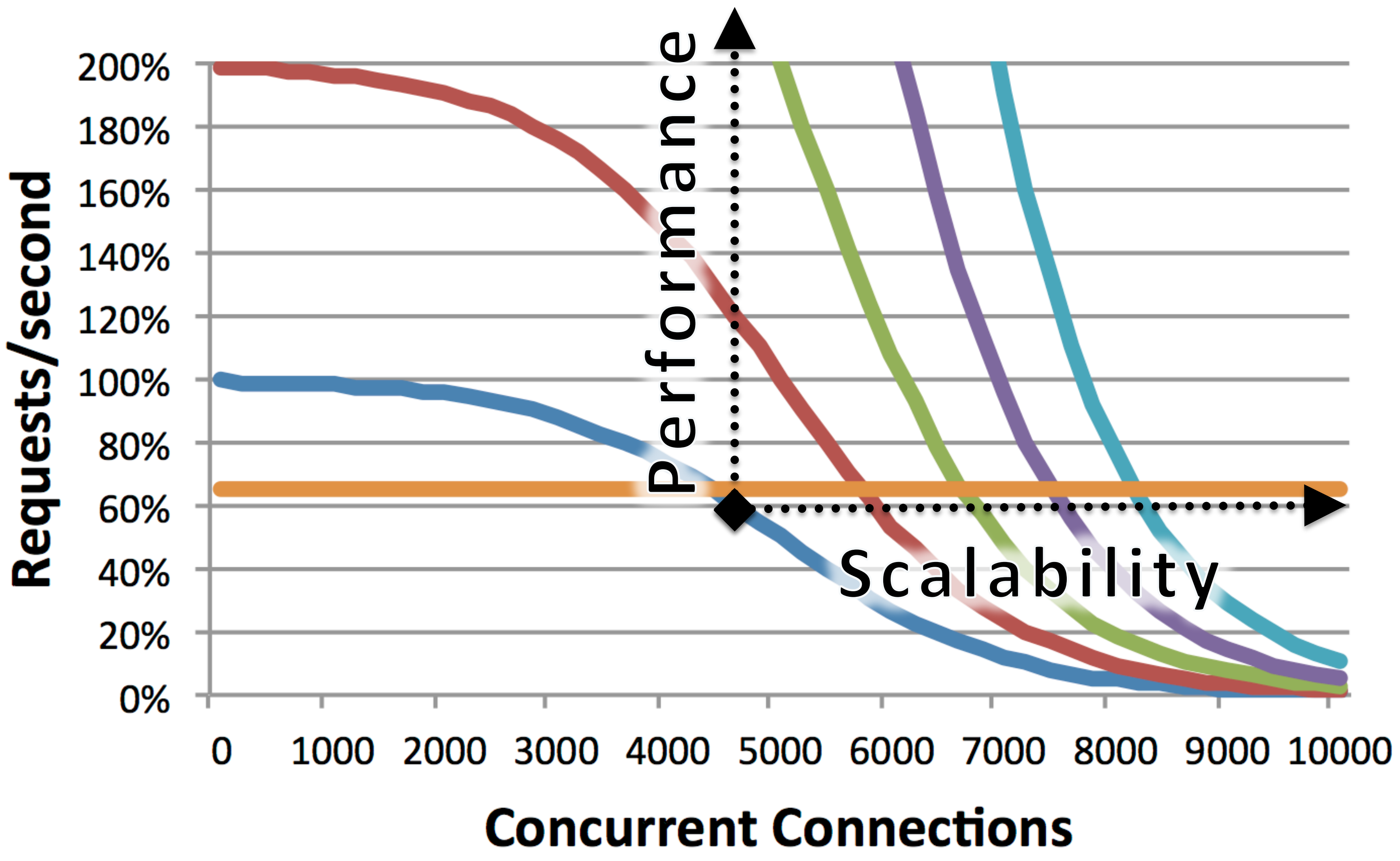
The cost of things at scale

Robert Graham

@ErrataRob

https://blog.erratasec.com


Requests/second
200%
180%
160%
140%
120%
100%
80%
60%
40%
20%
0%
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
Concurrent Connections
Performance
Scalability

The cost of things

- How fast can CPUs execute instructions
- How fast can CPUs access-memory
- How fast are kernel system calls
- How fast are synchronization primitives
- How fast are “context-switches”

Code

- https://github.com/robertdavidgraham/c10mbench

C10M defined

- 10 million concurrent connections
- 1 million connections/second
- 10 gigabits/second
- 10 million packets/second
- 10 microsecond latency
- 10 microsecond jitter
- 10 coherent CPU cores

