The reason is **always** in a kernel-user trace produced by KUtrace.

This is what makes the invisible visible -- seeing **causality** in action.



When repetitive code sometimes runs too slowly

The following slides show

"black box" execution times of some sample programs, the invisible

and then show

detailed observations of all of that execution time, the visible

revealing the root causes for slowdowns.

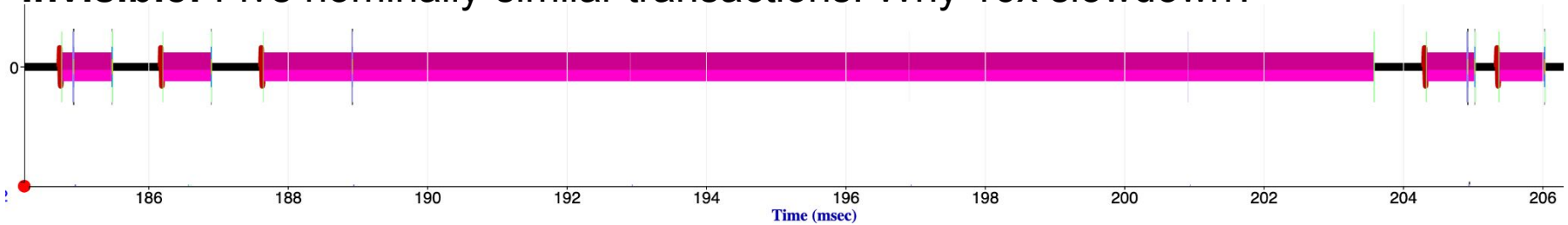
1. Executing Too Much Code

Executing Too Much Code

Invisible: Five nominally-similar transactions. Why 16x slowdown?

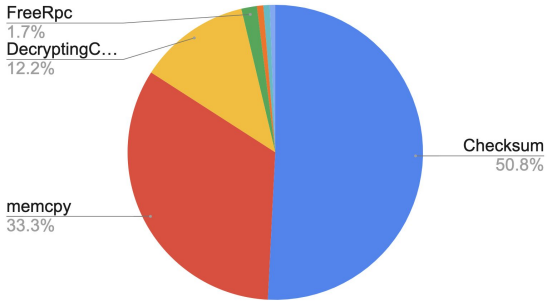
Executing Too Much Code

Invisible: Five nominally-similar transactions. Why 16x slowdown?



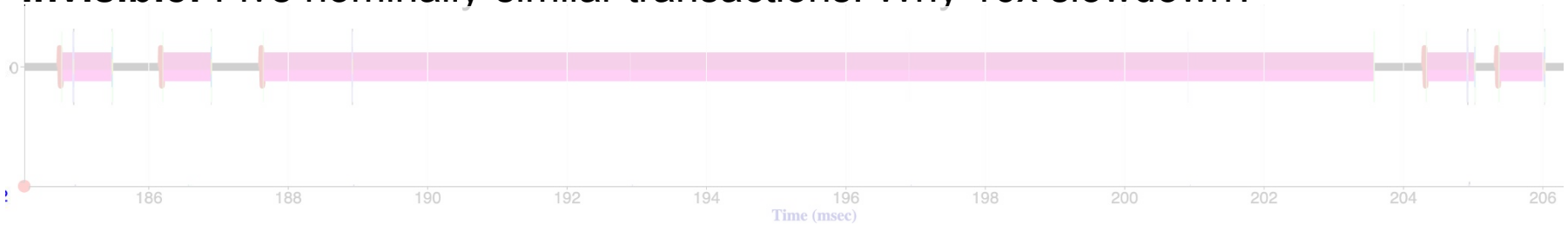
Visible: PC-sample profile

Checksum	50.8%
memcpy	33.3
DecryptingChecksum	12.2
FreeRpc	1.7
__tls_get_addr	0.7
finish_task_switch	0.7
get_page_from_freelist	0.6

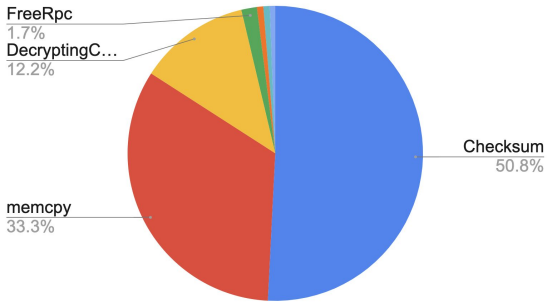


Executing Too Much Code

Invisible: Five nominally-similar transactions. Why 16x slowdown?

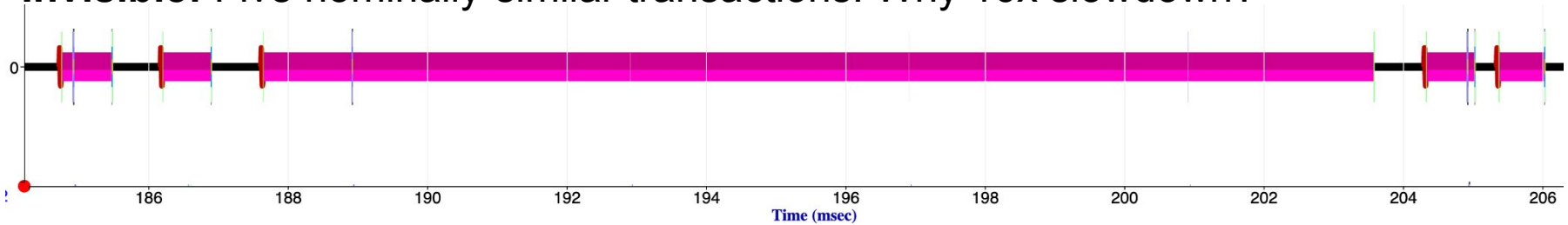


Where is the slow transaction in this profile?



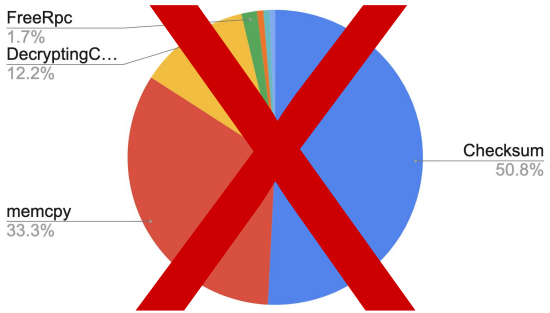
Executing Too Much Code

Invisible: Five nominally-similar transactions. Why 16x slowdown?



