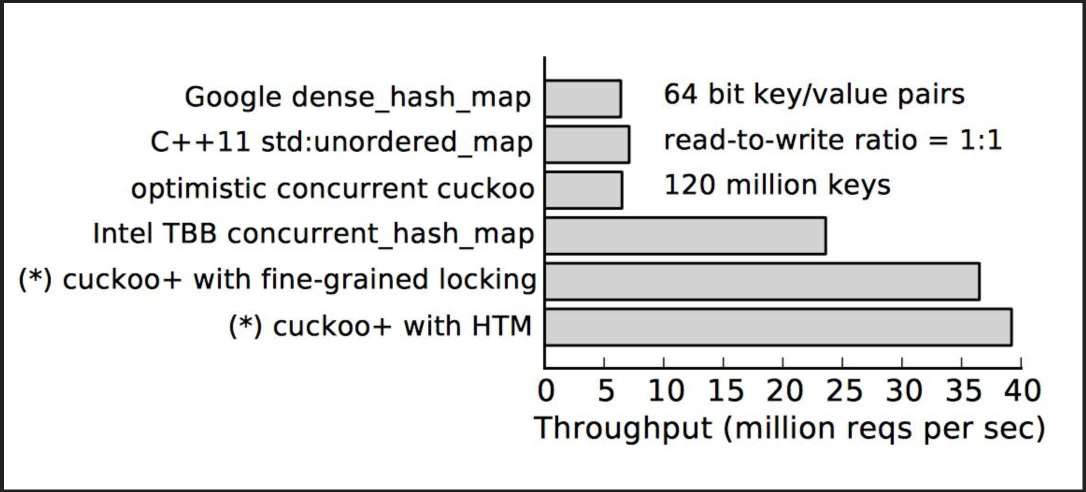
Hardware Transactional Memory – Why You Should Care

Reversim 2016
19 Sep 2016

Uri Shamay
Lead Developer, Juno

Research: Real Cases Improvements

- LevelDB = +25%
- Memcached = x2
- eliminating global interpreter locks in Ruby through HTM = x4.4
- data structures, e.g ::