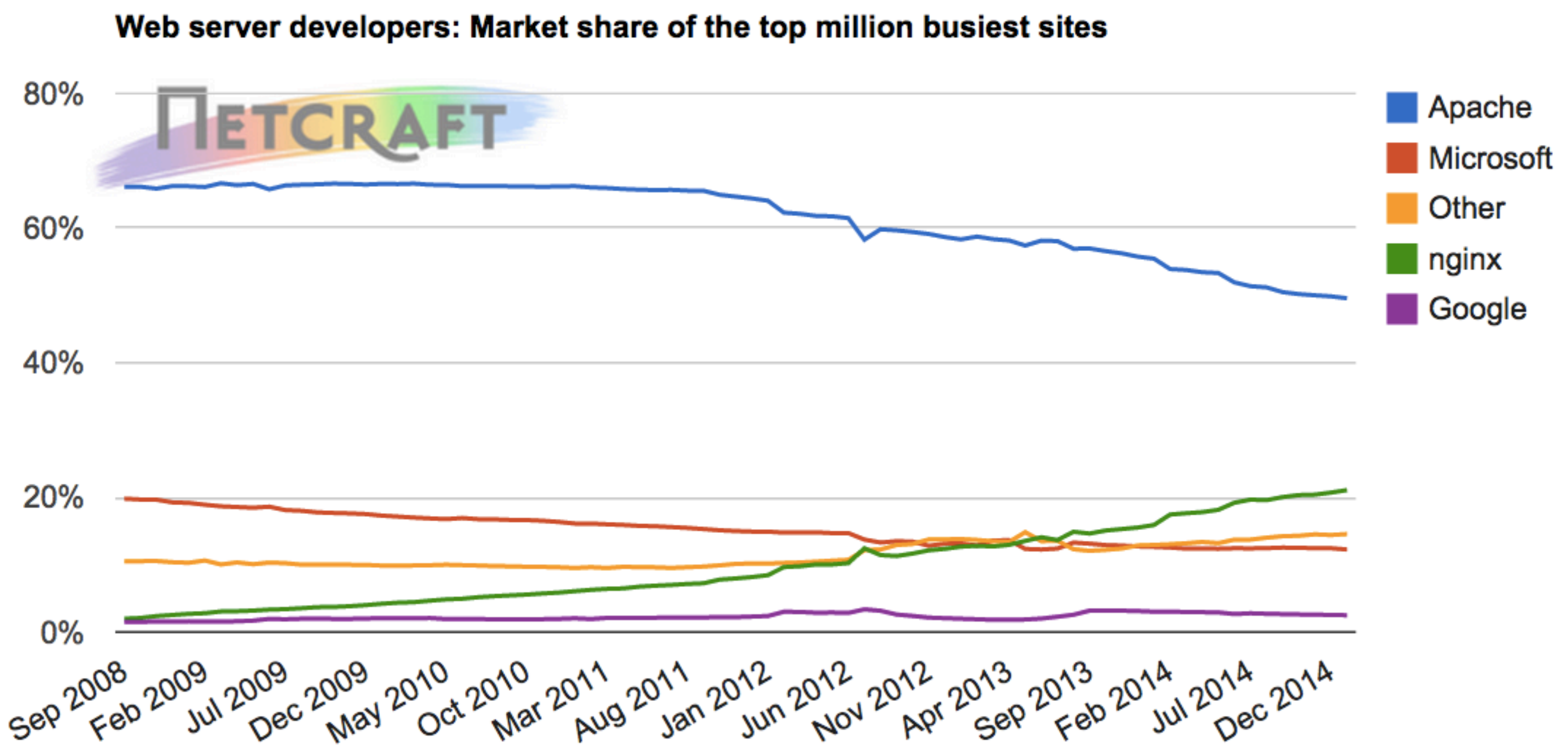
Classic definition: Context-switch

- Process/thread context switches

..but process context switches becoming rare

- NodeJS
- Nginx
- Libevent
- Java user-mode threads
- Lua coroutines

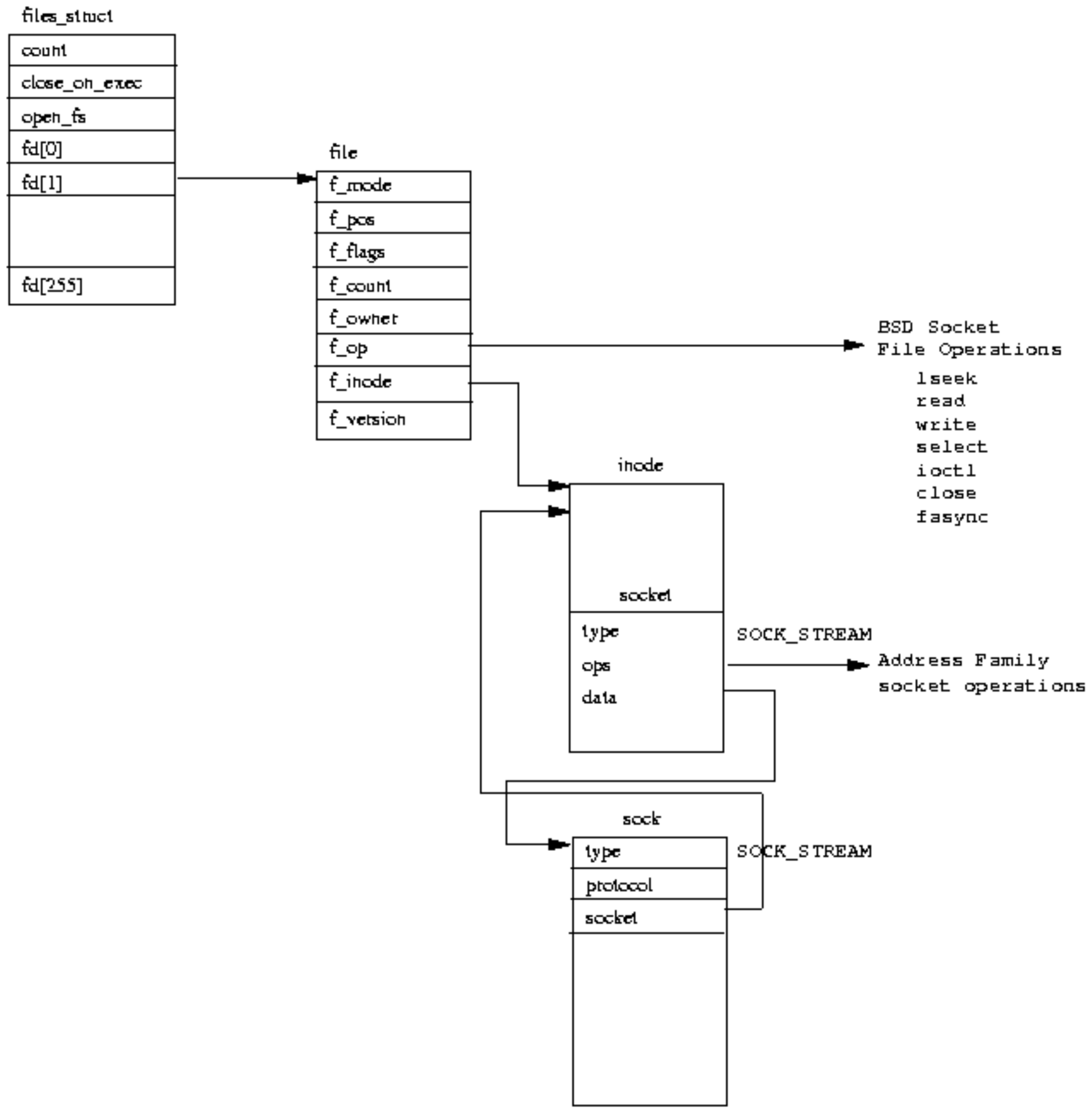
…but context switches becoming rare

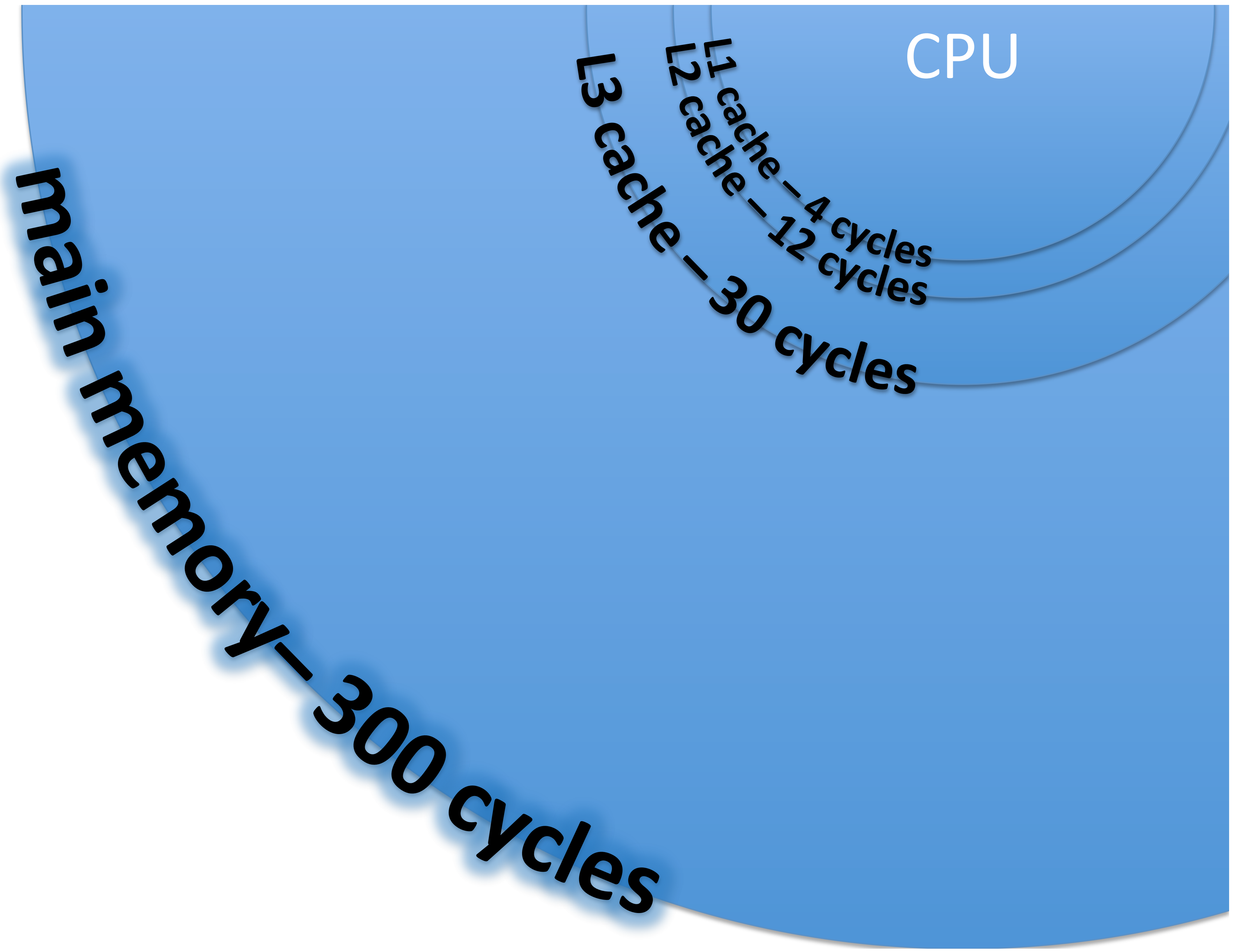


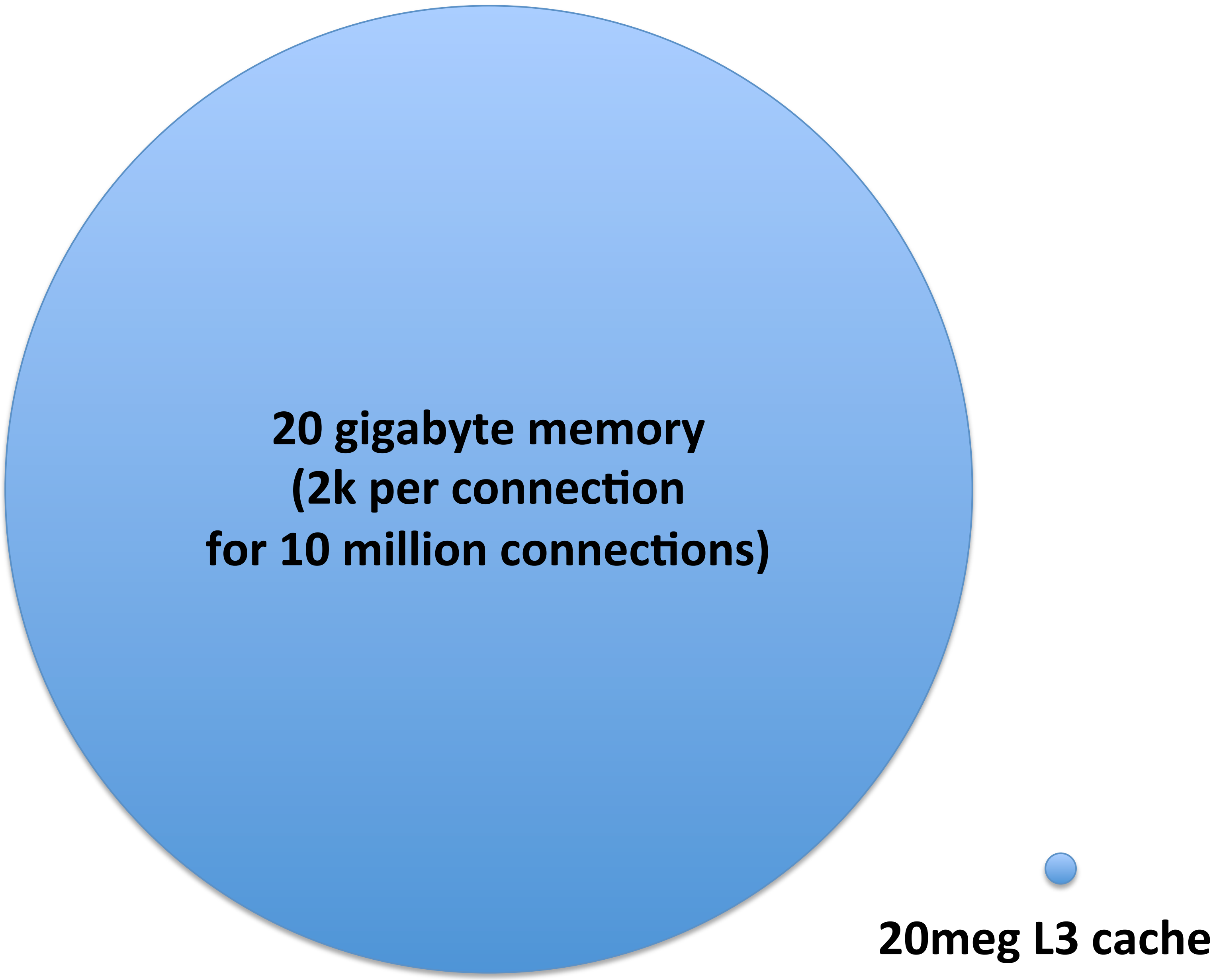
Real definition: Context-switch

- Each TCP connection is a task, with context
  - Whether you assign a thread to it, a closure, or a data structure
- Each incoming packet causes a random context switch
- A lot of small pieces of memory must be touched – *sequentially*
  - "pointer-chasing"