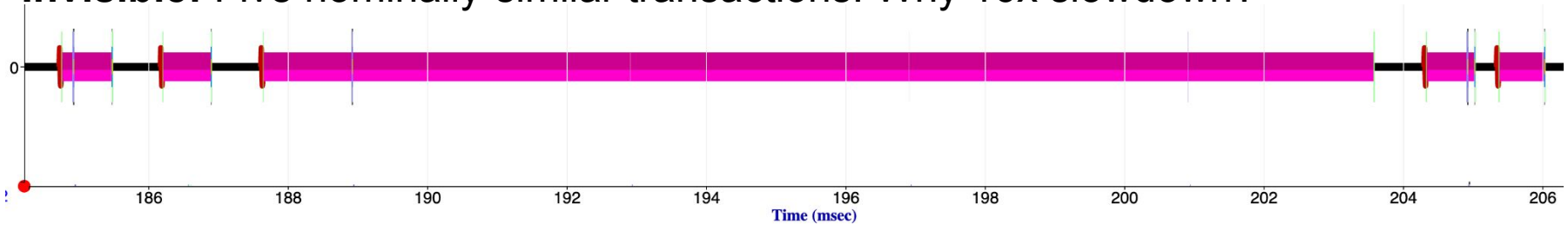
Visible: PC-sample profile **USELESS**

Checksum	50.8
memcpy	33.3
DecryptingChecksum	12.2
FreeRpc	1.7
__tls_get_addr	0.7
finish_task_switch	0.7
get_page_from_freelist	0.6

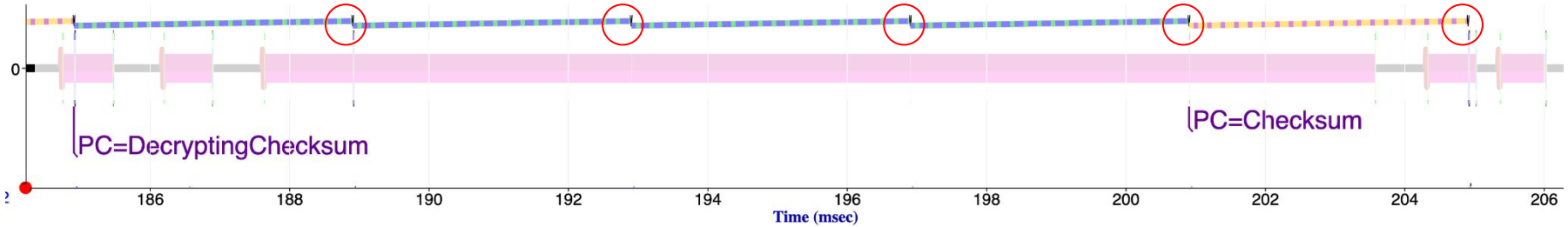


Executing Too Much Code

Invisible: Five nominally-similar transactions. Why 16x slowdown?

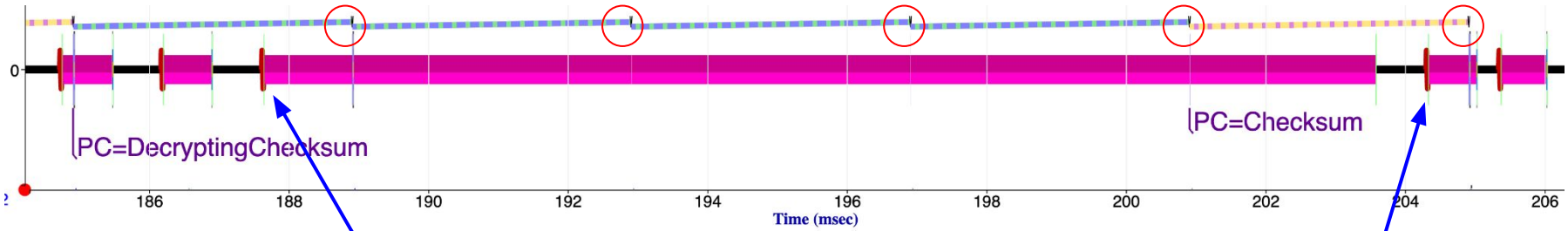


Visible: All timer-interrupt PC samples traced **in context**



Executing Too Much Code

Invisible: Five nominally-similar transactions. Why 16x slowdown?
Visible: All timer-interrupt PC samples traced **in context**



Long transaction executes different, slower, DecryptingChecksum code, which programmer "knew" was just as fast as Checksum. But it isn't.

Executing Too Much Code, summary

Too much code comes from unexpected ...

- Branching
- Calls
- Callbacks

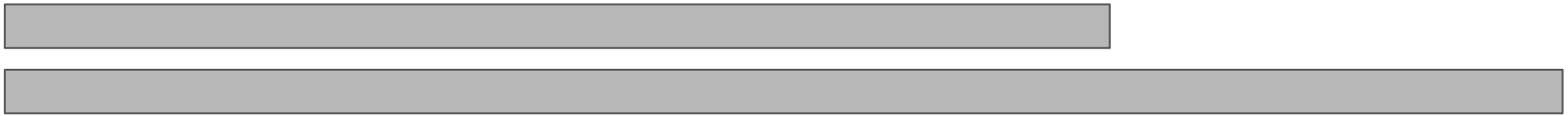
Profiles merge together many fast cases with a few slow cases, hiding what is different about the slow ones. Useless.

Traces reveal what is different.

2. Executing Too Slowly

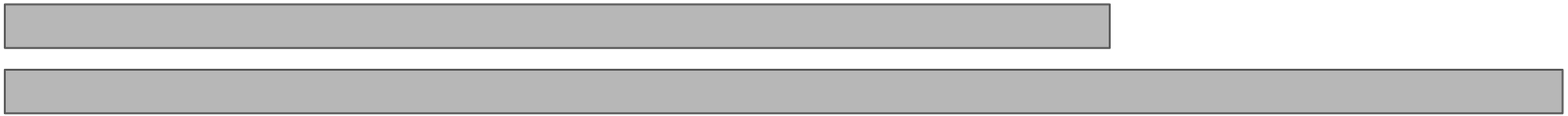
Executing Too Slowly

Invisible: Two runs of same identical benchmark. Why 40% slowdown?