http://www.cs.princeton.edu/~xl/Publications_files/cuckoo-eurosys14.pdf

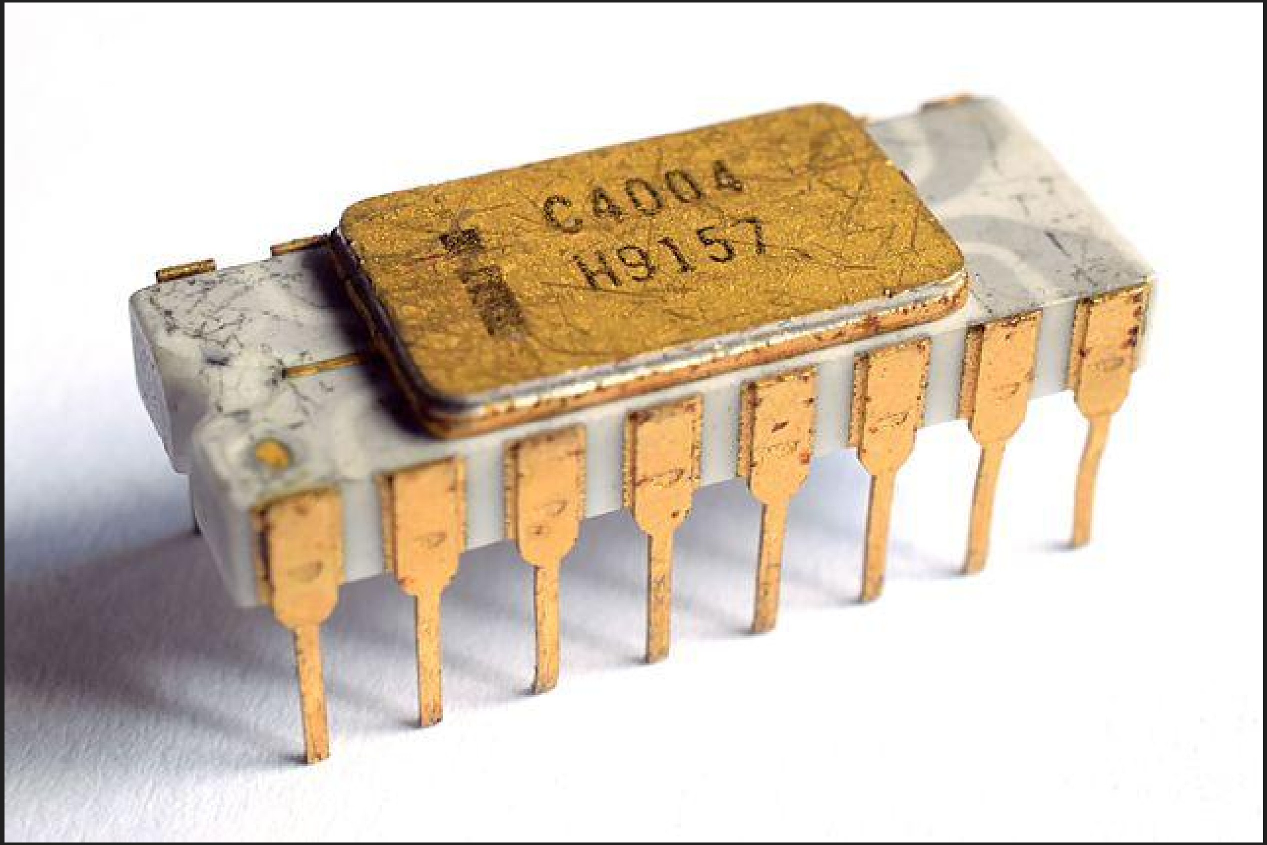
Agenda

- concurrency programming evolution
- become familiar with HTM mechanism
- why you should care when you code

$>whoami

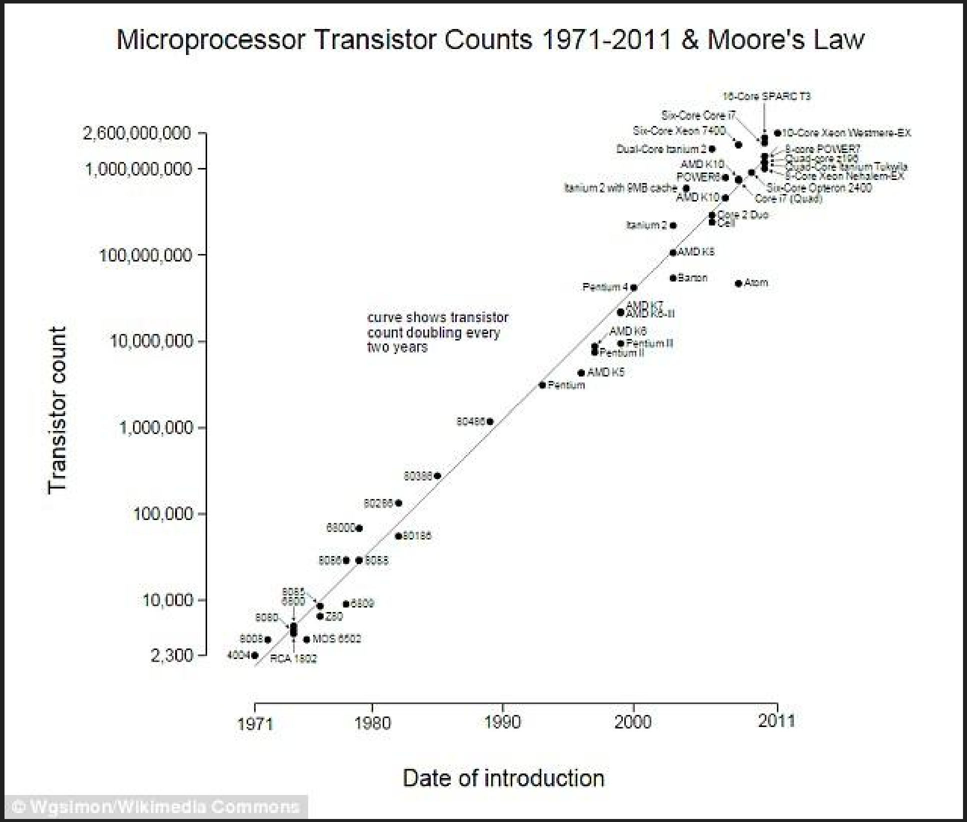
- pretty much… mechanical-sympathy junky
- hard-core infrastructure, scale & high performance solver
- cmpxchg16.me
- @cmpxchg16
- github.com/cmpxchg16

