Make your code faster by moving it out of the kernel

The anti-kernel

About this talk

- Old: OS principles
  - For doing things the OS wants you to do
  - Moving into the kernel
- New: User-land principles
  - For breaking the rules
  - Moving out of the kernel
  - That includes hardware drivers

What OS does for you

- Locks (spinlocks, mutexes)
- IPC
- Processes/threads/scheduling
- Memory allocation
- Security
- Network stack

What you can do for yourself in userland, better

- Locks (spinlocks, mutexes)
- IPC
- Processes/threads/scheduling
- Memory allocation
- Security
- Network stack

Q: Why is moving into the kernel faster?

- A: Cost of system calls
  - Function call = ~1.8 nanoseconds
  - System call = ~30ns to ~400ns
  - Security supervisory checks are expensive
- A: virtual memory vs. physical memory
  - Virtual addresses must be translated to physical addresses
  - TLB misses require page table walk

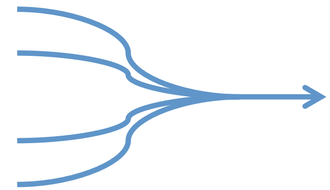
Q: So why is moving out of kernel better?

- A: because you can − even hardware drivers
- A: because you also avoid system calls and context switches
- A: because the cost of virtual memory can be mitigated

- A: because user-mode software is more robust, more secure, easier to debug

Multi-threading is not the same as multi-core

- Old: Multi-threading
  - Safety!
  - More than one thread per CPU core
  - Spinlock/mutex must therefore stop one thread to allow another to execute

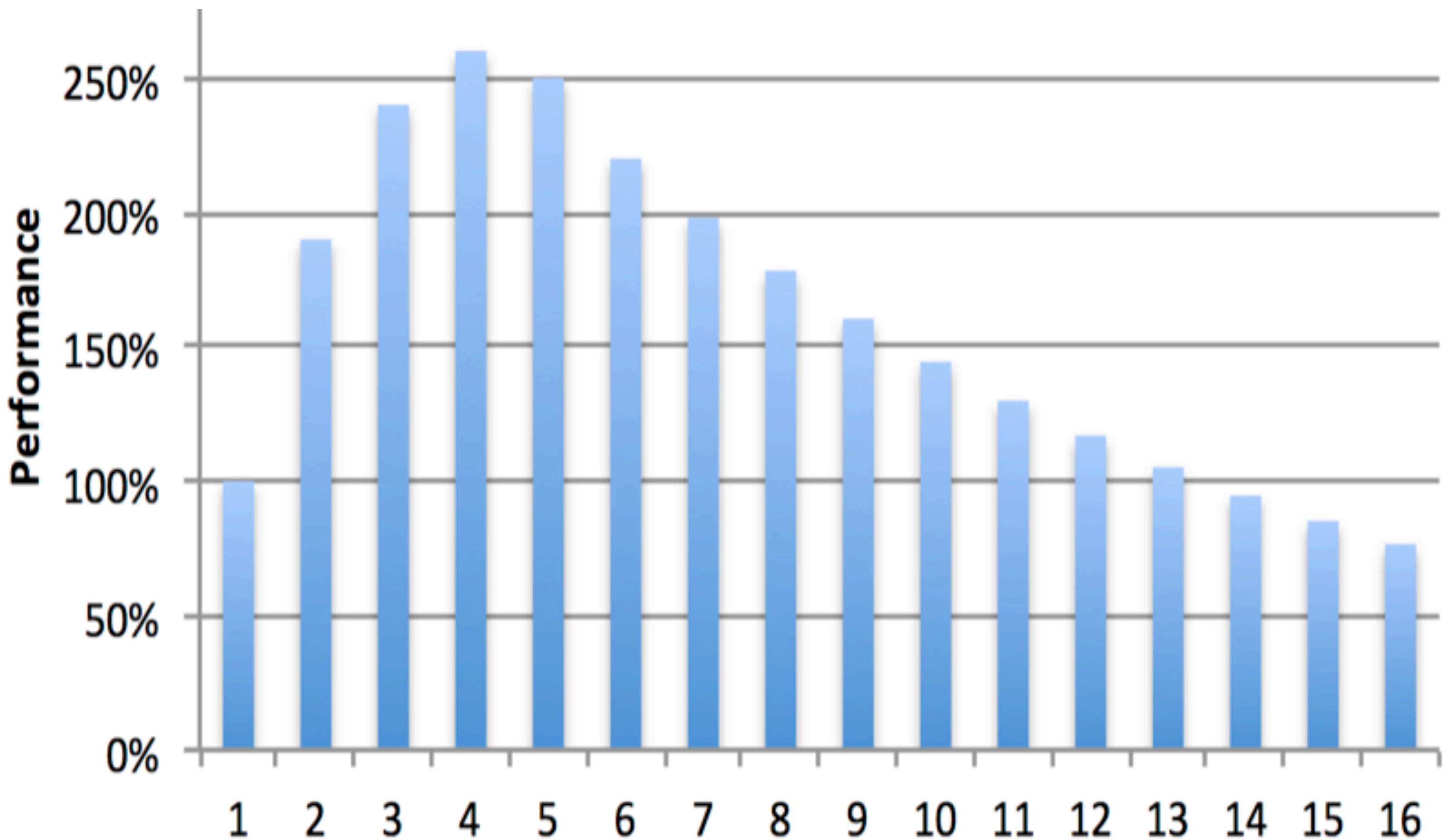


- New: Multi-core
  - Speed!
  - One thread per CPU core
  - When two threads/cores access the same data, they can't stop and wait for the other

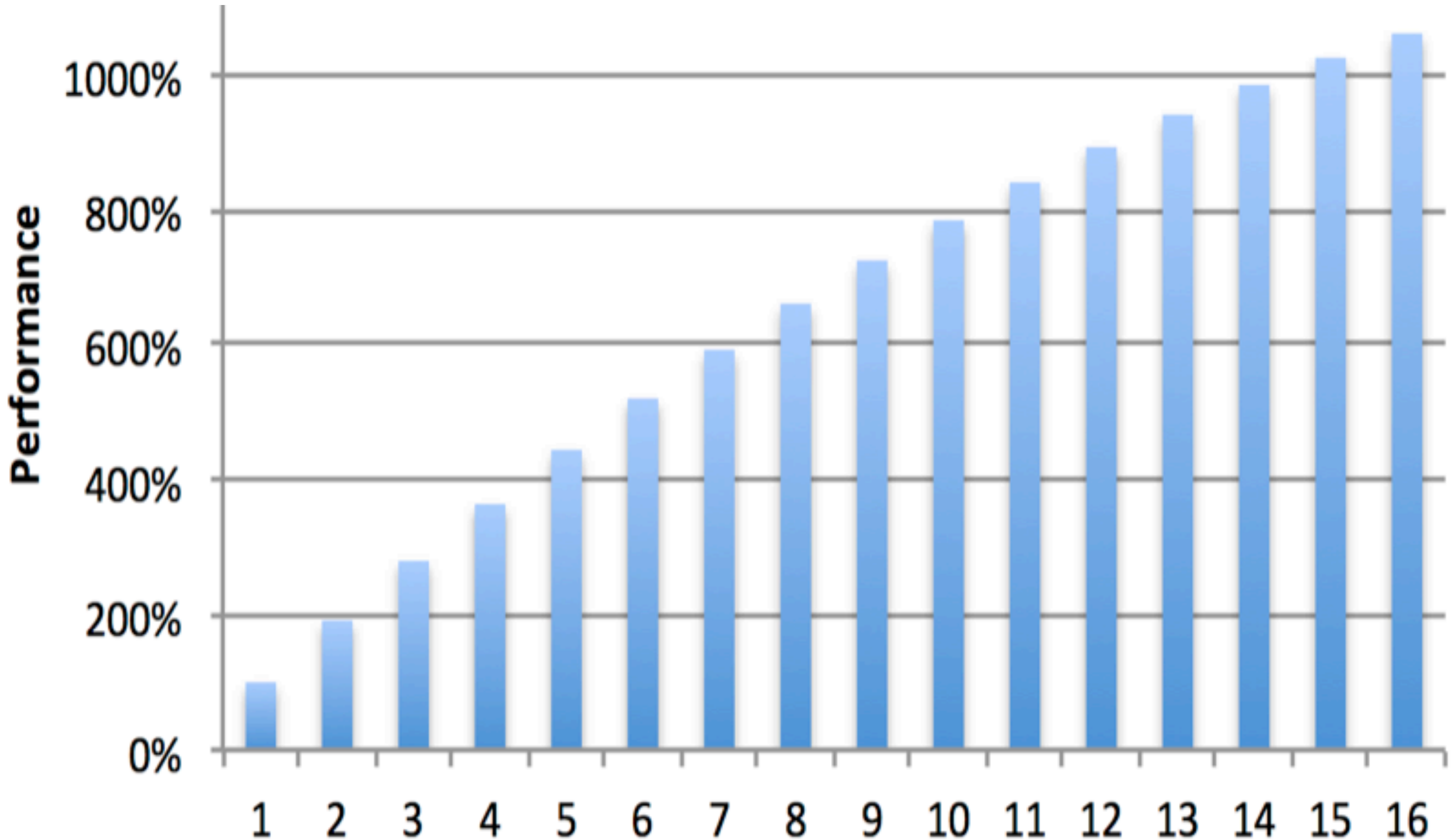
Contrast: spinlocks/mutexes

- What you were taught: better locks
- What you need to know: no locks
  - Linux kernel rapidly getting rid of spinlocks
  - Replacing with RCU and other lock-free/wait-free techniques