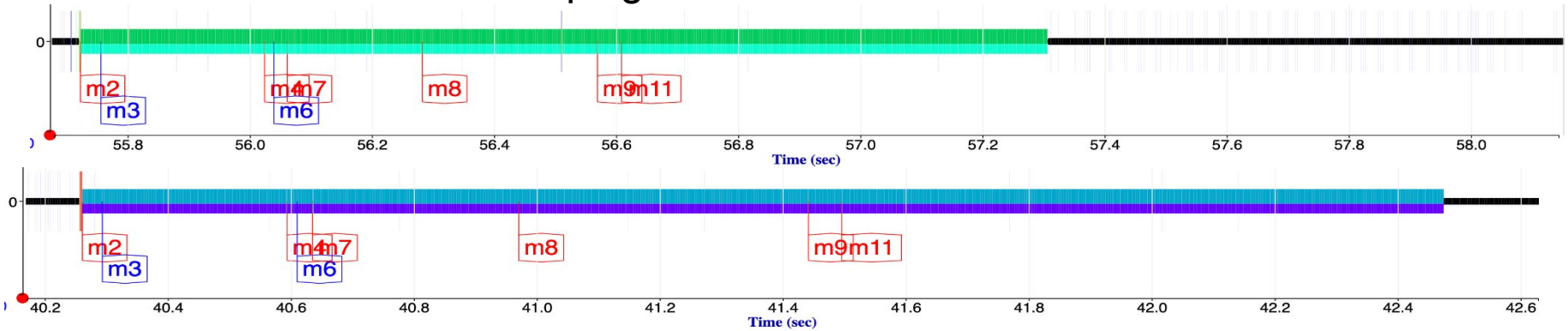


Executing Too Slowly

Invisible: Two runs of same identical benchmark. Why 40% slowdown?



Visible: Some but not all loops get 35-65% slower



Executing Too Slowly

The same code but sometimes executing slowly means that there is some form of **interference** --

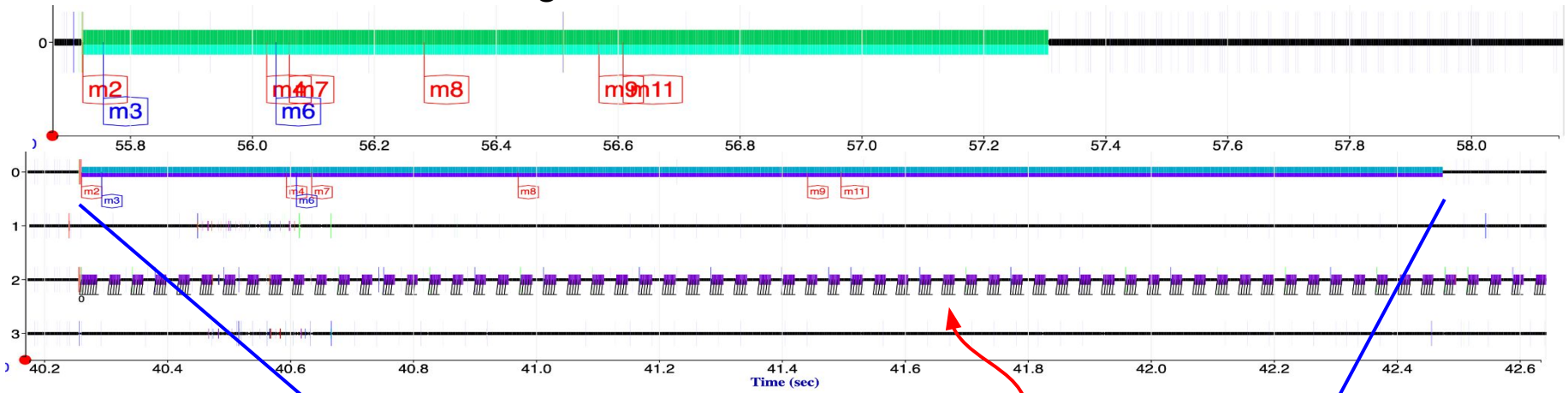
which can only come from use of shared hardware resources or shared software critical sections.

Interference comes from **what else** is running.

Executing Too Slowly

Invisible: Two runs. Why 40% slowdown?

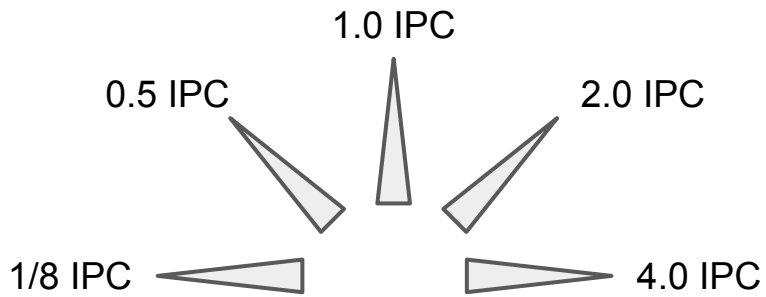
Visible: **What else** is running?



The long run executes slowly because of **another** program.
(Interference is at the floating divide execution unit.
Loops m2 to m6 do not use much floating-point.)

Executing Too Slowly

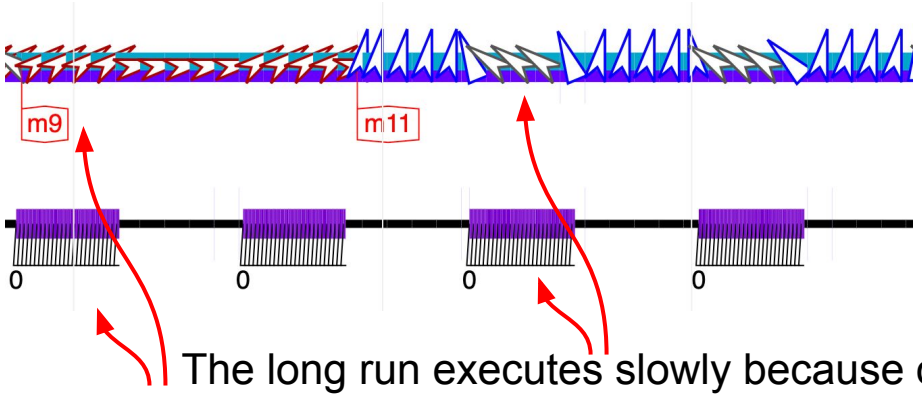
Instructions per cycle (IPC) speedometer



Executing Too Slowly

Invisible: Two runs. Why 40% slowdown?

Visible: What else is running?



The long run executes slowly because of **another** program.
When it runs, the benchmark IPC drops (speedometer triangles).
1.4x for m9 loop, **3x** for m11 loop.

Executing Too Slowly, summary

Executing too slowly comes from ...

- Other-thread, other-program, or operating-system *interference* from use of some shared resource: CPU, memory, disk, network, locks
- Power-saving slow CPU clock frequency
- Slow exit from power saving

Microsecond-scale IPC reveals the interference between tasks.

3 Waiting for something

Waiting for CPU, memory, etc.

