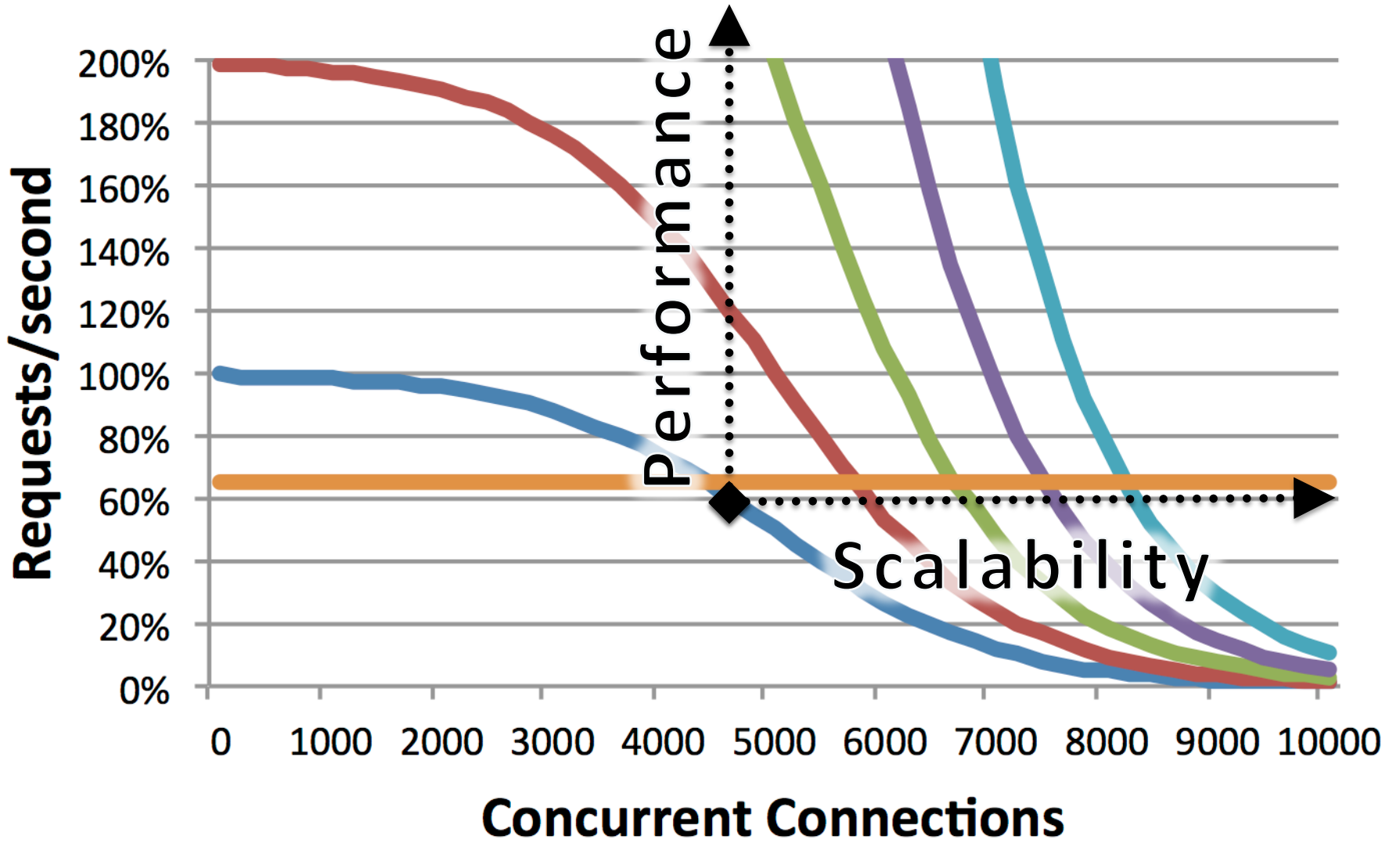


# C10M: Defending the Internet at scale

by Robert David Graham  
(Shmoocon 2013)



# who this talk is for

- coders
  - who write internet scale stuff
- everyone else
  - who manages internet scale stuff
  - who needs to know how stuff works
  - who needs to know what's possible
    - that the limitations you are familiar with are software not hardware

# how this talk is organized

- c10k – Internet scalability for the last decade
- C10M – Internet scalability for the next decade
- The kernel
- Asynchronous
- 1. packet scalability
- 2. multi-core scalability
- 3. memory scalability
- Questions
- ~~bonus. state machine parsers~~
- ~~bonus. attacks on scalability~~
- ~~bonus. language scalability~~

# c10k: a historical perspective



# Why servers could not handle 10,000 concurrent connections: $O(n^2)$

- Connection = thread/process
  - Each incoming packet walked list of threads
  - $O(n * m)$  where  $n$ =threads  $m$ =packets
- Connections = select/poll (single thread)
  - Each incoming packet walked list of sockets
  - $O(n * m)$  where  $n$ =sockets  $m$ =packets

# c10k solution

- First solution: fix the kernel
  - Threads now constant time context switch, regardless of number of threads
  - `epoll()/IOCompletionPort` constant time socket lookup

A word cloud centered around the word "asynchronous". The word "asynchronous" is the largest and most prominent, written in a dark blue, serif font. Other words are scattered around it in various sizes and colors, including brown, dark blue, and black. The words include: "event-driven", "node.js", "epoll", "libevent", "kernel", "thread", "posix", "callback", "select", "libev", "socket", "kqueue", "poll", "nginx", "lighttpd", and "aio". The words are arranged in a way that they appear to be floating or scattered around the central word.