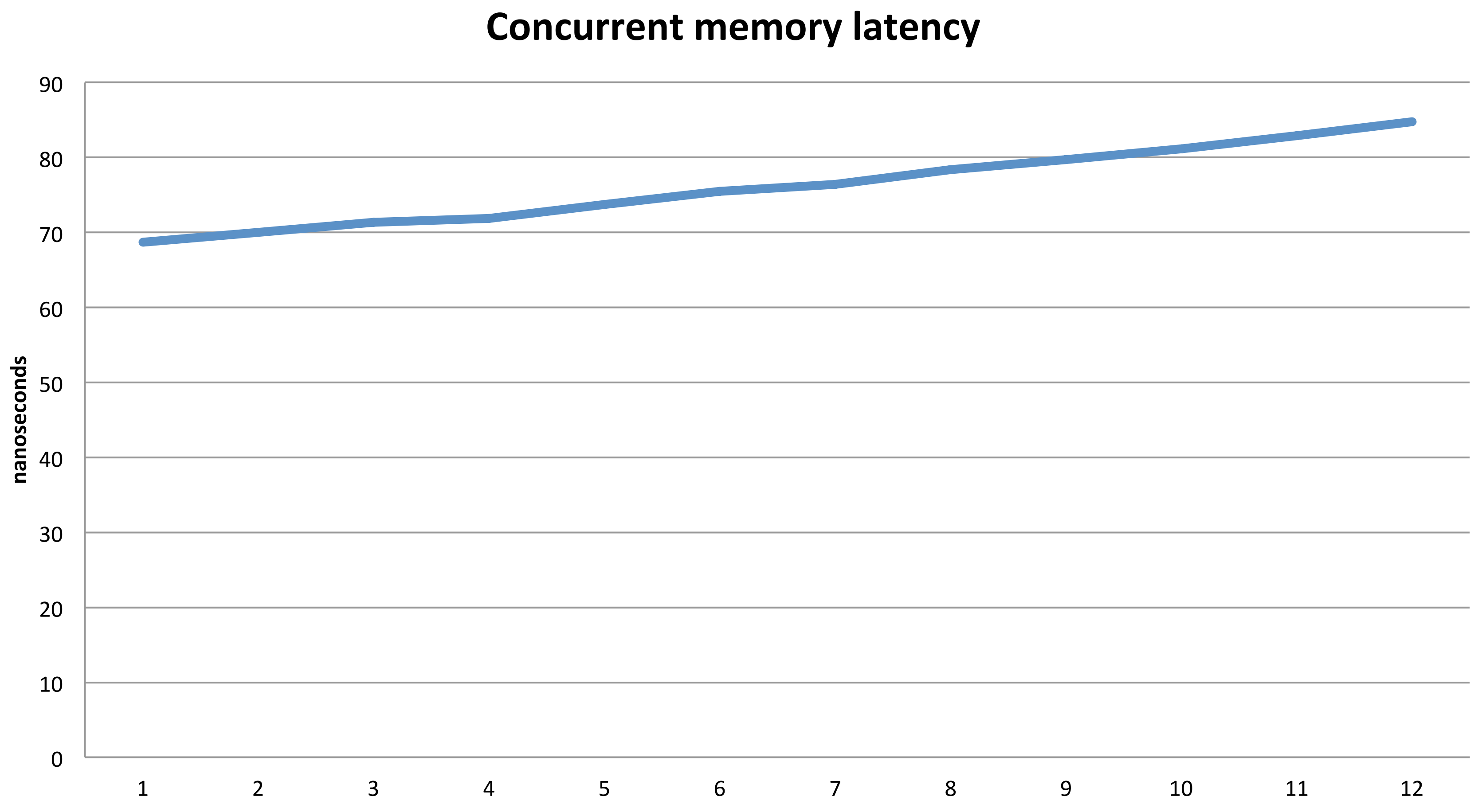
- .


files_struct
count
close_on_exec
open_fs
fd[0]
fd[1]
fd[255]
file
f_mode
f_pos
f_flags
f_count
f_owner
f_op
f_inode
f_version
BSD Socket
File Operations
lseek
read
write
select
ioctl
close
fasync
inode
socket
type
ops
data
SOCK_STREAM
Address Family
socket operations
sock
type
protocol
socket
SOCK_STREAM

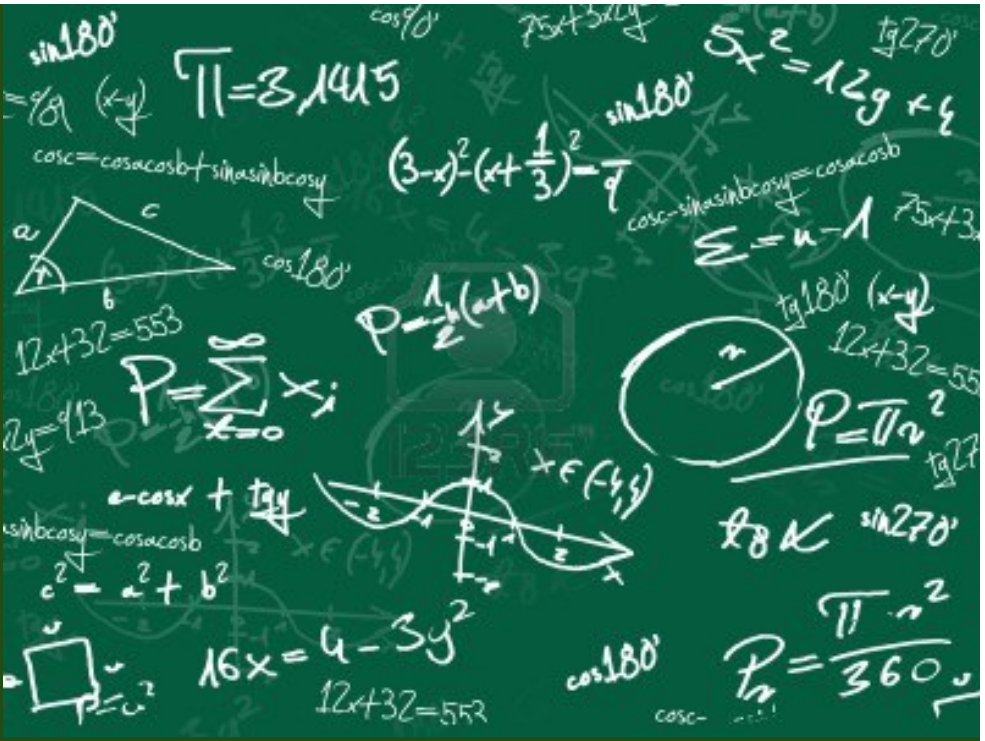

CPU
L1 cache – 4 cycles
L2 cache – 12 cycles
L3 cache – 30 cycles
main memory– 300 cycles


20 gigabyte memory
(2k per connection
for 10 million connections)
20meg L3 cache

Measured latency: 85ns



budget



10 million packets/second

divided by 10 cores

by 100 nanoseconds/miss

~10 cache misses per packet

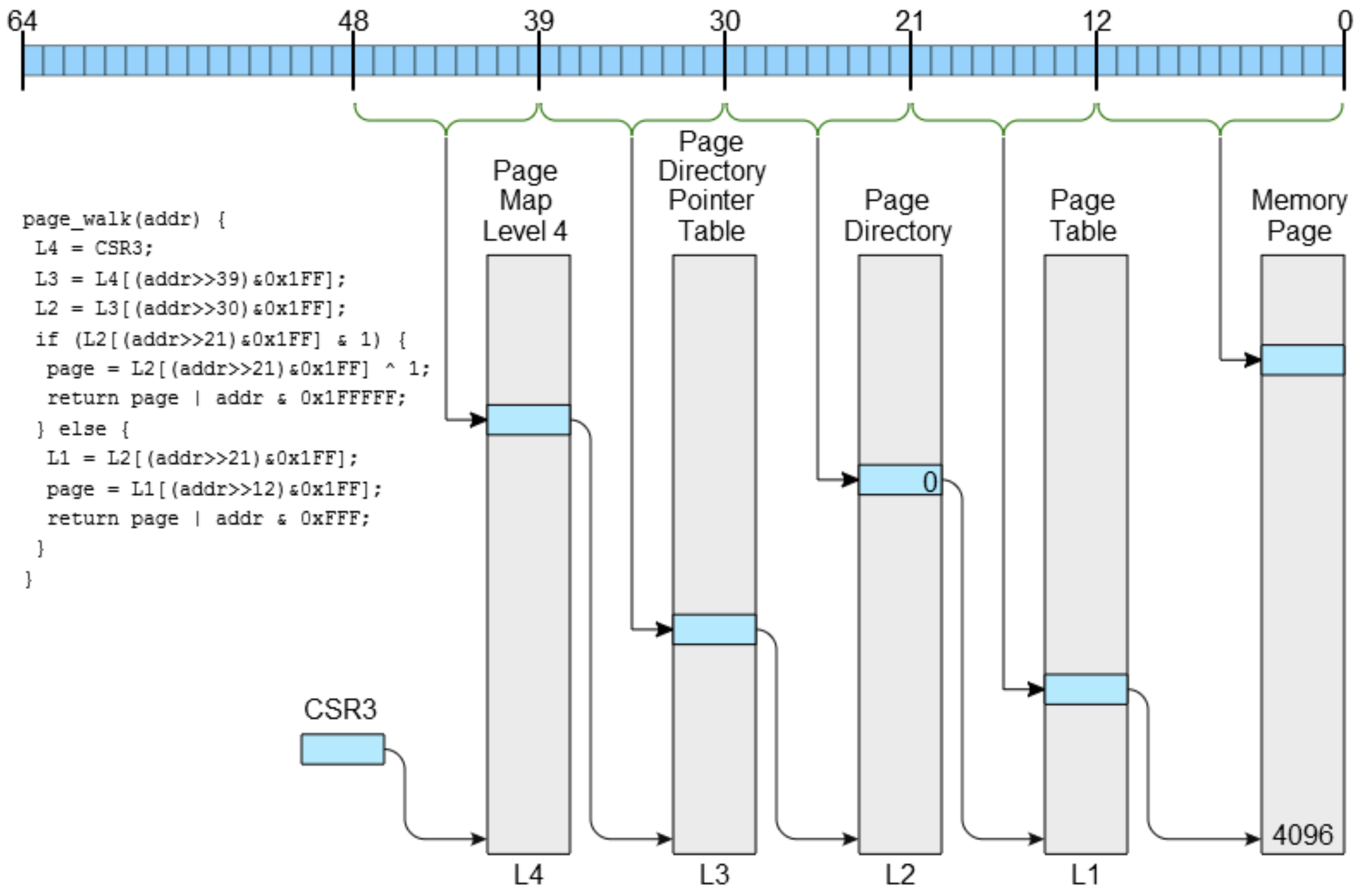
Now for user-mode

- Apps written in C have few data structures
- Apps written in high-level languages (Java, Ruby, Lua, JavaScript) have bits of memory strewn around