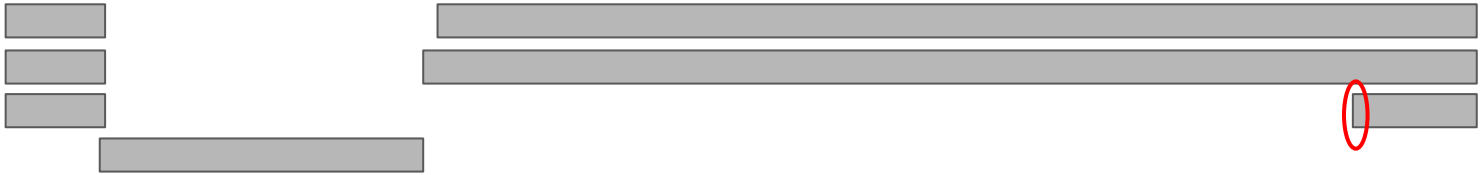
When a program is waiting for something, it is **not executing**.

Profiles only sample during execution. They reveal nothing about waiting. Useless.

3a. Waiting for CPU

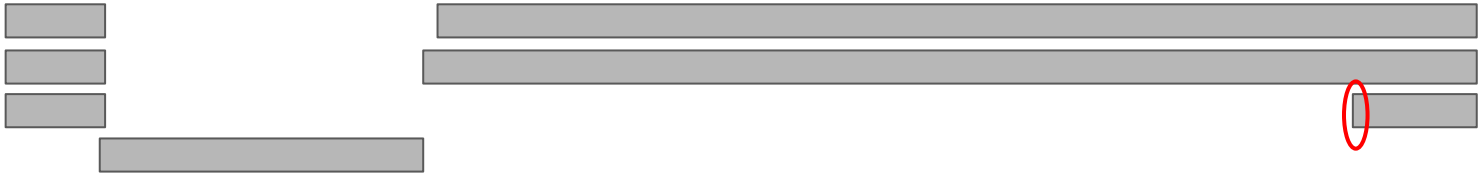
Waiting for CPU

Invisible: Three threads wait on a fourth, then resume. Why longer wait?

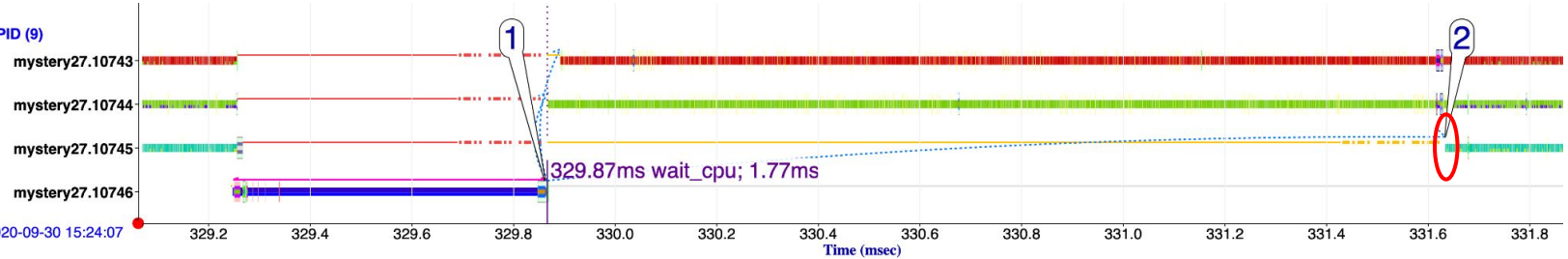


Waiting for CPU

Invisible: Three threads wait on a fourth, then resume. Why longer wait?



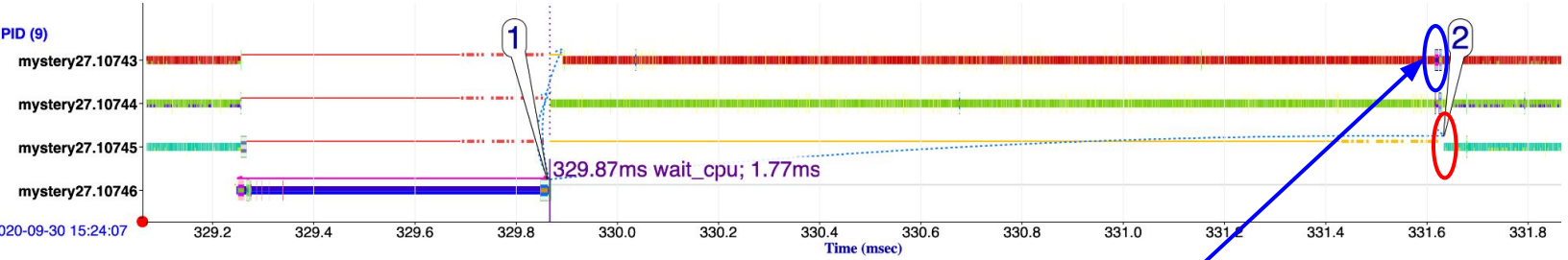
Visible: Long one is waiting almost 2 msec to get a CPU assigned



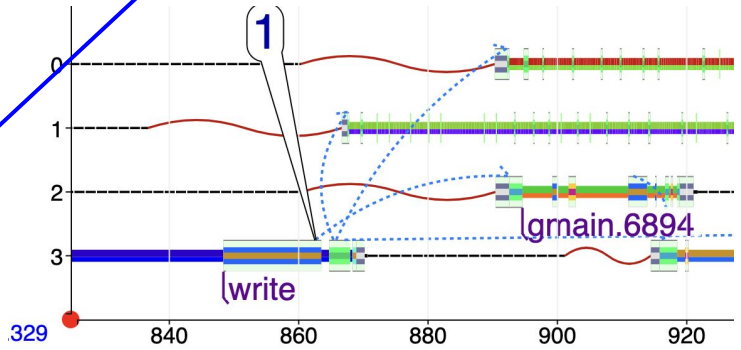
Waiting for CPU

Invisible: Why longer wait?

Visible: Long one is waiting almost 2 msec to get a CPU assigned



At (1), fourth thread does a write that wakes up gmain (Gnome display), and *then* restarts first three threads. Not enough CPUs to go around, so last wakeup waits. Linux scheduler fail: waits until a **timer interrupt** 1.77 msec later to restart.



Heisenbug !

Without the debugging write: no fifth thread and no performance bug.