Most code doesn’t scale past 4 cores



At Internet scale, code needs to use all cores



Reads *or* writes are atomic

- Reading an aligned integer from memory is always an atomic operating
  - You'll never get a partial integer
  - On any CPU
- Same with writes

Example: network interface stats

```
john@bt:~$ ifconfig eth1
eth1      Link encap:Ethernet  HWaddr 00:0c:29:34:90:e7
          inet addr:172.16.134.128  Bcast:172.16.134.255  Mask:255.255.255.0
          inet6 addr: fe80::20c:29ff:fe34:90e7/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:201255 errors:0 dropped:0 overruns:0 frame:0
          TX packets:133251 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:101236165 (101.2 MB)  TX bytes:13531654 (13.5 MB)
          Interrupt:19 Base address:0x2000

john@bt:~$
```

Read-Copy-Update

- How it works
  - Make a copy of a data structure
  - Make all your changes
  - Replace the pointer, to your new one vs old one
  - Wait until all threads move forward
    - Now point to new data structure
  - Free old data structure
- Non-blocking
  - …for readers
  - …only one writer/changer at a time

Atomics

- Special hardware instructions
  - `cmpxchg16b`
  - `lock add`
  - `TSX`
- Intrinsics/libraries
  - __sync_add_and_fetch()
  - InterlockedExchangeAdd()

Non-blocking algorithms

- “Lock-free”
  - At least one thread makes forward progress
- “Wait free”
  - All threads make forward progress
  - Upper bound to number of steps any thread make make

Non-blocking algorithms

- Queues
- Hash-tables
- Ring-buffers
- Stacks, sets, etc.

Performance tradeoffs

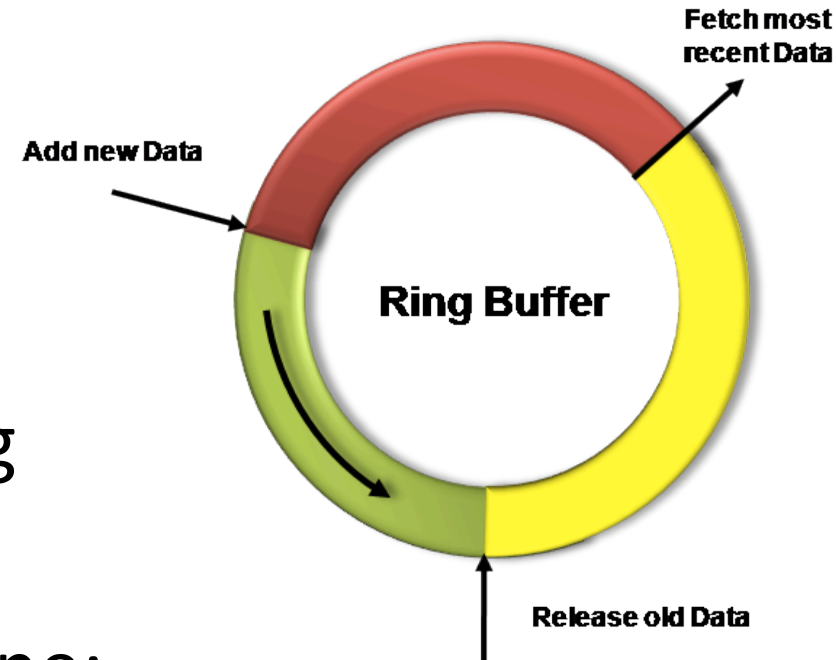
- You need to select the lock-free data structure that fits your needs
  - How frequent will contention be?
  - How many readers/writers?
  - Theoretical wait-free or lock-free?
- Sometimes the traditional spin lock is best
  - Especially for things with low contention

Most important synchronization issue

- Avoid more than one CPU changing data at the same time
- No amount of clever programming saves you if two CPUs are changing the same cache line on a frequent basis

Solution: lock-free ring-buffers Single reader, single writer

- 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



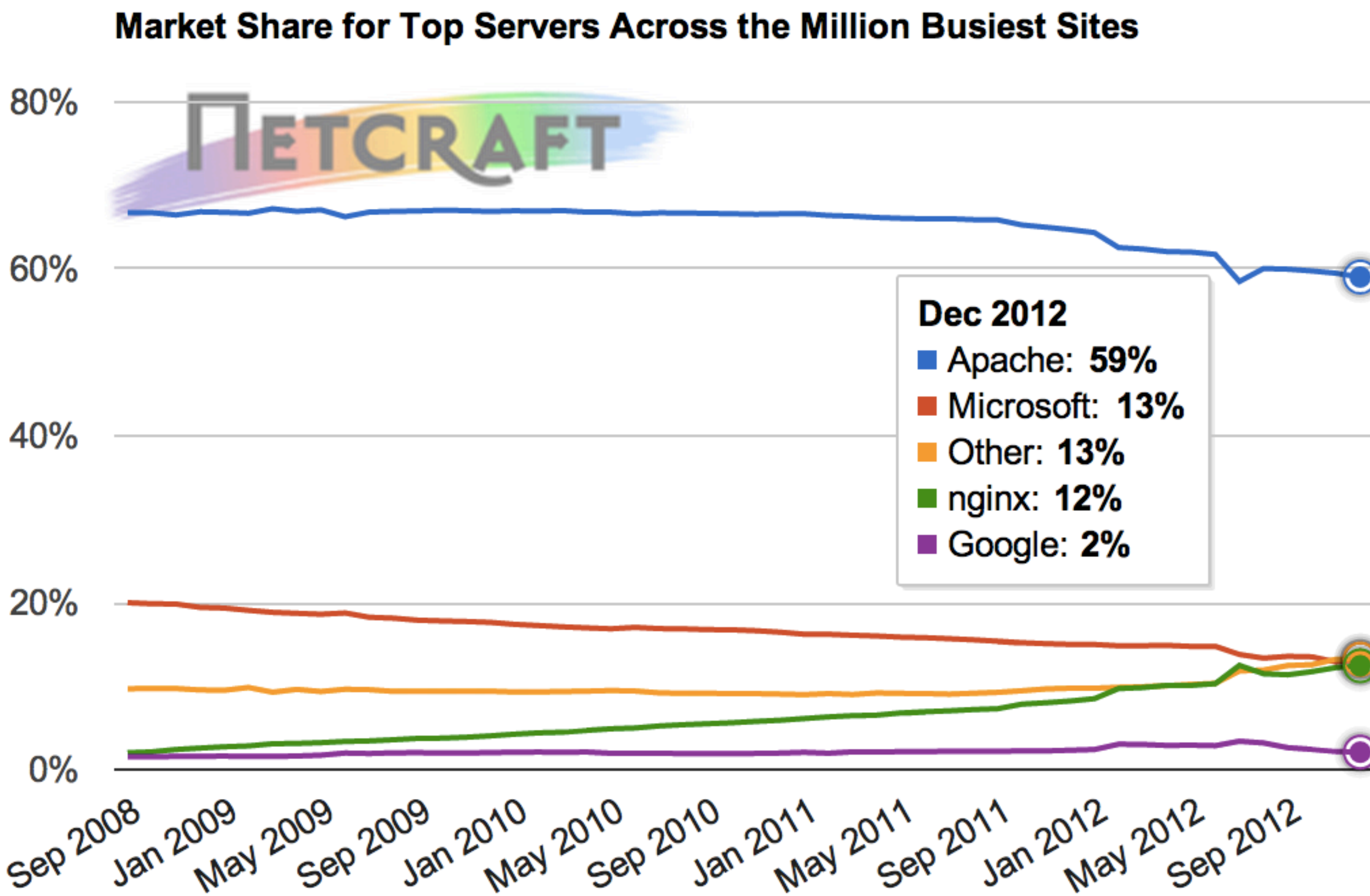
Contrast: Inter Process Communication

- Old school
  - Signals
  - Pipes
    - A hundred thousand messages/second
- New school
  - Shared memory
  - Lock-free queue
    - Millions of messages/second
  - Ring-buffer
    - Tens of millions of messages/second