User-mode memory is virtual

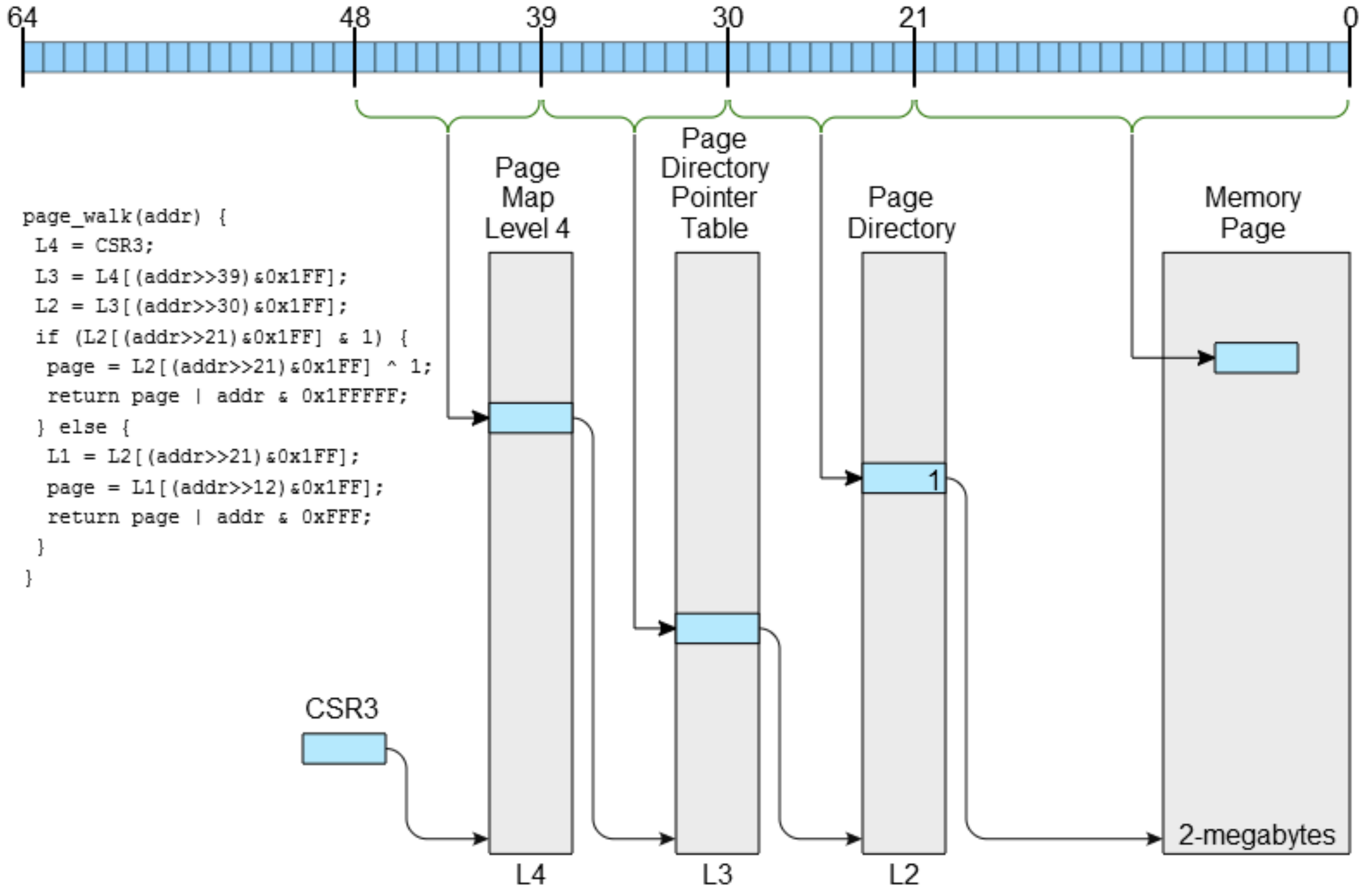
- Virtual addresses are translated to physical addresses on every memory access
  - Walk a chain of increasingly smaller page table entries
- But TLB cache makes it go fast
  - But not at scale
  - TLB cache is small
  - Page tables themselves may not fit in the cache

Small Page Diagram for x64 Virtual Memory

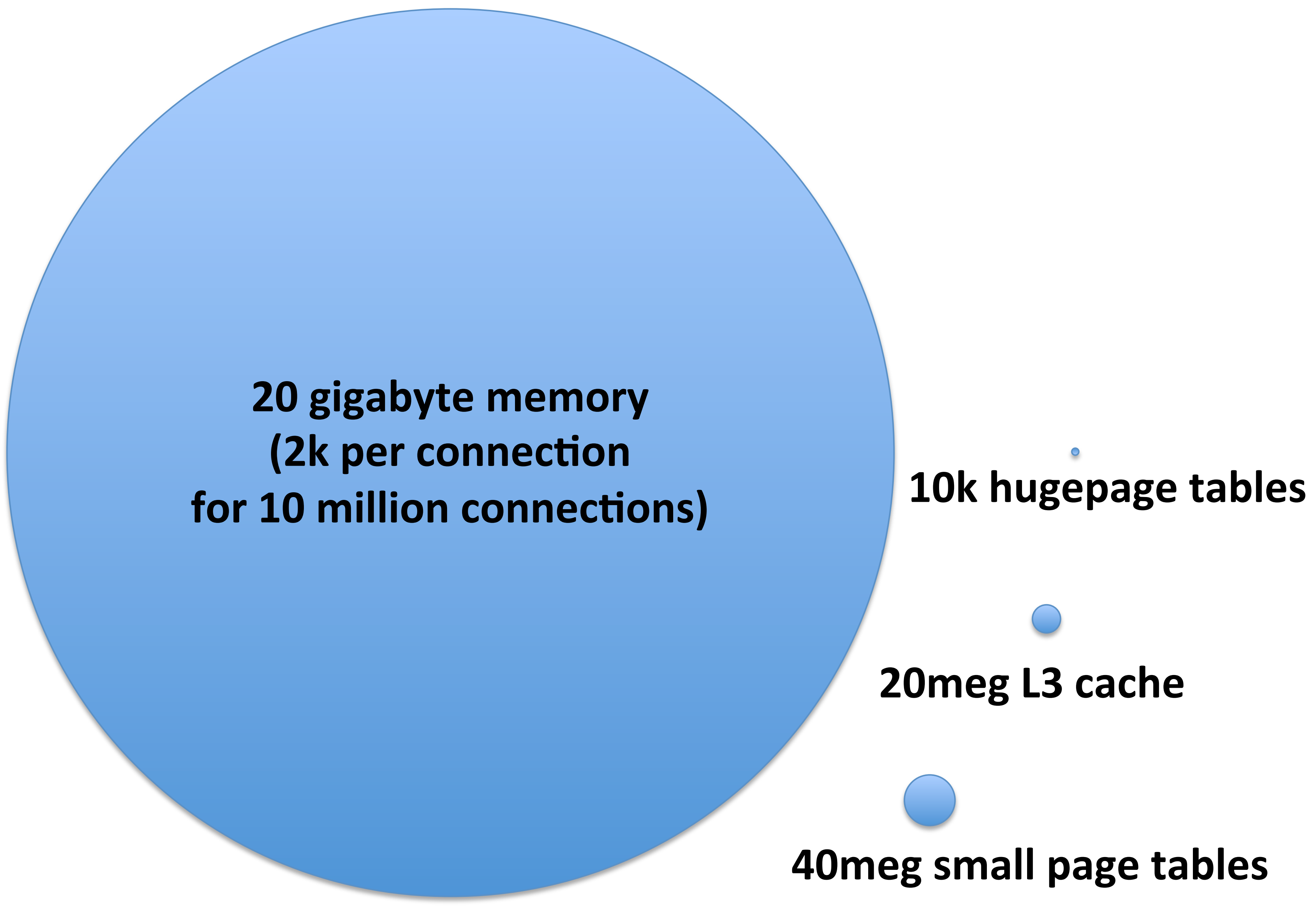


```
page_walk(addr) {
 L4 = CSR3;
 L3 = L4[(addr>>39)&0x1FF];
 L2 = L3[(addr>>30)&0x1FF];
 if (L2[(addr>>21)&0x1FF] & 1) {
  page = L2[(addr>>21)&0x1FF] ^ 1;
  return page | addr & 0x1FFFFF;
 } else {
  L1 = L2[(addr>>21)&0x1FF];
  page = L1[(addr>>12)&0x1FF];
  return page | addr & 0xFFF;
 }
}
```

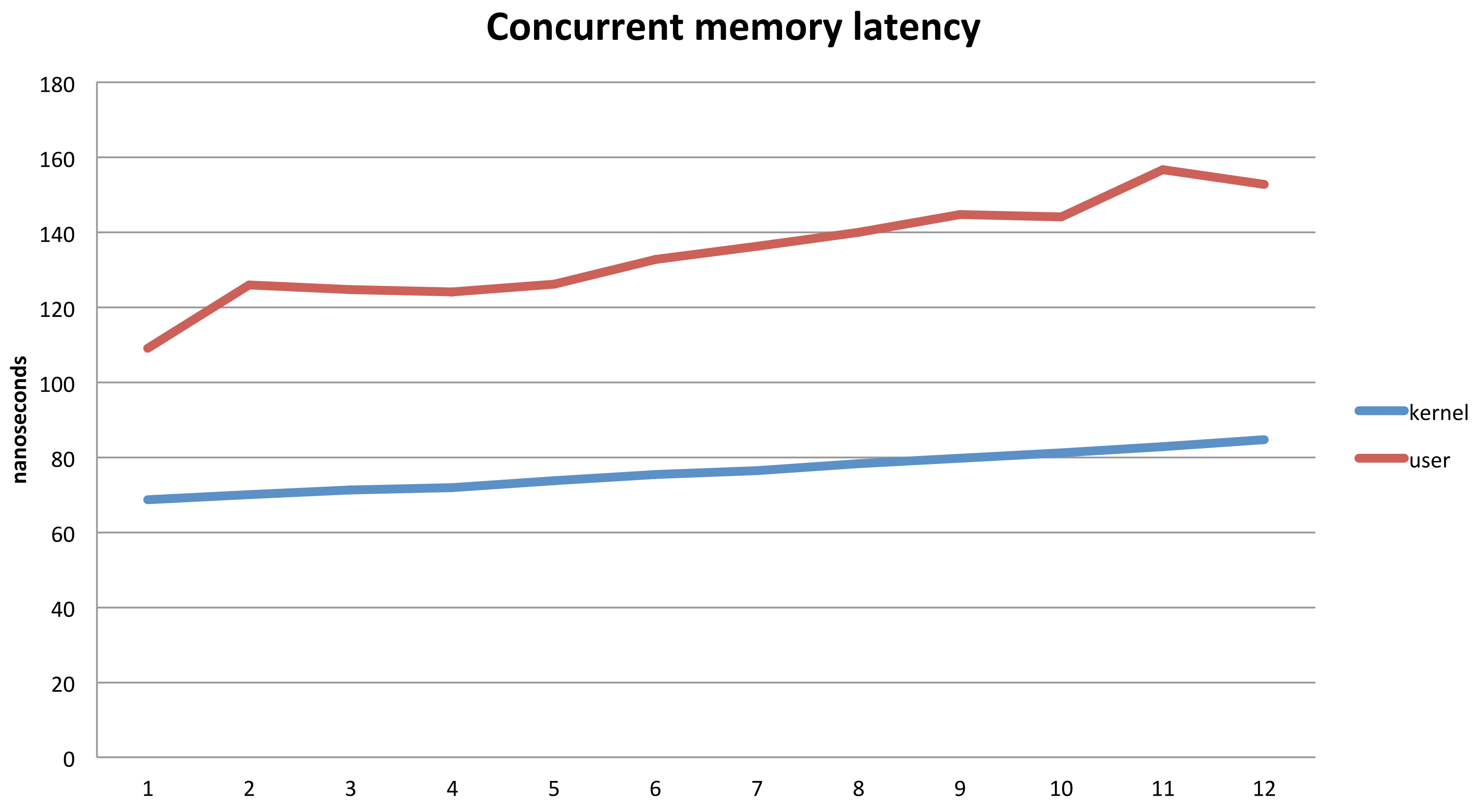
Large Page Diagram for x64 Virtual Memory



```
page_walk(addr) {
 L4 = CSR3;
 L3 = L4[(addr>>39)&0x1FF];
 L2 = L3[(addr>>30)&0x1FF];
 if (L2[(addr>>21)&0x1FF] & 1) {
  page = L2[(addr>>21)&0x1FF] ^ 1;
  return page | addr & 0x1FFFFF;
 } else {
  L1 = L2[(addr>>21)&0x1FF];
  page = L1[(addr>>12)&0x1FF];
  return page | addr & 0xFFF;
 }
}
```