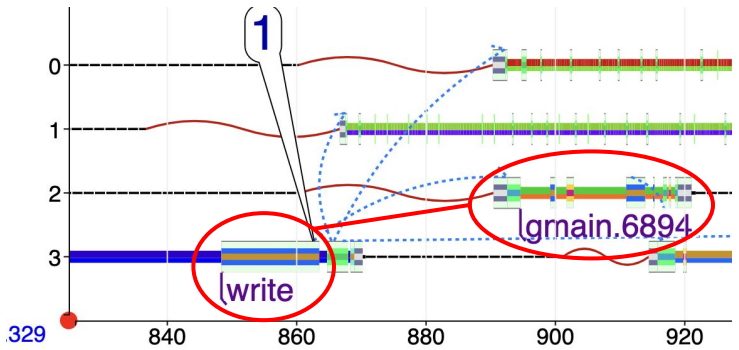
Debugging write to local disk: no fifth thread and no performance bug.

Without the scheduler screwup: no performance bug.

All 3: debug write and ssh access and scheduler screwup = performance bug.

If you can't see it, you can't fix it.

(A combination of causes is typical for difficult bugs)



Waiting for CPU, summary

Waiting for CPU comes from ...

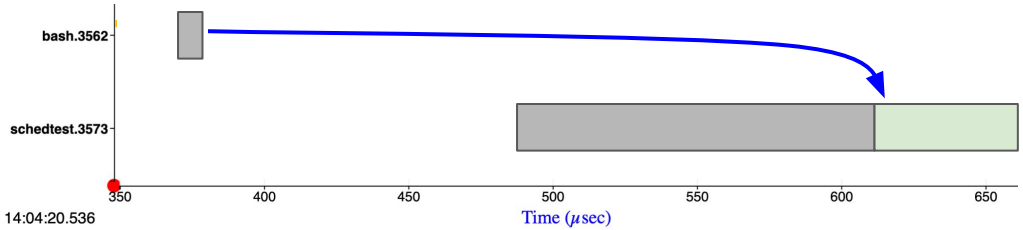
- Busy CPUs
- Scheduler's too-strong affinity to task's last-used core
- Delays coming out of power-saving states
- Complex interactions between user code, kernel code, and the scheduler

Wakeup events tell us what a thread was waiting for.
KUtrace has such low overhead that it does not disturb Heisenbugs.

3b. Waiting for Memory

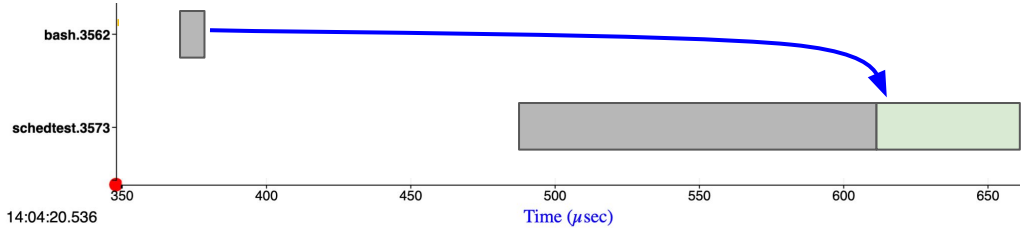
Waiting for Memory

Invisible: Pthread clone() takes a long time to start up

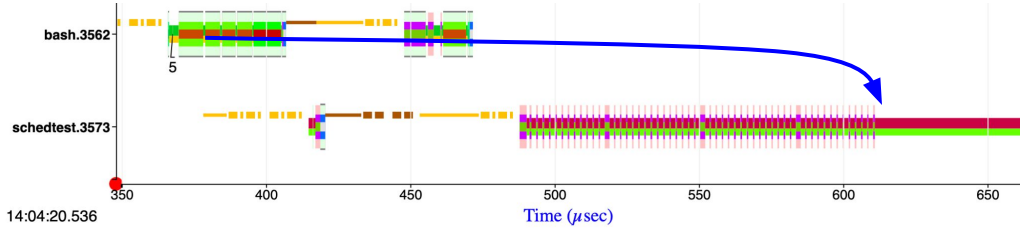


Waiting for Memory

Invisible: Pthread clone() takes a long time to start up



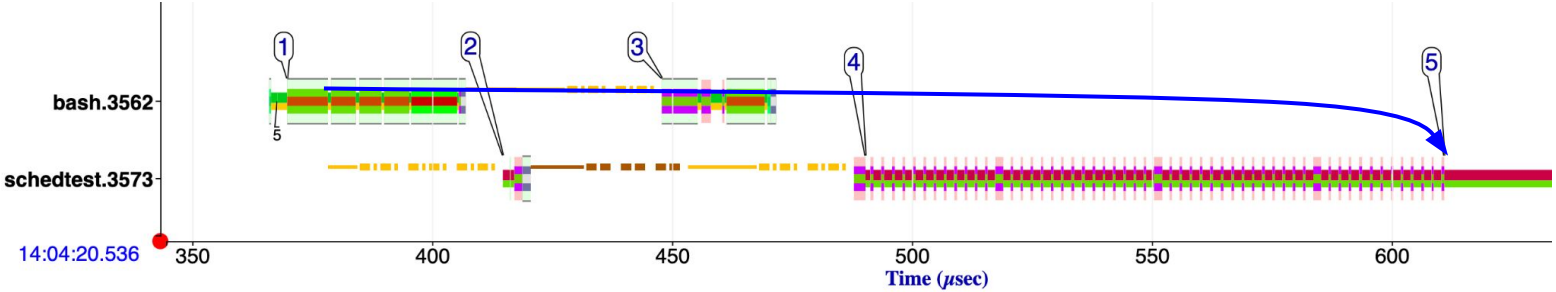
Visible: Started thread waits for CPU and memory



Waiting for Memory

Invisible: Long start up

Visible: Started thread waits for CPU and memory



1. Clone() makes a read-only copy of parent page tables to share its memory.
2. Child's first initialization write takes a page fault to do copy-on-write
3. That fault waits for parent to finish page-table-sharing setup; parent waiting for children
4. Rest of child's initialization takes 60 more page faults
5. Real child processing finally starts

Waiting for Memory, summary

Waiting for memory comes from ...

- All memory allocated
- Large allocation in badly-fragmented memory
- Paging to death
- Other threads manipulating page tables

Seeing the page faults in context matters.

3c. Waiting for Disk

Waiting for Disk

Invisible: Execution timespans for two different 40MB disk reads. Why slowdown?

