Premature Abstraction Is The Root of All Evil!!!
Premature Abstraction

Abstract away the finite speed of light???
Premature Abstraction

Abstract away the atomic nature of matter???
Who Cares About Laws of Physics?

Electrons move at 0.03C to 0.3C in transistors and, so lots of waiting. 3D???
CPUs at Their Best
Mispredicted Branch
Memory Barrier (AKA Fence)
Memory Ordering Not Created Equal

Exact relation depends strongly on hardware architecture and on compiler
Please stay on the line. Your call is very important to us...
## Costs of Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Cost (ns)</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock period</td>
<td>0.4</td>
<td>1</td>
</tr>
<tr>
<td>“Best-case” CAS</td>
<td>12.2</td>
<td>33.8</td>
</tr>
<tr>
<td>Best-case lock</td>
<td>25.6</td>
<td>71.2</td>
</tr>
<tr>
<td>Single cache miss</td>
<td>12.9</td>
<td>35.8</td>
</tr>
<tr>
<td>CAS cache miss</td>
<td>7.0</td>
<td>19.4</td>
</tr>
<tr>
<td>Single cache miss (off-core)</td>
<td>31.2</td>
<td>86.6</td>
</tr>
<tr>
<td>CAS cache miss (off-core)</td>
<td>31.2</td>
<td>86.5</td>
</tr>
<tr>
<td>Single cache miss (off-socket)</td>
<td>92.4</td>
<td>256.7</td>
</tr>
<tr>
<td>CAS cache miss (off-socket)</td>
<td>95.9</td>
<td>266.4</td>
</tr>
</tbody>
</table>
Loophole in Laws of Physics

Read-only data remains replicated in all caches
In Real Estate:
Location, location, location!!!
In Parallel Programming:
Locality, locality, locality!!!
But Can’t The Hardware Help???
Hardware Locality Optimizations

Big caches

Store buffer

Speculative execution

Big cachelines

Cache prefetching
Big Cachelines: The Good Time

CPU 0

r1 = a;

Long latency

CPU 1

r2 = b;
r3 = c;
r4 = d;

Almost no additional latency

Time
Big Cachelines: The Bad and Ugly

```
a = 42;  Long latency  a = 256;  Long latency
CPU 0

a b c d

CPU 1

a b c d

d = 1729;  Long latency

Time

False sharing!!!```
Big Cachelines: Alignment Directives

CPU 0

a = 42;
Long latency
Immediate effect!!!
a = 256;

CPU 1

da = 1729;
Immediate effect!!!

Time
Prefetching Good!!!

CPU 0

r1 = a;

Long latency

r2 = b;
r3 = c;
r4 = d;
r5 = e;
r6 = f;

Almost no additional latency

CPU 1

a b c d

abcd efgh

a b c d

abcd efgh

Time
Prefetching: Bad and Ugly!

CPU 0

a = 42;  \text{Long latency}  \quad a = 256;

CPU 1

\begin{align*}
a b & \quad c d \\
d = 1729; & \quad \text{Long latency}
\end{align*}

Time
Store Buffers: The Good

CPU 0

a = 42;

r2 = e;
r3 = f;
r4 = g;

Hidden latency

Almost no additional latency

Store to “a” can complete

e f g h

CPU 1

a b c d

Time

a b c d
Store Buffers: The Bad and Ugly

CPU 0:
- `a = 42;`
- `r2 = e;`
- `r3 = f;`
- `r4 = g;`
- `r5 = a; (42)`

CPU 1:
- `a b c d`

Hidden latency:
- Store of 42 to "a" can complete
- "Store to ‘a’ happened before load from ‘e’.”

Almost no additional latency:
- "No, store to ‘a’ happened after load from ‘e’!!!("
Store Buffers: The Bad and Ugly

Hidden latency

Almost no additional latency

Store of 42 to “a” can complete

“Store to ‘a’ happened before load from ‘e’.”

“No, store to ‘a’ happened after load from ‘e’!!!”

So we use ordering directives where needed!!!
Portable Ordering: Memory Model

C++11 Memory Model: Ordering Portability

CPU Family 1
Toolchain

CPU Family 1
Hardware

CPU Family 2
Toolchain

CPU Family 2
Hardware

CPU Family 3
Toolchain

CPU Family 3
Hardware
Big Caches & Speculative Execution

**Big caches:**
- Higher throughput when data fits in cache
- Higher latency for cache misses
- Energy inefficiency

**Speculative execution:**
- Hide latencies and memory misordering
- Higher worst-case latency, energy consumption
data = 42;
flag.store(1, mo_release)
while (!flag.load(mo_acquire))
    continue;
    r1 = data;
Maligned Workhorse: Locking

Stupidly simple lock acquisition:

```java
while (lk.exchange(memory_order_acquire))
    continue;
```

Lock release:

```java
lk.store(0, memory_order_release);
```

Works, but horrible high-contention behavior
Maligned Workhorse: Locking

Somewhat less stupidly simple lock acquisition:

```c
for (;;) {
    while (lk.exchange(1, memory_order_acquire))
        continue;
    if (!lk.load(memory_order_acquire))
        break;
}
```

Lock release:

```
lk.store(0, memory_order_release);
```

Works, but sub-optimal high-contention behavior
Suboptimal Lock Handoff

Lots of useless bus traffic after unlock time!!!
All CPUs spinning on same location...
Queued Locks

At most two CPUs content for given location

But Even Better...

Maintain low levels of contention!!!

Also, adaptive spin/sleep strategy (for example, algorithms using futex system calls!)
Reader-Writer Locking

Read-side parallelism! What not to like?

Textbook reader-writer lock:

Single atomically manipulated machine word

Flag bits (fairness, etc.)

Number of Readers

Number of Writers
Reader-Writer Locking

Textbook read-only acquisition and release:

Orders of magnitude slowdown from cache misses!!!
Might as well have an exclusive lock unless huge critical sections...
Software Must Help Hardware
One exclusive lock per thread: TLS
Reading threads acquire and release own lock
Writing threads acquire and release all locks

Very fast readers, but very slow writers
TLS Reader-Writer Lock

TLS read-only acquisition and release:

CPU 0

CPU 1

CPU 2

CPU 3

Very fast readers!!!
Too bad about those poor writers... Generality!!!
Reader-Writer Locking

And now you understand one motivation for Paul’s RCU and Maged’s Hazard Pointers...

But that is another presentation.

Now for Maged's single-producer single-consumer buffer!!!