P56 - Understanding Networked-Systems Performance

Friday 15:00-17:00 SE02

Andrew.W.Moore@cl.cam.ac.uk

Acknowledgements.

- With copious thanks to Richard Sites (formerly DEC, HP, MSR, Google) from whom I've cribbed many many ideas, reused slides and generally used and reused ideas and past work.

- Professor Noa Zilberman (Oxford), and many others whom I've bounced off ideas.

- Thank you to my students from 2023 – they survived and enjoyed the first incarnation of this module.

- Furthermore, a **huge** thank you to Eben Upton, Raspberry Pi Foundation, and PiHut people for enabling this incarnation of the module.

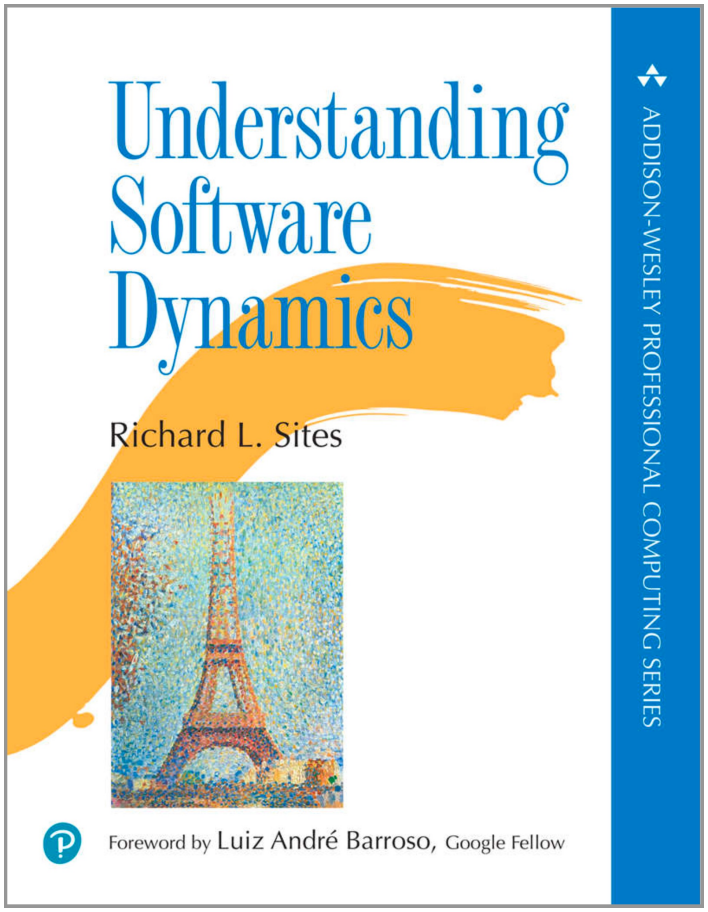
- Thank you to the sysadmins who answer my 11th hour pleas on a regular basis – and made the I51 machine.

High Performance Networked-Systems

On completion of this module, students should:

- Describe the role of high performance networked-systems and where they are used;
- Develop an appreciation of best-practices by performing hands-on measurement and analysis;
- Understand the architecture of a high performance networked-system;
- Understand with greater insight high performance networking devices;
- Understand challenges and solutions to high performance measurement;
- Understand how to select or implement tools to measure high performance measurement;
- Utilise representation techniques to understand and interpret networked-systems;
- Evaluate the performance of example high performance networked-systems

Textbook



Includes a number of discussions about getting information mostly leading to the ***kutrace*** tool

We will **loan** you a copy for the duration of the course.

The book says “Software” but don’t be fooled, computer software is our (human) gateway.

If the **hardware** doesn’t make the **software** go fast – it isn’t high-performance.

This book covers much more than we will have time to cover here – **consider investing** in a copy **for** your **professional library**.

Yes I **do** want the loaners back at the end of term.

High Performance Networked Systems

- The disks get faster
  - The CPUs get faster (at least a bit faster)
    - The Memory get faster
      - The Networks get faster – a lot faster
      - So why doesn't your program go faster too?

        Because nothing is simple. Sorry.

What is a *high-performance* system?

Supercomputer?

Maybe in the 1980's

Now it's datacenters

My program is too slow

If you don't have any idea how long your program should take, you are not in a position to complain that it is slow.

So as a performance professional, you are always making quick estimates of how long routines should take.

A factor of ten – a magnitude - is often good enough.

Estimates

Estimates

MIT freshman physics, 8.01

First assignment

"Estimate the number of blades of grass in the Great Court”

Cambridge challenge – which college court should we count?

Fermi estimate (1945)

About 40 seconds after the explosion the air blast reached me. I tried to estimate its strength by dropping from about six feet small pieces of paper before, during and after the passage of the blast wave. Since, at the time, there was no wind I could observe very distinctly and actually measure the displacement of the pieces of paper that were in the process of falling while the blast was passing. The shift was about 2½ meters, which, at the time, I estimated to correspond to the blast that would be produced by ten thousand tons of T.N.T.

CLASSIFICA... Unclassified
ON CHANGE... H. F. Cabrill
BY A... B. Wire 1-27-65

This document contains information affecting the National Defense of the United States within the meaning of the Espionage Act U. S. C. 50-31 and 32.

SECRET

Jeff Dean's 'Numbers Everyone Should Know'

L1 cache reference	0.5 ns	O(1) ns
Branch mispredict	3.0 ns	O(10) ns
L2 cache reference	4.0 ns	O(10) ns
Mutex lock/unlock	17.0 ns	O(10) ns
Main memory reference	100.0 ns	O(100) ns
Compress 1K bytes with Zippy/Snappy	2,000.0 ns	O(1) us
Read 1 MB sequentially from memory	4,000.0 ns	O(10) us
Send 2K bytes over 1 Gbps network	20,000.0 ns	O(10) us
Read 1 MB sequentially from SSD	62,000.0 ns	O(10) us
Round trip within same datacenter	500,000.0 ns	O(1) ms
Read 1 MB sequentially from spinning disk	947,000.0 ns	O(10) ms
Disk seek	3,000,000.0 ns	O(10) ms
Read 1 MB sequentially from network	10,000,000.0 ns	O(10) ms
Send packet CA->Netherlands->CA	150,000,000.0 ns	O(100) ms

Taken from a mashup of two slides stacks and some updated numbers from URL below
http://static.googleusercontent.com/external_content/untrusted_dlcp/research.google.com/en/us/people/jeff/stanford-295-talk.pdf
http://www.cs.cornell.edu/projects/ladis2009/talks/dean-keynote-ladis2009.pdf
https://kousiknath.medium.com/must-know-numbers-for-every-computer-engineer-6338a12c292c
https://norvig.com/21-days.html#answers

So….

- How long does it take to read a file?
  - a 10KByte file
    - With random seeking
      - Over the network
      - …..
- How long does it take to finish a matrix multiplication?
  - Of 10,000,000 elements
    - 8,000,000 times?
- How quickly will a particular web page be retrieved and rendered?
- How many frames of graphics per second can I render?
- How does everything affect me so?

Don’t be afraid to make an estimate

Networked Systems - nothing is straightforward

- Two computers connected with 100GbE will .never. NEVER move data from one computer memory to another computer memory at 100Gbit/s
- Spinning disks are very weird and techniques to make them go faster have been around since the 50’s
- Main memory is SERIOUSLY not sensible
- SSD and NVMe disks are just memory that (hopefully) remembers without power
- Don’t get me started on the sneaky tricks CPUs use!

Datacenter Servers are ‘Weird’

- Server programs start and run for months
  - Tens of programs per server
- Real-time user-facing transactions, not batch
  - Minimize latency, not CPU idle time
- Big fanout: one transaction to 1000+ servers
  - only 1-10ms CPU time per server
- 10K+ computers, 100K+ disks, 1000k+ network connections

What is different about datacenter software?

Datacenter Servers are not Weird just Different

① **(Routinely)** Move data: big and small

② **(Many)** Real-time transactions: many 1000s per second

③ **(limited)** Isolation between programs

④ Measurement underpinnings **(can help)**

Differences from desktop software

Tens of thousands of user-facing transactions per second

Distributed computation across thousands of servers

The important metric is response time, i.e. latency

Excessive tail latency is the most important performance problem

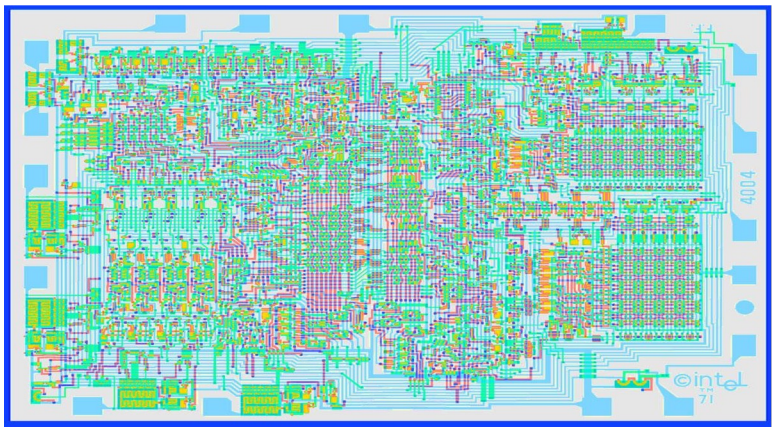
As an industry, we have poor tools for observing and understanding tail latency

① Move data: big and small

- Move *lots* of data
  - Disk to/from RAM
  - Network to/from RAM
  - SSD to/from RAM
  - Within RAM
- Bulk data
- *Short* data: variable-length items
- Compress, encrypt, checksum, hash, sort

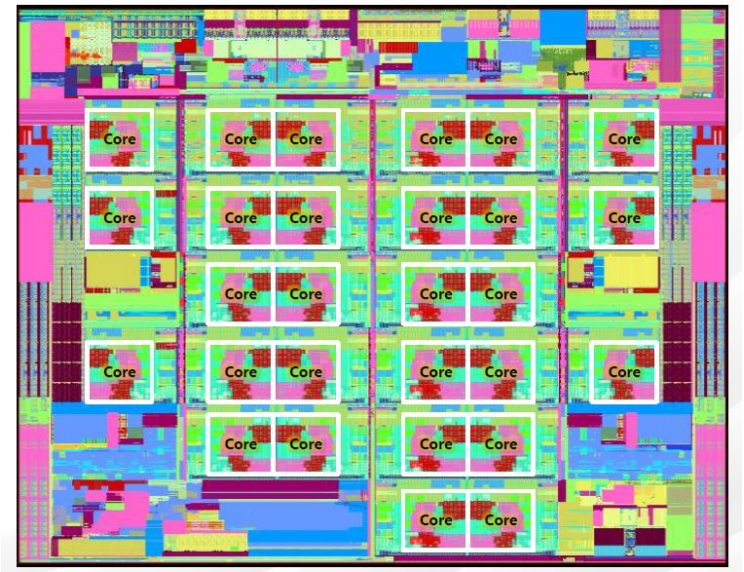
Lots of memory

- 4004: no memory



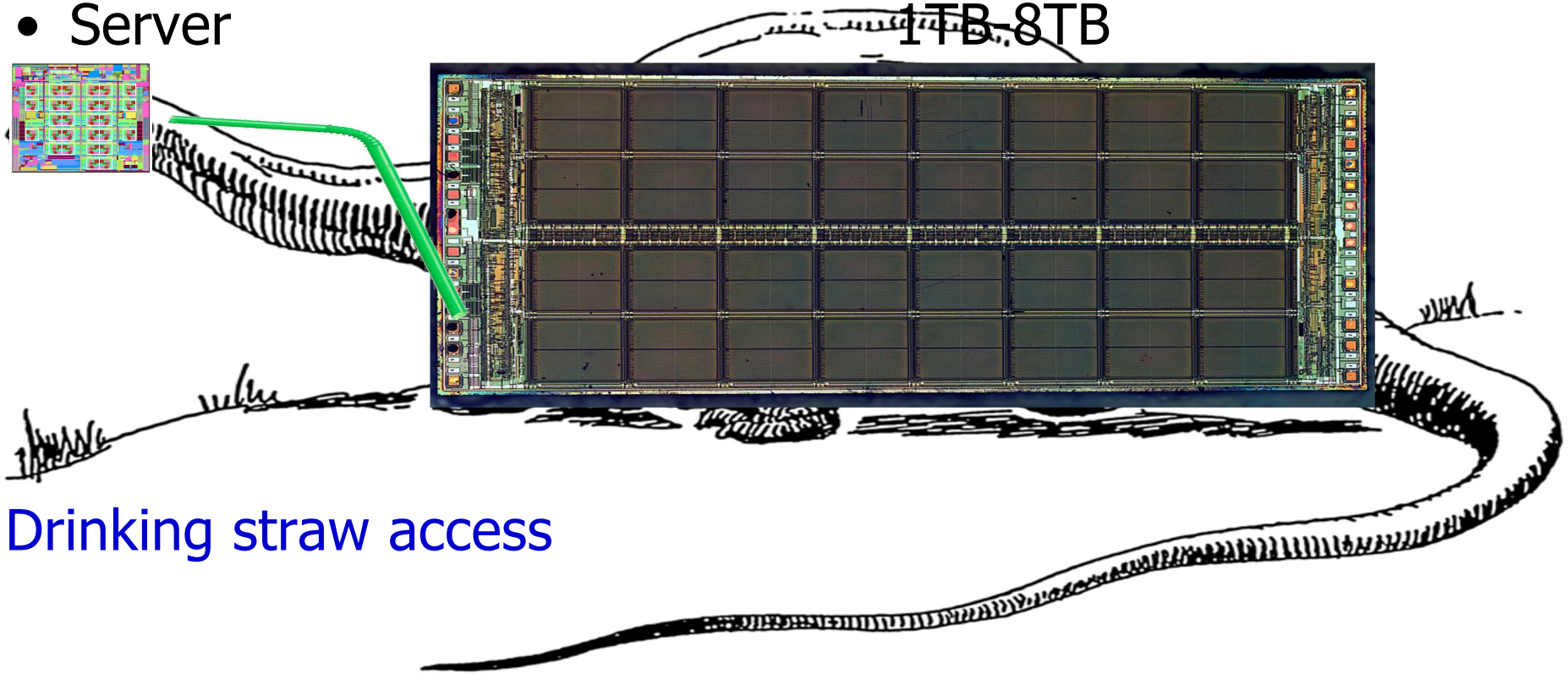
- Xeon Ice Lake: (2020 vintage)

L1-D 48KByte
L2 1.5MByte per core
L3 1.8MByte per core
(28 cores…. L3 = 50MByte)



Little brain, **LOTS** of memory

- Server

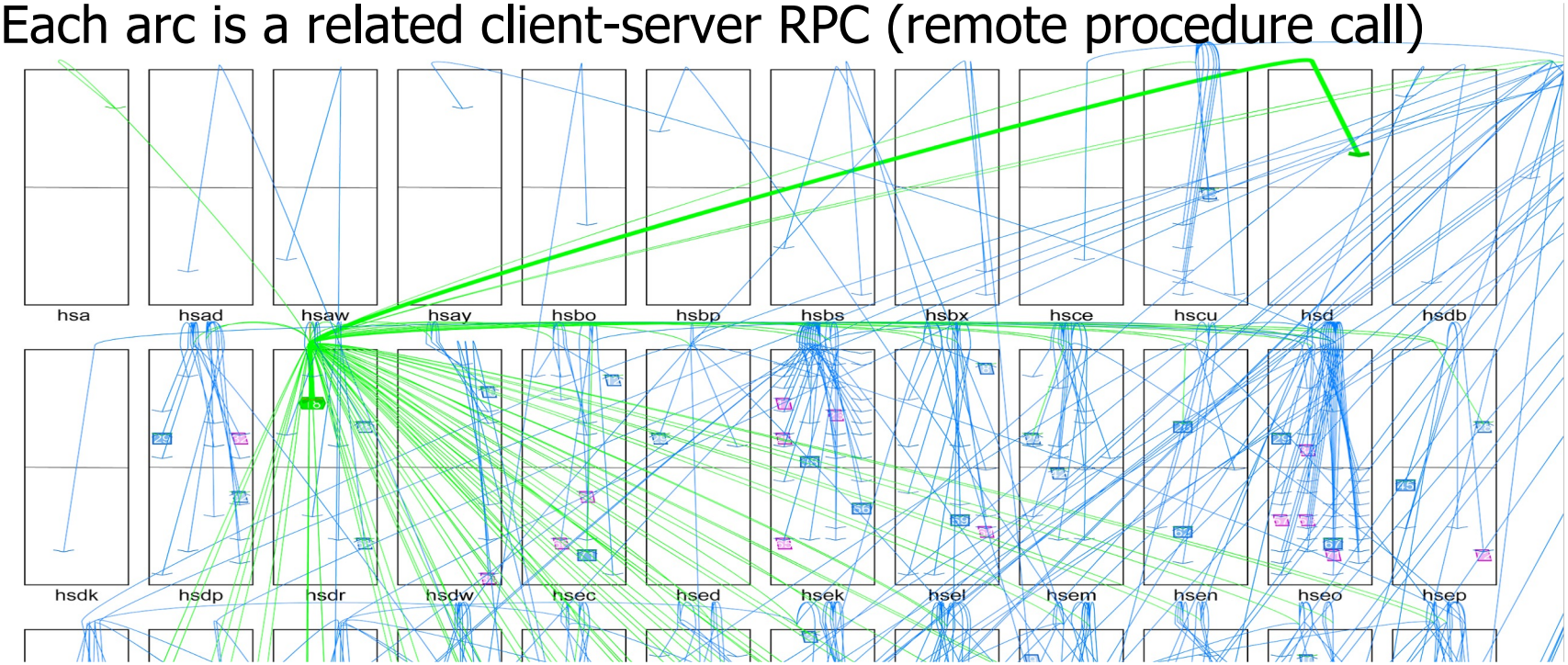


Drinking straw access

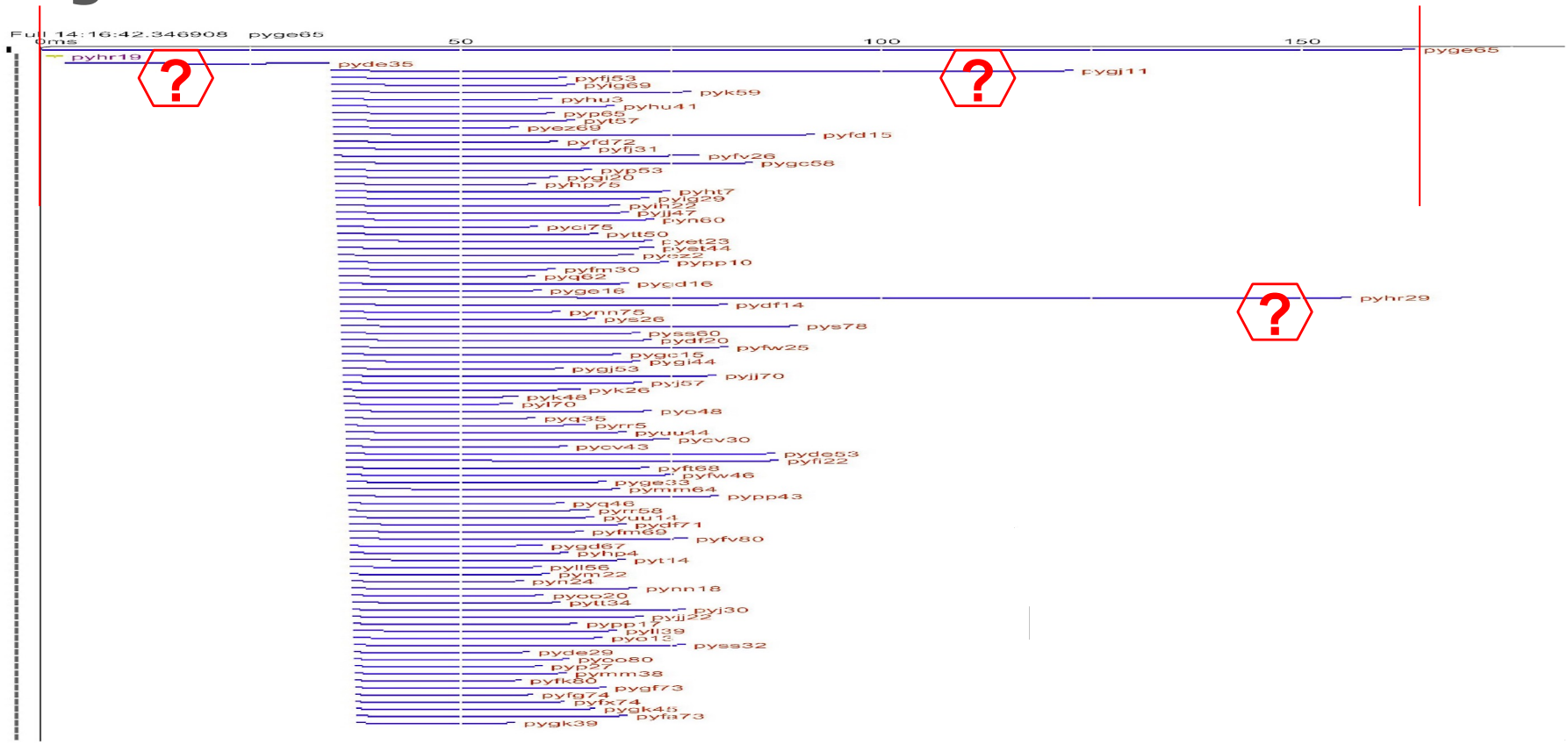
② Real-time transactions: 1000s per second