asynchronous

event-driven

node.js

epoll

libevent

kernel

thread

posix

callback

select

libev

socket

kqueue

poll

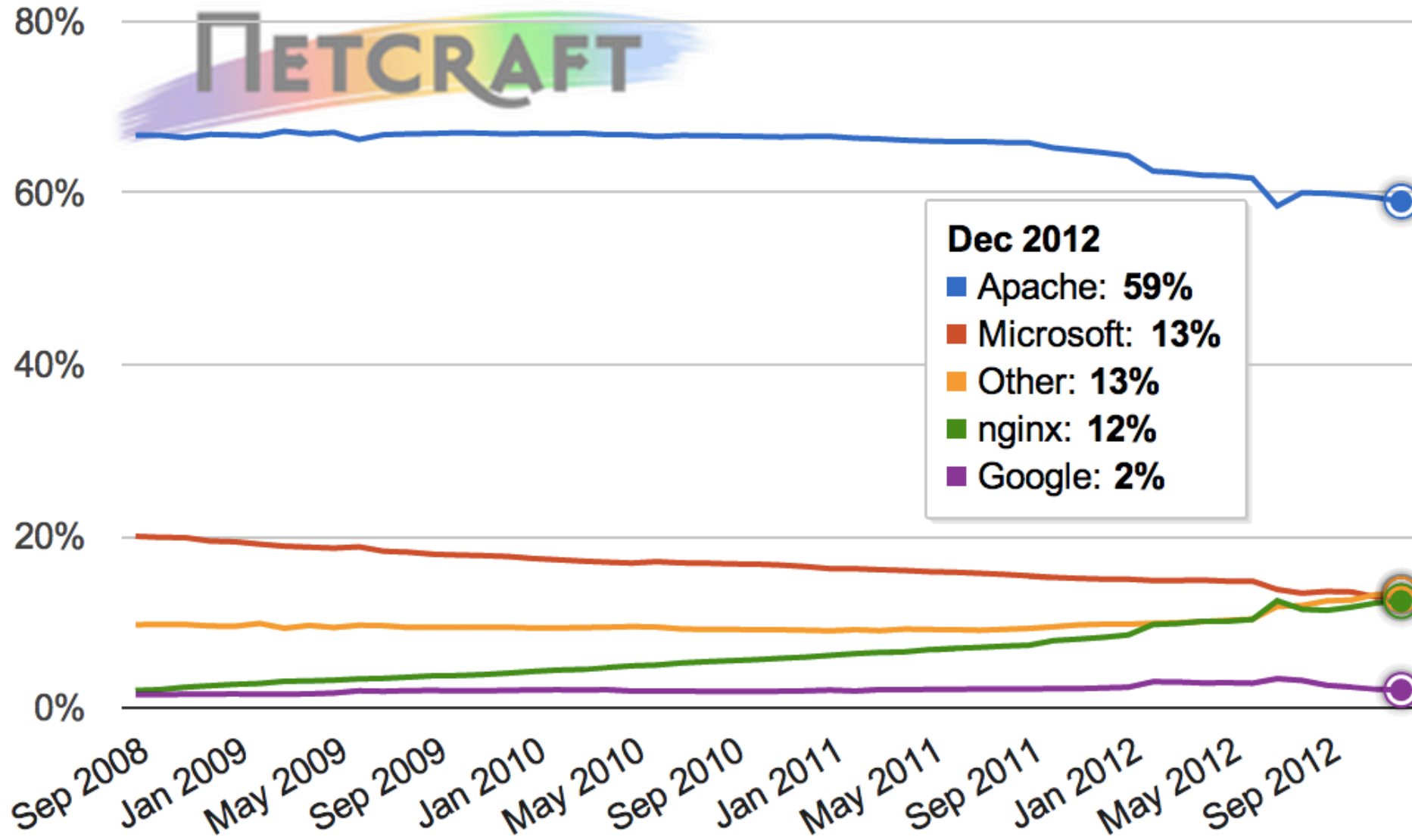
nginx

lighttpd

aio



# Market Share for Top Servers Across the Million Busiest Sites



**Dec 2012**  
■ Apache: **59%**  
■ Microsoft: **13%**  
■ Other: **13%**  
■ nginx: **12%**  
■ Google: **2%**

# C10M: the future



# 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

# Who needs Internet scale?

DNS root server

Carrier NAT

TOR node

VoIP PBX

Nmap of Internet

DPI, MitM

Video streaming

Load balancer

Banking

Web cache

Email receive

IPS/IDS

Spam send

Firewall

# Who does Internet scale today?

- “Devices” or “appliances” rather than “Servers”
  - Primarily closed-source products tied to hardware
  - But modern devices are just software running on RISC or x86 processors
    - “Network processors” are just MIPS/PowerPC/SPARC with lots of cores and fixed-function units
      - Running Linux/Unix

# X86 prices on Newegg Feb 2013

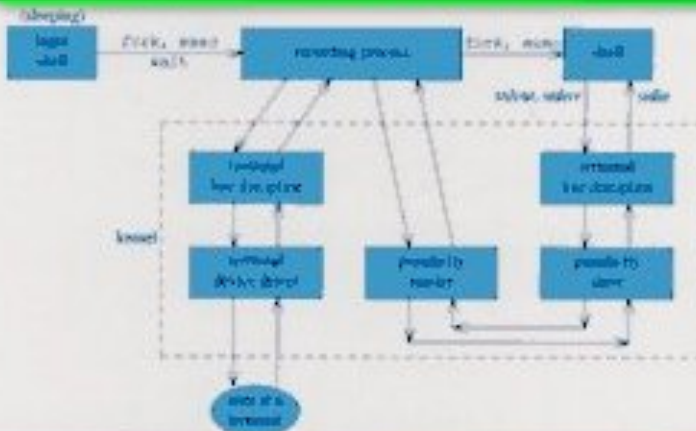
- \$1000 – 10gbps, 8-cores, 32gigs RAM
- \$2500 – 20gbps, 16-cores, 128gigs RAM
- \$5000 – 40gbps, 32-cores, 256gigs RAM

---

# UNIX<sup>®</sup>

## NETWORK PROGRAMMING

---



---

W. RICHARD STEVENS

---

PRENTICE HALL SOFTWARE SERIES

# How to represent an IP address?

```
char *ip1 = "10.1.2.3";
```

```
unsigned char ip2[] = {0xa,0x1,0x2,0x3};
```

```
int ip3 = 0x0a010203;
```

```
int ip4 = *(int*)ip2;
```

```
ip3 = ntohs(ip4);
```



```
rob$ cat test.c
#include <stdio.h>

int main()
{
    char *ip1 = "10.1.2.3.4";
    unsigned char ip2[] = {0xA, 0x1, 0x2, 0x3};
    int ip3 = 0x0A010203;

    int ip4 = *(int*)ip2;

    printf("ip3 = %x\n", ip3);
    printf("ip4 = %x\n", ip4);
    return 0;
}

rob$ gcc test.c
rob$ ./a.out
ip3 = a010203
ip4 = 302010a
rob$
```