The First CPU - Intel 4004



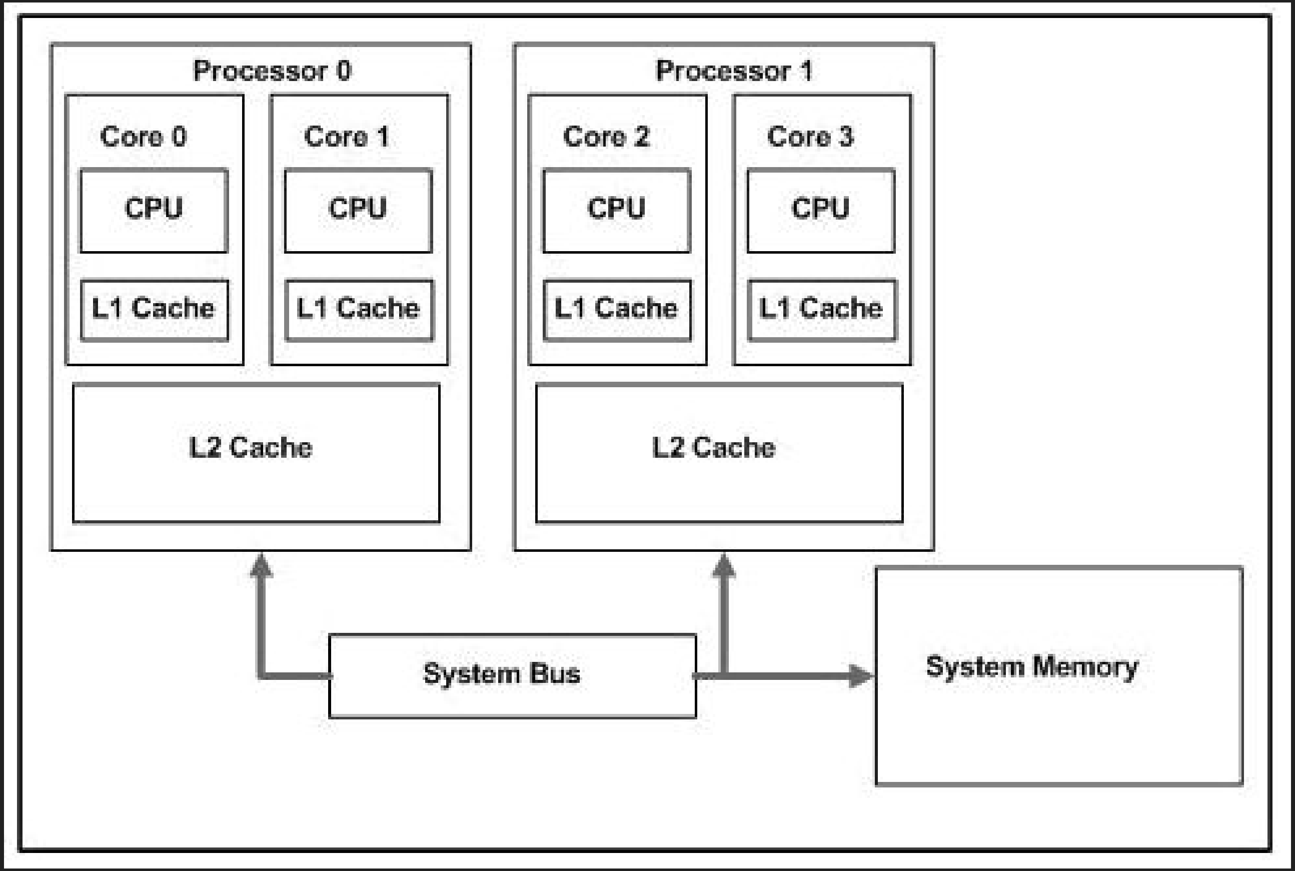
https://en.wikipedia.org/wiki/Intel_4004#/media/File:Intel_C4004.jpg

Moore's law



https://en.wikipedia.org/wiki/Moore%27s_law#/media/File:Transistor_Count_and_Moore%27s_Law_-_2011.svg

Multi-Core CPU



https://software.intel.com/en-us/articles/software-techniques-for-shared-cache-multi-core-systems

One CPU Programming - One Thread

global counter - no inconsistency

```
class Counter {
  void increment() {
    this.i++
  }
}
```

Multi-Core CPU Programming

shared counter - using lock

```
class Counter {
  void synchronized increment() {
    this.i++
  }
}
```

Multi-Core CPU Programming - hardware-based Optimization

```
class Counter {
  //shared counter - HW atomic add
  void increment() {
     // e.g Java AtomicInteger
     this.i.incrementAndGet()
  }
}
```

Multi-Core CPU Programming

2 shared counters

```
class Counter {
  void increment() {
     // e.g Java AtomicInteger
    this.counter1.incrementAndGet()
    this.counter2.incrementAndGet()
  }
}
```

but if we want both atomically.. ... .