A **Single** Transaction Across ~40 Racks of ~60 Servers

- Each arc is a related client-server RPC (remote procedure call)

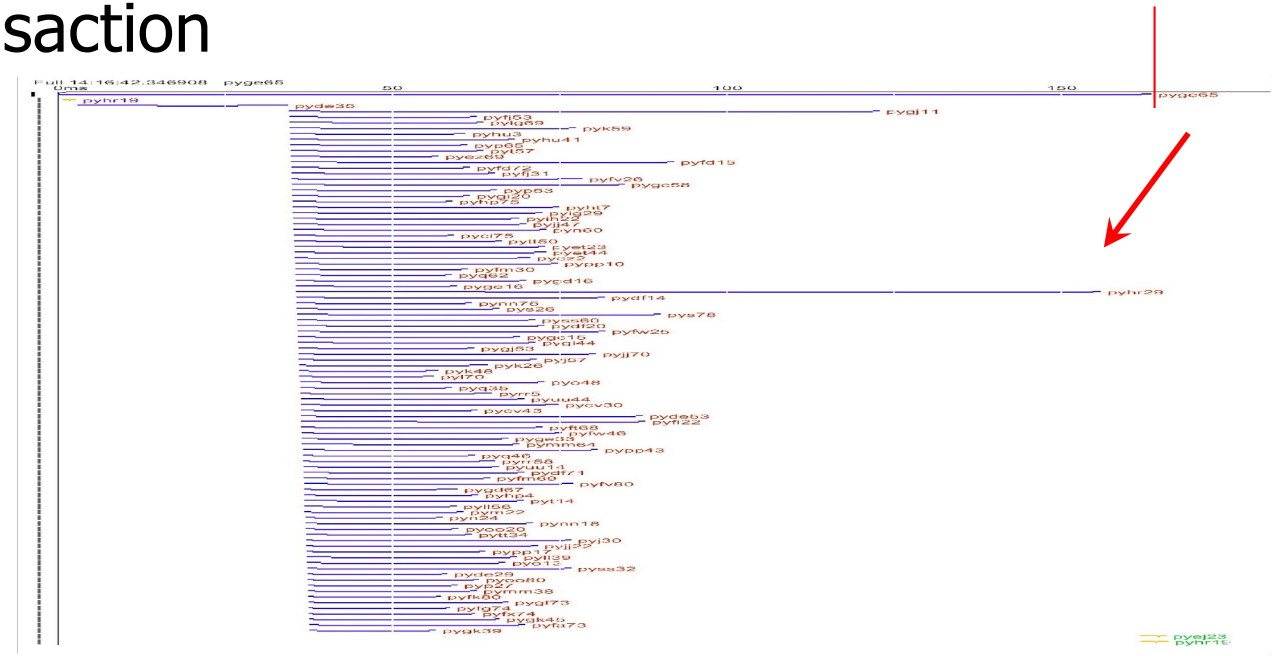


A **Single** Transaction RPC Tree: Client & 93 Servers



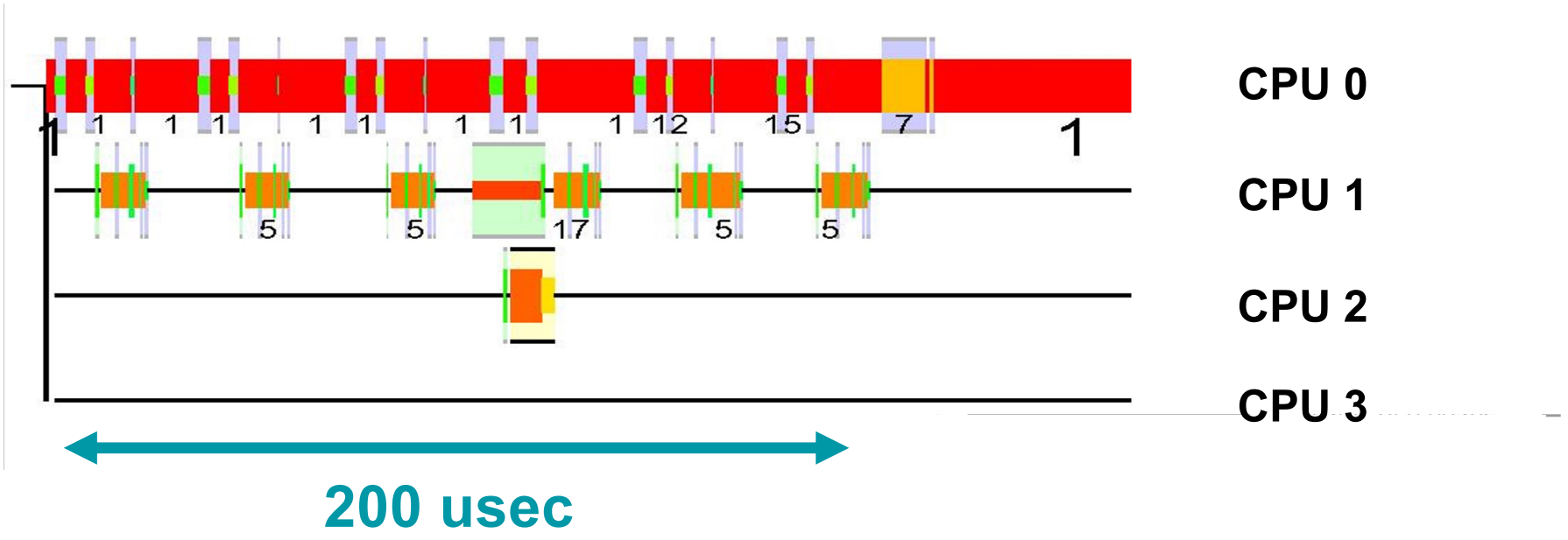
Single Transaction Tail Latency

- One slow response out of 93 parallel RPCs slows the *entire* transaction



One Server, Four CPUs: User/kernel transitions

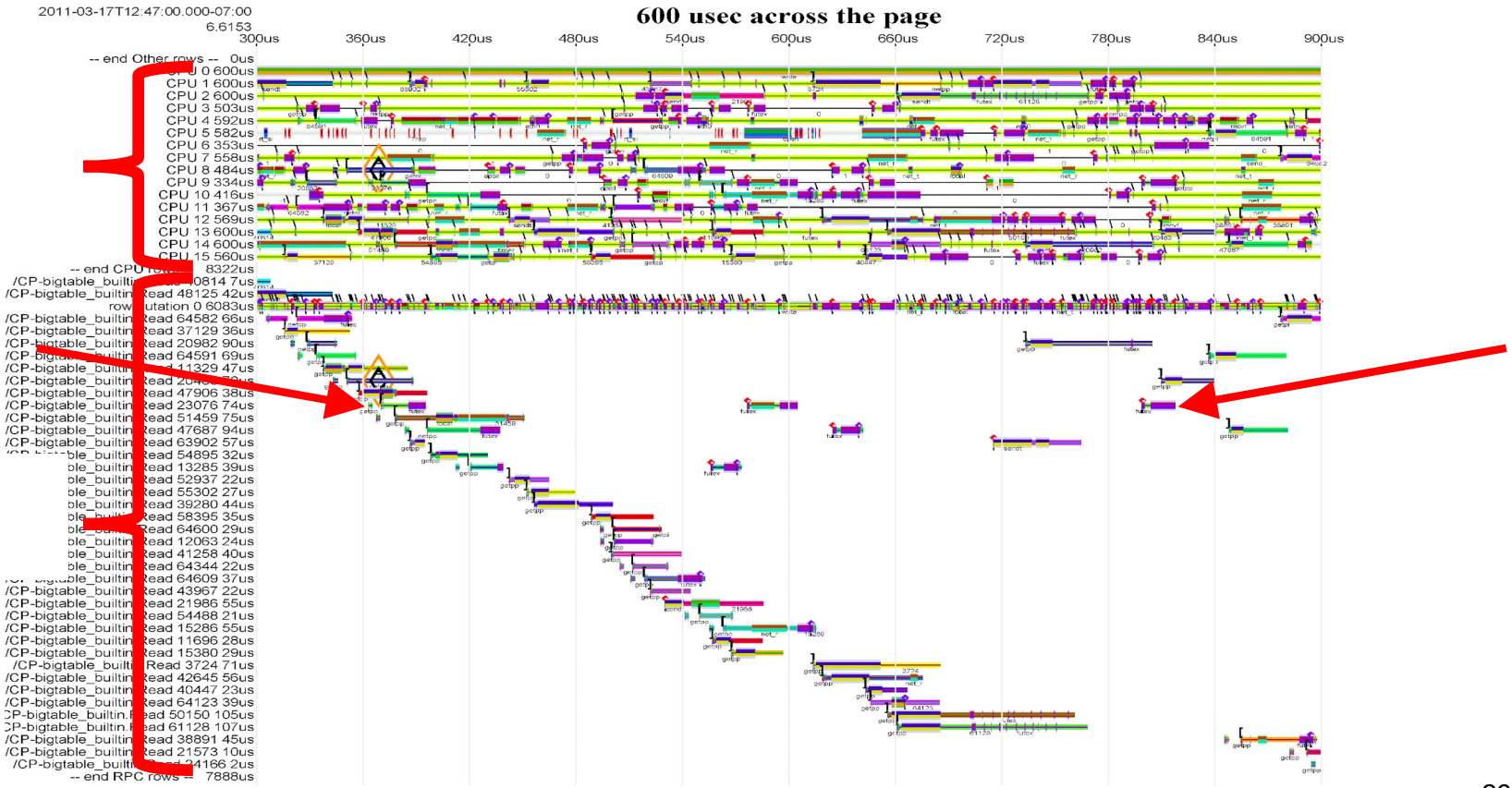
every CPU every nanosecond (Ktrace)