The kernel isn't the solution  
The kernel is the problem

asynchronous

the starting point

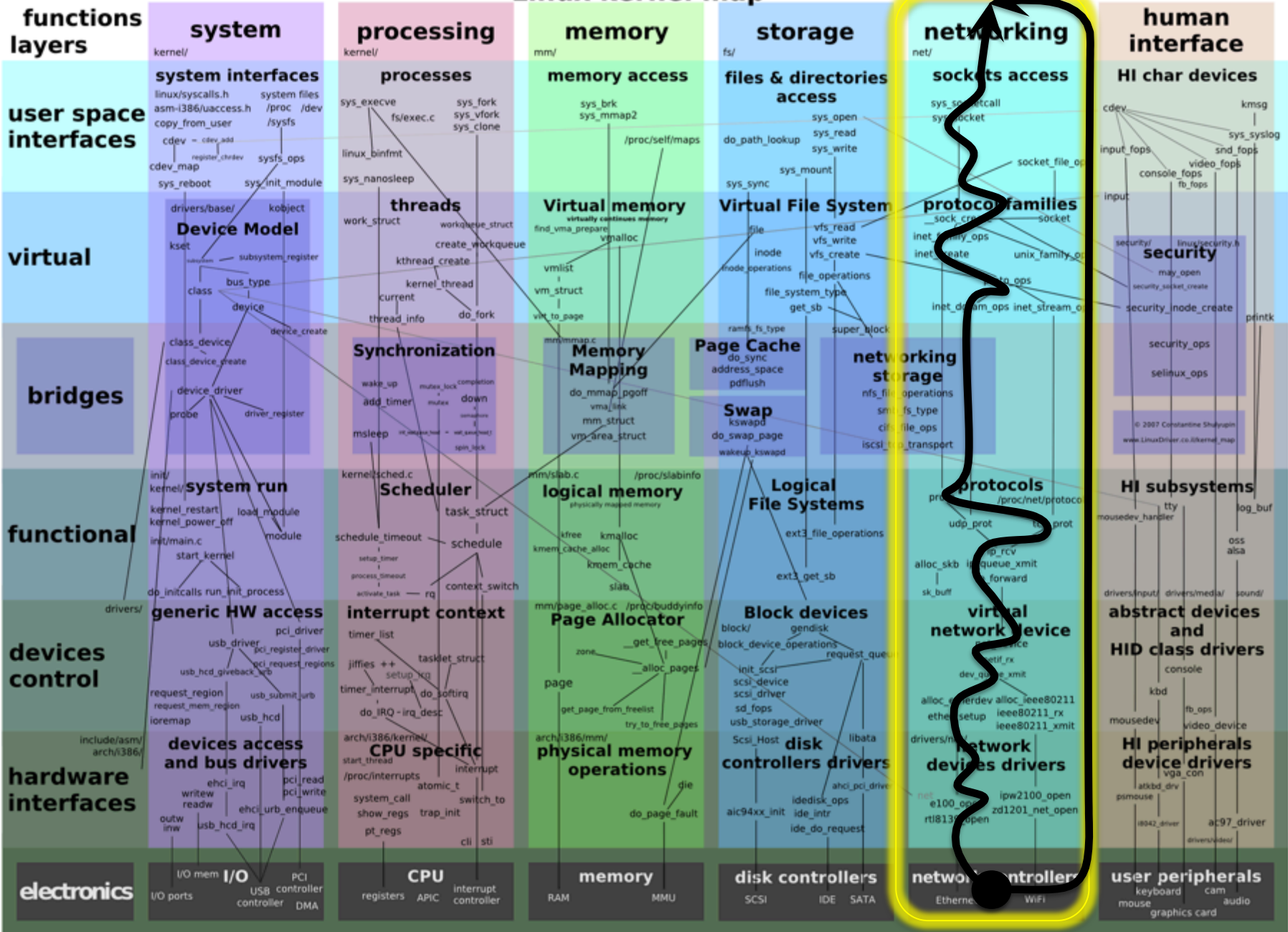
# Asynchronous at low level

- `read()` blocks
  - Thread scheduler determines which `read()` to call next
    - Depending on which data has arrived
  - Then `read()` continues
- `select()` blocks
  - Tells you which sockets have data waiting
  - You decide which `read()` to call next
  - Because data is available, `read()` doesn't block

- Apache, threads, blocking
  - Let Unix do all the heavy lifting getting packets to the right point and scheduling who runs
- Nginx, single-threaded, non-blocking
  - Let Unix handle the network stack
  - ...but you handle everything from that point on