Read 1: 680 msec

Read 2: 1330 msec

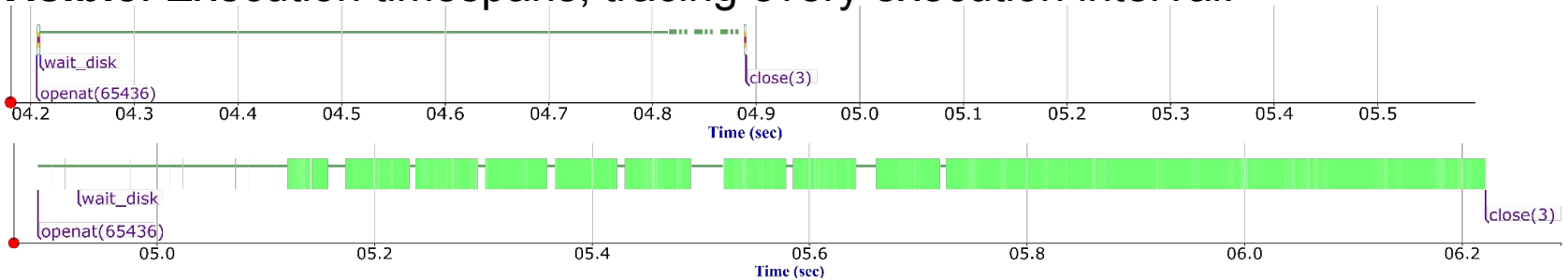
Waiting for Disk

Invisible: Execution timespans for two different 40MB disk reads. Why slowdown?

Read 1: 680 msec

Read 2: 1330 msec

Visible: Execution timespans, tracing every execution interval.



Read 1: **Single** 40MB read; almost all wait_disk for ~0.7 seconds

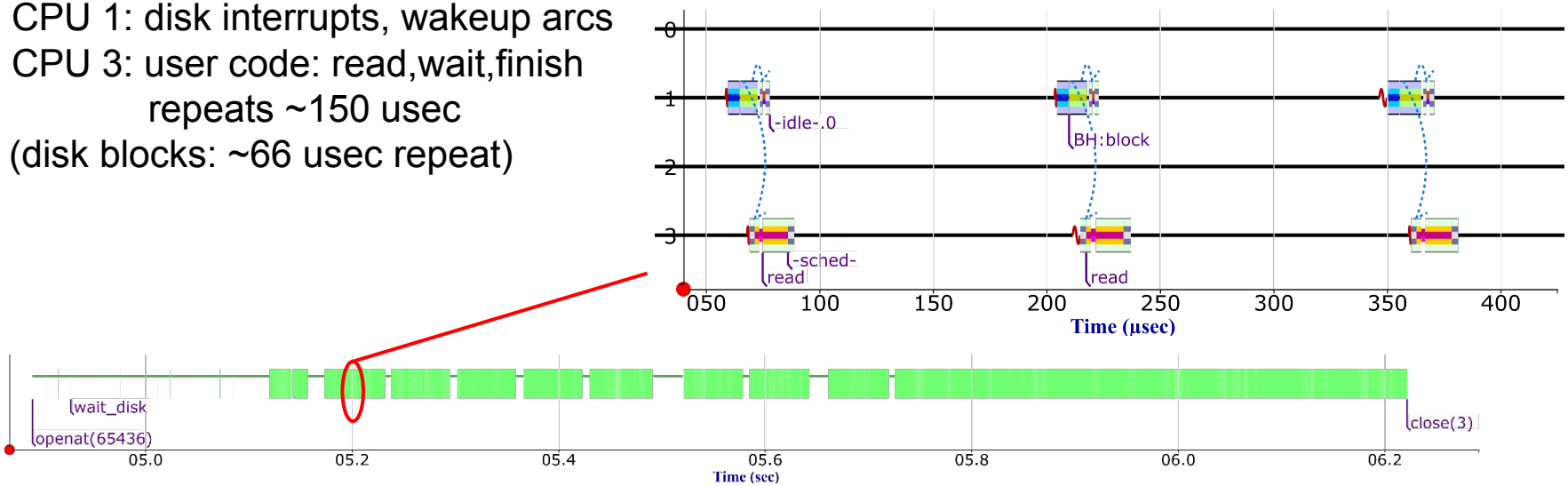
Read 2: 10240 reads, each 4KB, missing every other disk revolution for ~1.3 seconds

Waiting for Disk

Invisible: Why slowdown?

Visible: Detail of three 4KB reads

CPU 1: disk interrupts, wakeup arcs
CPU 3: user code: read,wait,finish
repeats ~150 usec
(disk blocks: ~66 usec repeat)



Waiting for Disk, summary

Waiting for disk/SSD comes from ...

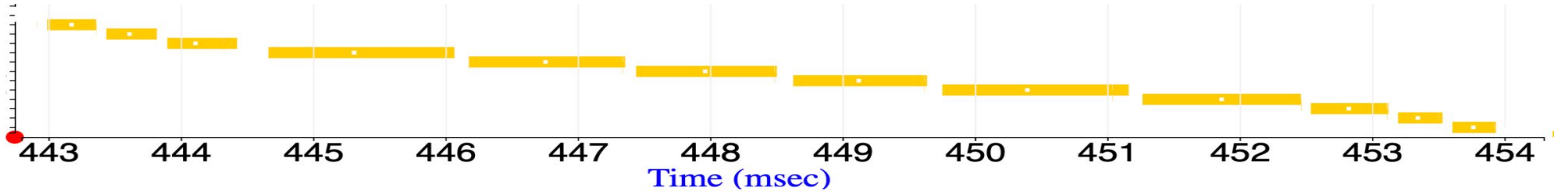
- Busy disks
- Small transfers
- Device emptying write buffer or erasing flash blocks
- Complex interactions between user code, kernel code, file system code, scheduler delays, and storage devices

Seeing the interrupt activity matters.

3d. Waiting for Network

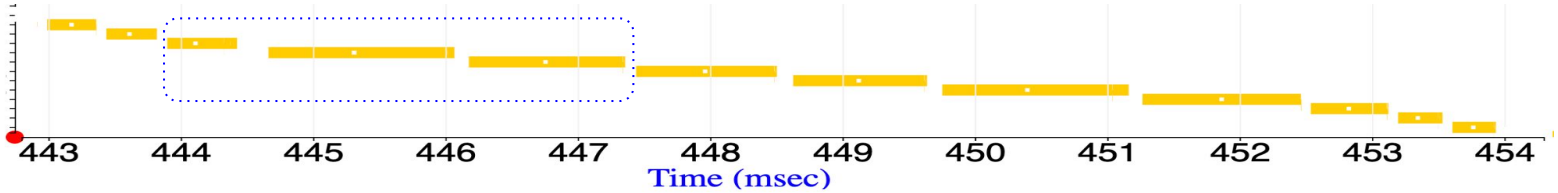
Waiting for Network

Invisible: Twelve RPCs sending 4KB each, receiving short answers. Why 6 slow?