20 gigabyte memory
(2k per connection
for 10 million connections)
10k hugepage tables
20meg L3 cache
40meg small page tables

User-mode latency



QED:

- Memory latency becomes a big scalability problem for high-level languages

How to solve

- Hugepages to avoid page translation
- Break the chain
  - Add “void *prefetch[8]” to the start of every TCP control block.
  - Issue prefetch instructions on them as soon as packet arrives
  - Get all the memory at once

Memory access is parallel

- CPU
  - Each core can track 72 memory reads at the same time
  - Entire chip can track ?? reads at the same time
- DRAM
  - channels X slots X ranks X banks
  - My computer: 3 * 2 * 1 * 4 = 24 concurrent accesses
  - Measured: 190-million/sec = 15 concurrent accesses

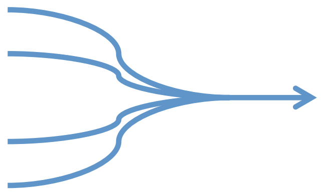
Some reading

- “What every programmer should know about memory” by Ulrich Draper
- http://www.akkadia.org/drepper/cpumemory.pdf

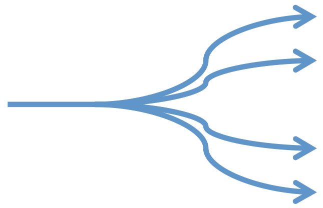
Multi-core

Multi-threading is not the same as multi-core

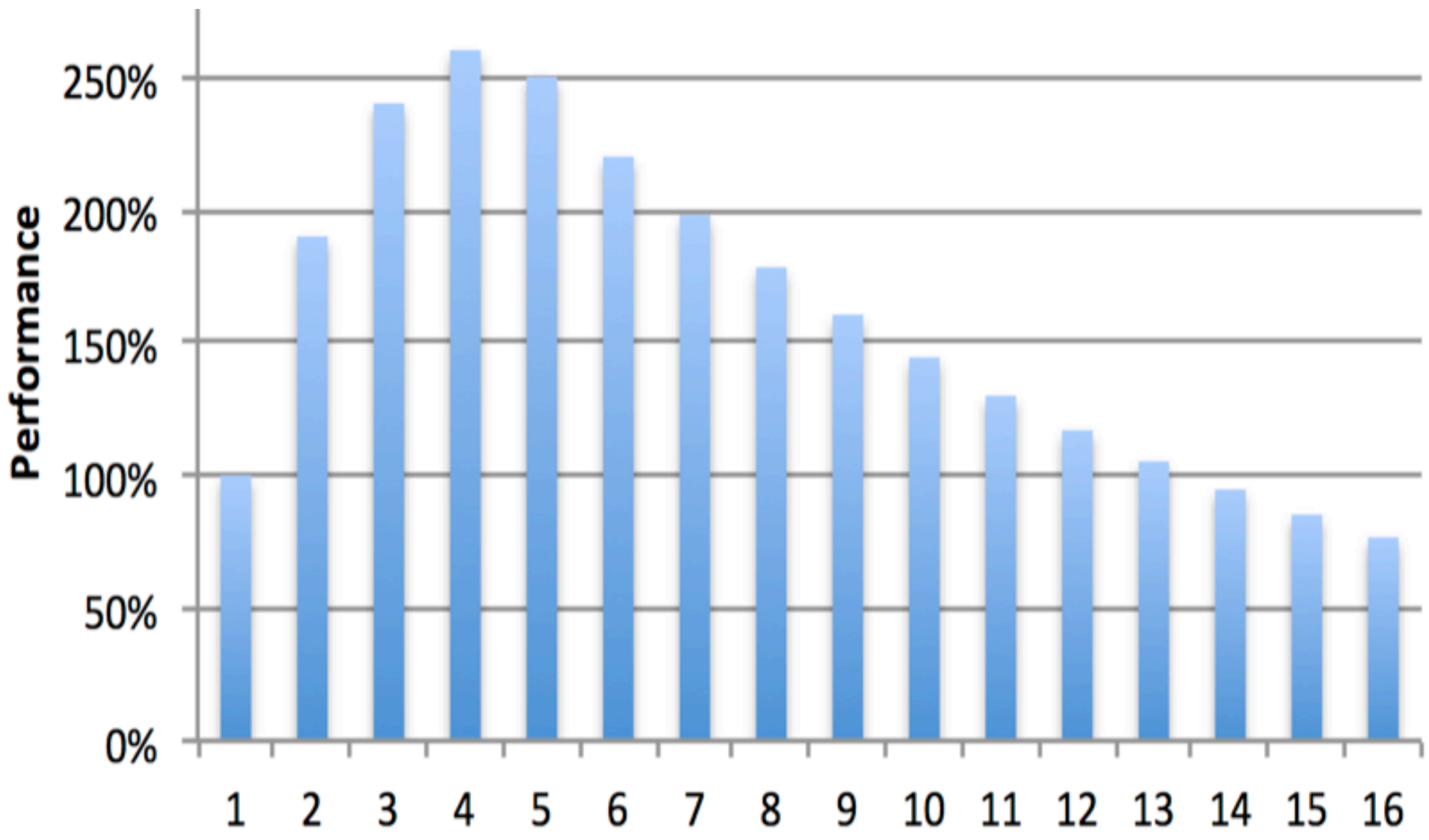
- Multi-threading
  - More than one thread per CPU core
  - Spinlock/mutex must therefore stop one thread to allow another to execute
  - Each thread a different task (multi-tasking)



- Multi-core
  - One thread per CPU core
  - When two threads/cores access the same data, they can't stop and wait for the other
  - All threads part of the same task



Most code doesn’t scale past 4 cores

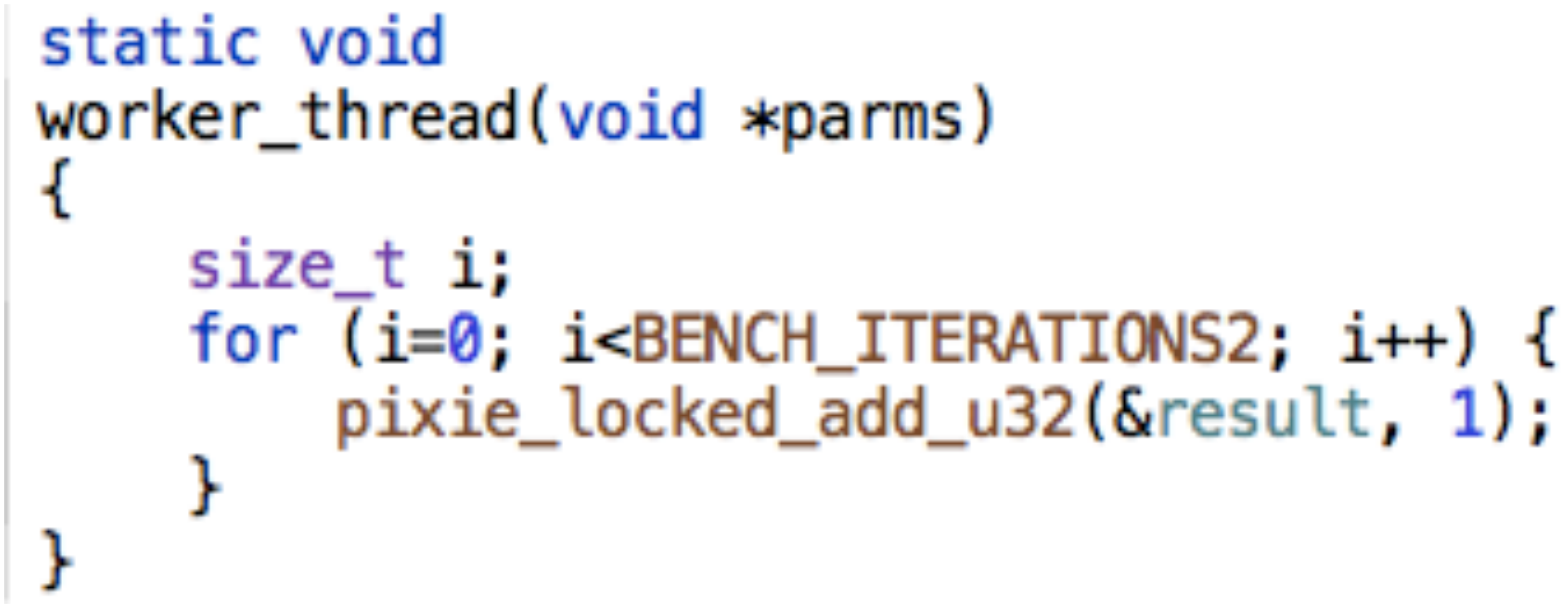


#1 rule of multi-core: don't share memory

- People talk about ideal mutexes/spinlocks, but they still suffer from shared memory
- There is exist data structures, "lock free", that don't require them