16 CPUs, 600us, Many RPCs

CPUs 0..15

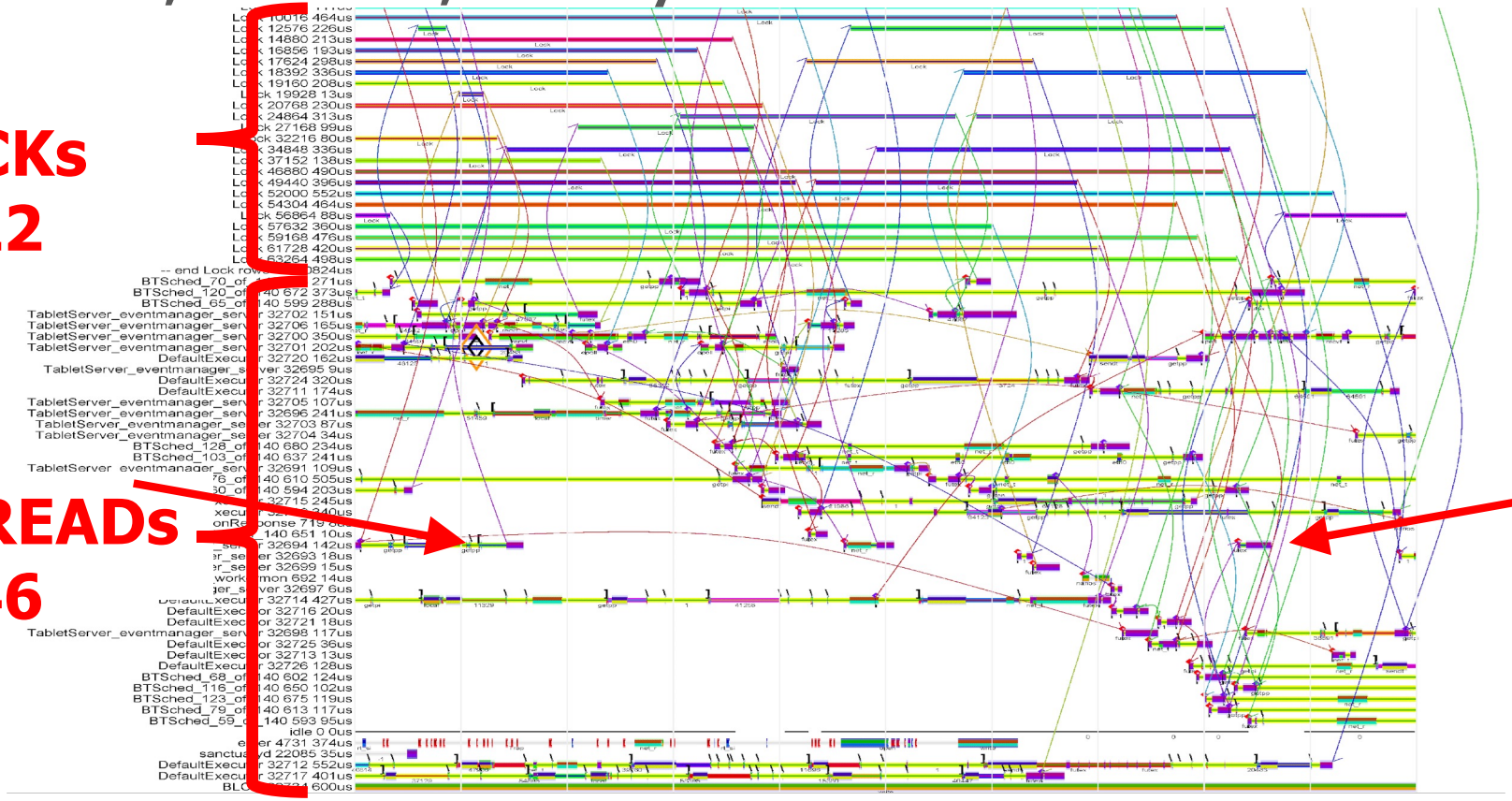
RPCs 0..39



16 CPUs, 600us, Many RPCs

LOCKs
0..22

THREADs
0..46



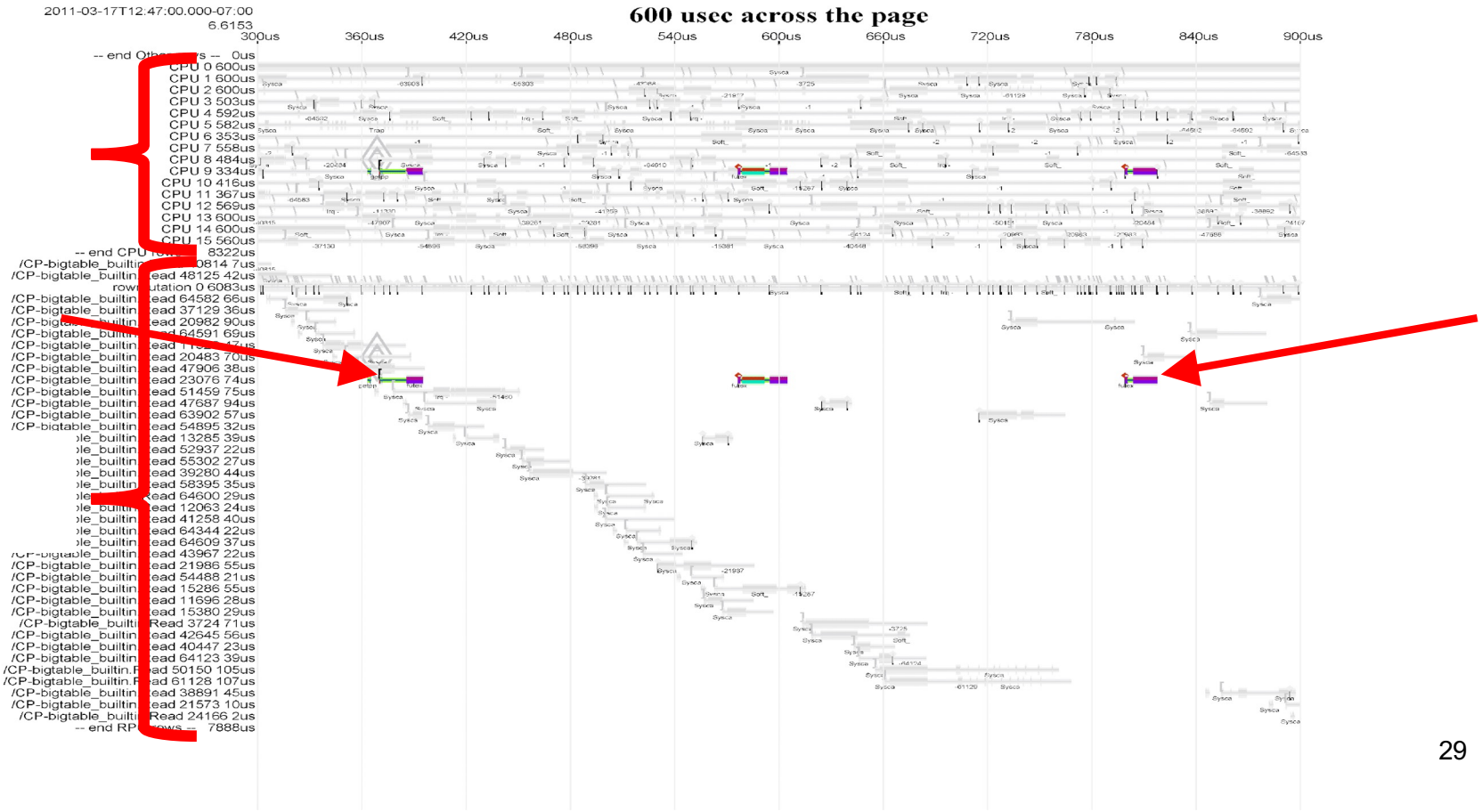
That is A LOT going on at once

- Let's look at just *one* long-tail RPC in context

16 CPUs, 600us, one RPC

CPUs 0..15

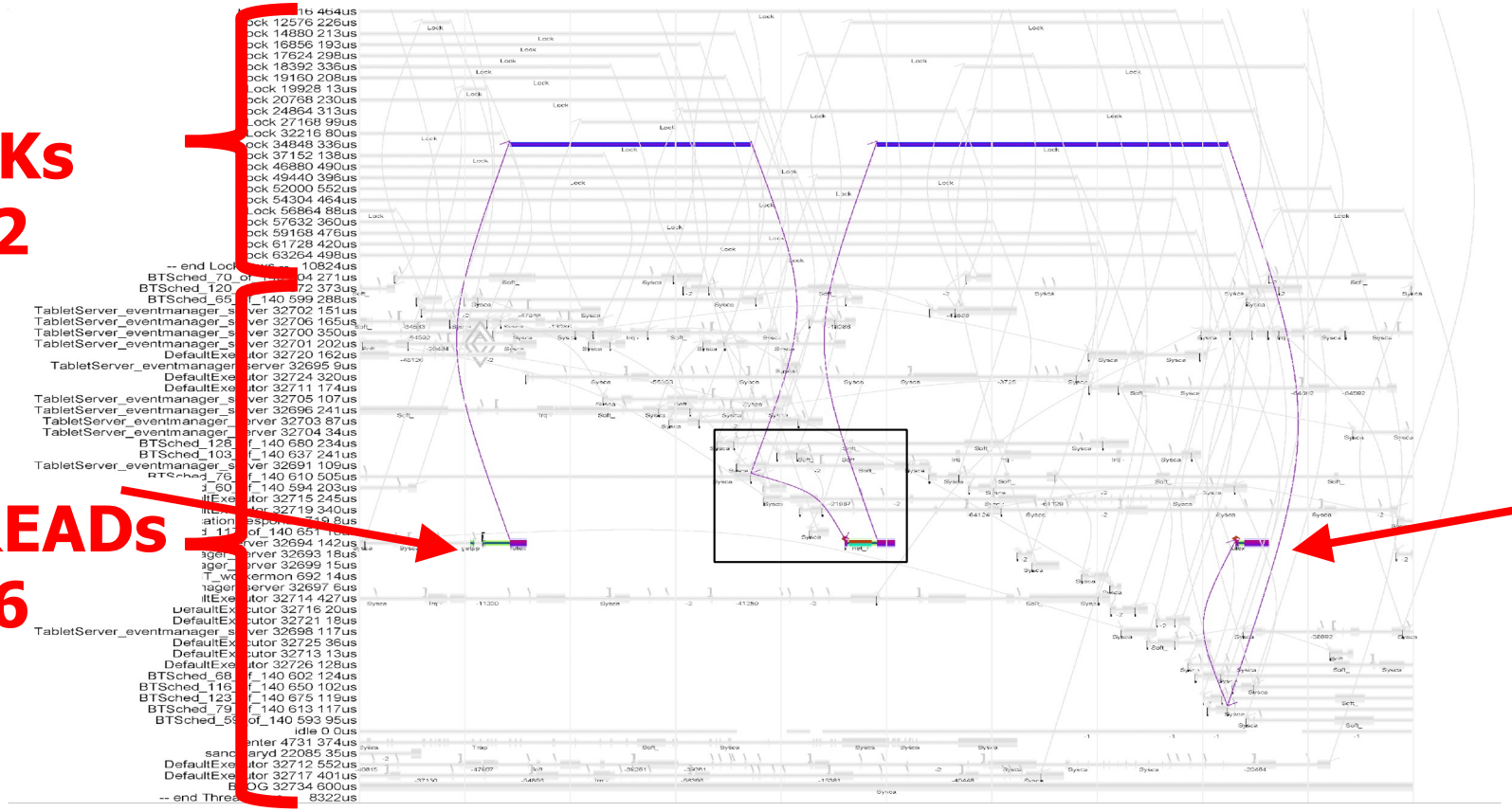
RPCs 0..39



16 CPUs, 600us, one RPC

LOCKs
0..22

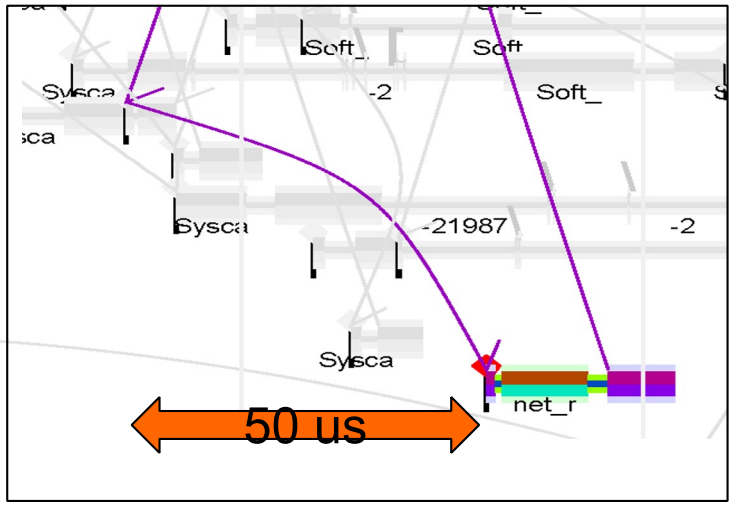
THREADs
0..46



Wakeup Detail

Thread 19 frees lock, sends wakeup to waiting thread 25

Thread 25 actually runs



50us wakeup delay ??

CPU Scheduling, 2 Designs

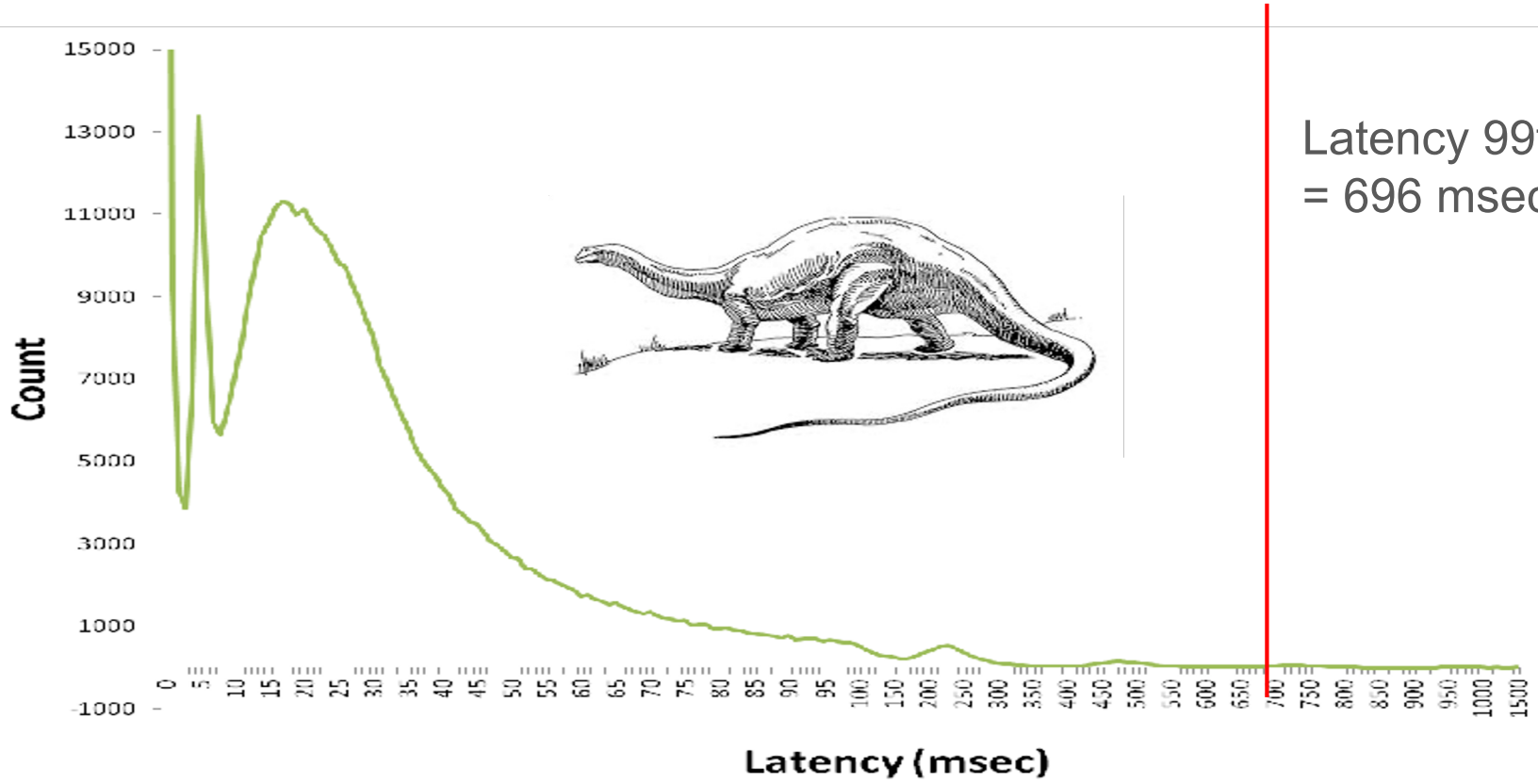
- Re-dispatch on any idle CPU core
  - But if idle CPU core is in deep sleep, can take 75-100us to wake up
- Wait to re-dispatch on previous CPU core, to get cache hits
  - Saves nothing if could use same L1 cache
  - Saves ~10us if could use same L2 cache
  - Saves ~100us if could use same L3 cache
  - Expensive if cross-socket cache refills
  - Don't wait too long…

Real-time transactions: 1000s per second

- Not your father's benchmark or even SPECmarks
- To understand delays, need to track simultaneous transactions across servers, CPU cores, threads, queues, locks

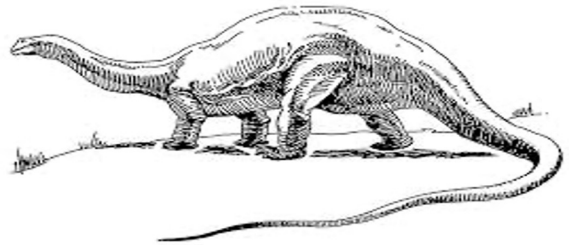
③ Isolation between programs

Histogram: Disk Server Latency; Long Tail



Latency 99th %ile
= 696 msec

Non-repeatable Tail Latency Comes from Unknown Interference



Isolation of programs reduces tail latency.

Reduced tail latency = higher utilization.

Higher utilization = more $$$.

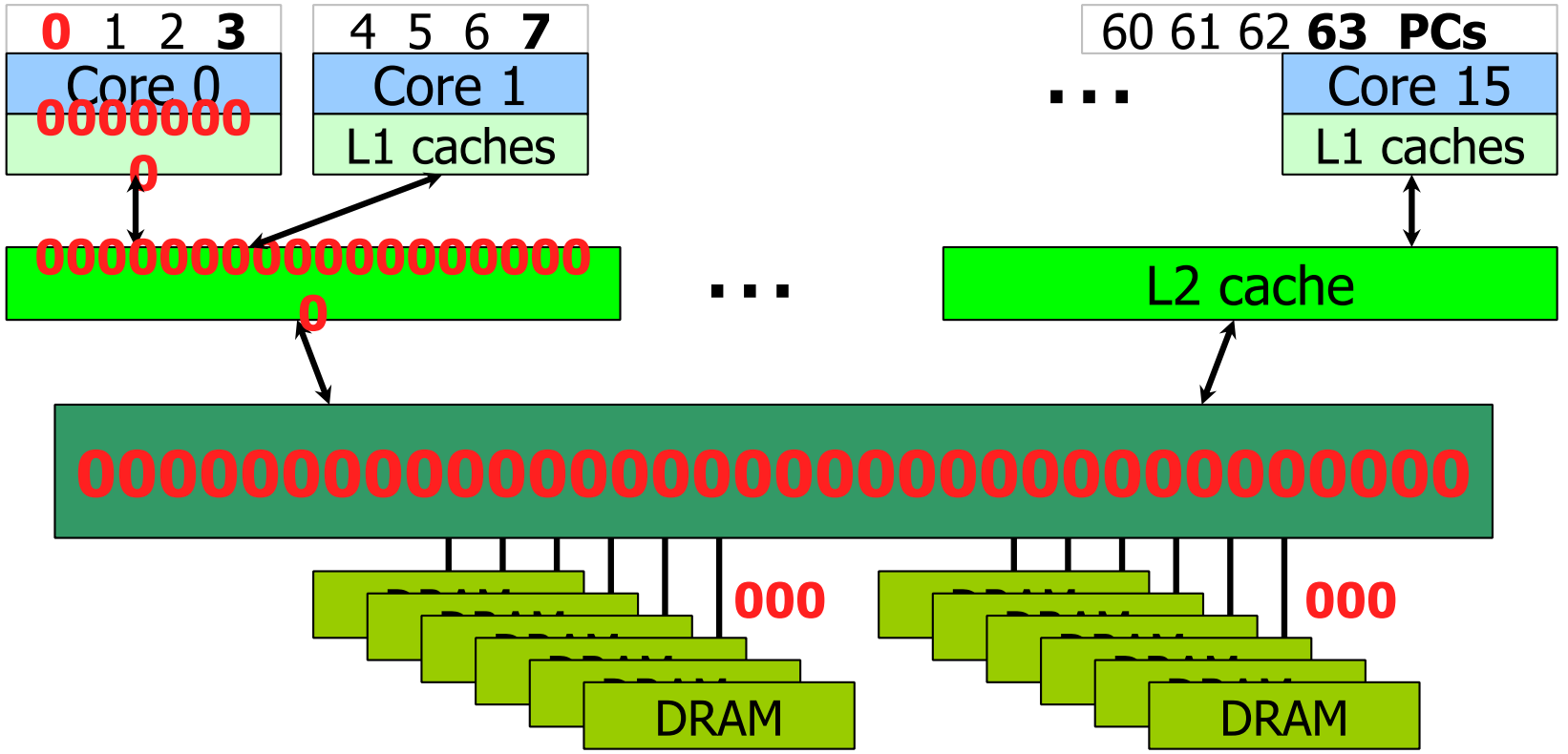
Many Sources of Interference

- Most interference comes from software
- But a bit from the hardware underpinnings

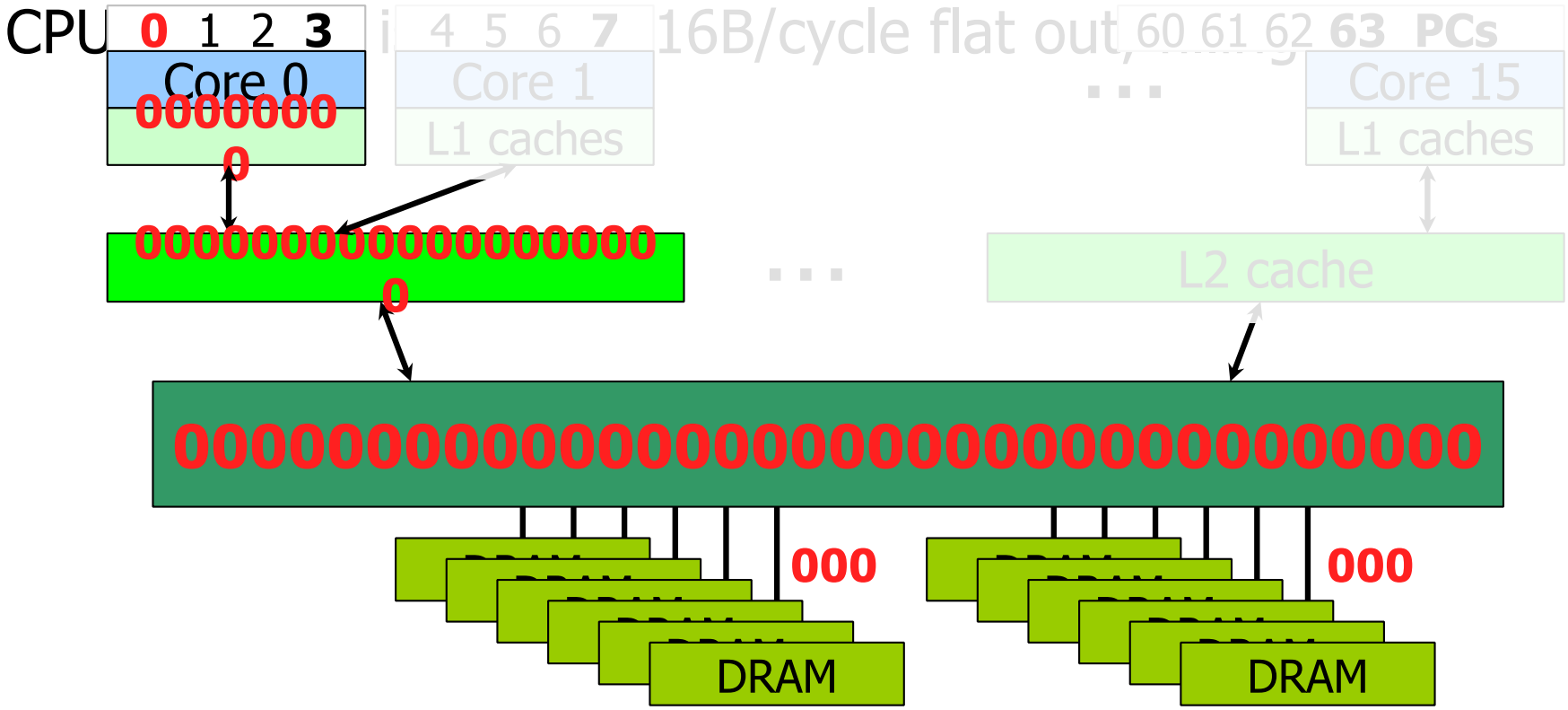
- In a shared apartment building, most interference comes from jerky neighbors
- But thin walls and bad kitchen venting can be the hardware underpinnings