[#1] packet scaling

# Linux kernel map

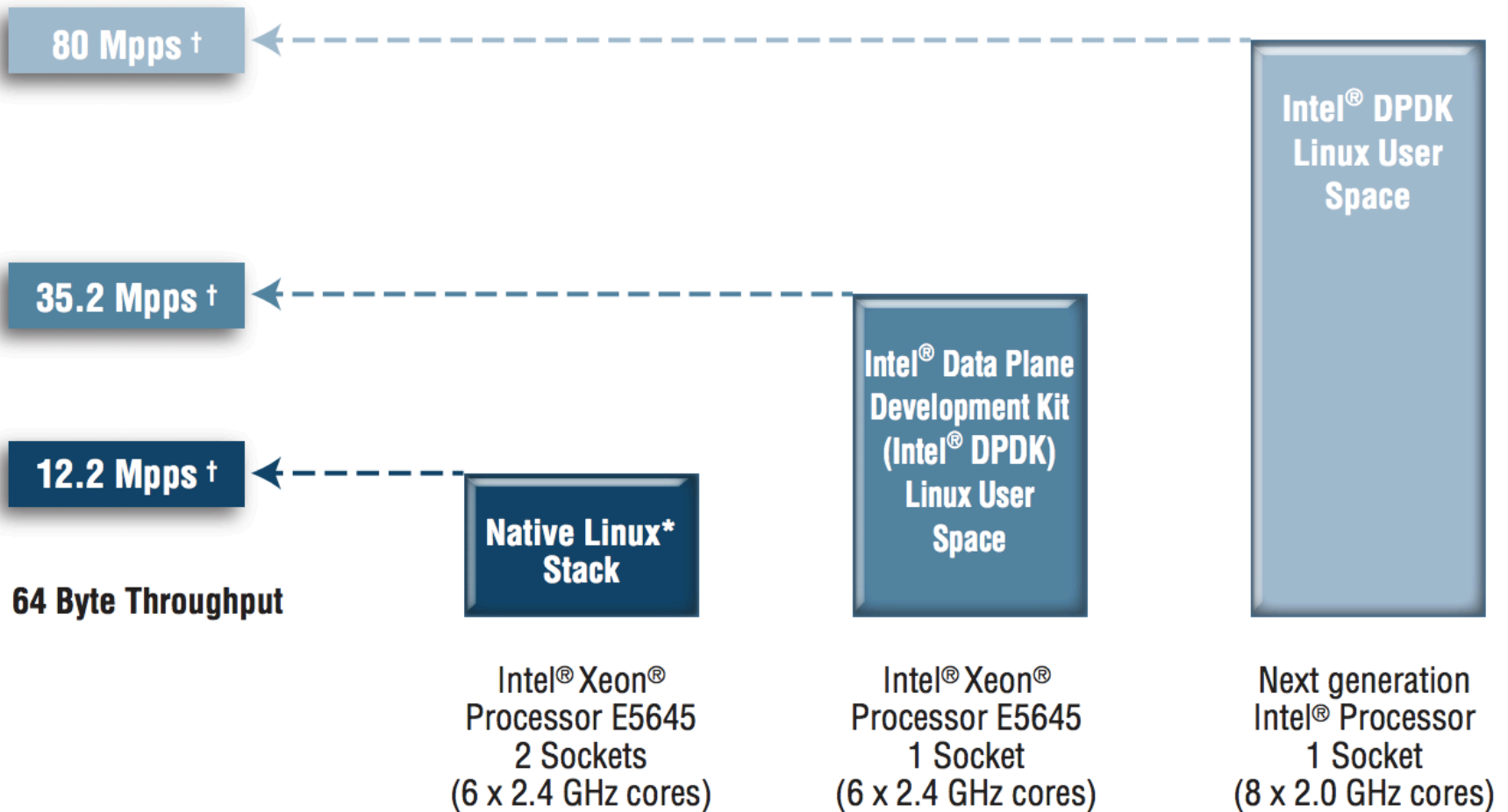


# Where can I get some?

- PF\_RING
  - Linux
  - open-source
- Netmap
  - FreeBSD
  - open-source
- Intel DPDK
  - Linux
  - License fees
  - Third party support
    - 6WindGate



# 200 CPU clocks per packet



# User-mode network stacks

- PF\_RING/DPDK get you raw packets without a stack
  - Great for apps like IDS or root DNS servers
  - Sucks for web servers
- There are many user-mode TCP/IP stacks available
  - 6windgate is the best known commercial stack, working well with DPDK