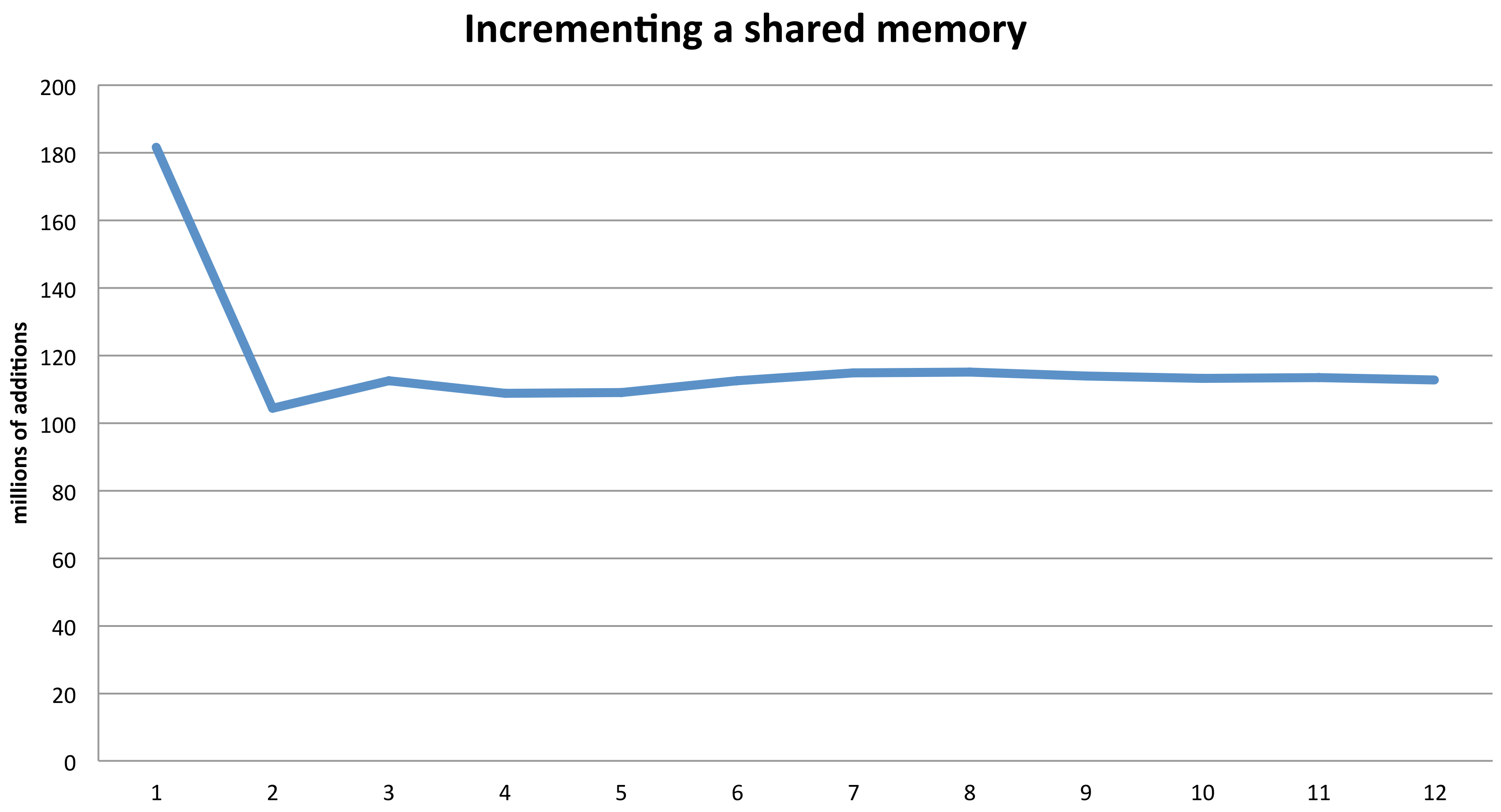
Let’s measure the problem

- A “locked add” simulates the basic instructions behind spinlocks, futexes, etc.

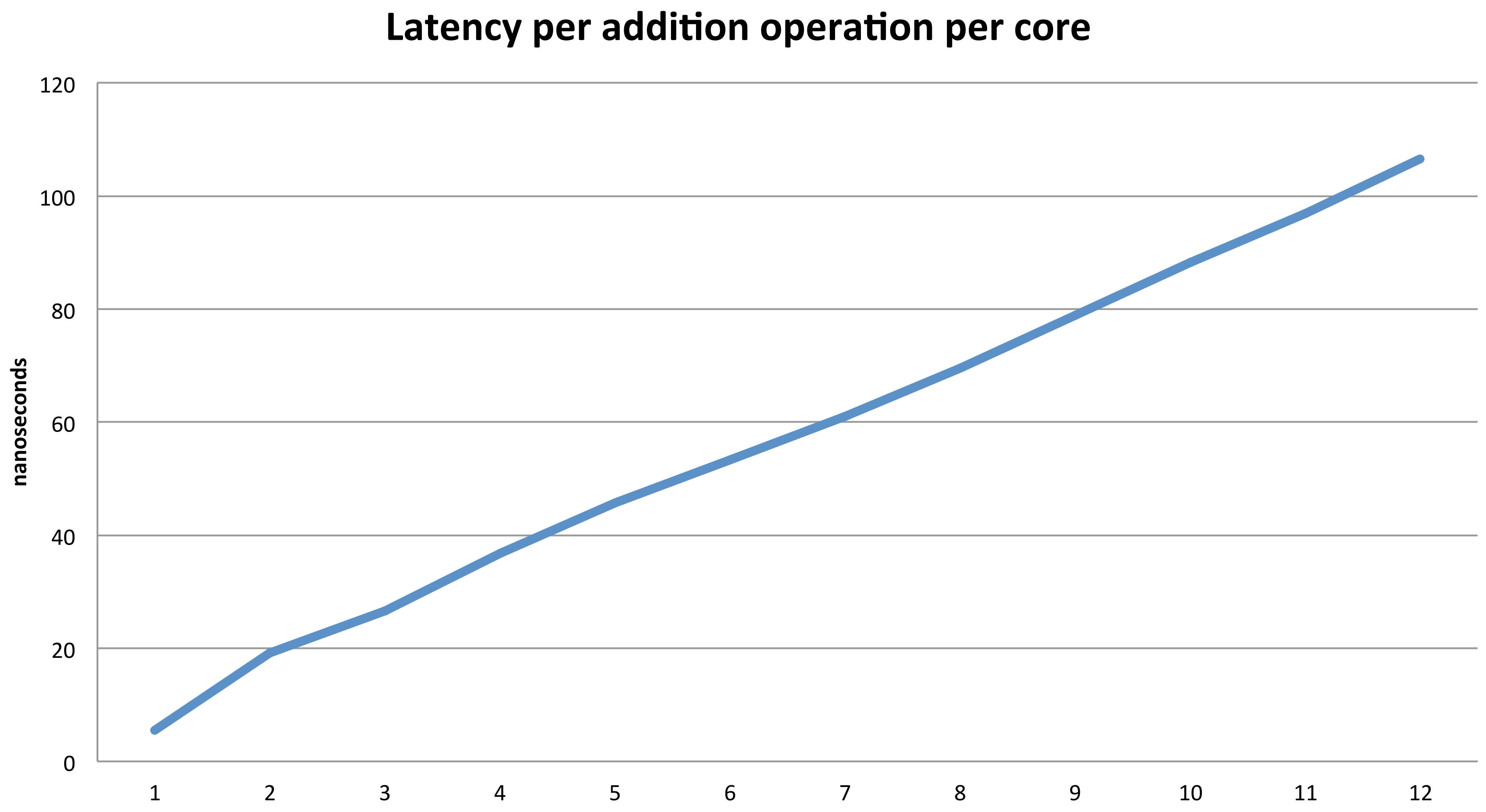


```
static void
worker_thread(void *parms)
{
    size_t i;
    for (i=0; i<BENCH_ITERATIONS2; i++) {
        pixie_locked_add_u32(&result, 1);
    }
}
```

Total additions per second



Latency per addition per thread



Two things to note

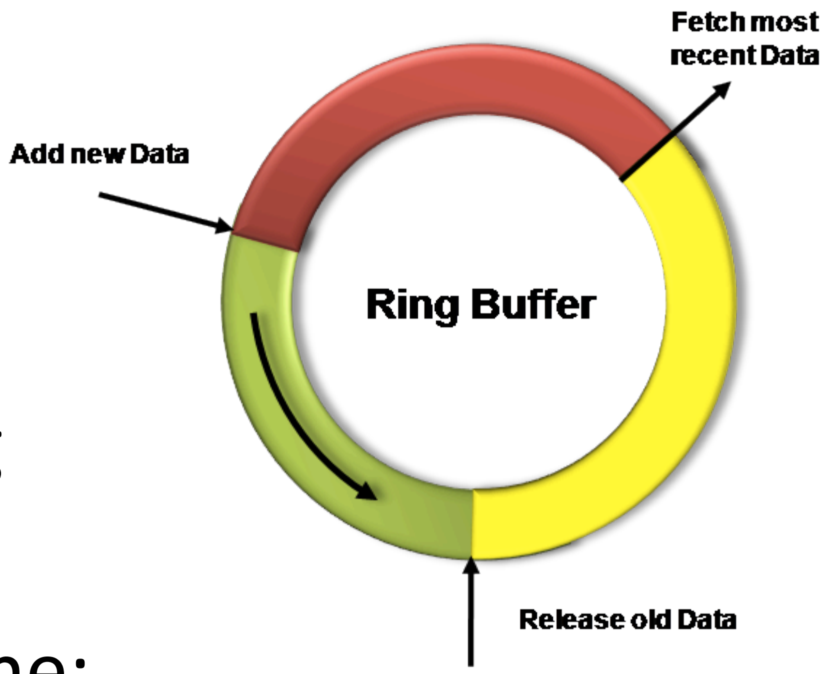
- ~5 nanoseconds
  - Cost of an L3 cache operation (~10ns)
  - Minus the out-of-order execution by the CPU (~5ns)
  - …and I'm still not sure
- ~100 nanoseconds
  - When many thread contending, it becomes as expensive as a main memory operation

Syscalls

- Mutexes often done with system calls
- So what's the price of a such a call?
  - On my machine
  - ~30 nanoseconds is minimum
  - ~60 ns is more typical idealized cases
  - ~400 ns in more practical cases

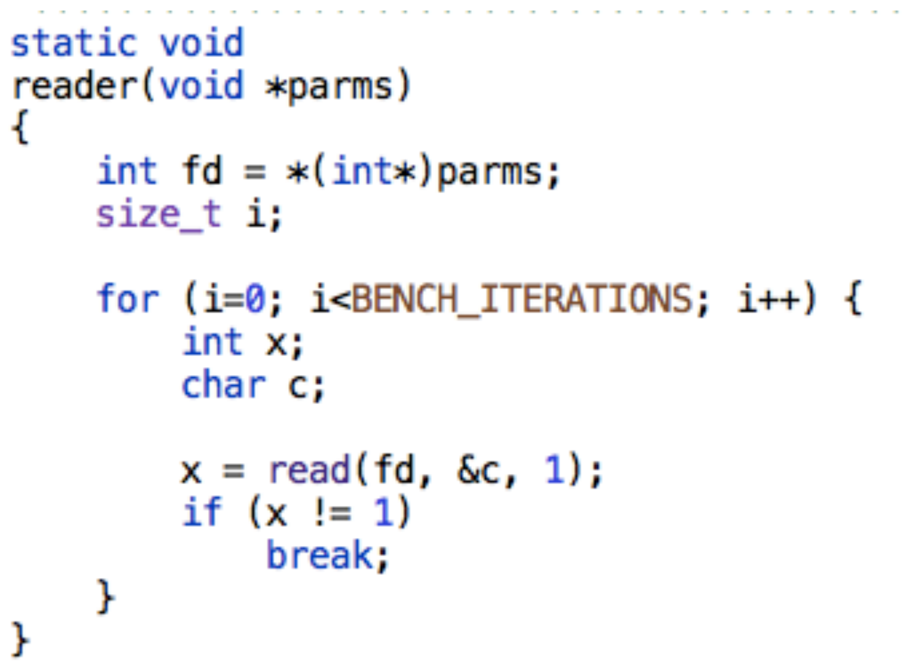
Solution: lock-free ring-buffers

- No mutex/spinlock
- No syscalls
- Since head and tail are separate, no sharing of cache lines
- Measured on my machine:
  - 100-million msgs/second
  - ~10ns per msg