Isolation issue: Cache Interference

CPU thread 0 is moving 16B/cycle flat out, filling caches

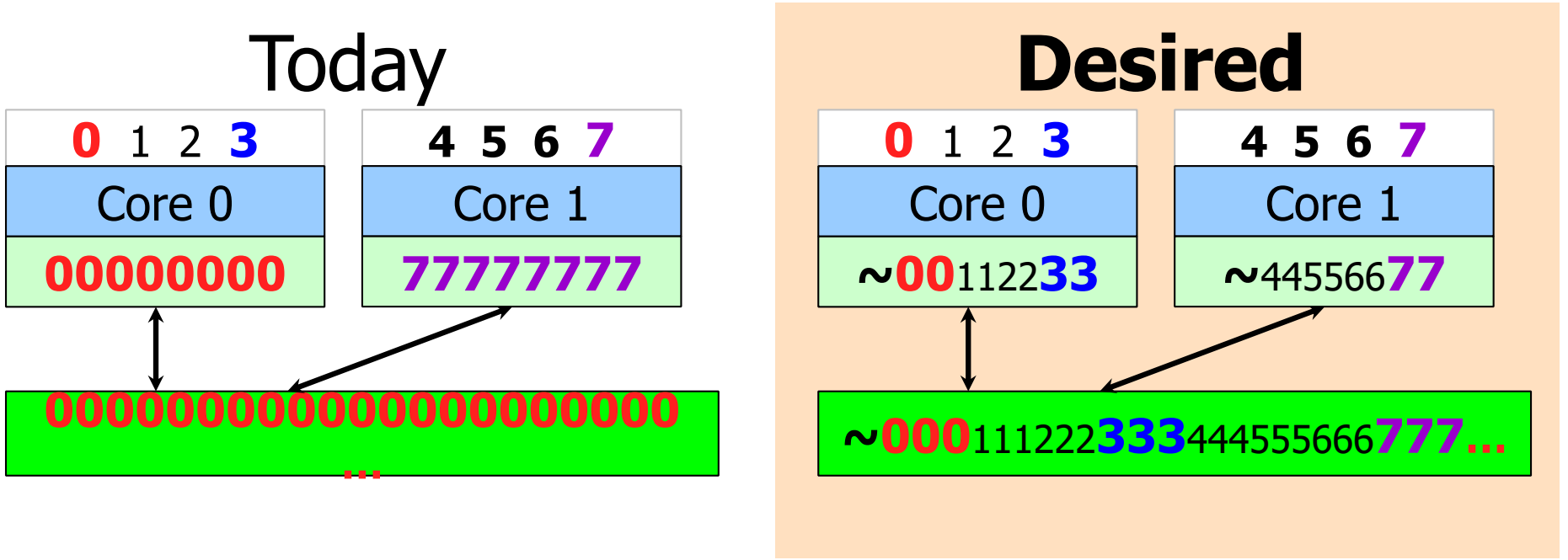


Isolation issue: Cache Interference



Isolation issue: Cache Interference

- CPU thread 0 is moving 16B/cycle flat out



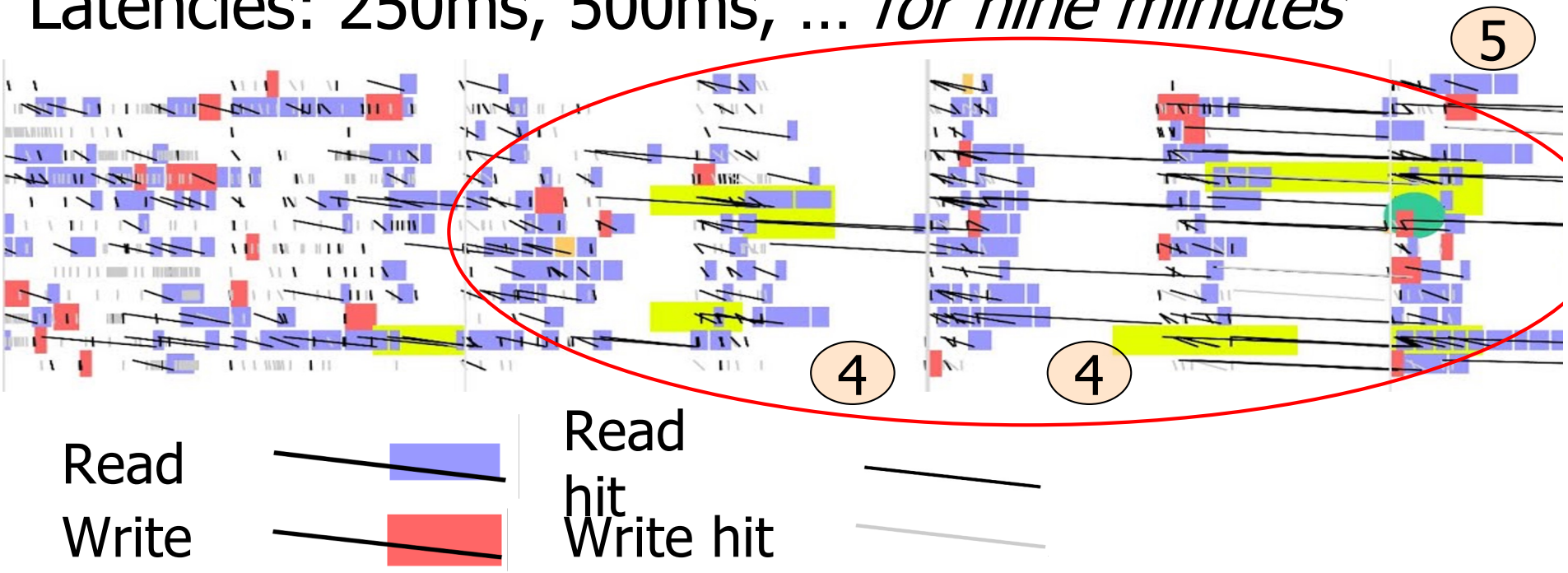
Isolation between programs

- Good fences make good neighbors
- We need better hardware support for program isolation in shared memory systems

④ Measurement underpinnings

Trace: Disk Server, 13 disks, 1.5 sec

- Latencies: 250ms, 500ms, … *for nine minutes*



Why?

- Probably not on your guessing radar…

- Kernel throttling the CPU use of any process that is over purchased quota

- Only happened on old, slow servers

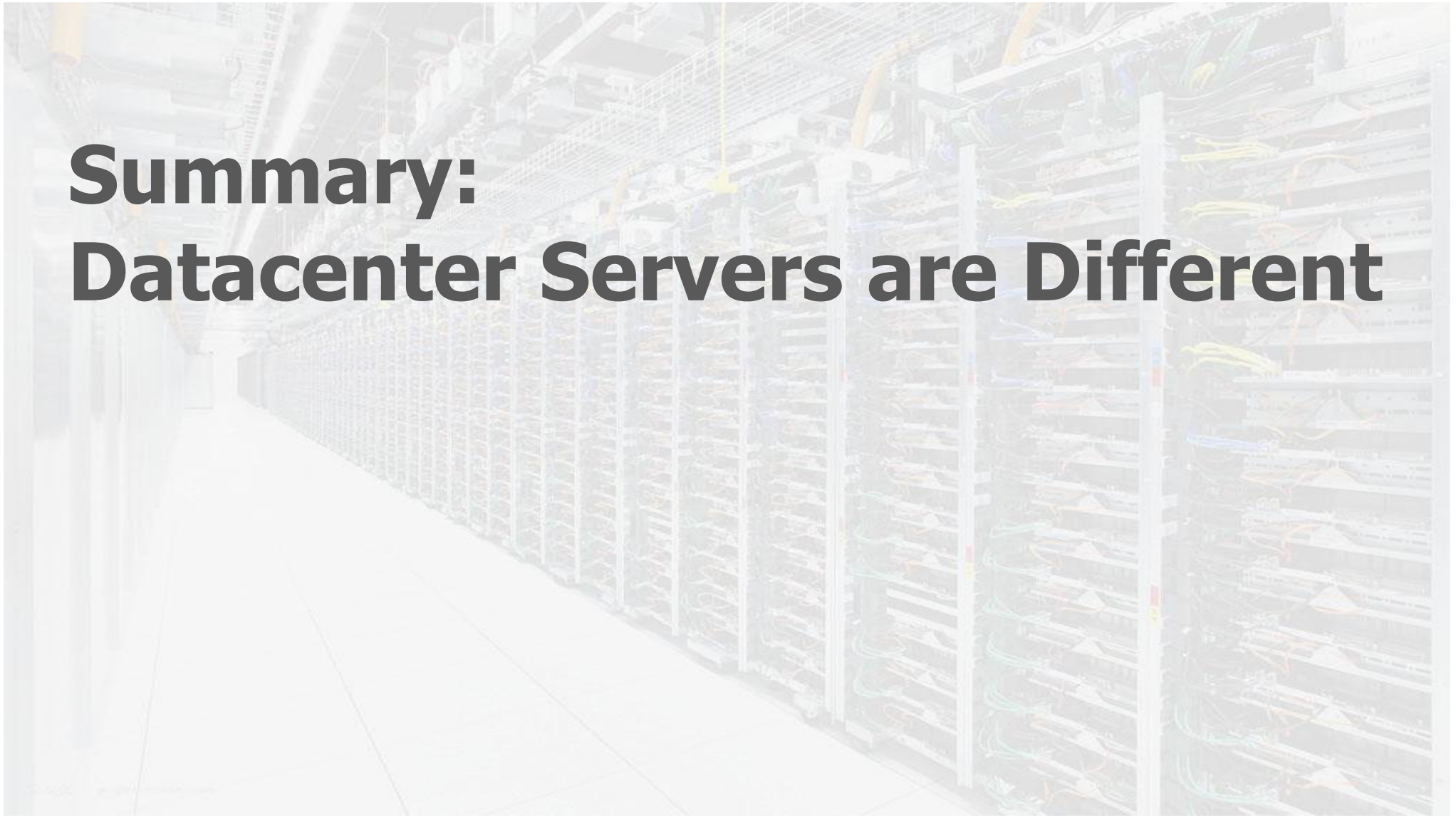
Disk Server, CPU Quota bug

- Understanding Why (4) sped up 25% of entire disk fleet worldwide!
  - Had been going on for three years
  - A few million savings right there
- Hanlon's razor: Never attribute to malice that which is adequately explained by stupidity.
- Dick Sites' corollary: Never attribute to stupidity that which is adequately explained by software complexity.

Measurement Underpinnings

- All performance mysteries are simple once they are understood

- "Mystery" means that the picture in your head is wrong; software engineers are singularly inept at guessing how their view differs from reality



Summary:
Datacenter Servers are Different

Datacenter Servers are Different

① Move data: big and small

② Real-time transactions: 1000s per second

③ Isolation between programs

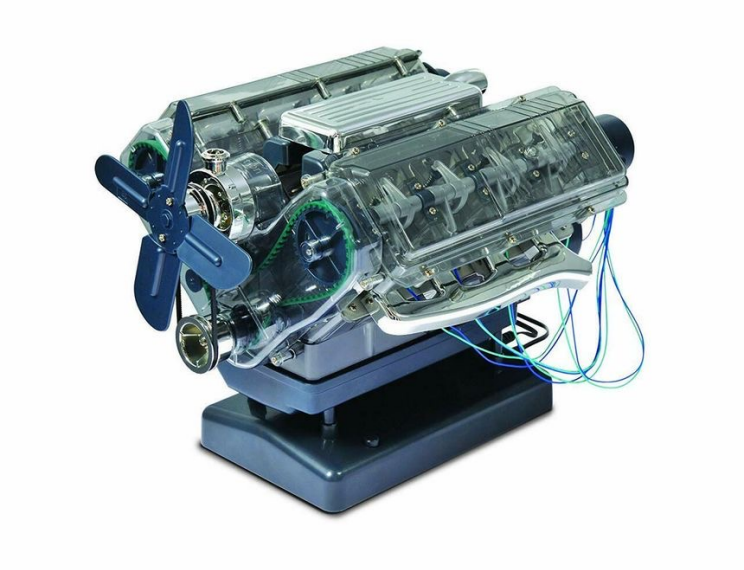
④ Measurement underpinnings

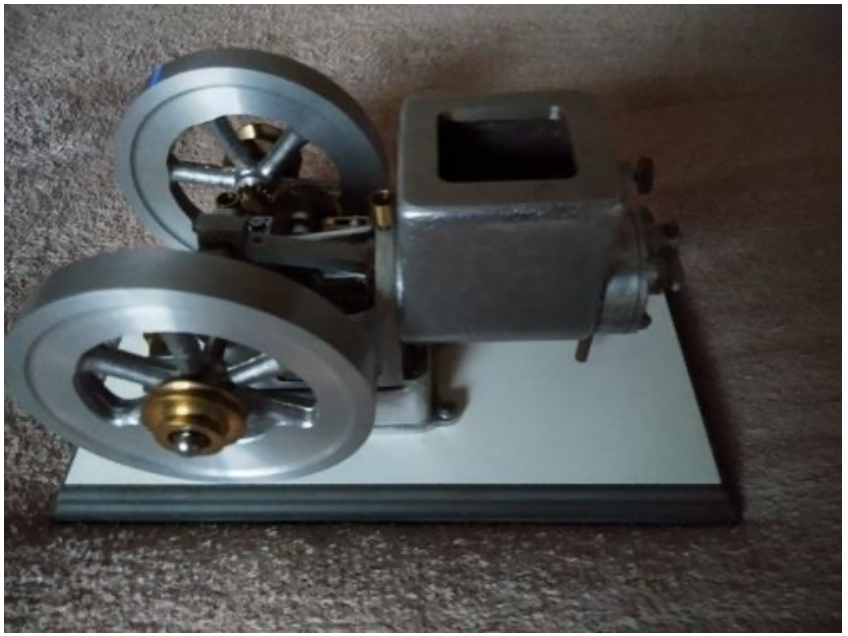
Measure what is measurable, and make measurable what is not so - Galileo Galilei

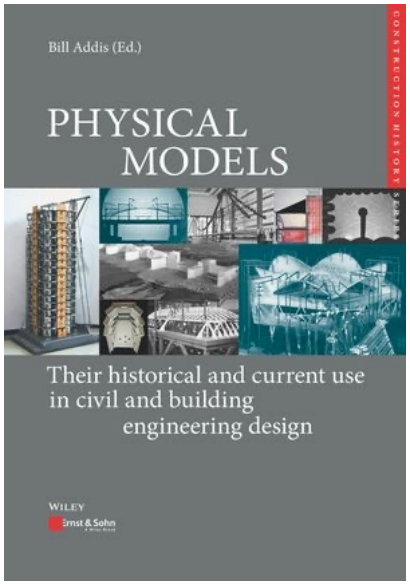
- So… you make an estimate of what you expect, but if you don't measure something – how do you know its too slow?

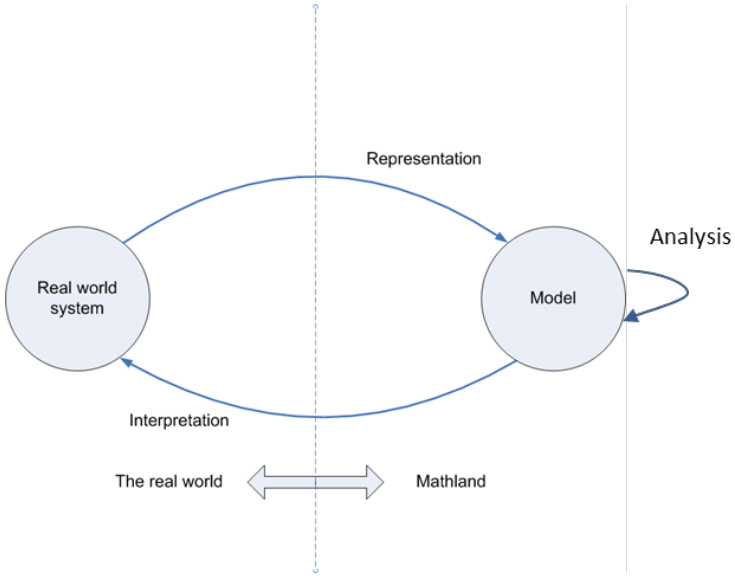
- Measurements are often "over time" – sometimes we want to understand quantities too.
    - How fast is the CPU? – frequency
    - How long will this job take? - seconds
    - How big is the file? – storage size
    - How much power is used? – watts (literally power)

In this process, we make models….