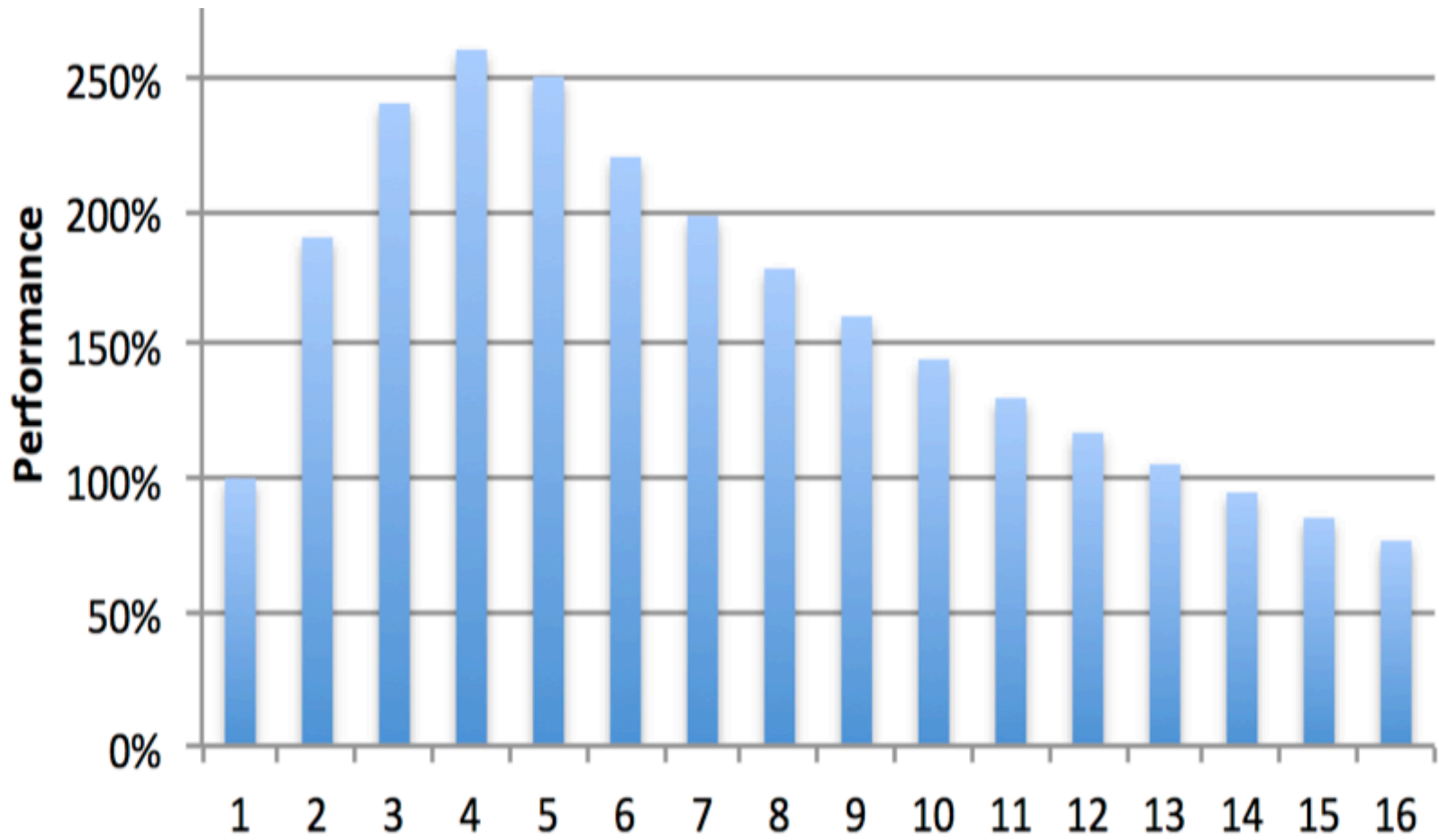
# Control plane vs. Data plane



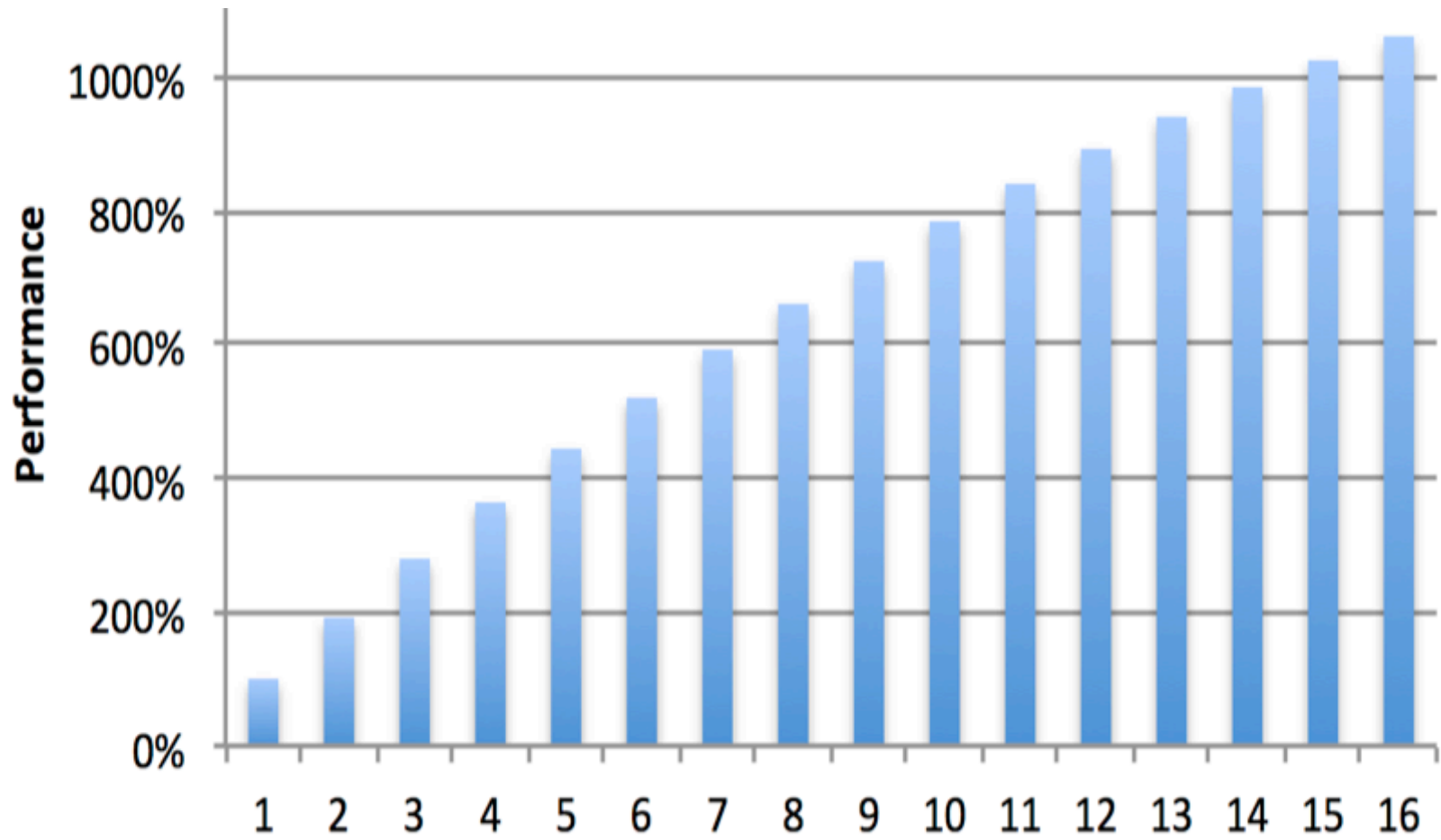
# [#2] multi-core scaling

multi-core is not the same thing as multi-threading

# Most code doesn't scale past 4 cores



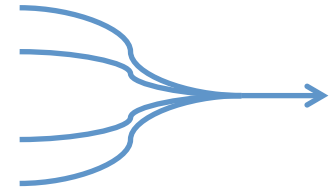
# At Internet scale, code needs to use all cores



# Multi-threading is not the same as multi-core

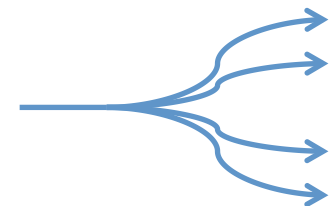
- Multi-threading

- 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)