Multi-Core CPU Programming

2 shared counters - what we want (semantically)

```
atomic {
  this.counter1++
  this.counter2++
}
```

Multi-Core CPU Programming

2 shared counters - how do we get it O_*

```
void synchronized increment() {
   this.counter1++
   this.counter2++
}
```

side note: 2CAS (cmpxchg16b) exists at HW, available just in low-level languages

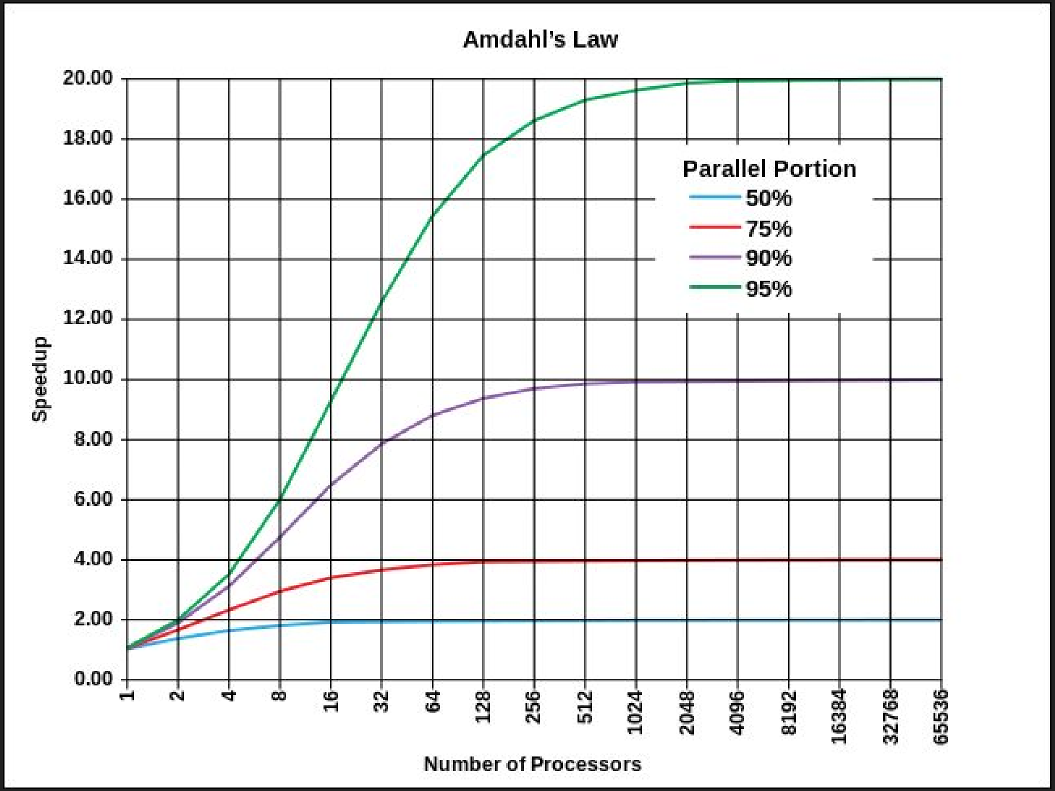
Multi-Core CPU Programming

and sometimes it becomes more complex...

```
void synchronized mutate(k) {
  this.hashmap1.remove(k)
  this.hashmap2.remove(f(k))
  this.size--
}
```

Amdahl’s Law

more processors != speedup (mostly..)



https://en.wikipedia.org/wiki/Amdahl%27s_law#/media/File:AmdahlsLaw.svg

Hardware Transactional Memory

- a new concurrency primitive to read/write memory atomically in HW (transactional)
- commodity in new Intel-based computers (Haswell/Broadwell microarchitecture)
- Java 8u40 support it -XX:+UseRTMLocking

Hardware Transactional Memory

an optimistic approach to pessimistic locking

```
public class HashMap<K,T> ...

private final array[] = ...

public synchronized T get(K key) {

    return array[key.hash % ...].get(...);

}
```

this code automatically switches to..