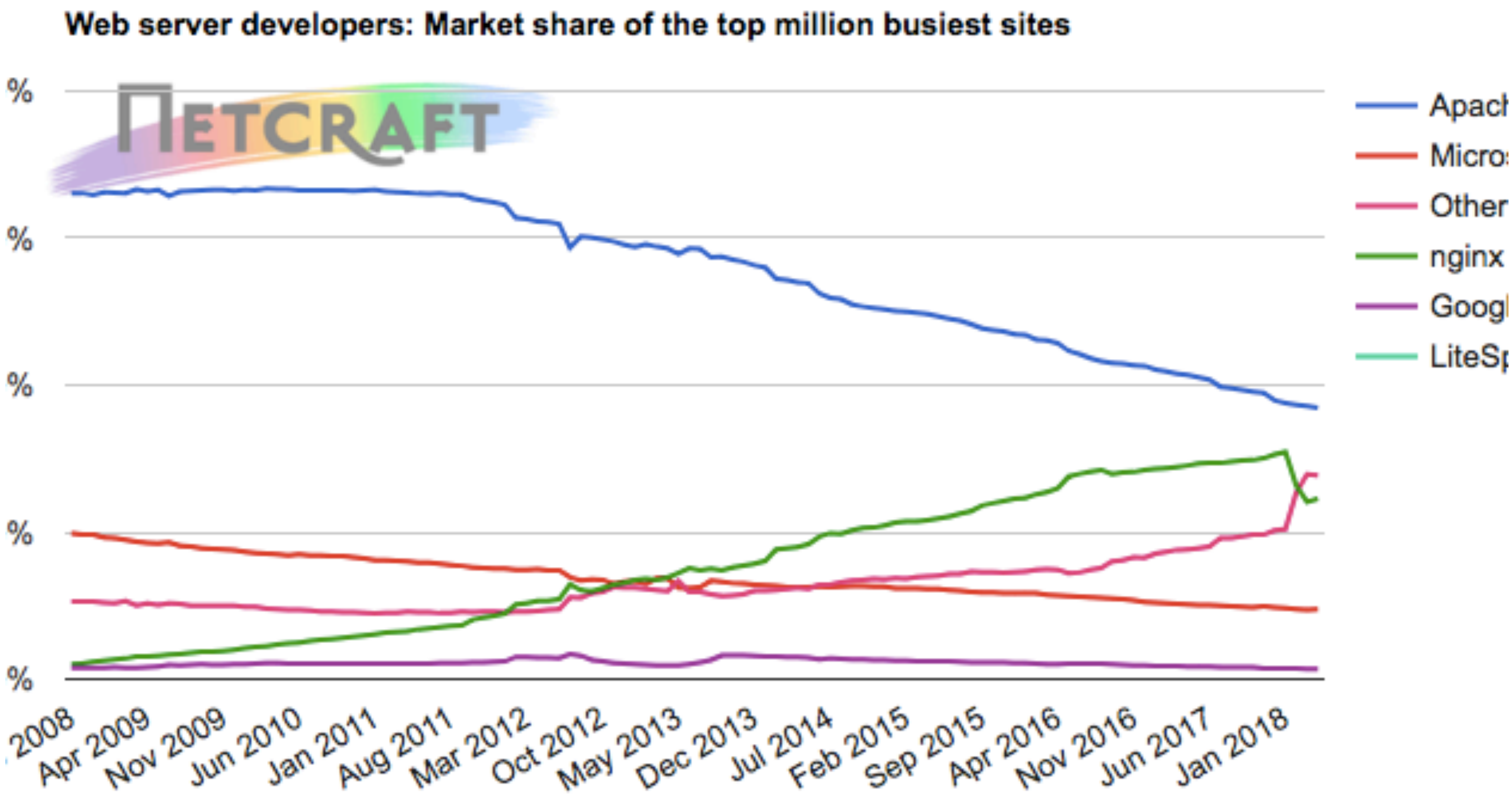
Contrast: thread scheduler

- When you do this
  - When there are more processes/threads than CPUs
  - E.g. Apache with has one process/thread per TCP connection
    - For 10,000 connections
- When you don’t do this
  - When you are spreading a single task across many CPUs
  - E.g. Nginx which has one thread per CPU
    - For 1-million TCP connections

Q: Why?
A: Scale


Market Share for Top Servers Across the Million Busiest Sites
80%
60%
40%
20%
0%
NETCRAFT
Dec 2012
Apache: 59%
Microsoft: 13%
Other: 13%
nginx: 12%
Google: 2%
Sep 2008
Jan 2009
May 2009
Sep 2009
Jan 2010
May 2010
Sep 2010
Jan 2011
May 2011
Sep 2011
Jan 2012
May 2012
Sep 2012

Scalable solutions



Asynchronous programing (one thread per CPU)

- API
  - Epoll, kqueue, completion-ports
- Examples
  - Libuv, libev, libevent
  - Nginx
  - NodeJS
    - Bare-metal programming with JavaScript

“User-mode threads”

- Co-operative multitasking
  - Old: read() context switches in kernel
  - New: read() is function call return
- Examples
  - Coroutines in Lua
  - Goroutines in Go
  - Java green threads
  - Windows ‘fibers’

Case study: Erlang programming language

- Everything is a user-mode “process”
- No memory sharing…
- …message passing between processes instead

Case study: go

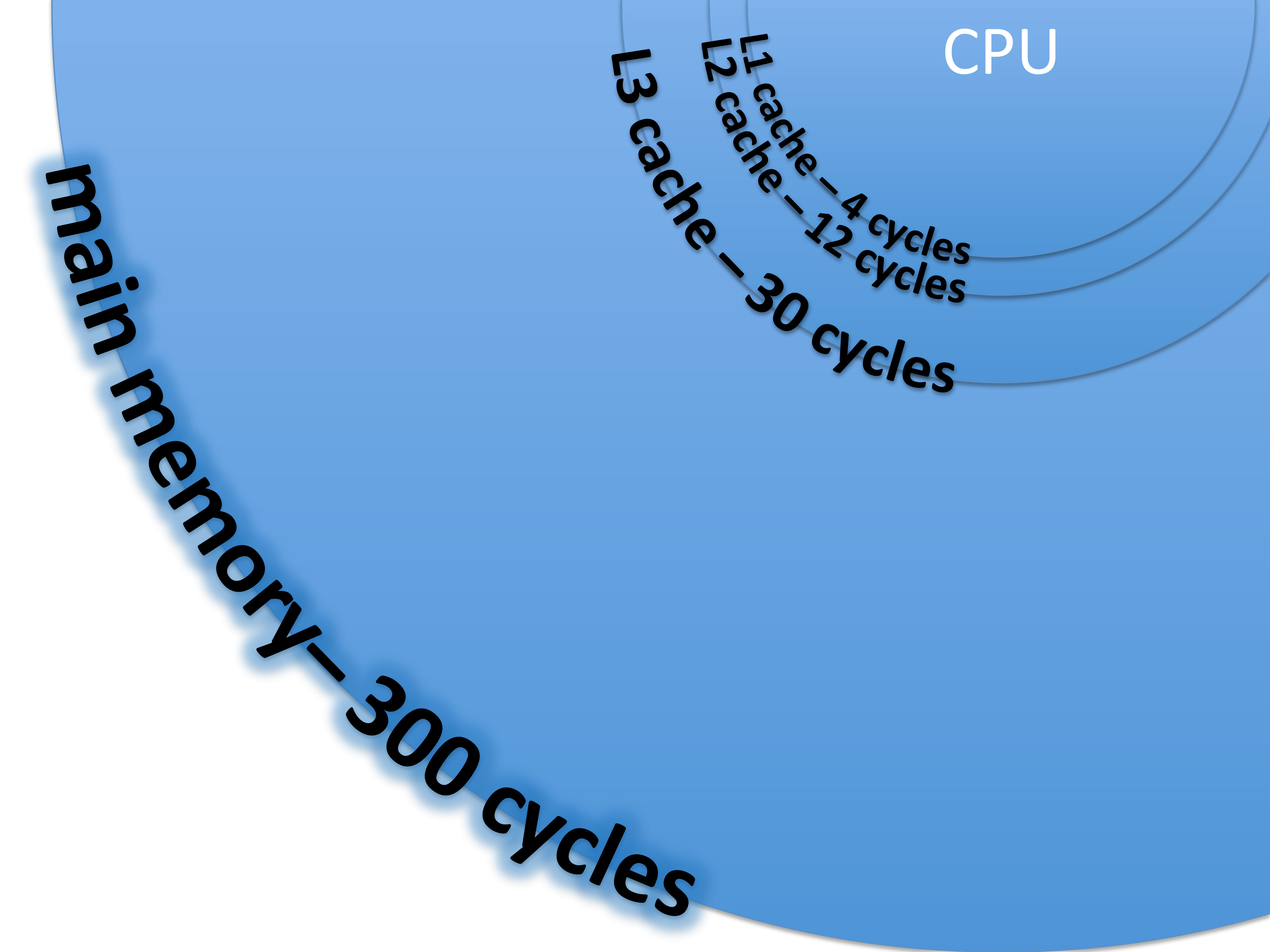
- Massive use of ‘goroutines’
- Message passing via ‘channels’ between goroutines