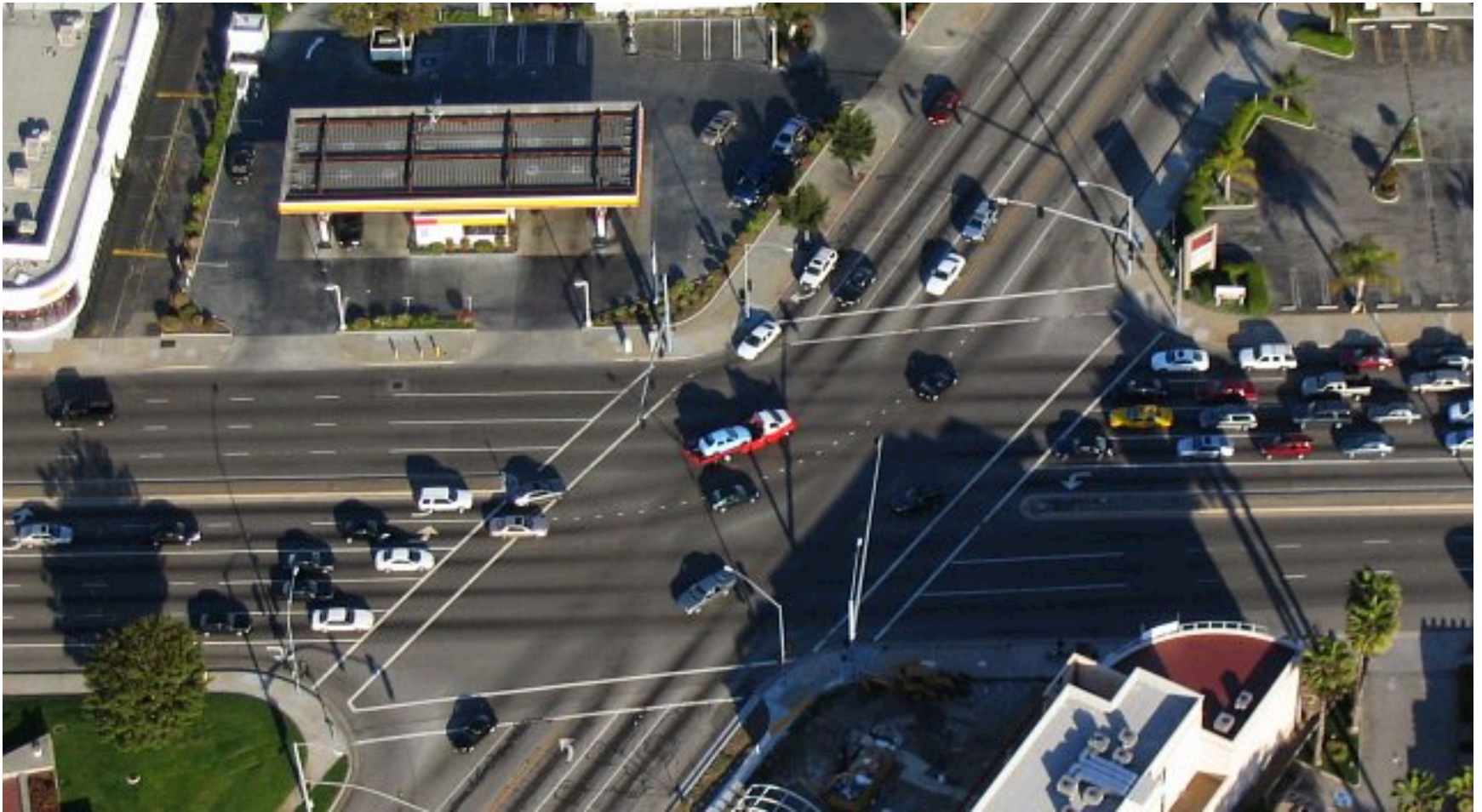


- Multi-core

- 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



spin-locks, mutexes, critical sections,  
semaphores





no waiting

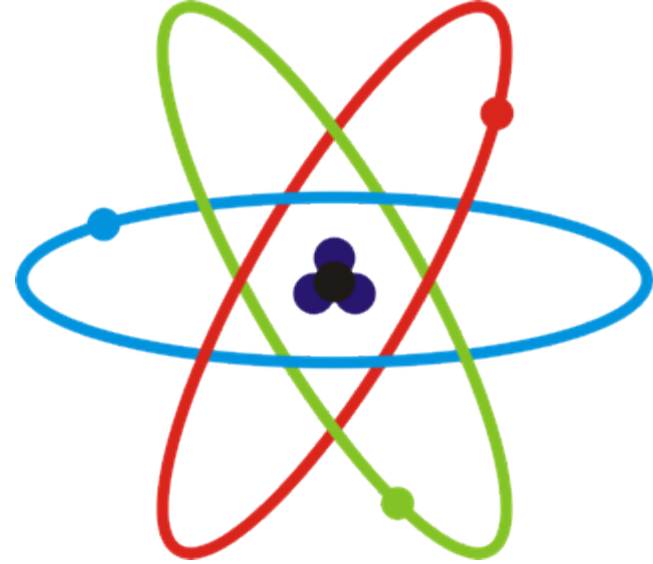


# un-synchronization

```
john@bt: ~  
File Edit View Terminal Help  
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:~$
```

- core local data
- ring buffers
- read-copy-update (RCU)

# atomics



`cmpxchg`

`lock add`

`__sync_fetch_and_add()`

`__sync_bool_compare_and_swap()`

Costs one L3 cache transaction (or 30 – 60 clock cycles, more for NUMA)

# “lock-free” data structures



- Atomics modify one value at a time
- Special algorithms are needed to modify more than one one value together
- These are known by many names, but “lock-free” is the best known
- Data structures: lists, queues, hash tables, etc.
- Memory allocators aka. malloc()

# Be afraid

- The ABA problem
  - You expect the value in memory to be A
  - ...but in the meantime it's changed from A to B and back to A again
- The memory model problem
  - X86 and ARM have different memory models
  - Multi-core code written for one can mysteriously fail on the other