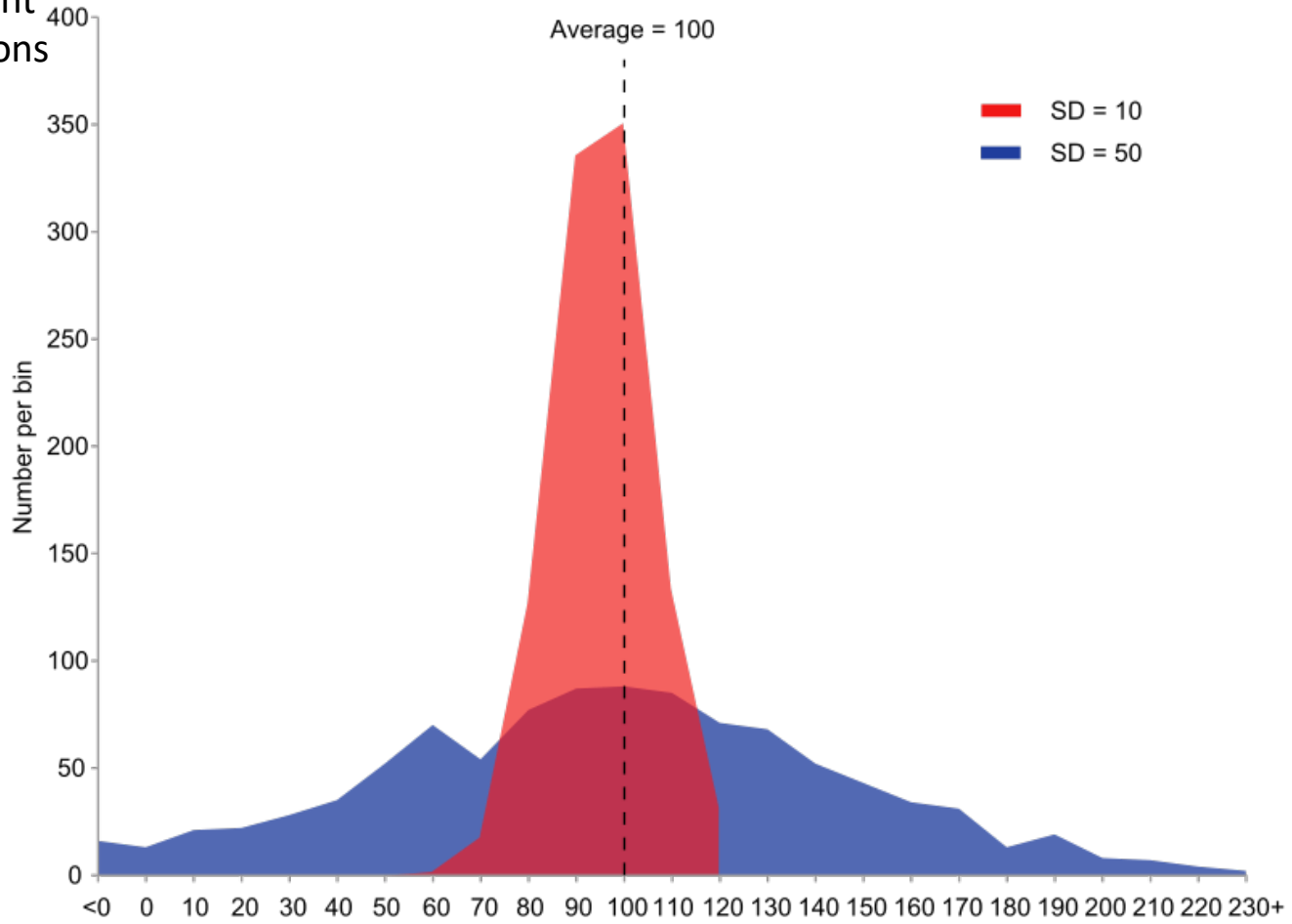






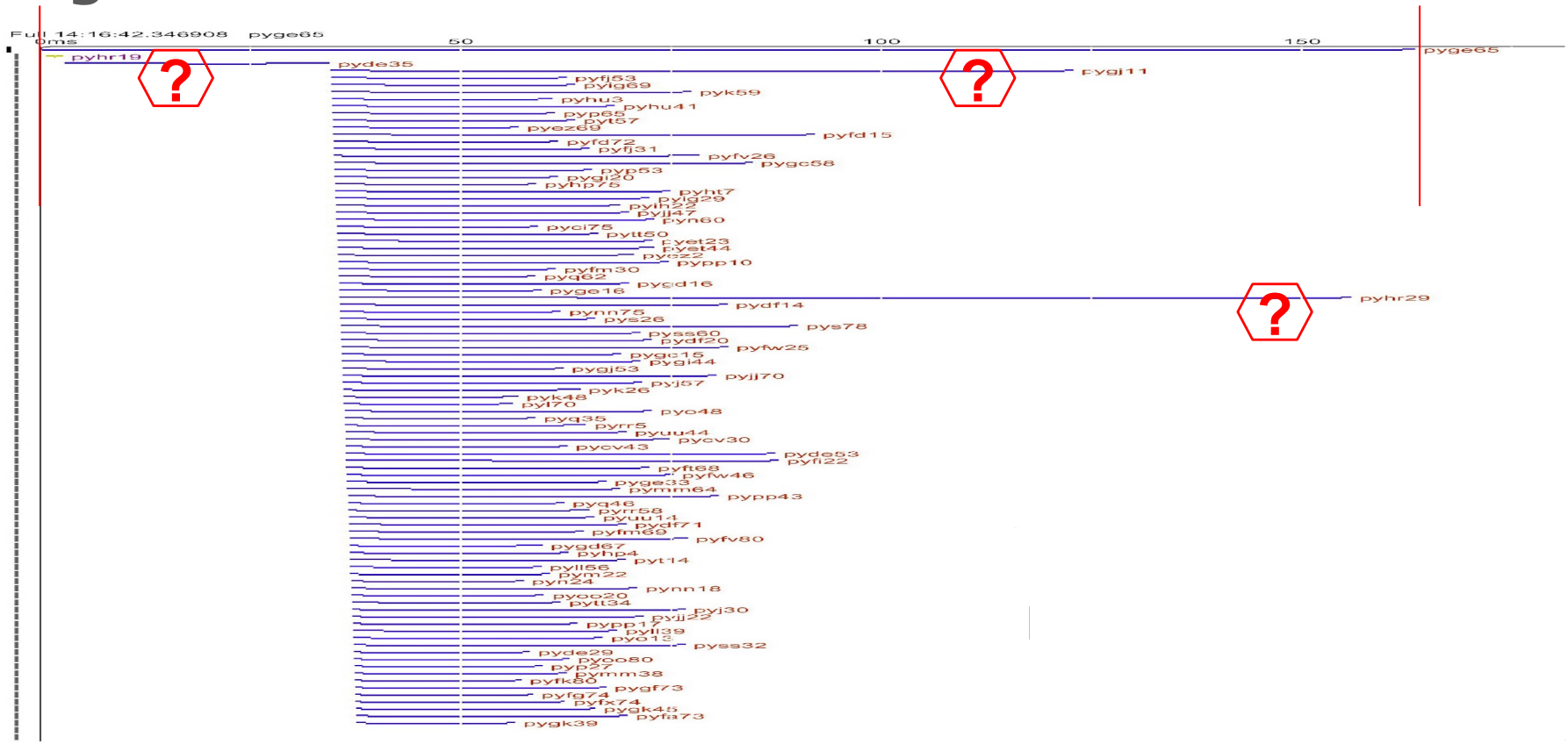
Simple models can oversimplify… (e.g. mean)

Two sets of samples with the same mean and different Standard Deviations



Keep this “*one rogue RPC*” in the back of your mind…

A **Single** Transaction RPC Tree: Client & 93 Servers



Easy case

A particular transaction is slow every time it is run

It is straightforward to run the transaction repeatedly offline on a few load-test servers

Existing profiling tools, disk byte counts, network byte counts, etc. will reveal where the time goes

"Interesting" case

A particular transaction is usually fast, but occasionally quite slow

It is only slow under live load during the busiest hour of the day

Running it again is fast

Until the *reason* for slowness is found, we cannot reproduce the problem offline

Existing tools are unable to reveal where the time goes

"Interesting" case -- why it matters

Until the *reason* for slowness is found, we cannot reproduce the problem offline

At 10,000 transactions/second and no call tree, the 99th percentile slow cases happen 100 times per second

But, for software with 100:1 fanout transaction call trees, *almost everything runs at the 99th percentile slow rate*

Existing tools are unable to reveal where the time goes

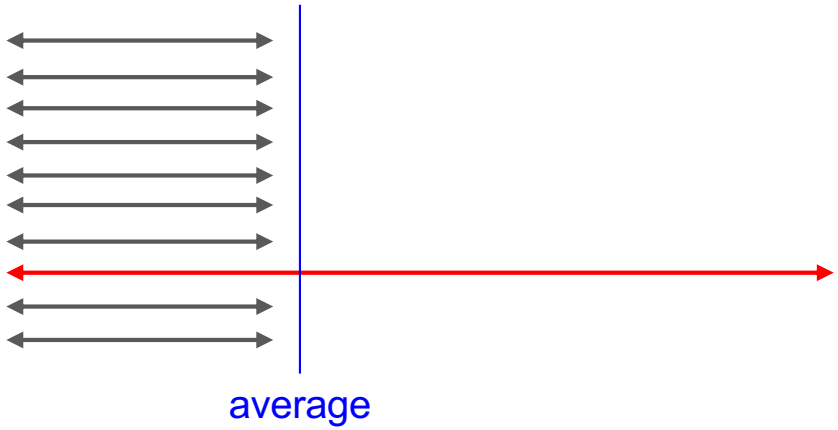
Tail latency

Some transactions, total latency

We seek to understand why the one is slow

Tail latency

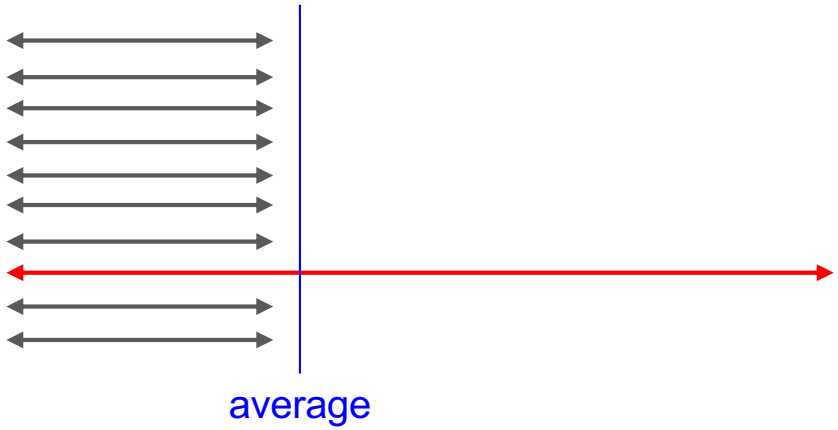
Possible tool: average latency



Tells us **nothing** about the slow one

Tail latency

Possible tool: average latency

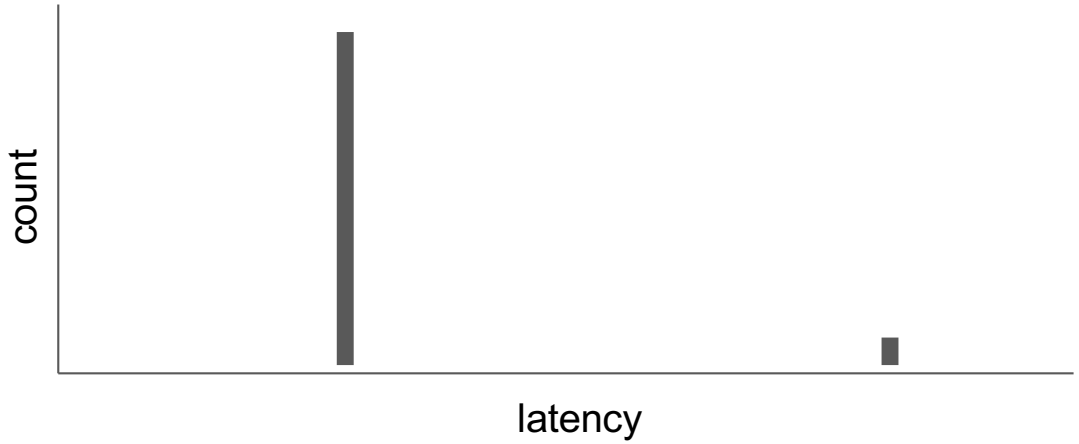


Tells us **nothing** about the slow one

Average is the wrong tool for understanding variance

Tail latency

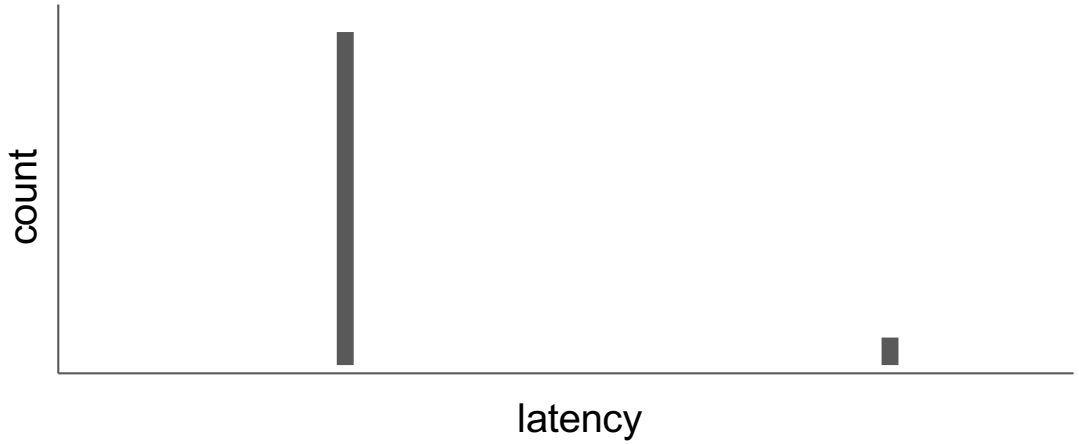
Possible tool: latency histogram



Tells us **what** we have but not **why**

Tail latency

Possible tool: latency histogram

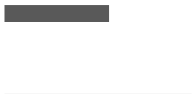


Tells us **what** we have but not **why**

Histogram is the wrong tool for understanding variance

Tail latency

Possible tool: Profile of CPU time per source function

gather_inputs()	
process_item()	
process_more()	
calculate()	
produce_output()	

But where is the slow transaction?

Tail latency

Possible tool: Profile of CPU time per source function

gather_inputs()	
process_item()	
process_more()	
calculate()	
produce_output()	

But where is the slow transaction?

Profiling is the wrong tool for understanding variance

Profiling: the wrong tool for understanding variance

Possible tool: Profile of CPU time per source function

gather_inputs()	
process_item()	
process_more()	
calculate()	
produce_output()	

It merges together many normal transactions with a few slow ones:
hiding the 1% signal in 99% noise

Tail latency

Possible tool: Profile of CPU time per source function

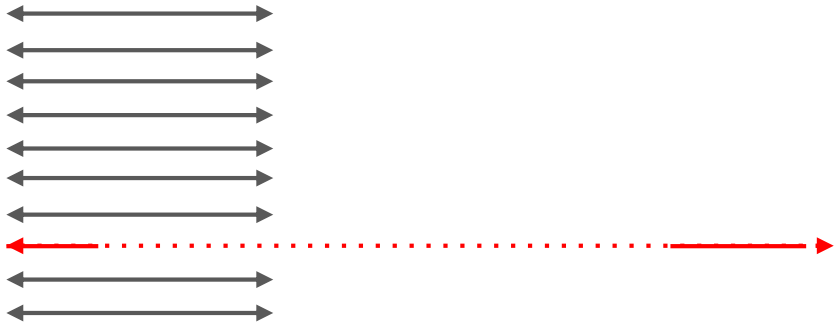
gather_inputs()	
process_item()	
process_more()	
calculate()	
produce_output()	

But where is the slow transaction?

CPU Profiling is the wrong tool for a second reason ...

CPU Profiling: the wrong tool for a second reason

Some transactions, total time



The delay may not be using CPU time at all; it may be *waiting* for something. CPU profiling is *blind* to non-CPU wait time

Tail latency

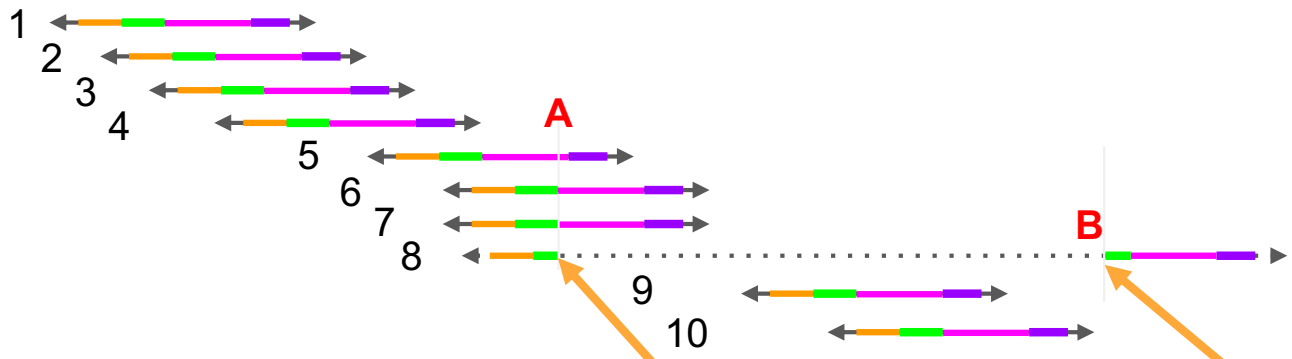
Possible tool: **trace** events for each transaction

Now we can see **what** is different about the slow one.

But we don't know **why** it is different

Tail latency

Possible tool: trace events for each transaction, lined up in time

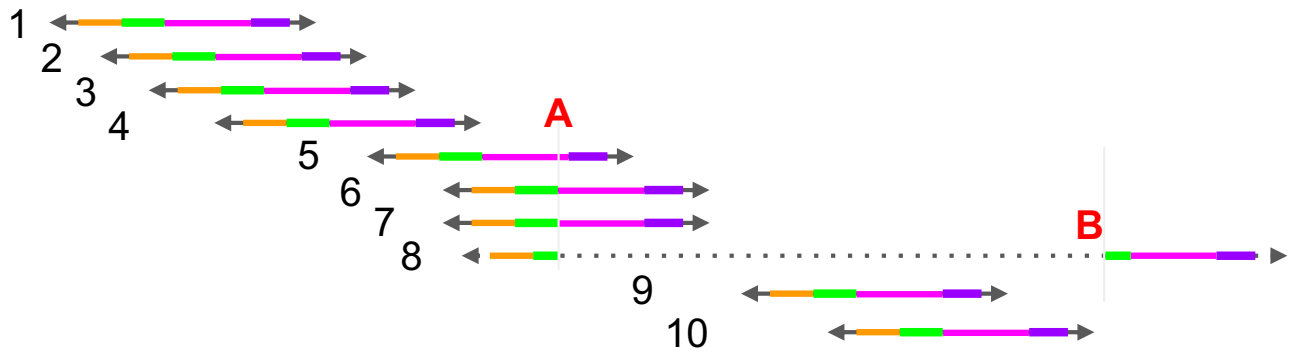


Now we can see **what** is different about the slow one.

And the time alignment of 6's end-of-green (A) with 8's green blocking until B.

Tail latency

Possible tool: trace events for each transaction, lined up in time



More detailed events at A or B will reveal the reason for starting or stopping blocking. (Possibly a race condition or a contended software lock.)

Tail latency, summary

Average is the wrong tool for understanding variance

Histogram is the wrong tool for understanding variance

Profile is the wrong tool for understanding variance

Event tracing is a good tool for understanding variance

Event tracing lined up in time is a great tool for understanding variance --
it can show causes of delay directly

Measuring CPU time

The Four Fundamental Hardware Resources

CPU

Memory

Disk/SSD

Network

Five

The ~~Four~~ Fundamental ~~Hardware~~ Resources

CPU

Memory

Disk/SSD

Network

Critical software sections (locks)

Measuring CPU time

Time base

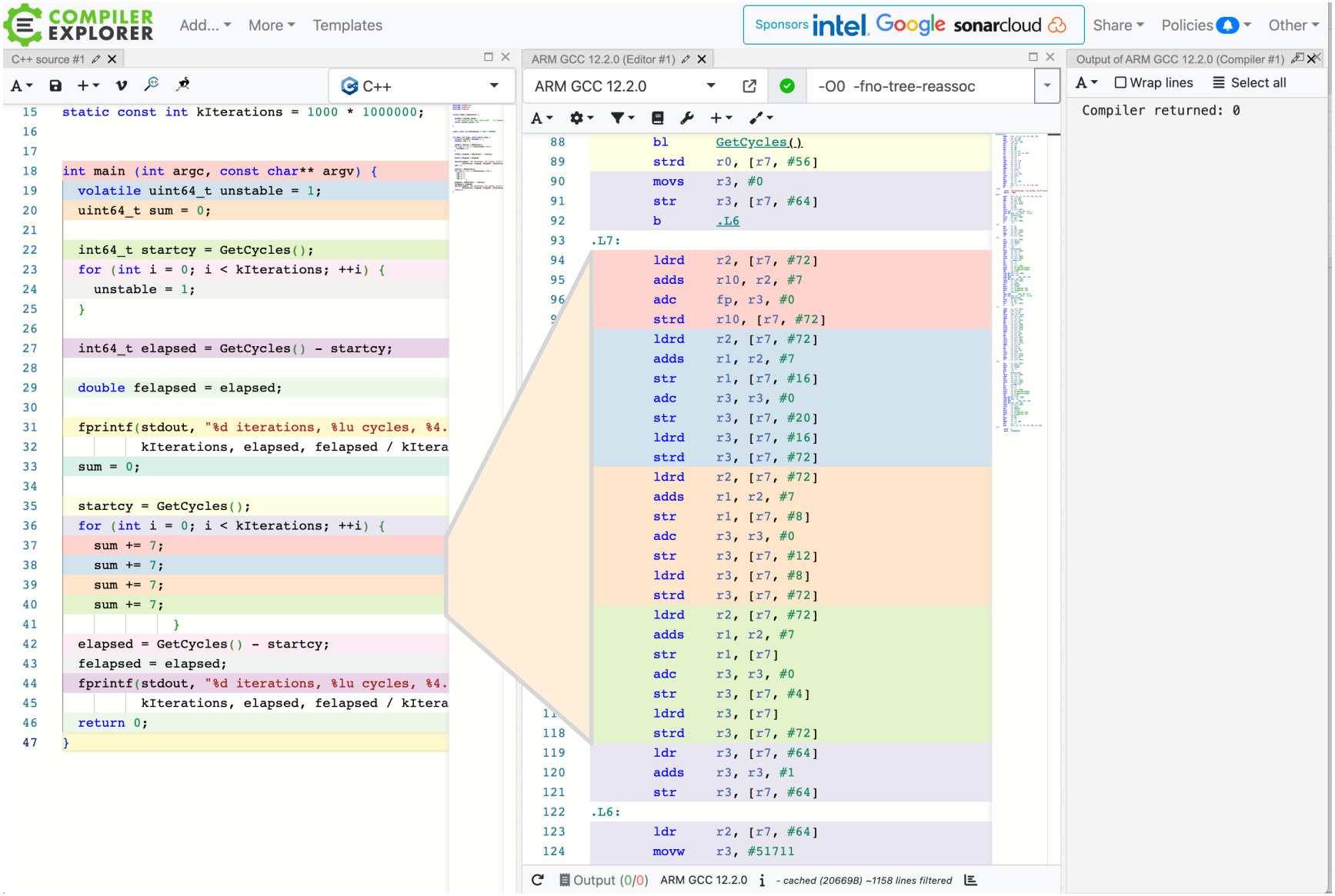
We will use two different time bases in this module

- CPU cycle counter, accessed via the RDTSC instruction
- gettimeofday() system call

The first is fast and has resolution on the order of 1 nsec, but cycle counters on different machines have different values and different increment rates

The second is slower, can have a resolution of 1 usec, with *nearly-same* times across multiple machines

An interestng tool: Compiler Explorer https://godbolt.org/



From here on, **do not measure performance of unoptimized code** -- too much of the time is wasted on things that should not be there, and the results are unrelated to deployed, production, optimized code. Mostly measures memory access time!