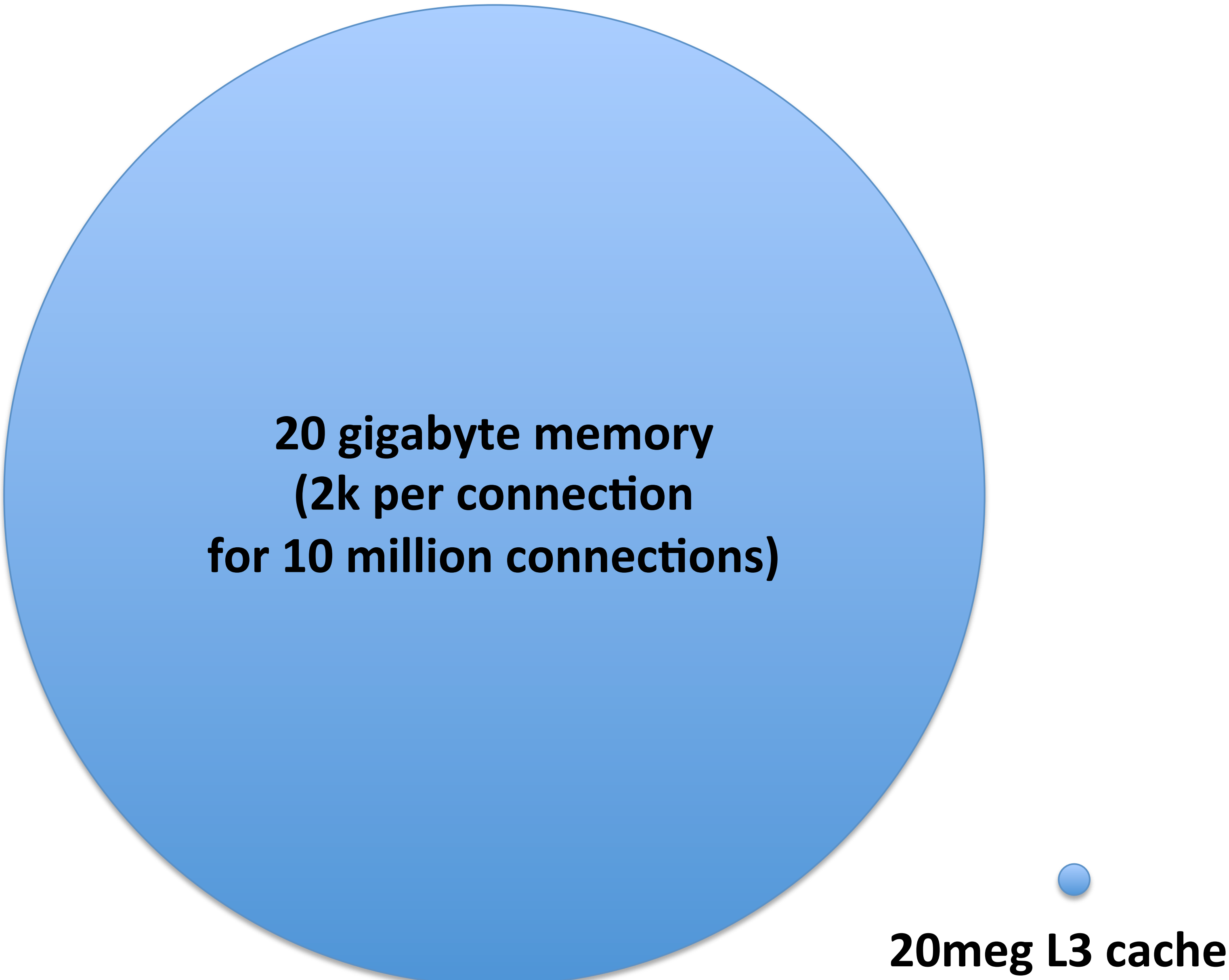
Case study: OpenResty

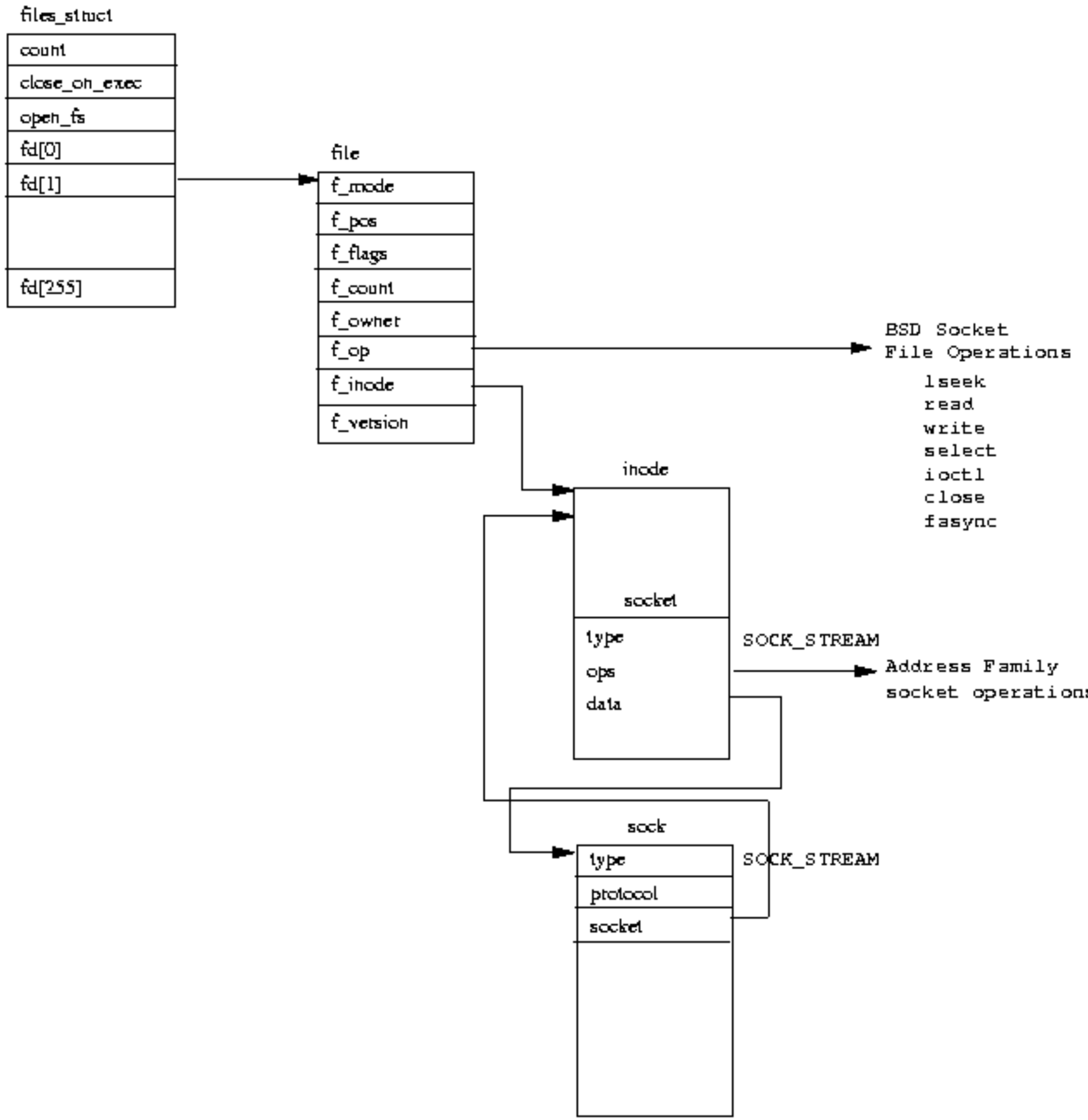
- Lua within nginx
- Lowest cost coroutines per TCP connection

Contrast: dealing with memory

- Old school
  - Many processes
  - All doing moderate memory
  - Fairness
- New school
  - One process (dedicated appliance)
  - Allocating most all the memory in a system
  - Unfairness


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


files_stuct
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

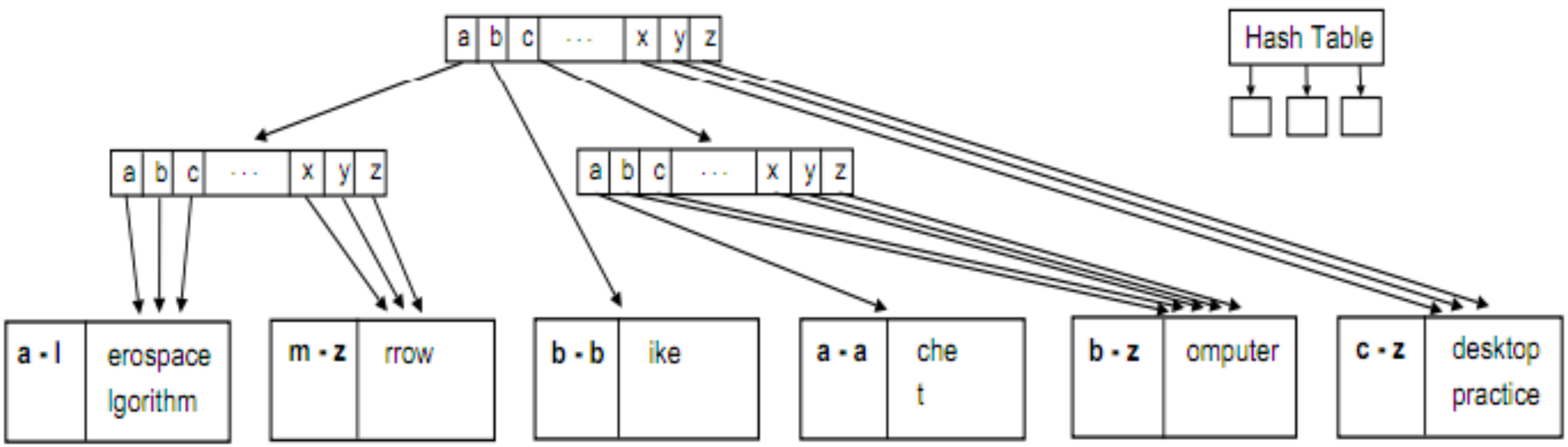
co-locate data

- Don't: data structures all over memory connected via pointers
  - Each time you follow a pointer it'll be a cache miss
  - [Hash pointer] -> [TCB] -> [Socket] -> [App]
- Do: all the data together in one chunk of memory
  - [TCB | Socket | App]

compress data

- Bit-fields instead of large integers
- Indexes (one, two byte) instead of pointers (8-bytes)
- Get rid of padding in data structures