Shared ring vs. pipes

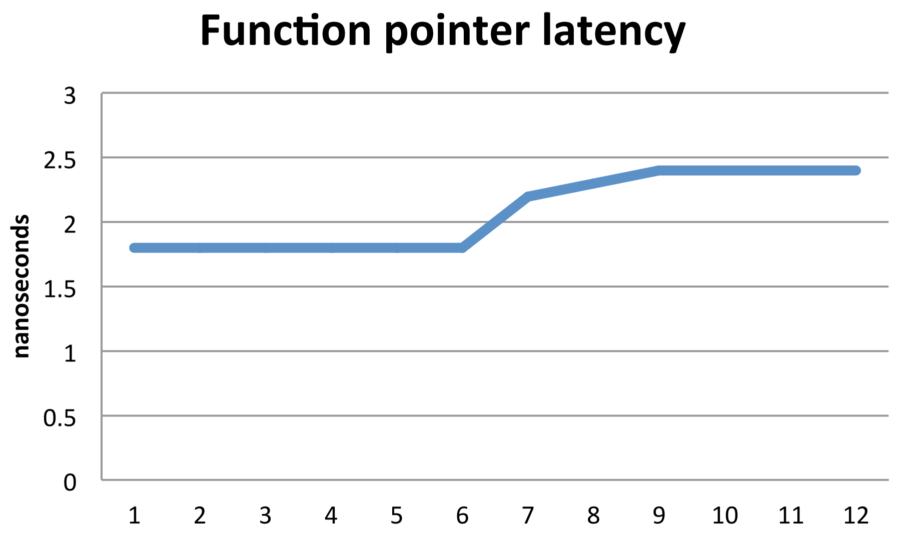
- Pipes
  - ~400ns per msg
  - 2.5 m-msgs/sec
- Ring
  - ~10ns per msg
  - 100 m-msgs/sec



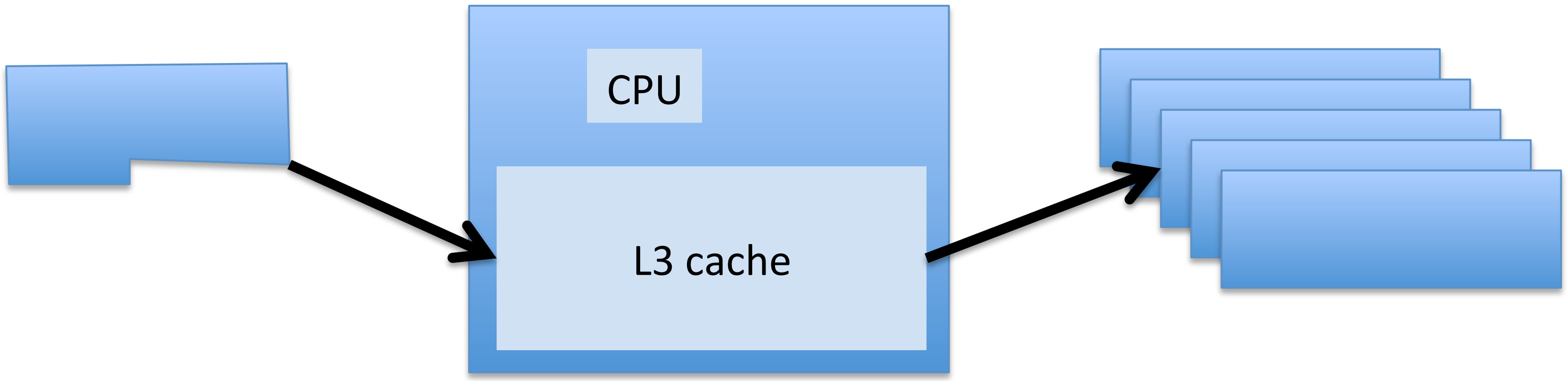
```
static void
reader(void *parms)
{
    int fd = *(int*)parms;
    size_t i;

    for (i=0; i<BENCH_ITERATIONS; i++) {
        int x;
        char c;

        x = read(fd, &c, 1);
        if (x != 1)
            break;
    }
}
```

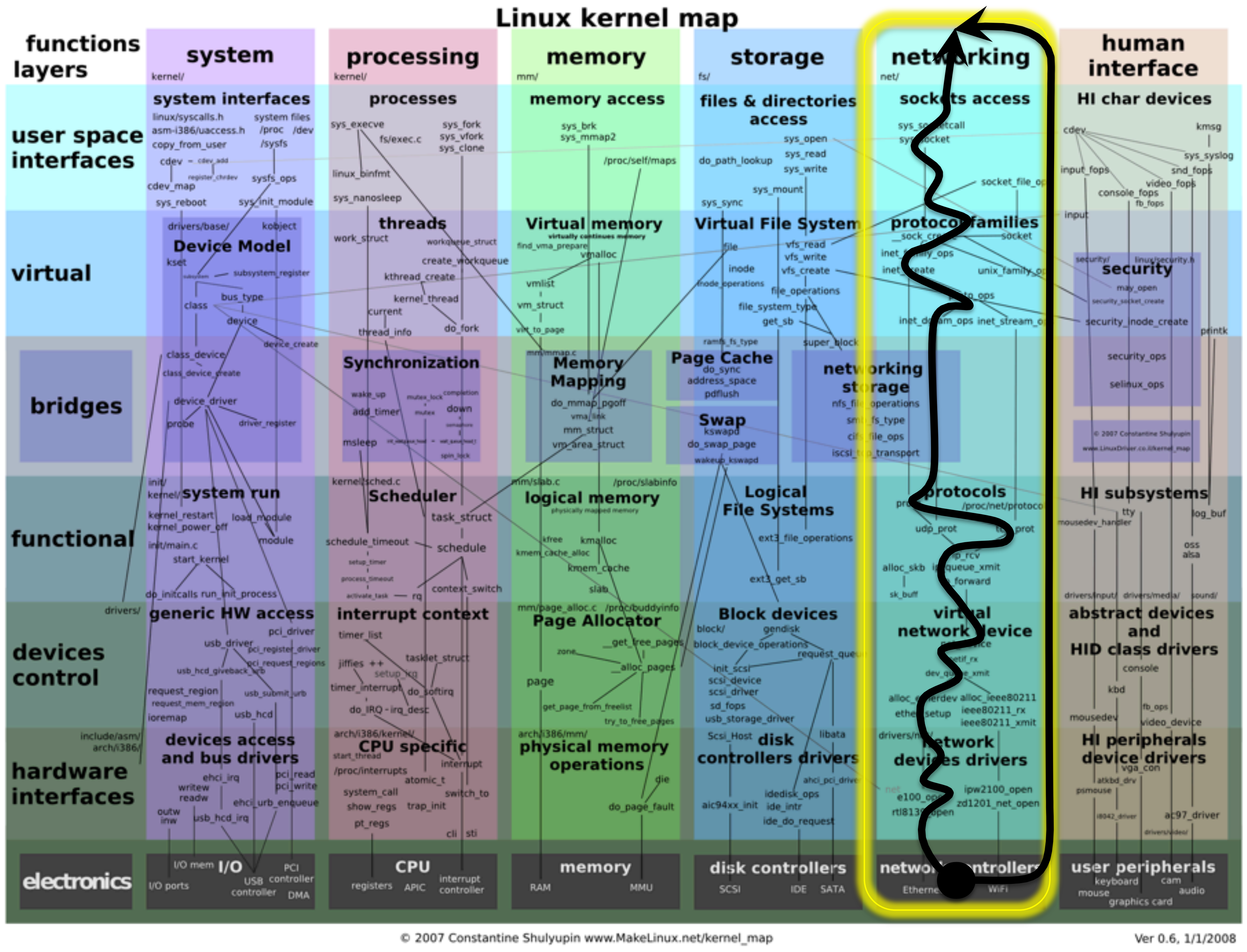
Function call overhead

- ~1.8ns
- Note the jump for “hyperthreading”
  - My machine has 6 hyperthreaded cores
- 6 clock cycles