Memory timing, caches

The Four Fundamental Resources

CPU

Memory ☜

Disk/SSD

Network

This week: Memory timing, problems to avoid

Modern CPUs: Multiple loads outstanding
  Defeat by making second depend on first's loaded value

Modern CPUs: Cache prefetching of line N+1 when N is accessed
  Defeat by making cache line access pattern irregular

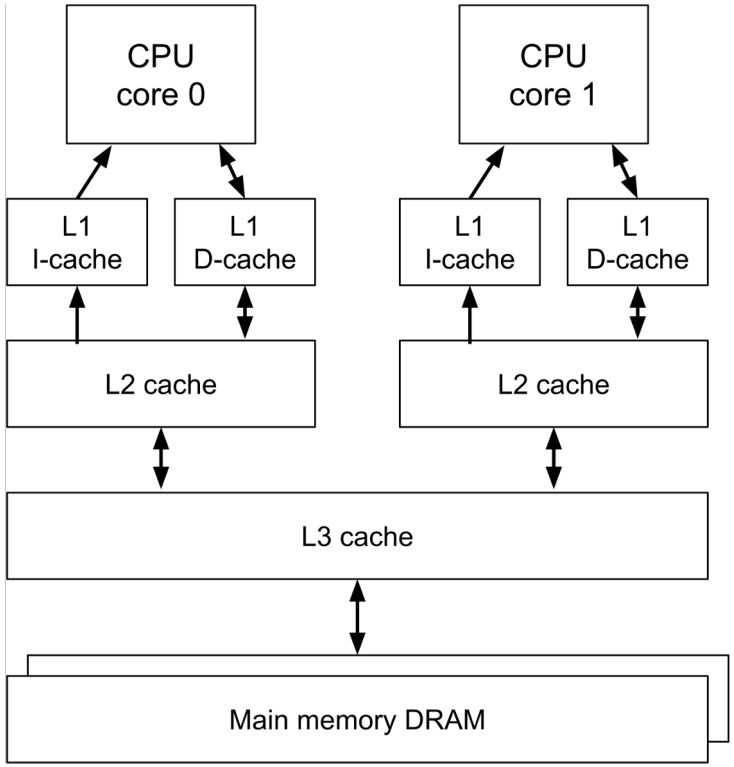
Modern DRAM: Not random: nearby references to previous are 3x faster
  Defeat by deliberately going far enough away in sequences

Reference:

Structural aspects of the System/360 model 85: the cache (1968) J. S. Liptay
https://people.cs.umass.edu/~emery/classes/cmpsci691st/readings/Arch/liptay68.pdf

Maurice Wilkes (Cambridge) original cache paper (1964)
https://www.cl.cam.ac.uk/teaching/current/P56/files/wilkes1965cache.pdf
(TW: deprecated language)

Example cache hierarchy



4KB..32KB each

64KB..1MB each

1MB..32MB+

64GB..1TB+

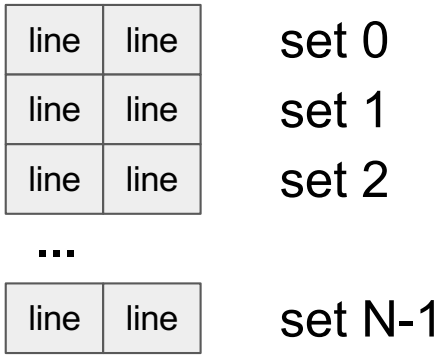
Cache line

Tag has the (physical) address of the data

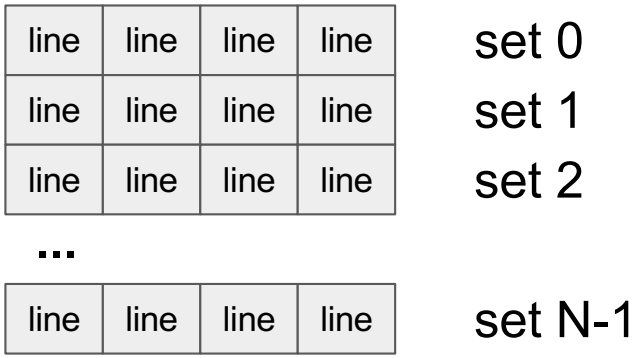
Line contains the data, usually read or written by the CPU 8 or 16 bytes at a time

Typical cache setup

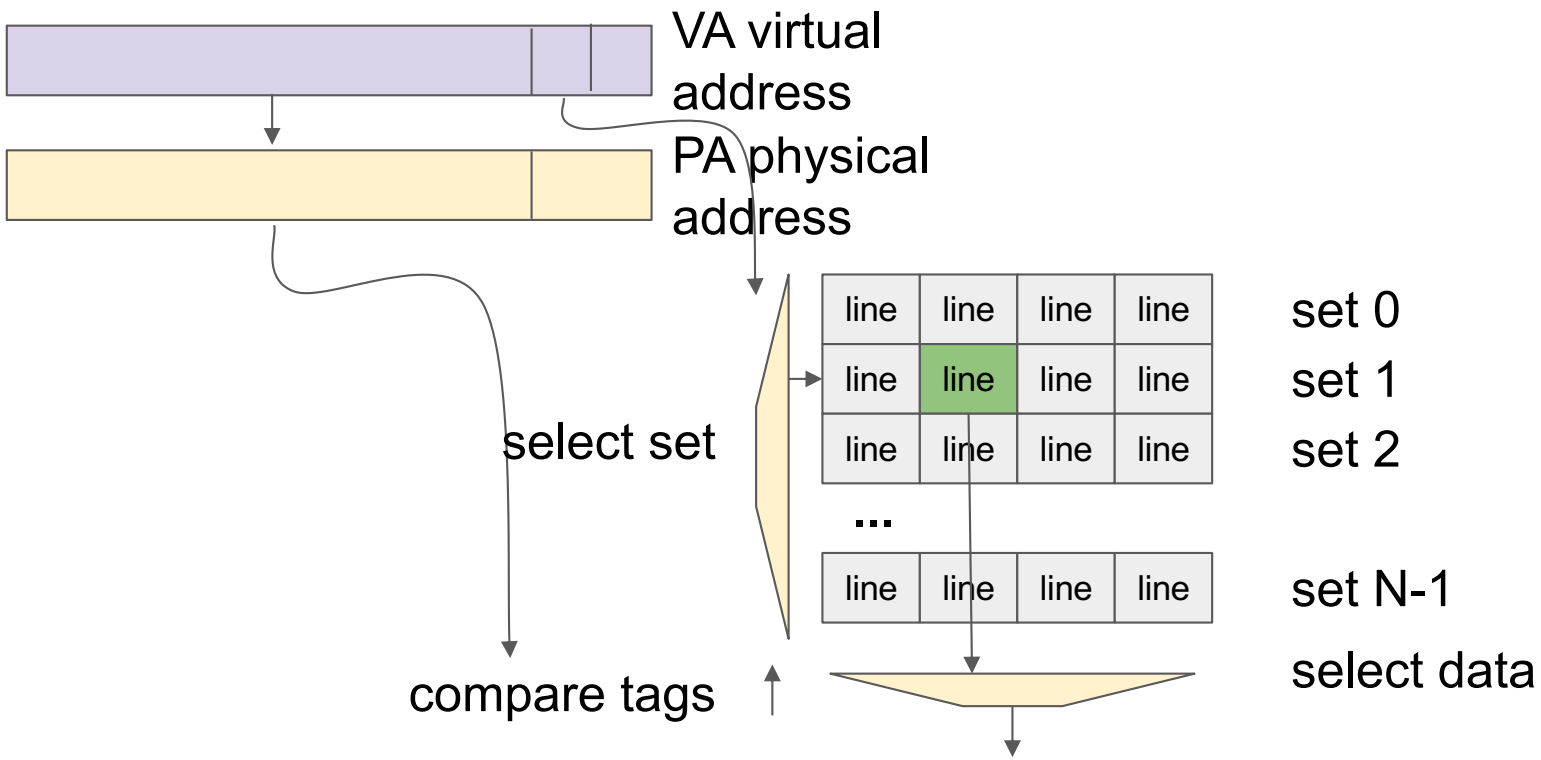
Two-way associative:
every cache line can be in
either of two places in a set



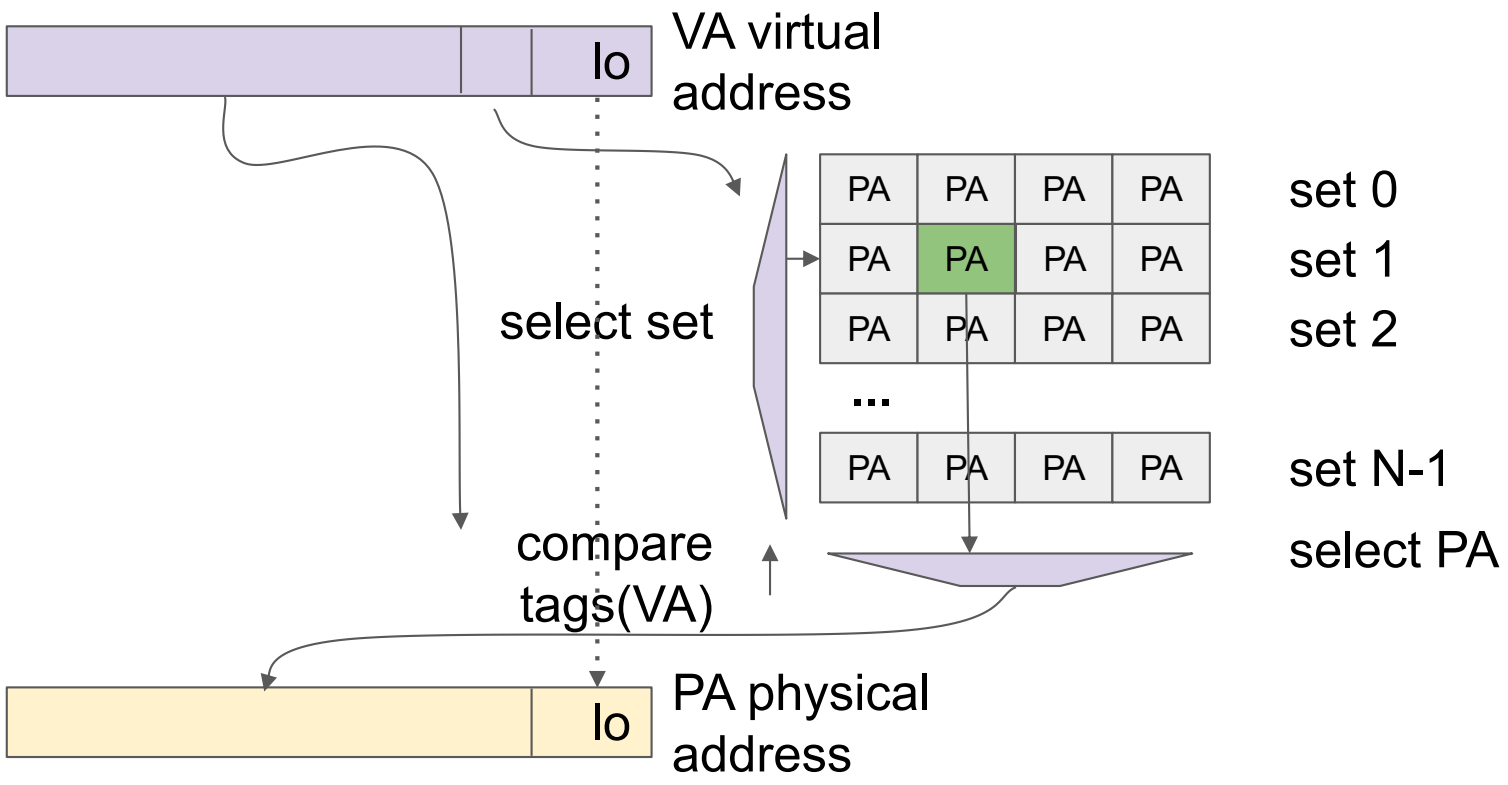
Four-way associative



Typical cache setup



Translation Lookaside Buffer, TLB



Memory timing, cache line size

Memory timing: How big is one cache line?

Strategy: Build a linked list of items of sizes 16, 32, 64, etc. bytes. Trash the caches. Access the first N=256 linked list items.

If item is >= cache line size, each will be a cache miss all the way out to main memory, O(100-200 cycles).

If item is < cache line size, 1/2, 3/4, etc. will be cache hits because some earlier access brought many of them in.

Issues:

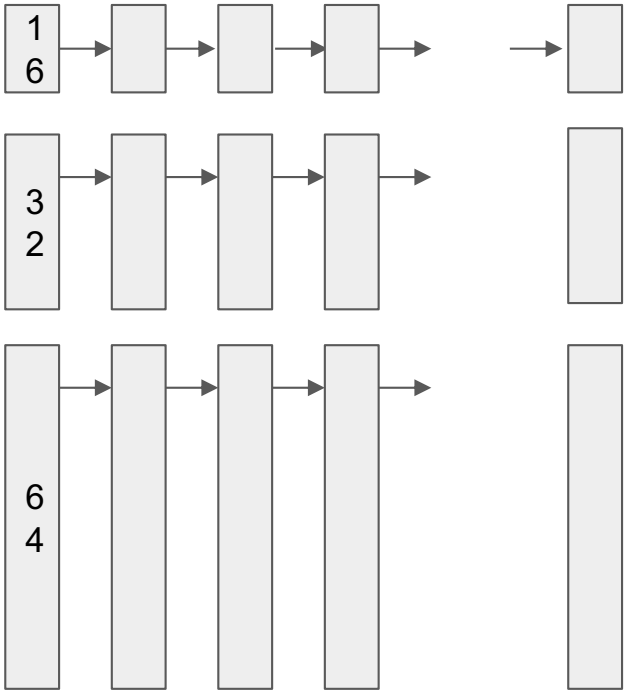
How to empty the cache.

How to make sure misses are worst-case slow.

Memory timing: How big is one cache line?

Strategy: Build a linked list of items of sizes 16, 32, 64, etc. bytes.

Access the first N=256



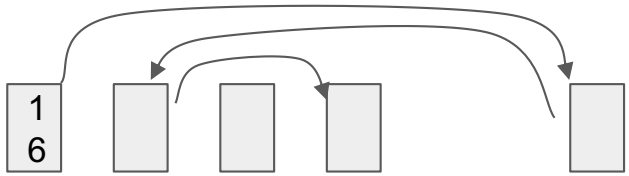
If item is >= cache line size, each will be a cache miss all the way out to main memory, O(100-200 cycles).

If item is < cache line size, 1/2, 3/4, etc. will be cache hits because some earlier access brought many of them in.

Memory timing: How big is one cache line?

N+1 prefetching may ruin the previus strategy

Strategy 2: Linked list with irregular access pattern



How to empty the cache?

Assume 64B block, four sets, two-way associative, **True LRU** replacement

0	256	set 0
64	320	set 1
128	384	set 2
192	448	set 3

Loading from 0, 64, 128, 192 will fill one entry in each set.
Loading from 256, 320, 384, 448 will fill the second entry in each set.
All eight blocks overwritten.

How to empty the cache?

Assume 64B block, four sets, two-way associative, **Random** replacement

0	256
320	?
384	?
192	448

Loading from 0, 64, 128, 192 will fill one entry in each set.
Loading from 256, 320, 384, 448 will fill the second entry in each set half the time and **overwrite the first entry** half the time.

On average, only 6 blocks overwritten, 2 left untouched.

How to empty the cache?

Need to load substantially more than the original stuff you want to evict.
May need to overwrite several times to increase odds of eviction
No one builds a true LRU 8-way associative cache: 40,320 combinations

Memory timing, cache total size

Memory timing: How big is each cache level?

Strategy: Build a linked list of items that are linesize bytes. Trash the caches. Access the first N=16, 32, ..., 512K linked list items. Then access again, 3 times.

[512K items at 64 bytes each = 32MB, the largest cache size we expect.]

If all N items fit in the cache, the first pass will miss a lot, but after that the other three passes will hit.

If the N items total more the the cache size, the first pass will miss a lot and only leave 1/2, 1/4, etc. of the items still in the cache. The other three passes will get 1/2, 3/4 etc. misses, or more depending on the replacement algorithm.

Memory timing: Associativity of each cache level?

If the cache is 1MB 1-way associative (i.e. direct-mapped), it is 1MB x 1
If the cache is 1MB 2-way associative, it is 512K x 2 ... 256K x 4, etc.

If the cache set is determined by low address bits, addresses 1MB apart are all mapped to the same set

Access 1, 2, 3, etc. addresses 1MB apart inside a loop of perhaps 1000 times.
If the cache is in fact 4-way associative, 1,2,3,4 addresses will get nearly 100% hits, while 5+ addresses will start missing part of the time. Overall time will differ.