“cache efficient” data structures

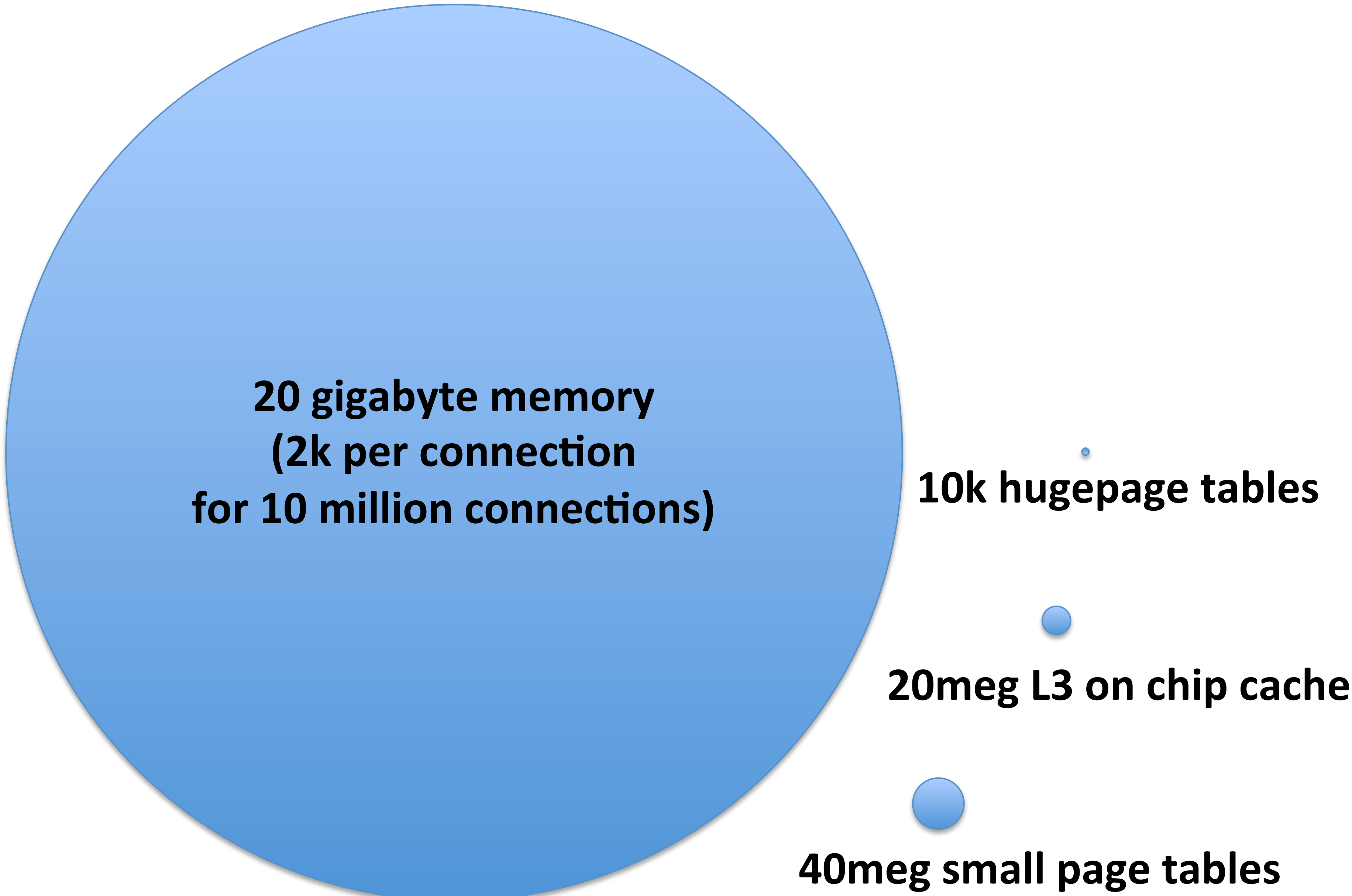


Virtual memory (aka. memory address translation)

- Review why moving code to kernel faster
  - Avoid system call
  - Avoid virtual to physical memory address translation
- Spectre/Meltdown
  - Has made this cost much worse
  - Old: kernel memory was mapped into user-memory, memory cache is the same
  - New: cache flushes a lot on each system call

“huge pages”

- What is it?
  - 2 megabyte pages instead of 4 kilobytes
  - (a chunk of 512 smaller pages)
- At scale page tables won’t be in cache
  - Thus, uncached memory lookups require two memory lookups
  - It’s a big reason why kernel code (no virtual memory) is faster than user-mode
    - That, and no transitions
- Linux auto-hugepage
  - Linux now automatically gives huge pages underneath


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

Contrast

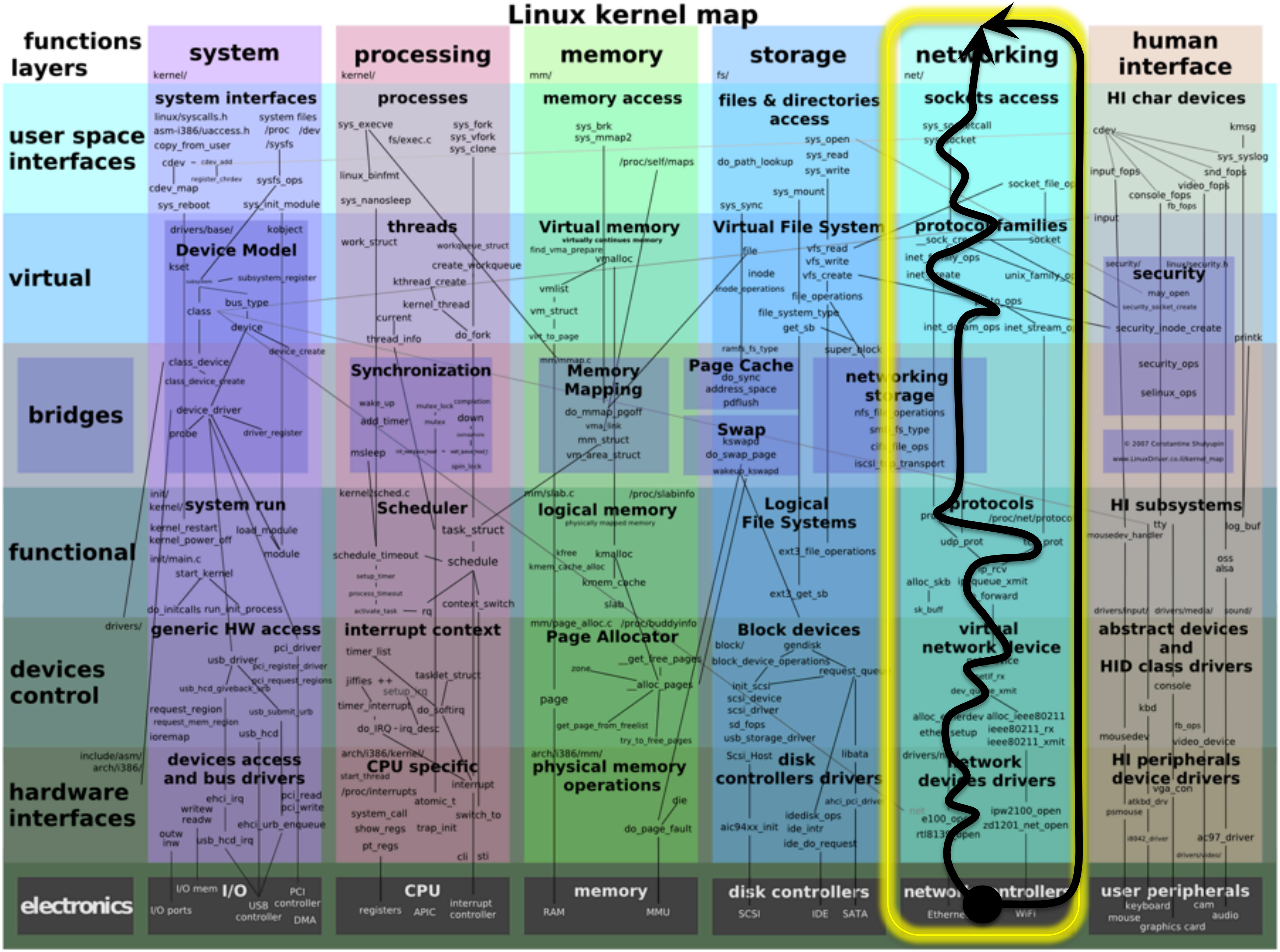
- Old school
  - Many users on the same system
- New school
  - Appliance dedicated to a single task
  - Any security you'll have to do yourself
  - E.g. CloudFlare revealing uninitialized memory

Review: system call speed

- System calls require security check on every call
- Unnecessary for dedicated appliances
- Application code still needs to do network security checks

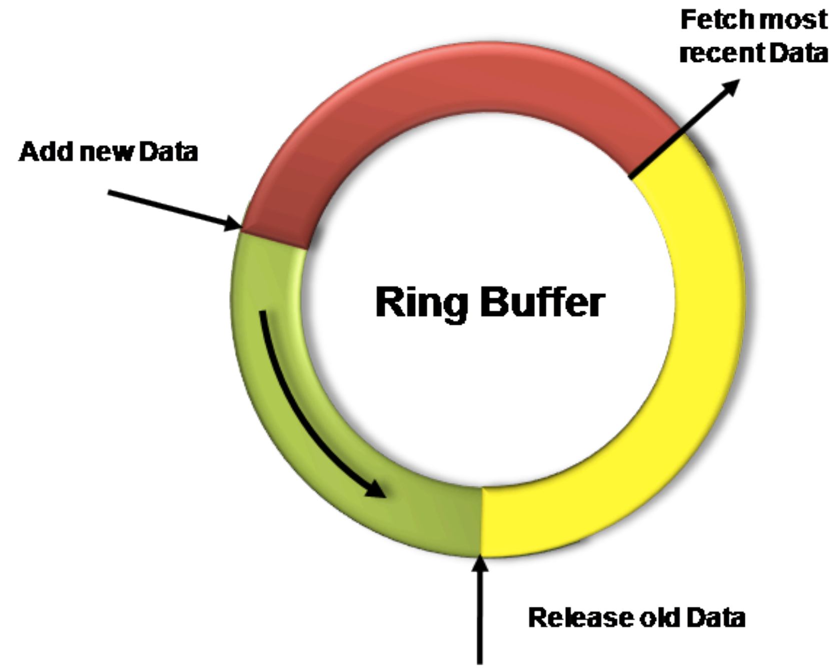
Contrast: network

- Old school
  - A network stack for many processes
- New school
  - Your own network stack, not shared with others
  - Yes, even hardware drivers, especially hardware drivers