# Threading models



Pipelined – each thread does a little work, then hands off to another

Worker – each thread does the same sort of work, from begin to end



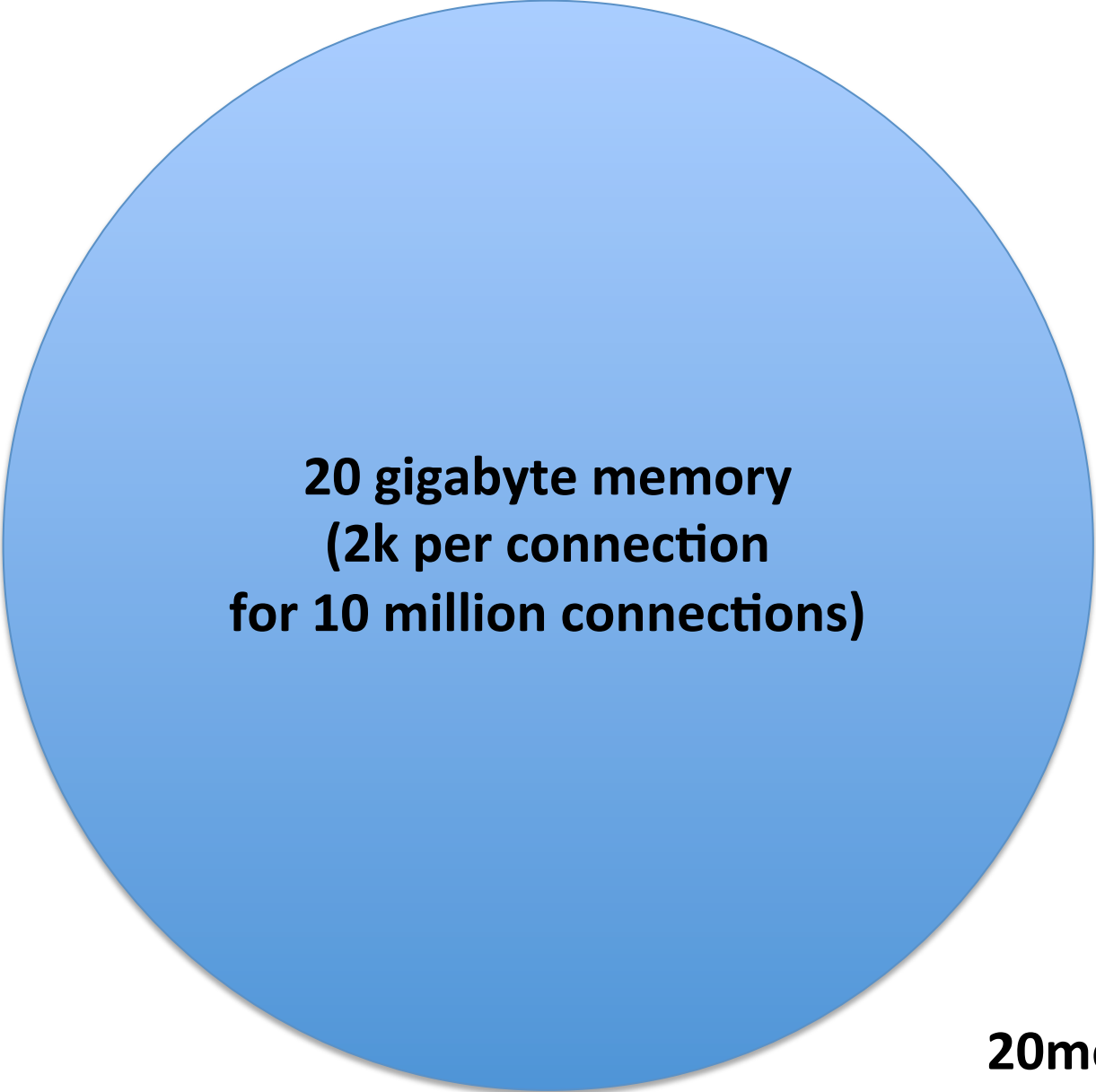
# Howto: core per thread

- `maxcpus=2`
  - Boot param to make Linux use only the first two cores
- `pthread_setaffinity_np()`
  - Configure the current thread to run on the third core (or higher)
- `/proc/irq/smp_affinity`
  - Configure which CPU core handles which interrupts

# [#3] CPU and memory

at scale, every pointer is a cache miss





**20 gigabyte memory  
(2k per connection  
for 10 million connections)**



**20meg L3 cache**

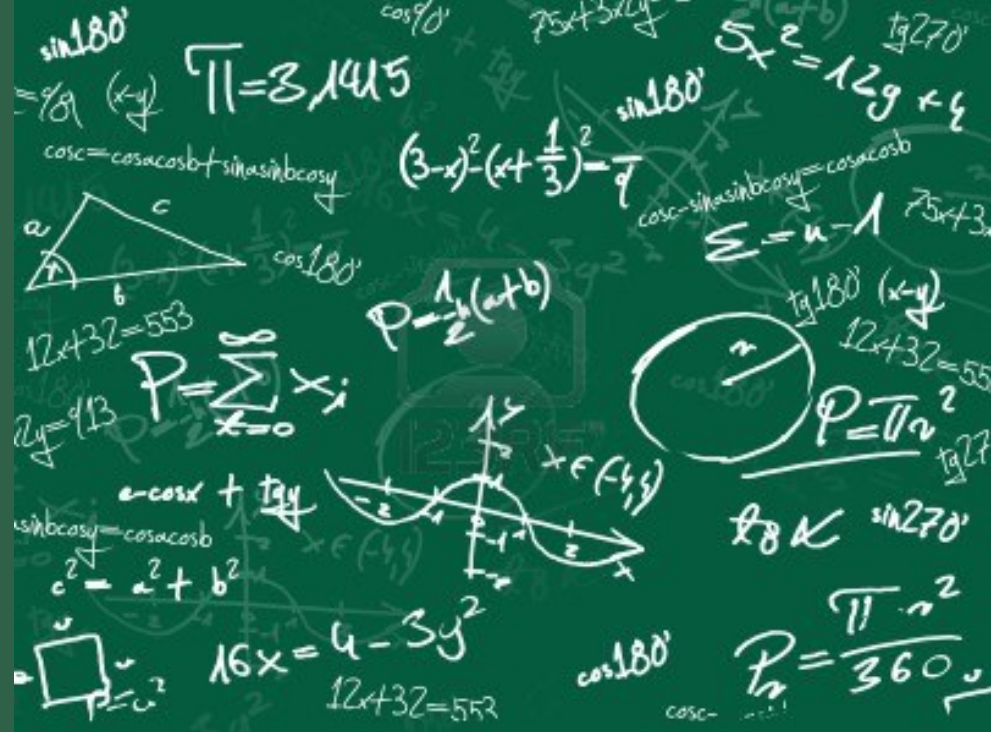
CPU

L1 cache — 4 cycles  
L2 cache — 12 cycles

L3 cache — 30 cycles

main memory — 300 cycles

budget



200 clocks/pkt overhead

1400 clocks/pkt remaining

300 clocks cache miss

-----

4 cache misses per packet

# paging

- 32-gigs of RAM needs 64-megs of page tables
  - page tables don't fit in the cache
  - every cache miss is doubled
- solution: huge pages
  - 2-megabyte pages instead of 4k-pages
  - needs to be set with boot param to avoid memory fragmentation