Memory timing, cache associativity

Memory timing: Associativity of each cache level?

Strategy: Build a linked list of items that are N bytes apart, where N is the total cache size. Cycle the associativity A through about twice the likely associatively possibilities: A = 1, 2, 3, 4, 5, ... maybe 40.

Trash the caches. Access the first A linked list items. Then access again, ~1000 times.

If the actual associativity is >= A, the first pass will leave every item in the cache and the other passes will all get hits.

If the actual associativity is < A, the first pass will leave only some items in the cache and the other passes will get some misses.

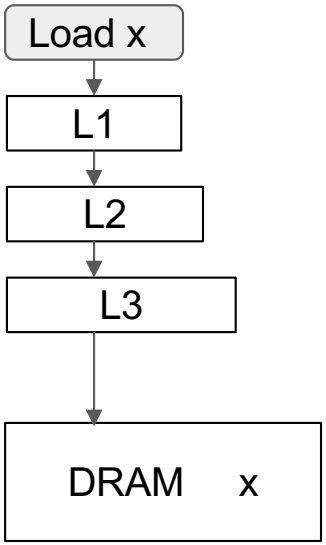
Memory timing, DRAM

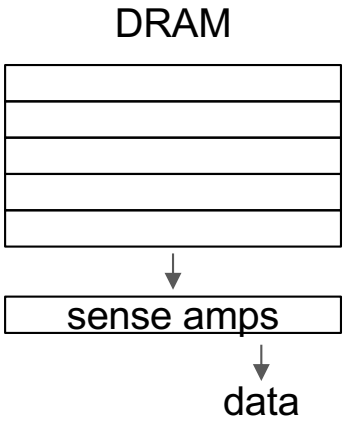
Dynamic Random Access Memory (DRAM)

Not actually particularly random!

Access speed depends on prior accesses

How to make sure misses are worst-case slow?



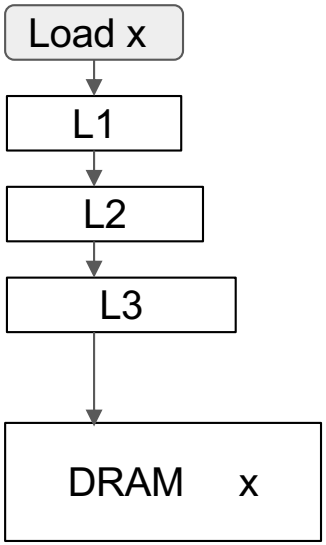


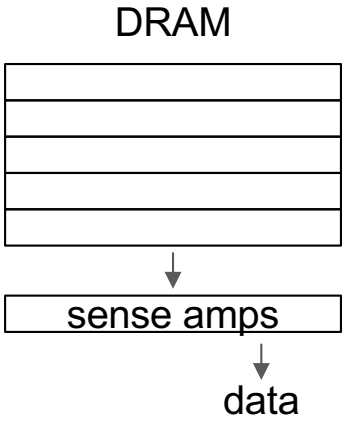
Precharge 15ns
row 0
row 1
...

Row select RAS 15ns

Column select CAS 15ns
Total access time: 45ns

How to make sure misses are worst-case slow?





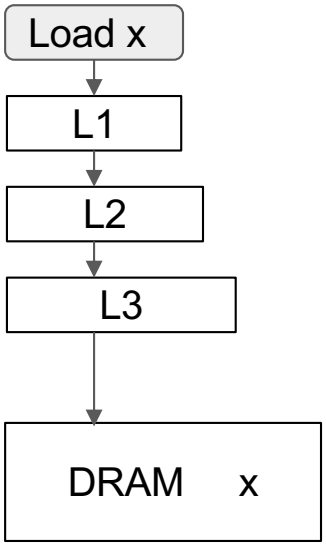
Precharge 15ns
row 0
row 1
...

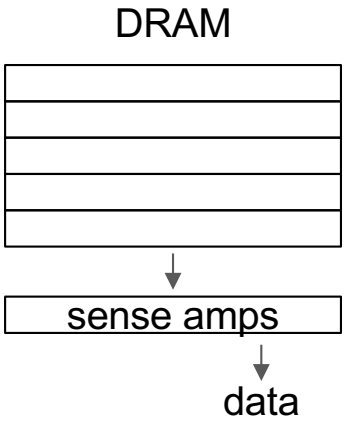
Row select RAS 15ns

Column select CAS 15ns
Total access time: 45ns

BUT, if next data Y is in the open row, just CAS: **15ns**

How to make sure misses are worst-case slow?





Precharge 15ns
row 0
row 1
...

Row select RAS 15ns

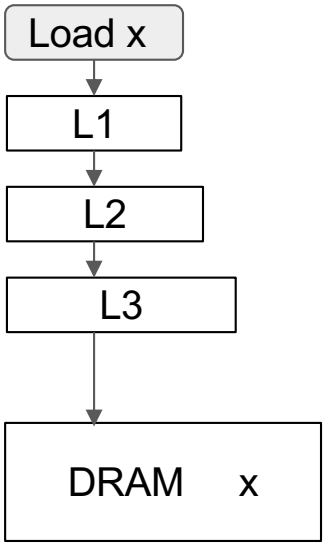
Column select CAS 15ns
Total access time: 45ns

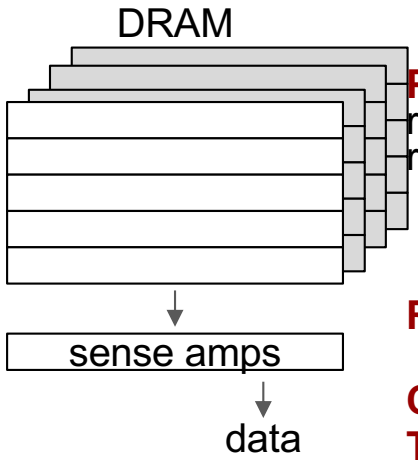
Row = 1KB, typical

Eight DRAM chips together for 64-bit bus = 8KB row

Make Y at least 8KB away to force different row

How to make sure misses are worst-case slow?





Precharge 15ns

Row select RAS 15ns

Column select CAS 15ns

Total access time: 45ns

Row = 1KB, typical

Eight DRAM chips together for 64-bit bus = 8KB row

Four interleaved memory cards = 32KB row

Make Y at least 32KB away to force different row

Memory timing, cache line size

Cache line size

Assume for a moment a 32-byte cache line, an empty cache, and A = an address at the start of a cache line.

Load A, A + 8, A + 16, ... A+72 (ten loads, stride 8)

How many cache misses will we get?

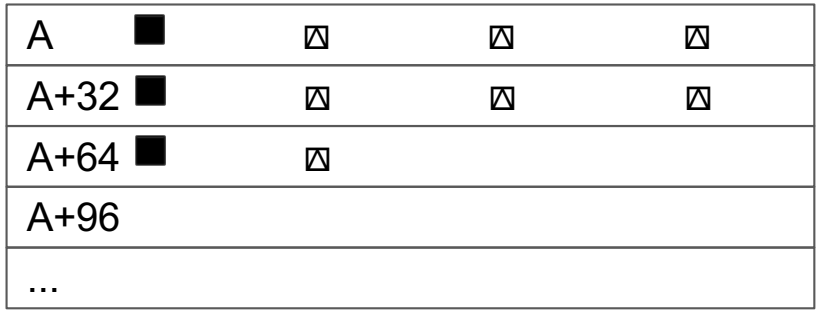
A
A+32
A+64
A+96
...

Cache line size

Assume for a moment a 32-byte cache line, an empty cache, and A = an address at the start of a cache line.

Load A, A + 8, A + 16, ... A+72 (ten loads)

How many cache misses will we get? Three



Cache line size

Assume for a moment a 32-byte cache line, an empty cache, and A = an address at the start of a cache line.

Load A, A + 16, A + 32, ... A+144 (ten loads, stride 16)

How many cache misses will we get? Five

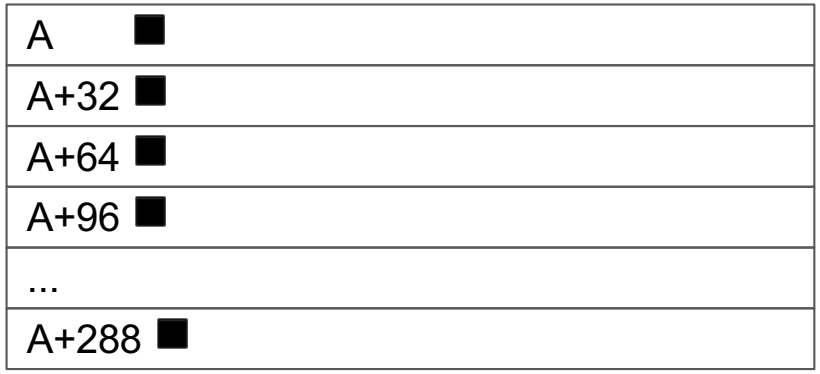
A	■	⍰
A+32	■	⍰
A+64	■	⍰
A+96	■	⍰
A+128	■	⍰

Cache line size

Assume for a moment a 32-byte cache line, an empty cache, and A = an address at the start of a cache line.

Load A, A + 32, A + 64, ... A+288 (ten loads, stride 32)

How many cache misses will we get? Ten

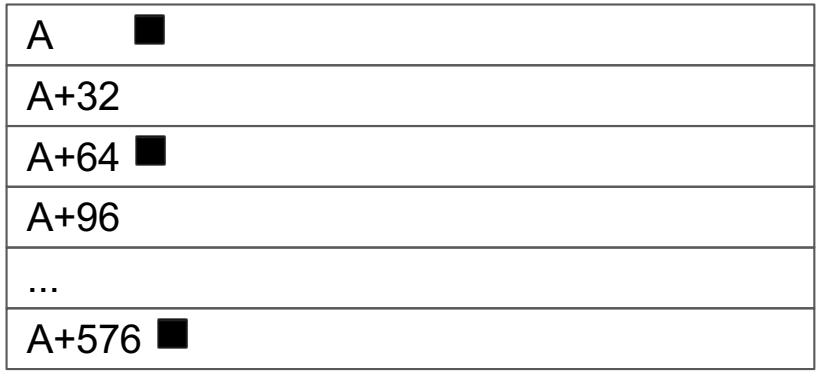


Cache line size

Assume for a moment a 32-byte cache line, an empty cache, and A = an address at the start of a cache line.

Load A, A + 64, A + 128, ... A+576 (ten loads, stride 64)

How many cache misses will we get? **Ten, and ten for 128, 256, ...**



Cache line size

Assume for a moment a 32-byte cache line, an empty cache, and A = an address at the start of a cache line.

Load A, A + n, A + 2n, ... A+9n (ten loads)

How many cache misses will we get?

Stride	**Misses**
8	3
16	5
32	10
64	10
128...	10 ...

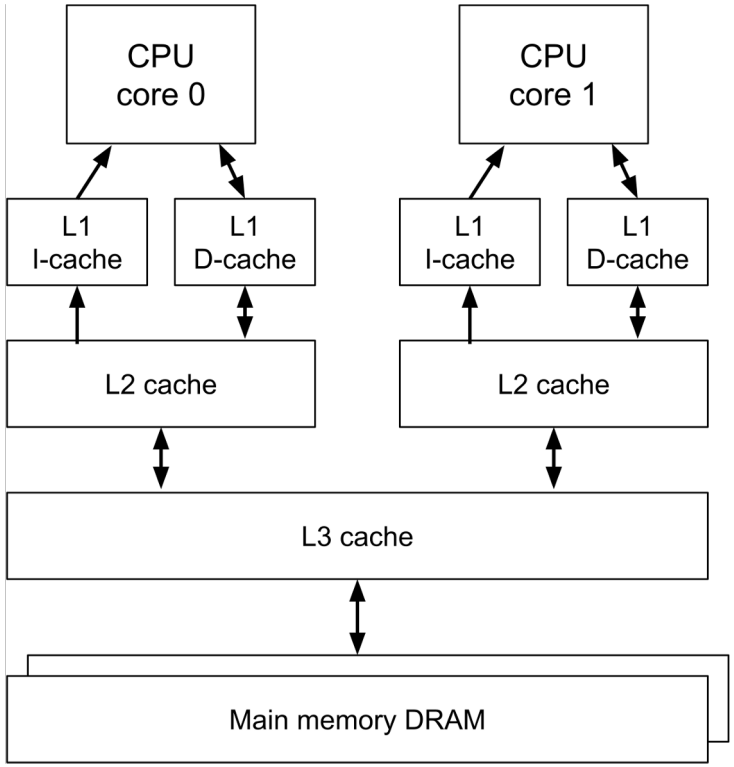
Cache line size, L1 data cache

Strategy:
Build a linked list (to defeat independant loads), scrambled (to try to defeat prefetching) that likely fits in the L1 cache. Use various possible line sizes.

After filling all the caches with other data, time how long it takes to walk the list, hoping to see delays that are proportional to the number of cache misses that go all the way out to main memory.

The miss count tells you the L1 line size. Other cache levels will have the same or a larger line size. After the total sizes are known, you could build L2 and L3 lists to check their line sizes.

Example cache hierarchy



4KB..32KB each

64KB..1MB each

1MB..32MB+

64GB..1TB+

Memory timing, cache total size

Memory timing: How big is each cache level?

Strategy: Build a linked list of items that are linesize bytes each. Trash the caches. Access the first N=16, 32, ..., 512K linked list items. Then access again, ~3 times.

[512K items at 64 bytes each = 32MB, the largest cache size we expect to see.]

The first pass into an empty cache will miss a lot, but if all N items fit in the cache, the other three passes will hit.

If the N items total more the the cache size, the first pass will miss a lot and only leave 1/2, 1/4, etc. of the items still in the cache. The other three passes will get 1/2, 3/4 etc. misses, or more depending on the replacement algorithm.

Memory timing, cache associativity

Memory timing: What is the associativity per level?

Strategy: For each possible associativity A = 1, 2, 4, ... and given the line size and total cache size measured above, build a list of items at distinct addresses that will all map to **a single set** of the cache if the associativity is A. For example, addresses that map into the same set of a 32KB 4-way associative cache will be 32KB/4 = 8KB apart. For a 256KB 4-way cache, 64KB apart, etc.

Walk the list once, trying to load all the A distinct items into the cache. Time how long it takes to walk again, hoping to distinguish all-hits from a mixture of hits and misses. If A is <= the true associativity, you should see all hits.
Different cache levels may well have different amounts of associativity.

Memory timing, complications

Complications

Similar considerations apply to TLB hits and misses for every 4KB accessed -- TLB entry size, total size, associativity.

Complications

Similar considerations apply to TLB hits and misses for every 4KB accessed -- TLB entry size, total size, associativity.

Oh wait, all the previous cache-timing stuff is slightly BOGUS with virtual memory

Complications

Similar considerations apply to TLB hits and misses for every 4KB accessed

Oh wait, all the previous cache-timing stuff is slightly BOGUS with virtual memory

The physical addresses presented to the caches will skip around at every 4KB boundary, upsetting the access patterns you are trying to establish.

Complications

Similar considerations apply to TLB hits and misses for every 4KB accessed

Oh wait, all the previous cache-timing stuff is slightly BOGUS with virtual memory

The physical addresses presented to the caches will skip around at every 4KB boundary, upsetting the access patterns you are trying to establish.

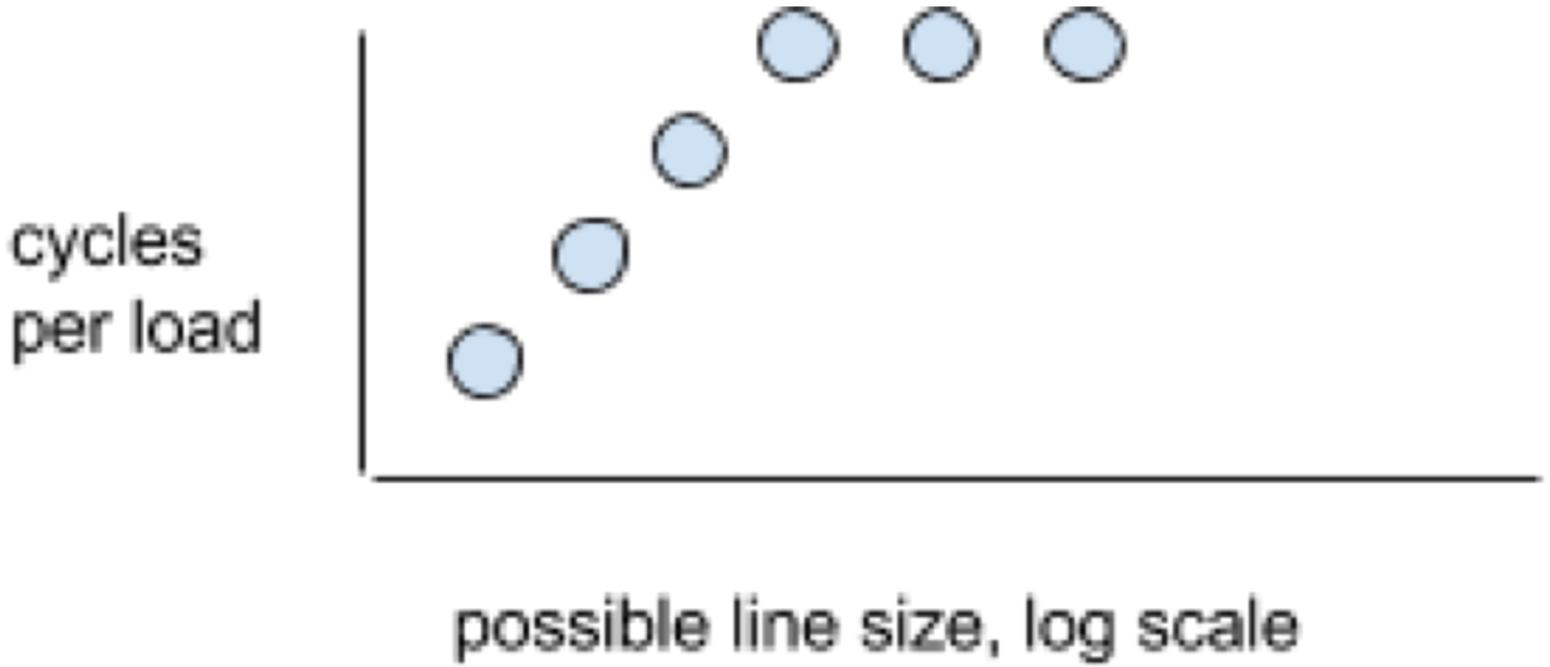
C'est la vie.