Ver 0.6, 1/1/2008

All drivers use packet ring buffers



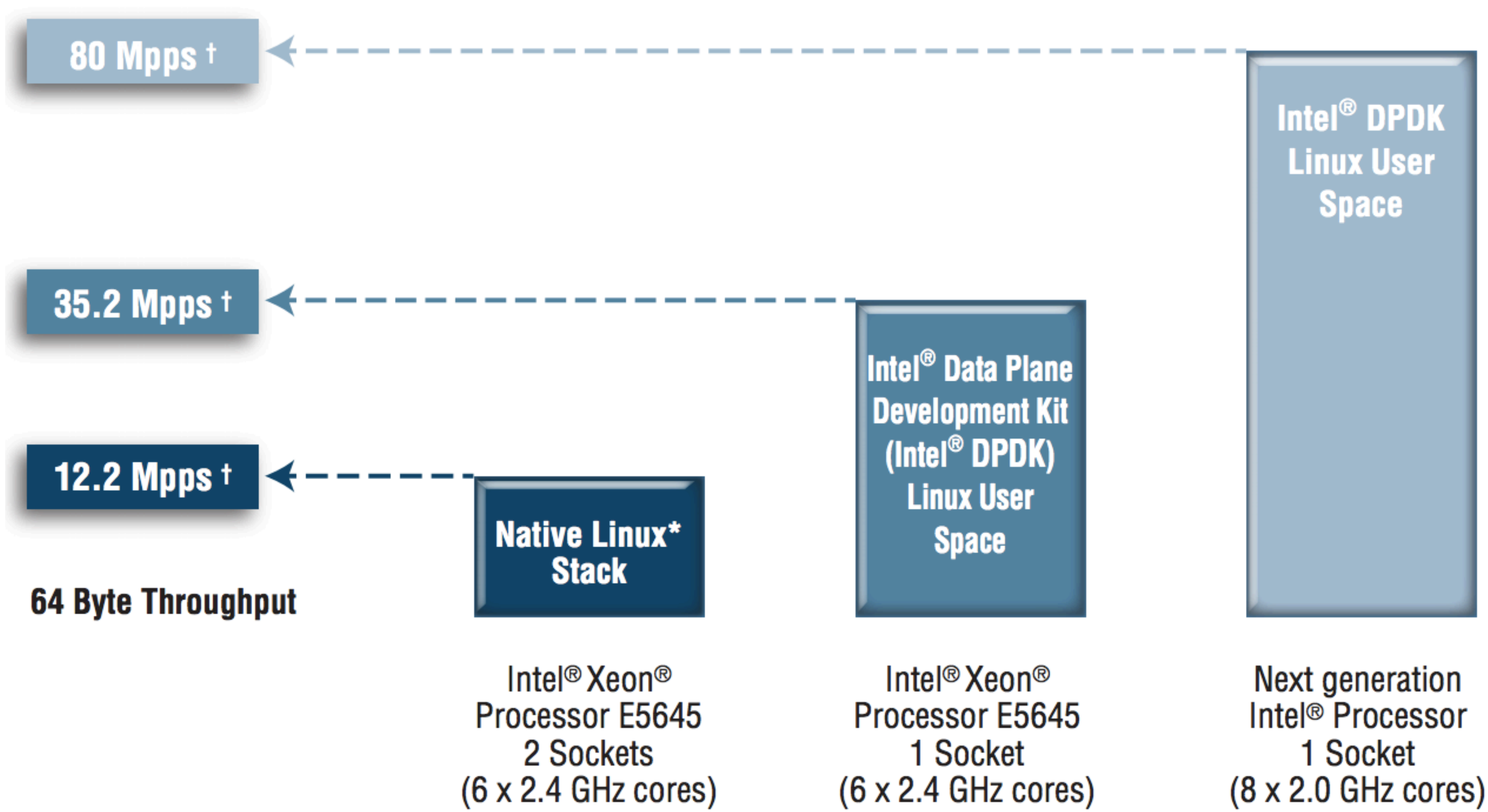
Trick

- Memory map ring buffer into user-mode
- User-mode cost
  - ~100 clock cycles to mark buffer as free
  - Packet parsed in-place inside the ring buffer
- Kernel cost
  - Must move packet out of ring buffer before processed
  - Because many apps may be using the network
  - Must either copy or alloc new buffer to replace old one
  - ~1000s clock cycles per packet

Where can I get some?

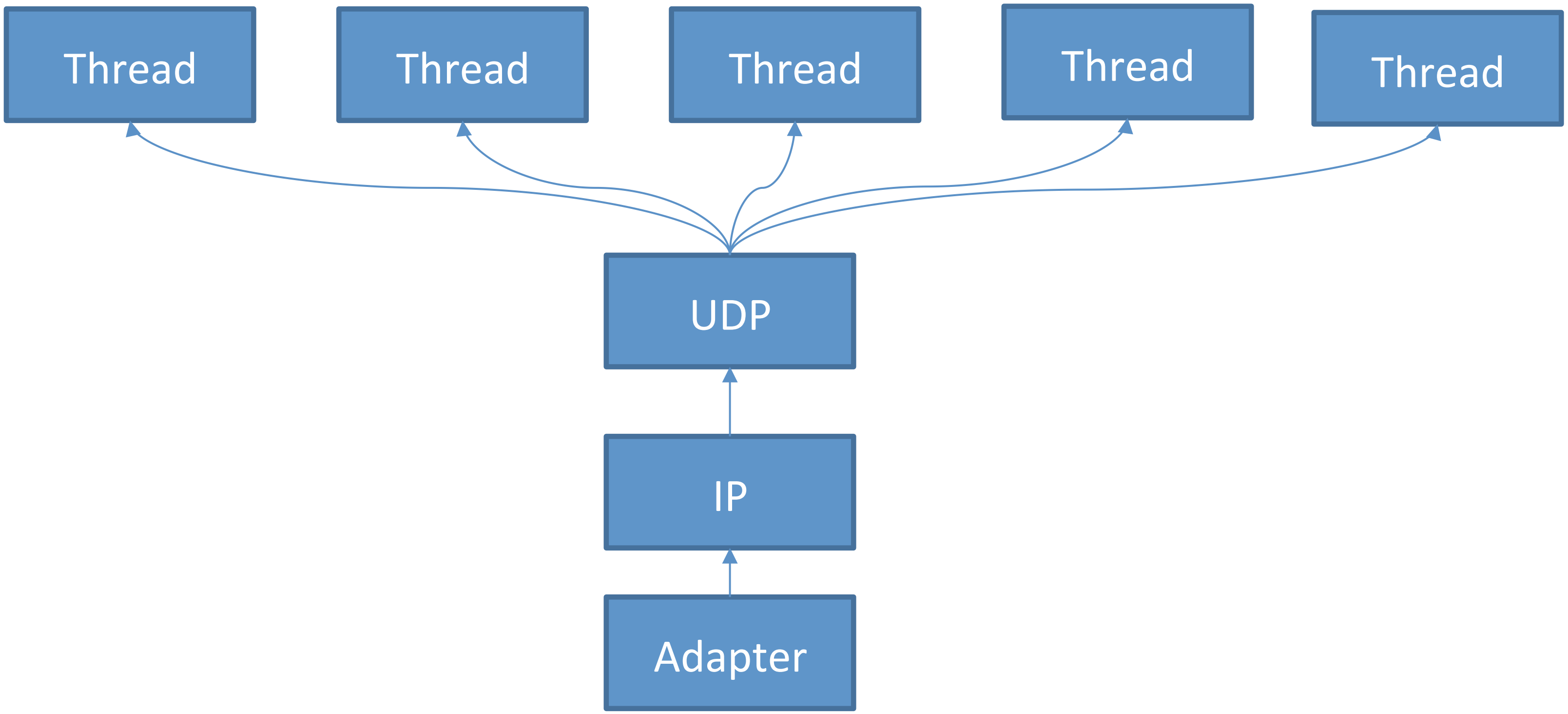
- PF_RING
  - Linux
  - open-source
- Netmap
  - FreeBSD
  - open-source
- Intel DPDK
  - Linux
  - License fees
  - Third party support
    - 6WindGate