Hardware Transactional Memory
auto lock elision - using transaction

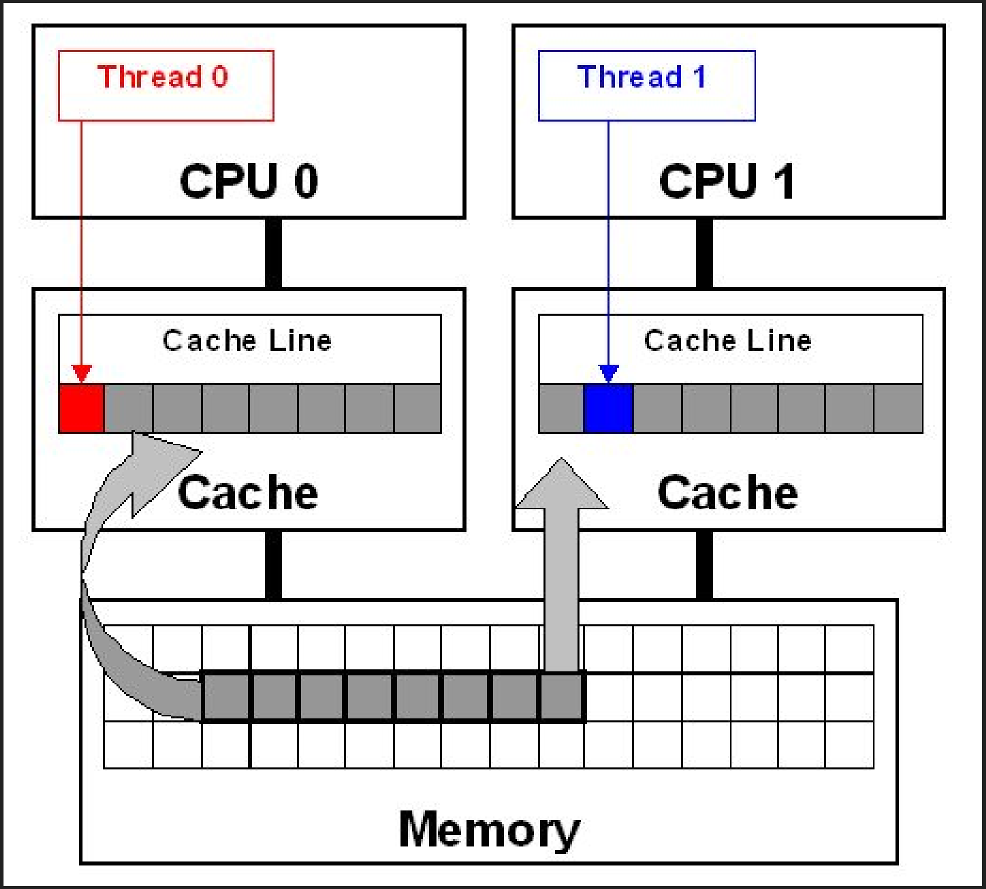
```
public ~~synchronized~~ T get(K key) {
    for (int i = 0; i < RETRIES; ++i) {
      xbegin();
      T t = array[key.hash % ...].get(...);
      if (xend() == SUCCEED) {
        return t
      }
    }

    //failed on transaction - fallback to lock
    lock()
    ...
    unlock()
```

Hardware Transactional Memory
under the hood

- system actions == failed transaction
- no conflict == commit
- conflict == abort transaction & retry
- too many retries == fallback to lock

Multi-Core CPU - Memory Access

transaction limit = number of lines in L1 cache



https://software.intel.com/en-us/articles/avoiding-and-identifying-false-sharing-among-threads

Hardware Transactional Memory
data contention

```
private int size;

public synchronized void put(K key, T val) {
    // different buckets == no data contention
    array[key.hash % ...].append(val);
    ++size; // data contention
}

public synchronized int size() {
     return this.size;
}
```

Hardware Transactional Memory
reduce data contention

```
private final sizes[] = ...;

public synchronized void put(K key, T val) {
    // different buckets == no data contention
    array[key.hash % ...].append(val);
    // different buckets == no data contention
    sizes[key.hash % ...]++;
}

public synchronized int size() {
     for (...)     {
          size += sizes[i];
}
```

Hardware Transactional Memory
under the hood

- uncontended lock - not interesting
- contended locks with no hotspot gain great improvement
- no conflicts == more parallelization
- data contention should be reduced
- smart implementation (e.g Java), knows to fallback immediate incase of “bad” code

Hardware Transactional Memory
wrap-up

- split locks increase parallelization, but with HTM can be more
- HTM != black magic
- profilers provides statistics on HTM, Java: -XX:+PrintPreciseRTMLockingStatistics
- HTM usage continues growing

Hardware Transactional Memory
getting started - check cpu support it

Option 1:

goto http://ark.intel.com && enter the cpu model

look for: **TSX-NI: Yes**

Option 2:

```
$>lscpu | grep rtm
Flags:                  ... fsgsbase bmi1 hle avx2
smep bmi2 erms rtm xsaveopt
```

Thank You!

Uri Shamay
Lead Developer, Juno

Standing on the shoulders of giants