Revisiting caches

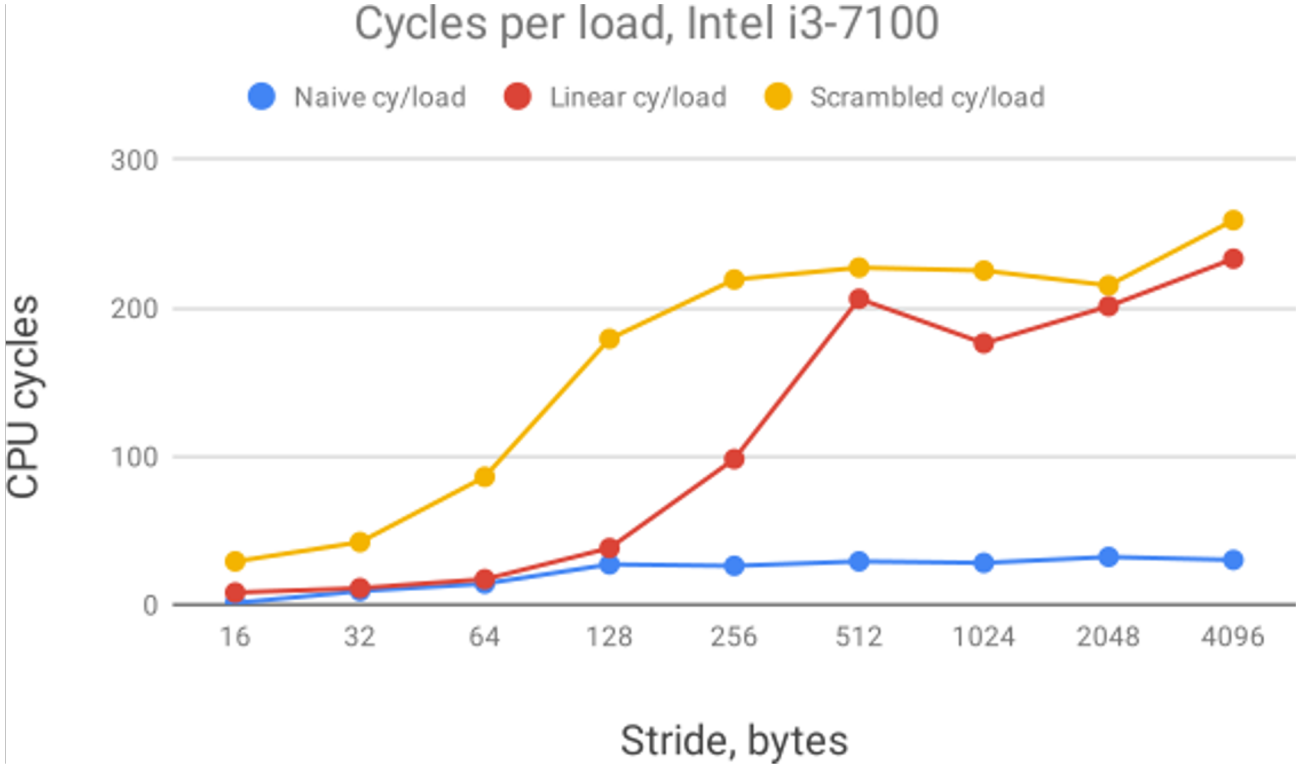
Hint: Draw yourself some sketches/diagrams of the expected dynamics

Memory timing, cache **line** size

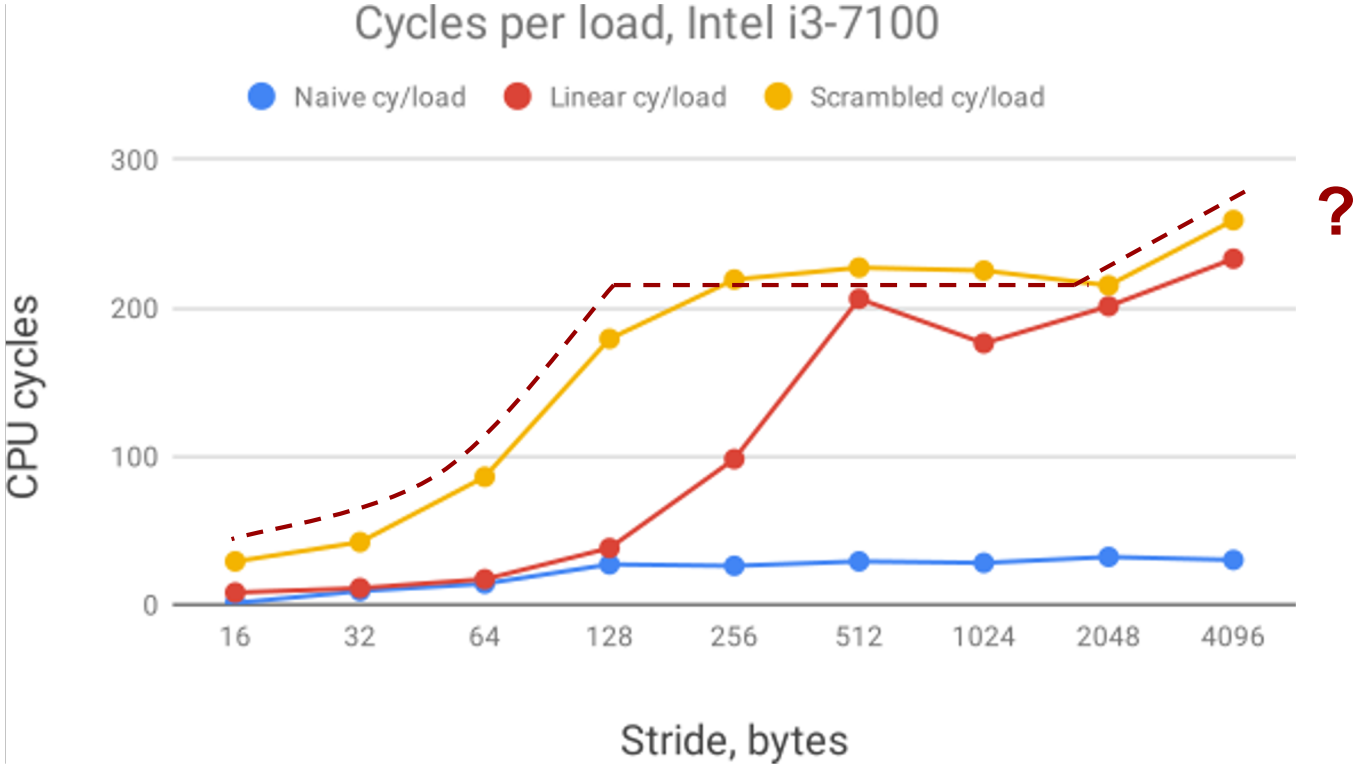
Cache line size, **sketch** of expected results



Cache line size, measured

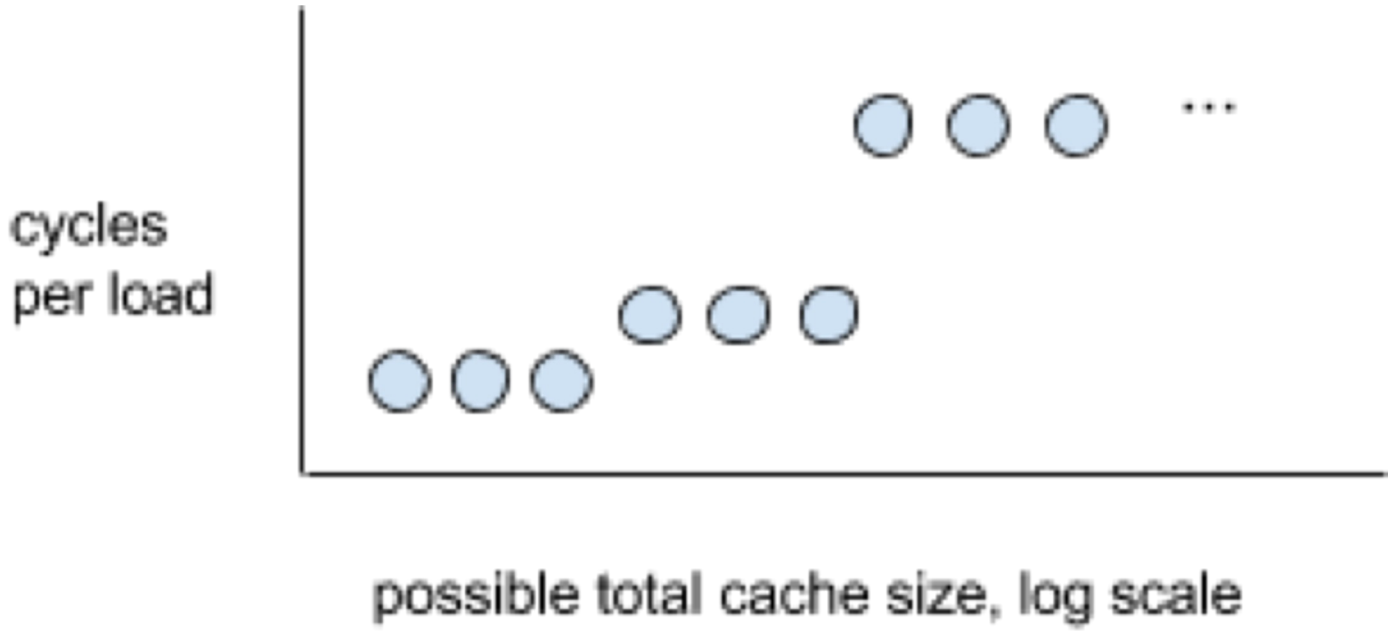


Cache line size, measured

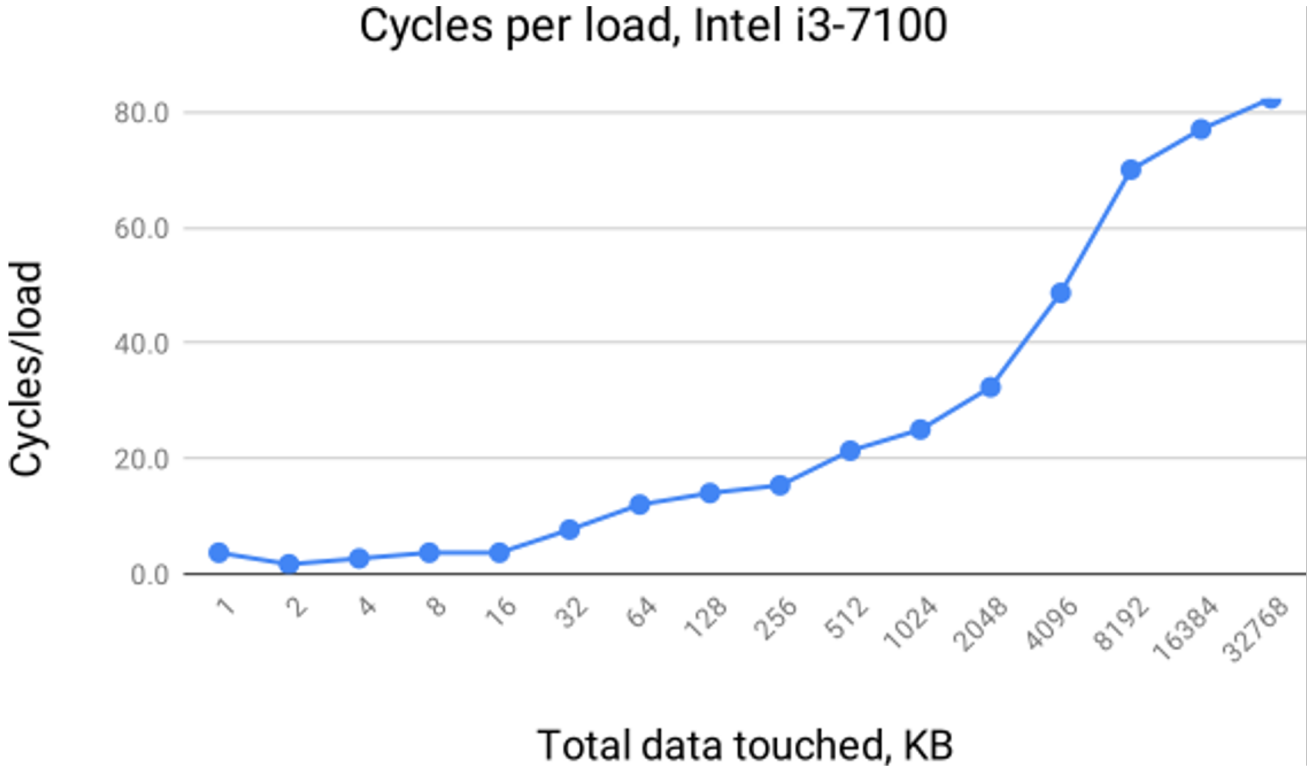


Memory timing, cache **total** size

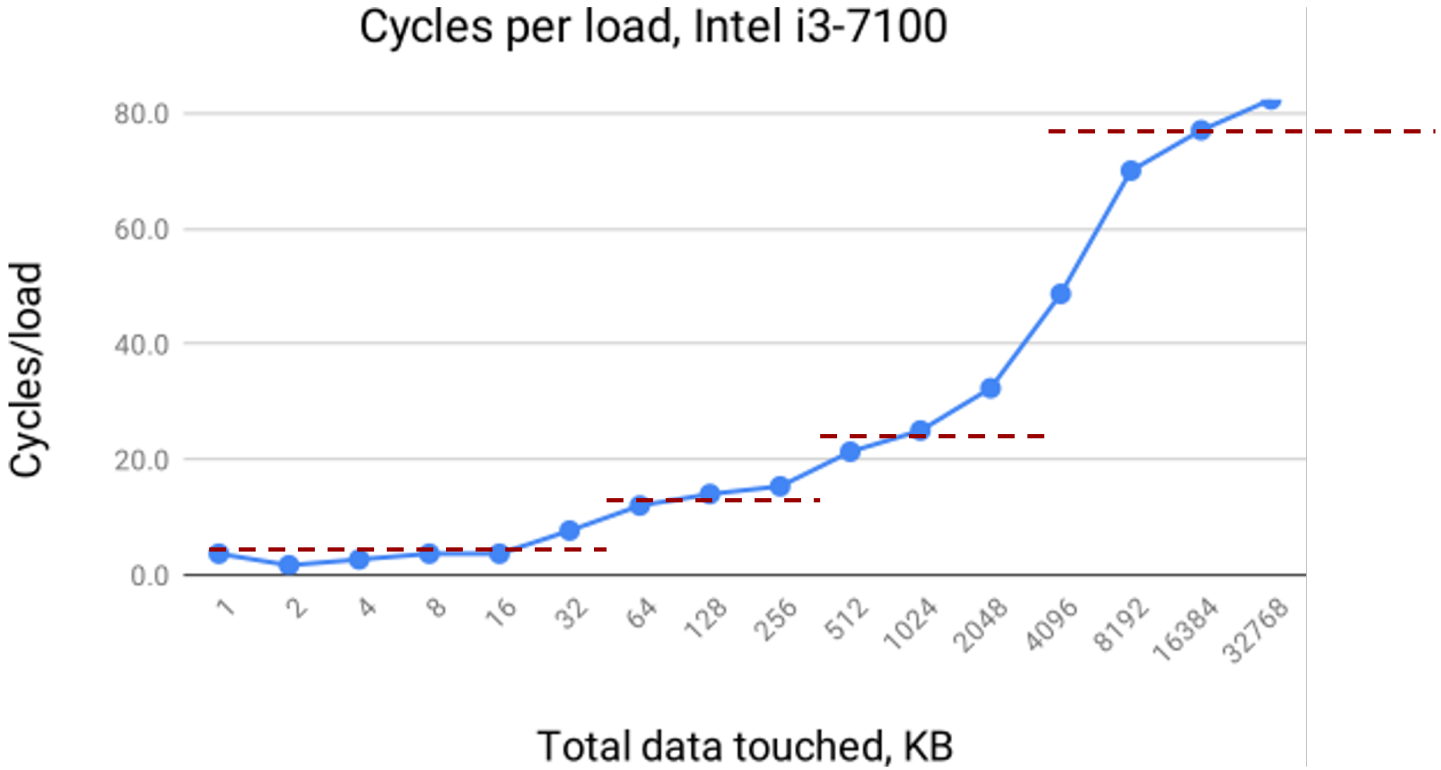
Cache total size, **sketch** of expected results



Cache total size, measured



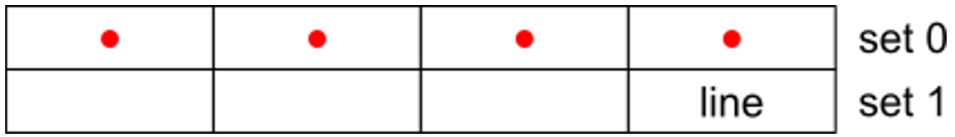
Cache total size, measured



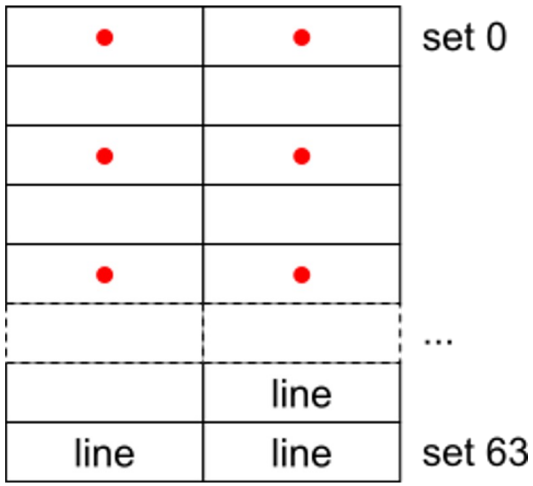
Memory timing, cache associativity

L2 associativity

Assume L1 is 4-way associative with two sets, and L2 is 2-way associative with 64 sets.



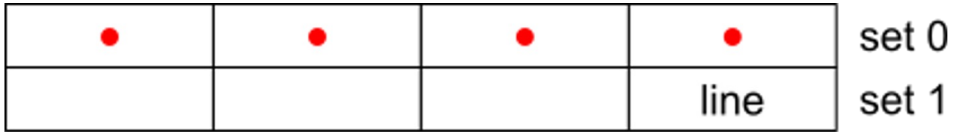
L1 cache



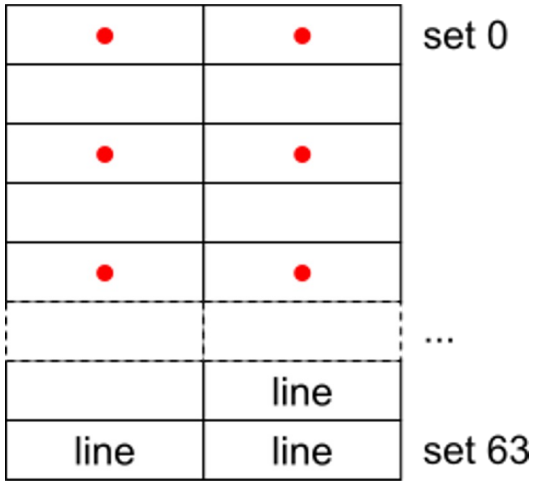
L2 cache

L2 associativity

Loading the six items in L2 lines 0, 2, and 4 will try to put all six in L1 set 0, but only **four** of the six will fit in L1. Repeatedly loading the six will keep missing in L1.



L1 cache



L2 cache

L2 associativity

Loading the six items in L2 lines 0, 2, and 4 will try to put all six in L1 set 0, but only **four** of the six will fit in L1. Repeatedly loading the six will keep missing in L1.

L1 cache

This kind of pattern lets us measure L2 without much interference from L1

L2 cache