200 CPU clocks per packet

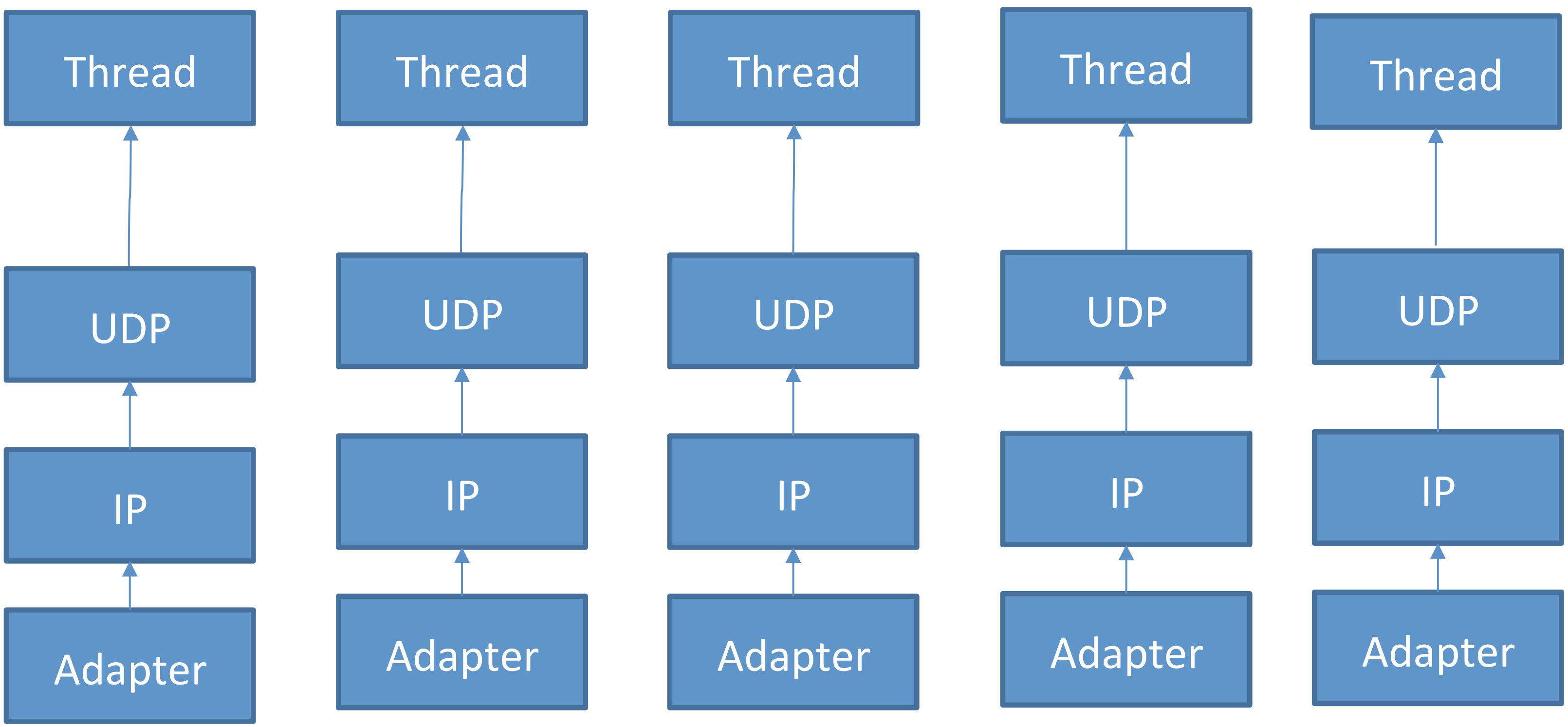


http://www.intel.com/content/dam/www/public/us/en/documents/solution-briefs/communications-packet-processing-brief.pdf

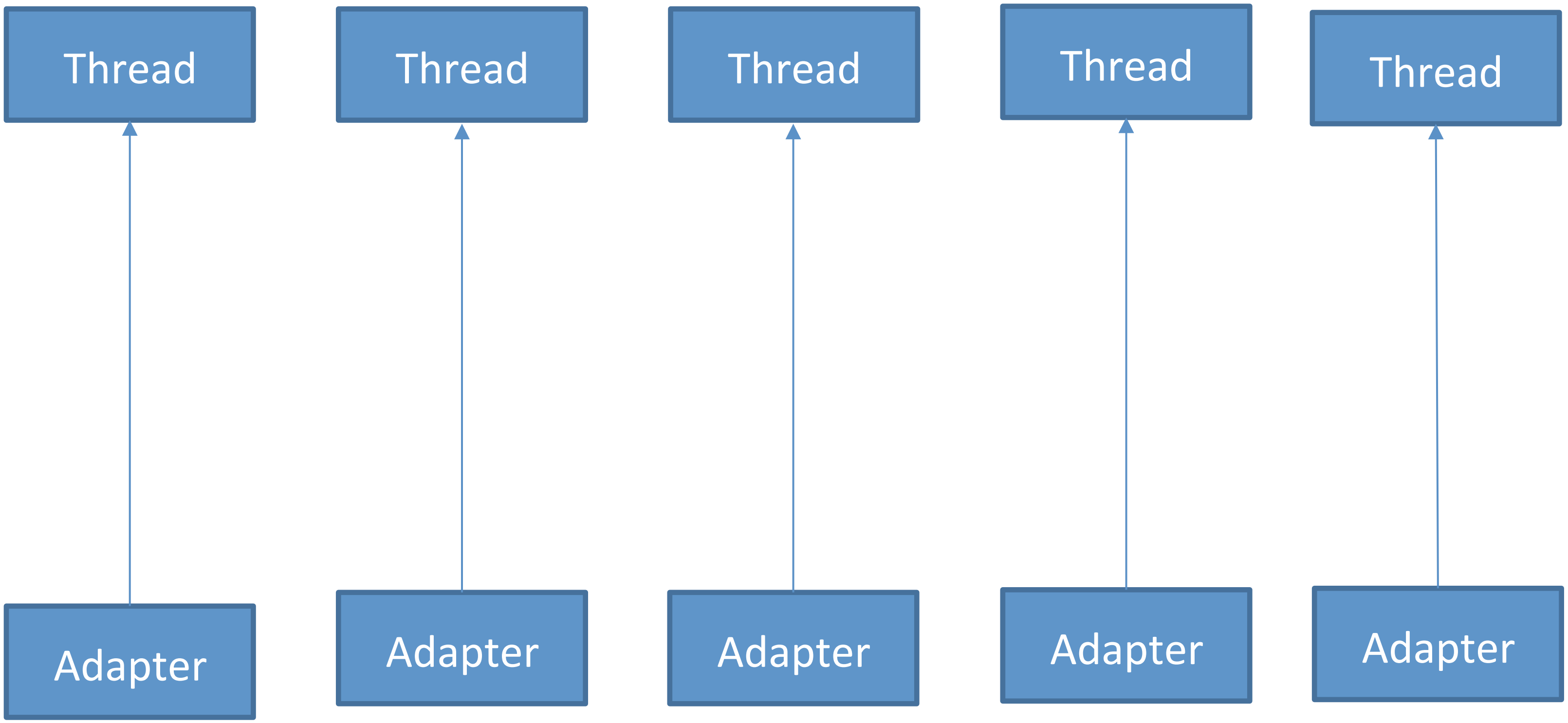
Old UDP: 500 kpps



UDP + receive queues + SO_REUSEPORT = 3 mpps



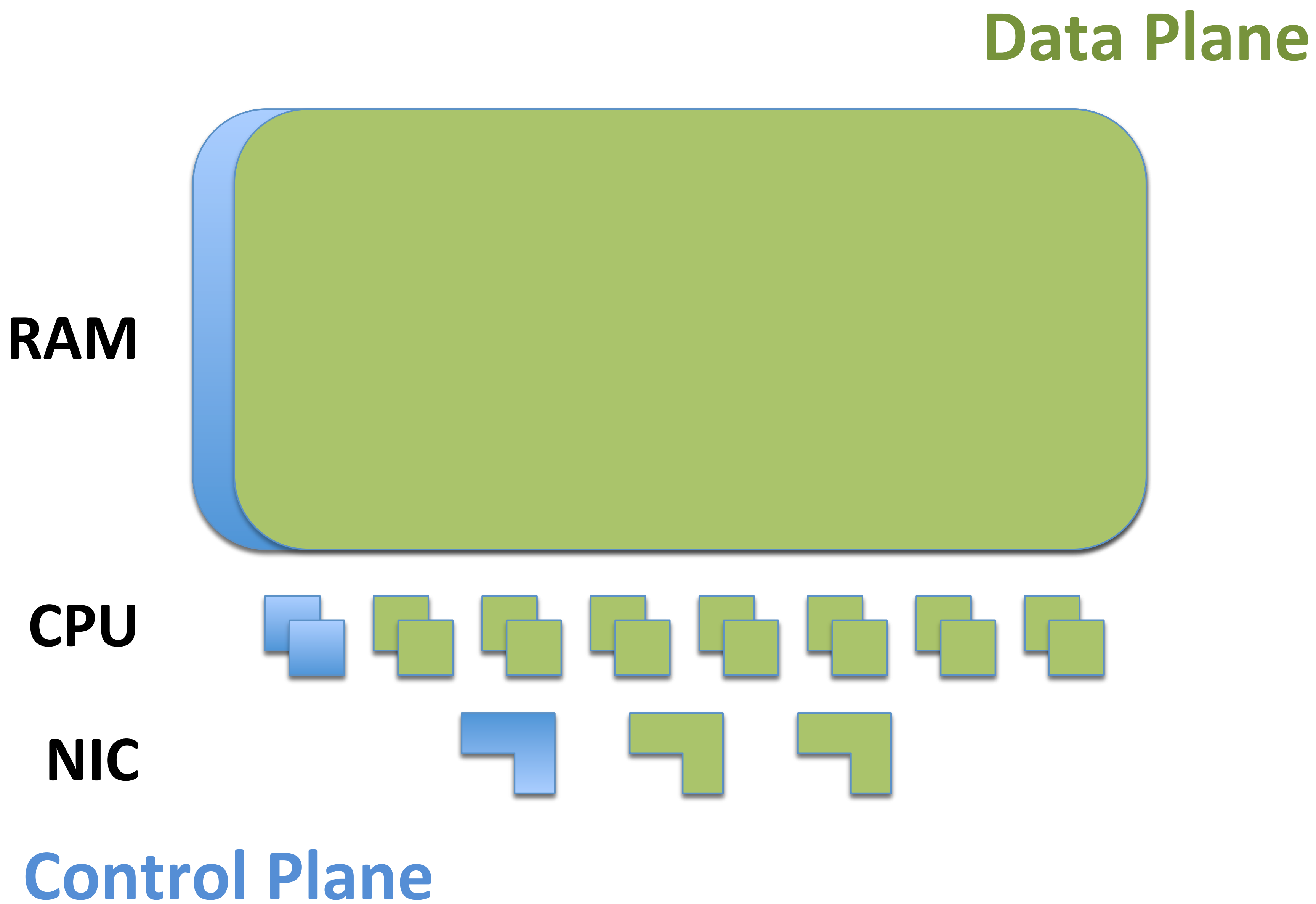
Custom = 30 mpps



User-mode network stacks

- PF_RING/DPDK get you raw packets without a stack
  - Great for apps like IDS or root DNS servers
- For TCP, there are commercial stacks available
  - 6windgate is the best known commercial stack, working well with DPDK
  - Also, some research stacks
  - Requires change in your software to exploit them, such as asynchronous