DMA isn't

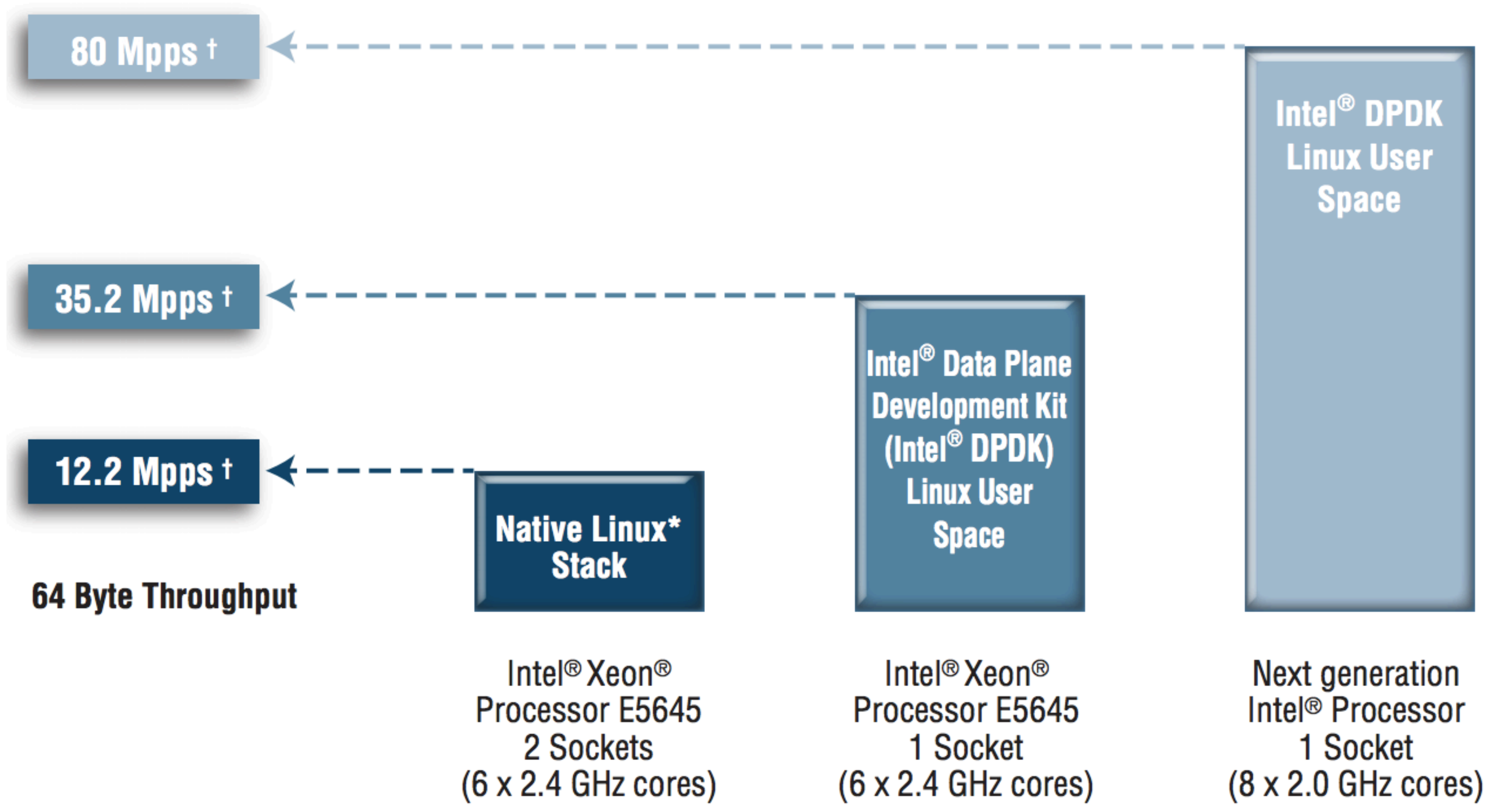



Linux kernel map
functions
layers
system
kernel/
processing
kernel/
memory
mm/
storage
fs/
networking
net/
human
interface
user space
interfaces
virtual
bridges
functional
devices
control
hardware
interfaces
electronics
system interfaces
processes
memory access
files & directories
access
sockets access
HI char devices
Device Model
threads
Virtual memory
Virtual File System
protocol families
security
Synchronization
Memory
Mapping
Page Cache
Swap
networking
storage
system run
Scheduler
logical memory
Logical
File Systems
protocols
HI subsystems
generic HW access
interrupt context
Page Allocator
Block devices
virtual
network device
abstract devices
and
HID class drivers
devices access
and bus drivers
CPU specific
physical memory
operations
disk
controllers drivers
network
devices drivers
HI peripherals
device drivers
I/O
CPU
memory
disk controllers
network controllers
user peripherals
sys_execve
sys_fork
sys_brk
sys_mmap2
sys_open
sys_read
sys_write
sys_socketcall
sys_socket
socket_file_ops
kmalloc
kmem_cache
vfs_read
vfs_write
inet_create
alloc_skb
dev_queue_xmit
e100_open
ipw2100_open
© 2007 Constantine Shulyupin www.MakeLinux.net/kernel_map
Ver 0.6, 1/1/2008

Where can I get some?

- PF_RING
  - Linux
  - open-source
- Netmap
  - FreeBSD
  - open-source
- Intel DPDK
  - Linux
  - License fees
  - Third party support
    - 6WindGate

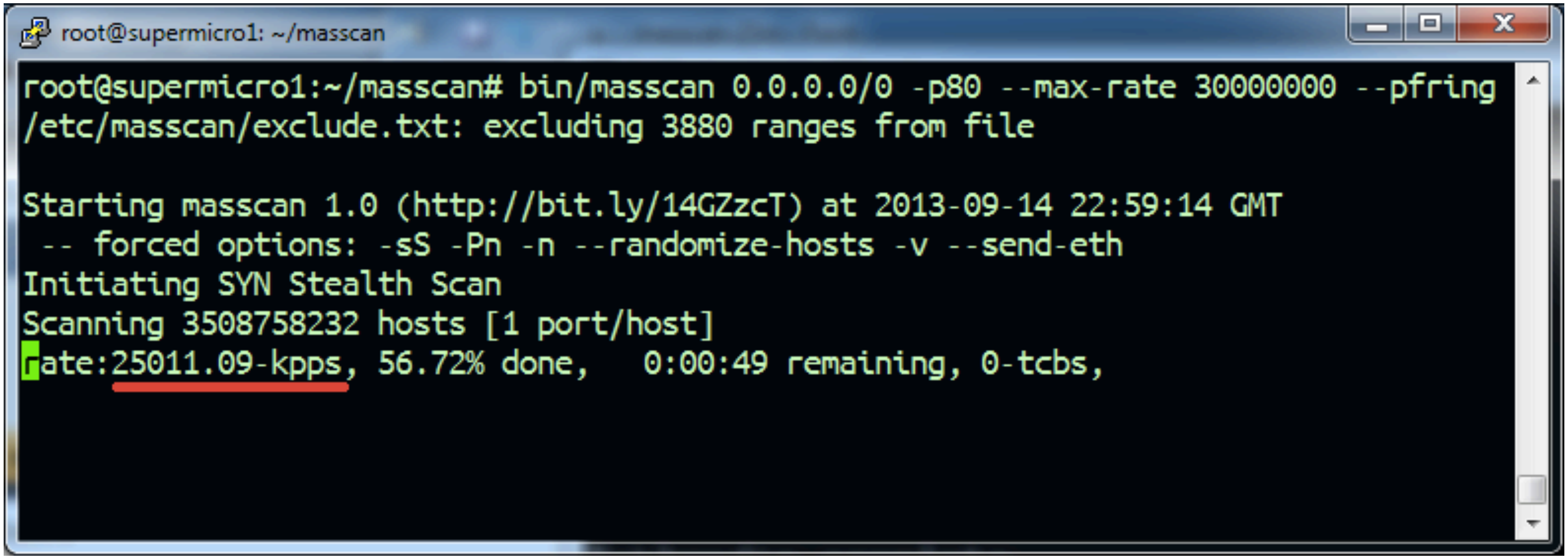
200 CPU clocks per packet



http://www.intel.com/content/dam/www/public/us/en/documents/solution-briefs/communications-packet-processing-brief.pdf

masscan

- Quad-core Sandy Bridge 3.0 GHz



```
root@supermicro1:~/masscan# bin/masscan 0.0.0.0/0 -p80 --max-rate 30000000 --pfring
/etc/masscan/exclude.txt: excluding 3880 ranges from file

Starting masscan 1.0 (http://bit.ly/14GZzcT) at 2013-09-14 22:59:14 GMT
 -- forced options: -sS -Pn -n --randomize-hosts -v --send-eth
Initiating SYN Stealth Scan
Scanning 3508758232 hosts [1 port/host]
rate:25011.09-kpps, 56.72% done,   0:00:49 remaining, 0-tcbs,
```

Premature optimization is good

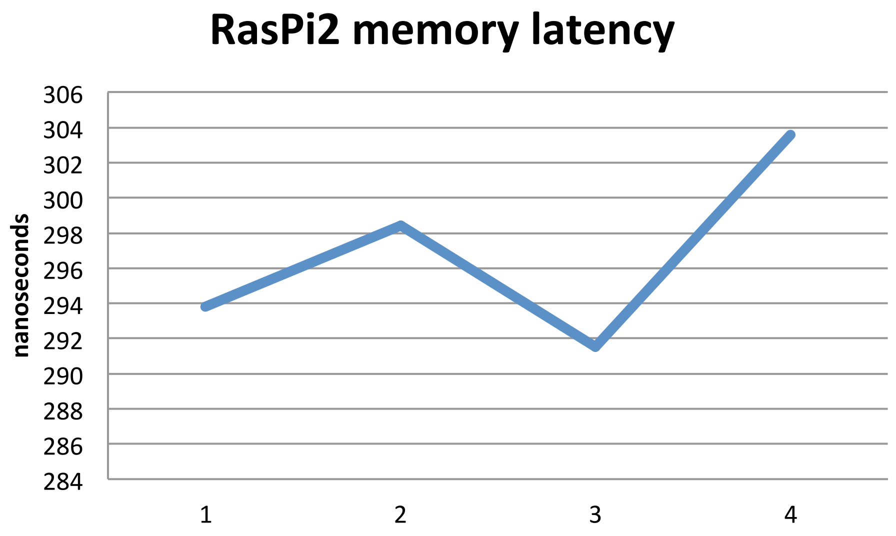
- Start with prototype that reaches theoretical max
  - Then work backwards
- Restate the problem so that it can be solved by the best solutions
  - Ring-buffers and RCU (read-copy-update) are the answers, find problems solved by them
- Measure and identify bottlenecks as they occur

Raspberry PI 2

900 MHz quad core ARM w/ GPU

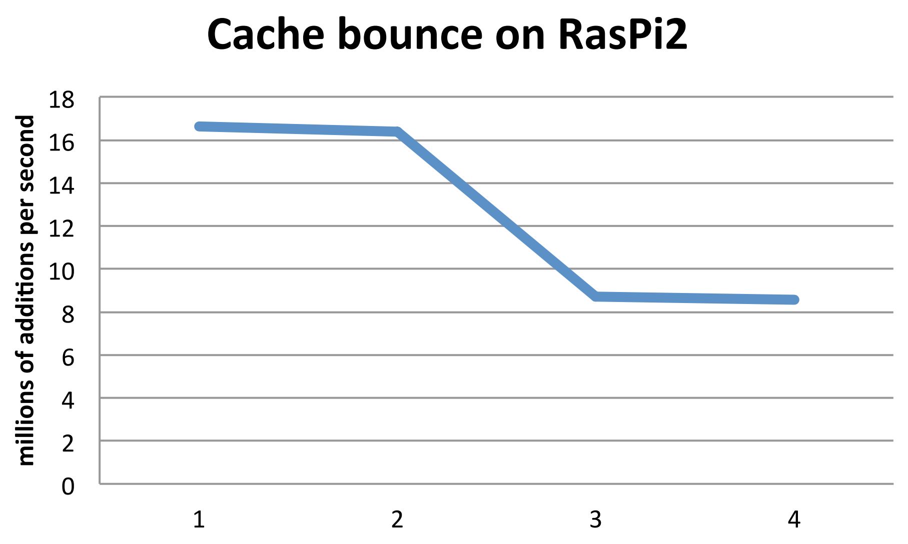
Memory latency

- High latency Probably due to limited TLB resources
- Didn't test max outstanding transactions, but should be high for GPU



Cache Bounce

- Seems strange
- No performance loss for two threads



- Answer: ARM Cortex-A8 comes in 2-cpu modules that share cache

Compared to x86

- .

	ARM	x86	Speedup
Hz	0.900	3.2	3.6
syscall	0.99	2.5	2.6
funcall	59.90	556.4	9.3
pipe	0.17	2.5	14.8
ring	3.90	74.0	19.0

Todo:

- C10mbench work
  - More narrow benchmarks to test things
  - Improve benchmarks
  - Discover exactly why benchmarks have the results they do
  - Benchmark more systems
    - Beyond ARM and x86