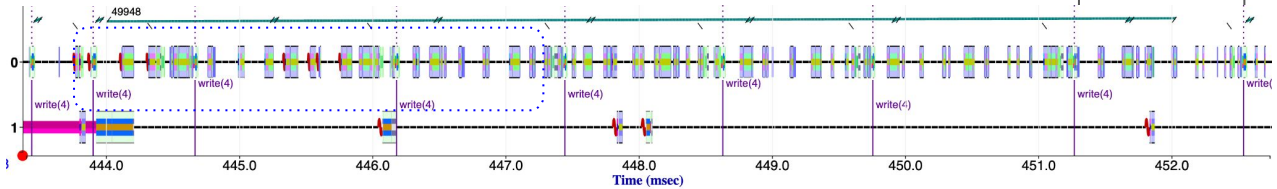


Waiting for Network

Invisible: Twelve RPCs sending 4KB each, receiving short answers. Why 6 slow?



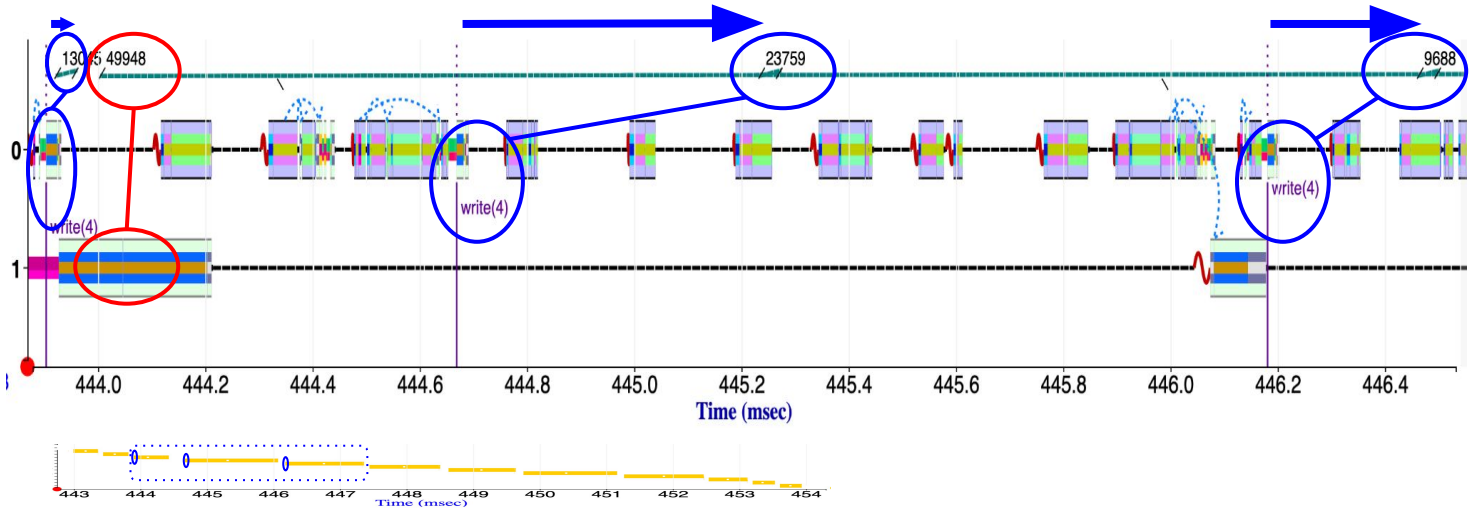
Visible: Time-aligned CPU writes



Waiting for Network

Invisible: Why slow? **Other-thread** 1MB write #49948 delays network access

Visible: Write #13045 not delayed. Writes #23759, ... delayed getting on the wire



Waiting for Network, summary

Waiting for network comes from...

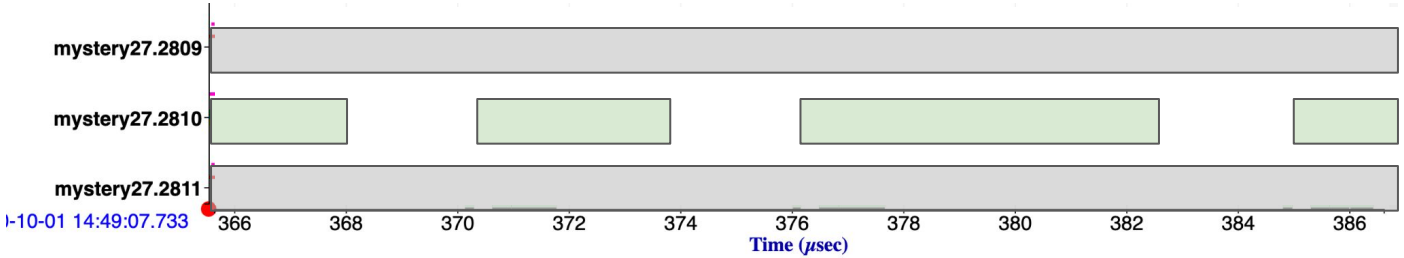
- Outbound kernel-code delay
- Outbound network-access delay
- Network hardware path
- Inbound network-interrupt delay
- Inbound kernel-code delay
- Inbound user-code fetch delay

Seeing when the packets cross the wire is important.