Control plane vs. Data plane


Data Plane
RAM
CPU
NIC
Control Plane

Case study: masscan

- What is…
  - Ports scans entire Internet
  - Like nmap, but more scalable
- Transmits 30-million packets/second
  - PF_RING user-mode ring-buffer
- Pointer to TCB, then TCB, containing all data
- Has it’s own IP address(s)
  - Even when shared with machine

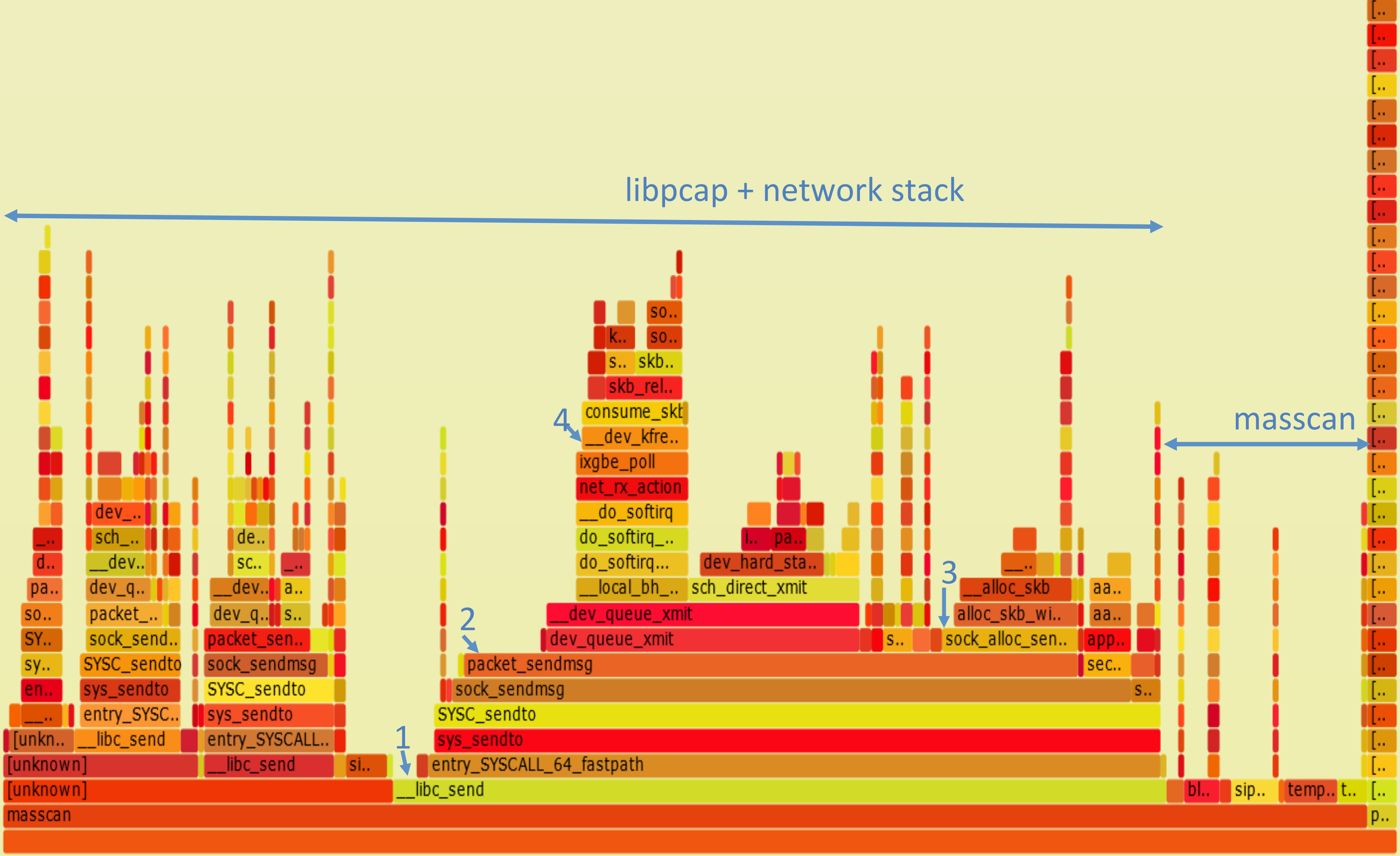
masscan

- Quad-core Sandy Bridge 3.0 GHz

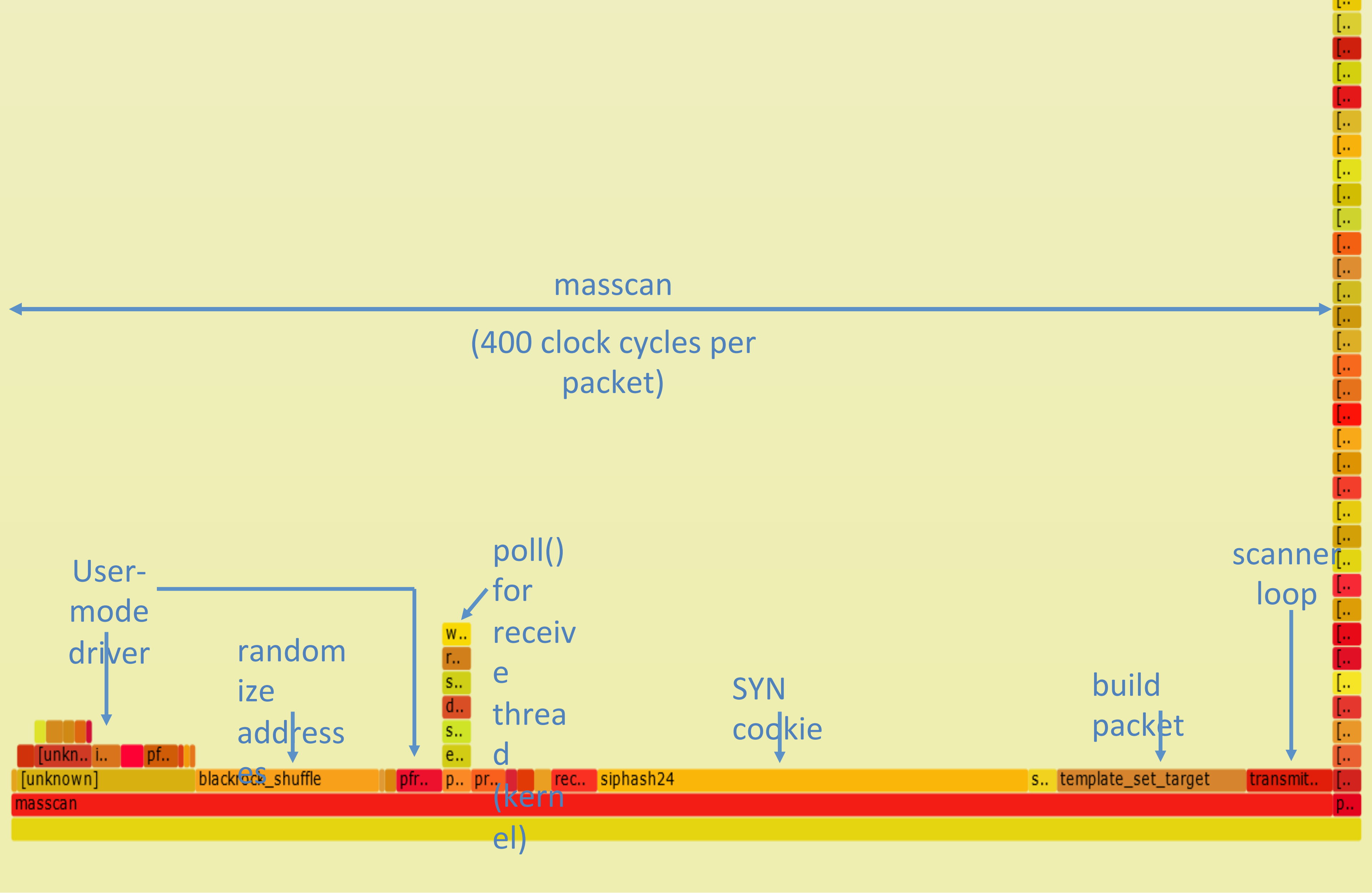
root@supermicro1: ~/masscan

```
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,
```



libpcap + network stack
masscan
1
2
3
4
so..
k..
s..
skb..
skb_rel..
consume_skb
__dev_kfre..
ixgbe_poll
net_rx_action
__do_softirq
do_softirq_..
do_softirq...
__local_bh_..
sch_direct_xmit
dev_hard_sta..
i..
pa..
__alloc_skb
alloc_skb_wi..
sock_alloc_sen..
aa..
app..
sec..
__dev_queue_xmit
dev_queue_xmit
packet_sendmsg
sock_sendmsg
SYSC_sendto
sys_sendto
entry_SYSCALL_64_fastpath
__libc_send
[unknown]
masscan
bl..
sip..
temp..
t..
p..
dev_..
sch_..
__dev..
dev_q..
packet_..
sock_send..
entry_SYSC..
[unkn..
de..
sc..
a..
packet_sen..
entry_SYSCALL..
si..
so..
SY..
sy..
en..
pa..
d..


masscan
(400 clock cycles per packet)
User-mode driver
randomize addresses
poll() for receive thread (kernel)
SYN cookie
build packet
scanner loop
[unkn..
i..
pf..
[unknown]
blackrock_shuffle
pfr..
p..
pr..
rec..
siphash24
s..
template_set_target
transmit..
masscan
p..
w..
r..
s..
d..
s..
e..

Conclusion

- Questions?
- @ErrataRob
- https://blog.erratasec.com
- https://github.com/robertdavidgraham/masscan