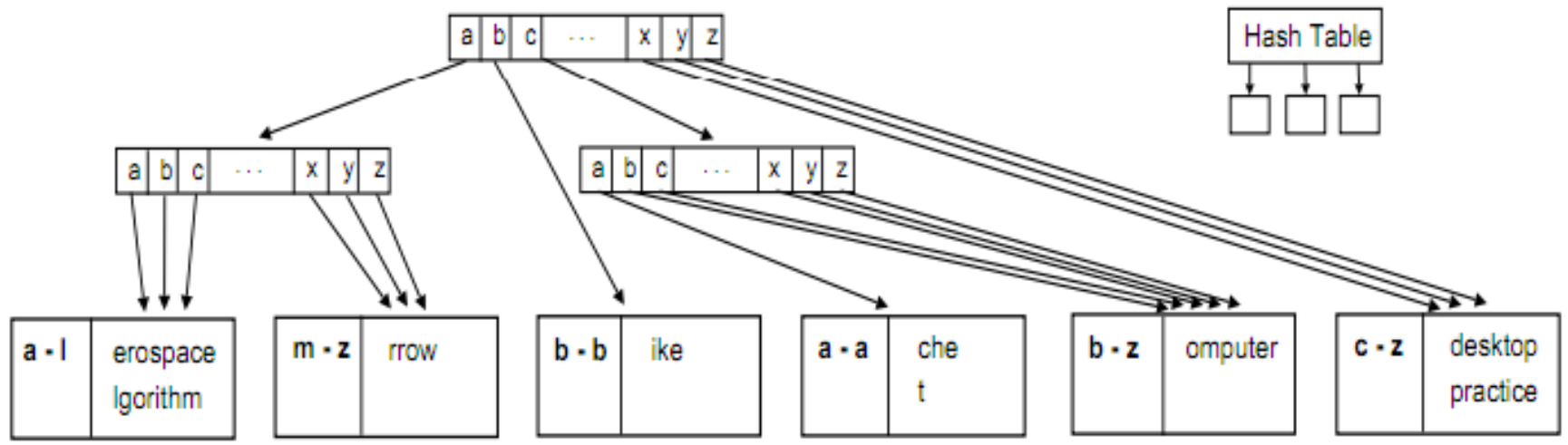
# co-locate data

- 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

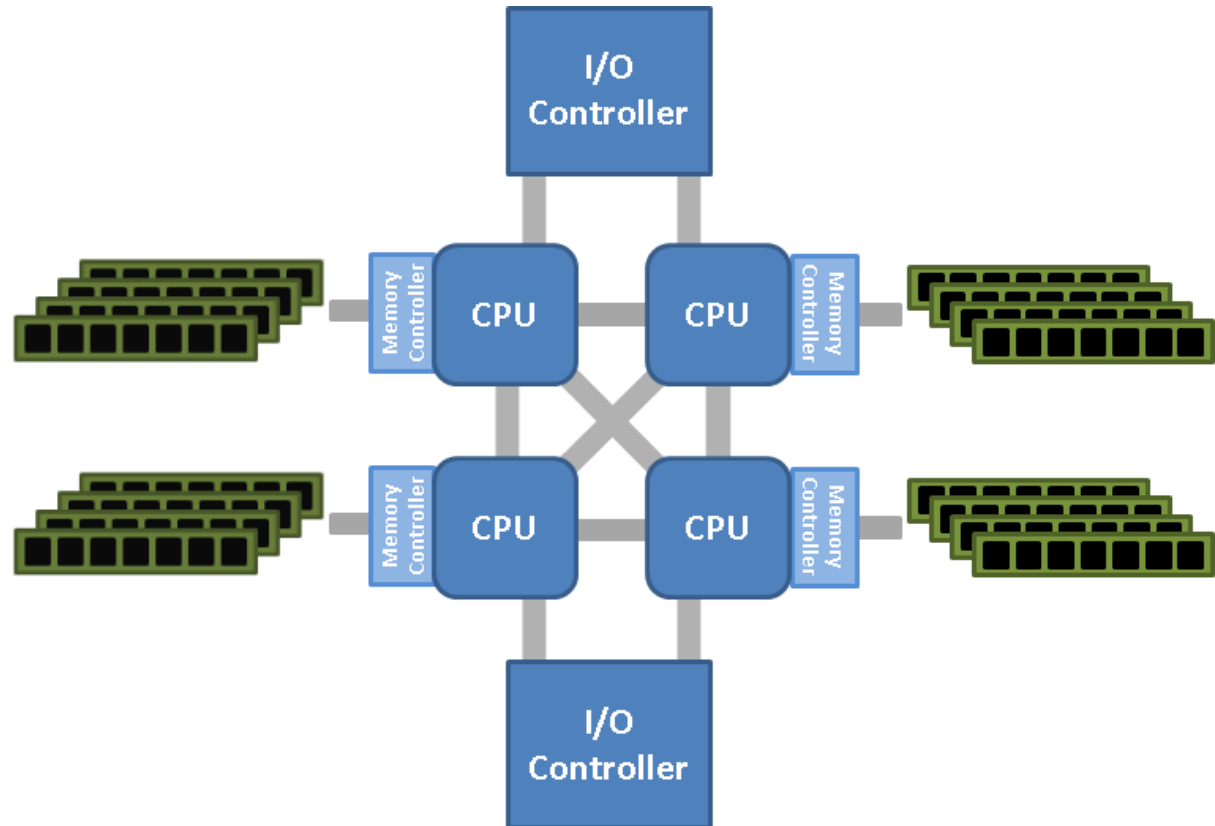
- Bit-fields instead of large integers
- Indexes (one, two byte) instead of pointers (8-bytes)
- Get rid of padding in data structures

# “cache efficient” data structures



# “NUMA”

- Doubles the main memory access time





# “memory pools”

- Per object
- Per thread
- Per socket
- Defend against resource exhaustion

# “prefetch”

- E.g. parse two packets at a time, prefetch next hash entry

# “hyper-threading”

- Masks the latency because when one thread waits, the other goes at full speed
- “Network processors” go to 4 threads, Intel has only 2

# Linux bootparam

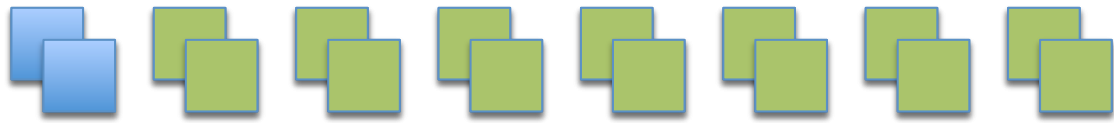
`hugepages=10000`

# Data Plane

RAM



CPU



NIC



# Control Plane

# Takeaways

- Scalability and performance are orthogonal
- C10M devices exist today
  - ...and it's just code that gets them there
- You can't let the kernel do you heavy lifting
  - Byte parsing, code scheduling
  - Packets, cores, memory

<http://c10m.robertgraham.com>