3e. Waiting for Locks

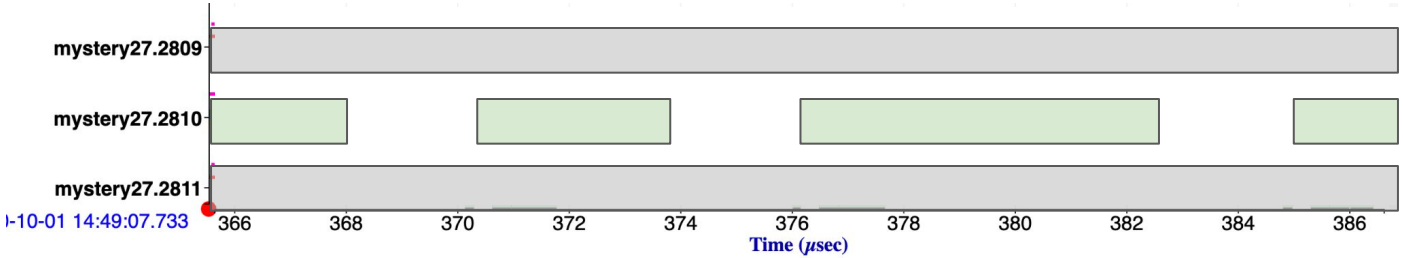
Waiting for Locks

Invisible: Two threads wait a long time for lock; middle thread has it

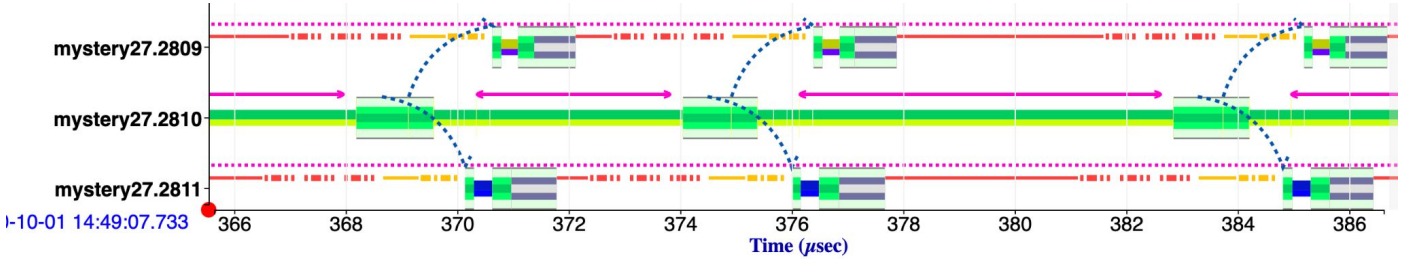


Waiting for Locks

Invisible: Two threads wait a long time for lock; middle thread has it



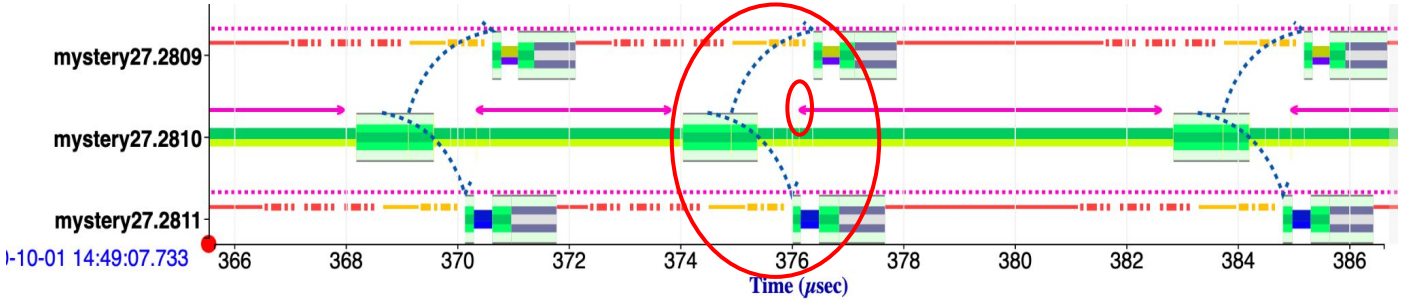
Visible: Middle thread **re-acquires** lock multiple times



Waiting for Locks

Invisible: Middle thread **starves out** the others

Visible: Middle thread re-acquires lock multiple times



Each time middle thread frees the lock, it wakes up the other two. But **before they can run**, it re-acquires the lock. Rinse and repeat ... goes on for 84 msec!

Waiting for Locks, summary

Waiting for locks comes from

- Other threads that are holding the lock
- (Hint: fix those, not the waiting thread)
- (But first you have to know which ones)

Seeing lock acquire, hold, release is important.
Recording *which* lock is important.

The Knuth Challenge

Make a thorough analysis of everything your computer does during one second of computation. -- Don Knuth 1989

The Knuth Challenge

Make a thorough analysis of everything your computer does during one second of computation. -- Don Knuth 1989

"Sites and KUtrace met my 33-year-old one-second Challenge"
-- Don Knuth, March 2022

Overall Summary

Summary

We looked at reasons for unexpected delays in complex software

- Executing too much code
- Executing too slowly
- Waiting: CPU, memory, disk, network, locks

Being able to see what every CPU core is doing every nanosecond makes the invisible visible and reveals root causes. **KUtrace makes the invisible visible.**

Summary

We looked at reasons for unexpected delays in complex software

- Executing too much code
- Executing too slowly
- Waiting: CPU, memory, disk, network, locks

Being able to see what every CPU core is doing every nanosecond makes the invisible visible and reveals root causes. **KUtrace makes the invisible visible.**

As an industry, we generally have poor tools for observing what is really happening in time-sensitive production code.

Reference

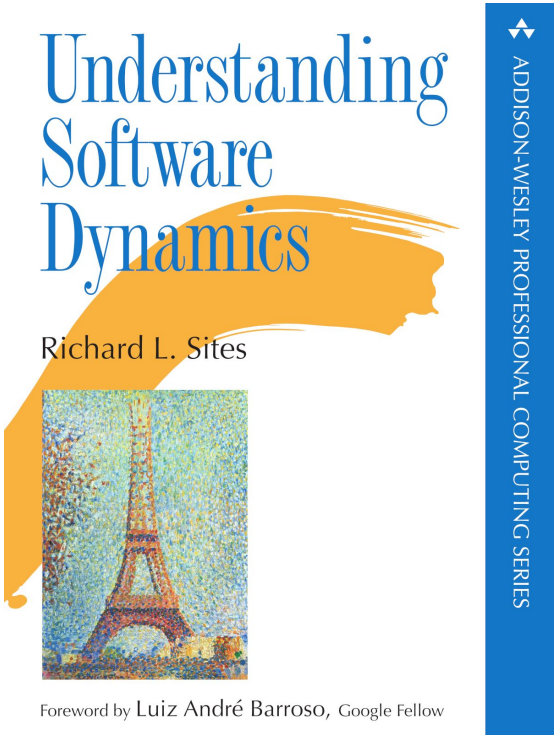
Reference

Richard L. Sites, "Understanding Software Dynamics."
Addison-Wesley, December 2021.

informit.com/dsites Use code DSITES to save 35%

Matching software and HTML at
https://www.informit.com/store/understanding-software-dynamics-9780137589739
(Downloads tab partway down)

or at https://github.com/dicksites/KUtrace



Questions, that perhaps I could answer

abstract

Making the Invisible Visible
Observing Complex Software Dynamics

From mobile and cloud apps to video games to driverless vehicle control, more and more software is time-constrained: it must deliver reliable results seamlessly, consistently, and virtually instantaneously. If it doesn't, customers are unhappy--and sometimes lives are put at risk. When complex software underperforms or fails, identifying the root causes is difficult and, historically, few tools have been available to help, leaving application developers to guess what might be happening. How can we do better?

The key is to have low-overhead observation tools that can show exactly where *all* the elapsed time goes in both normal responses and in delayed responses. Doing so makes visible each of the seven possible reasons for such delays, as we show.