The cost of things at scale

Robert Graham
@ErrataRob
https://blog.erratasec.com
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”
20 meg L3 cache

20 gigabyte memory
(2k per connection for 10 million connections)
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
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;
    }
}
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 & 0x1FFFFFF;
    } else {
        L1 = L2[(addr>>21) & 0x1FF];
        page = L1[(addr>>12) & 0x1FF];
        return page | addr & 0xFFFF;
    }
}
20 gigabyte memory (2k per connection for 10 million connections)

- 10k hugepage tables
- 20meg L3 cache
- 40meg small page tables
User-mode latency

Concurrent memory latency

- Blue line: Kernel
- Red line: User

Y-axis: Nanoseconds
X-axis: Concurrent memory 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.

```c
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

Incrementing a shared memory
Latency per addition per thread

Latency per addition operation per core

nanoseconds

0 10 20 30 40 50 60 70 80 90 100 110 120

1 2 3 4 5 6 7 8 9 10 11 12
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
Function call overhead

- ~1.8ns
- Note the jump for "hyperthreading"
  - My machine has 6 hyperthreaded cores
- 6 clock cycles
DMA isn’t
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

masscan

- Quad-core Sandy Bridge 3.0 GHz
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

<table>
<thead>
<tr>
<th></th>
<th>ARM</th>
<th>x86</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hz</td>
<td>0.900</td>
<td>3.2</td>
<td>3.6</td>
</tr>
<tr>
<td>syscall</td>
<td>0.99</td>
<td>2.5</td>
<td>2.6</td>
</tr>
<tr>
<td>funcall</td>
<td>59.90</td>
<td>556.4</td>
<td>9.3</td>
</tr>
<tr>
<td>pipe</td>
<td>0.17</td>
<td>2.5</td>
<td>14.8</td>
</tr>
<tr>
<td>ring</td>
<td>3.90</td>
<td>74.0</td>
<td>19.0</td>
</tr>
</tbody>
</table>
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