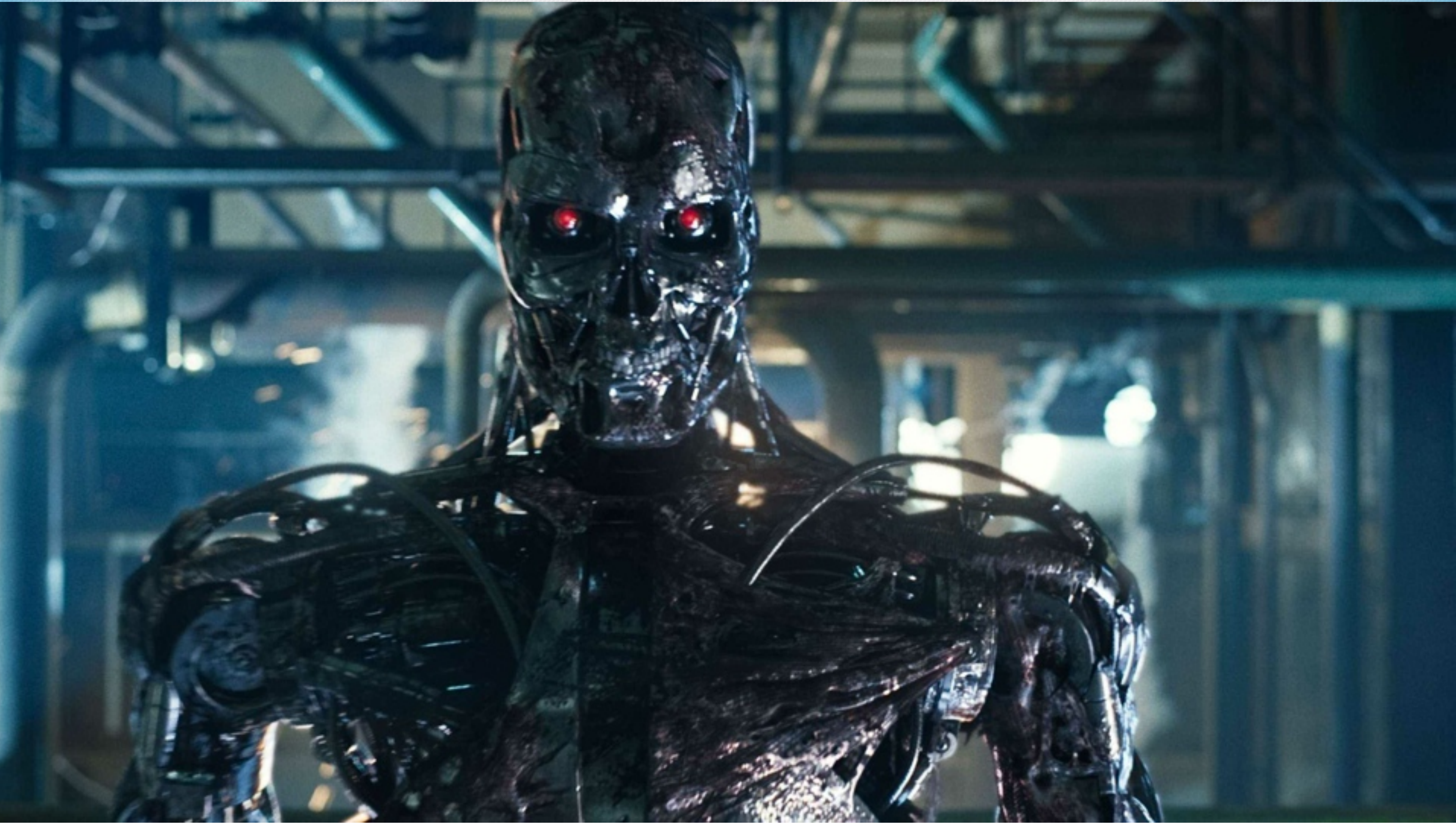
Java and the Machine

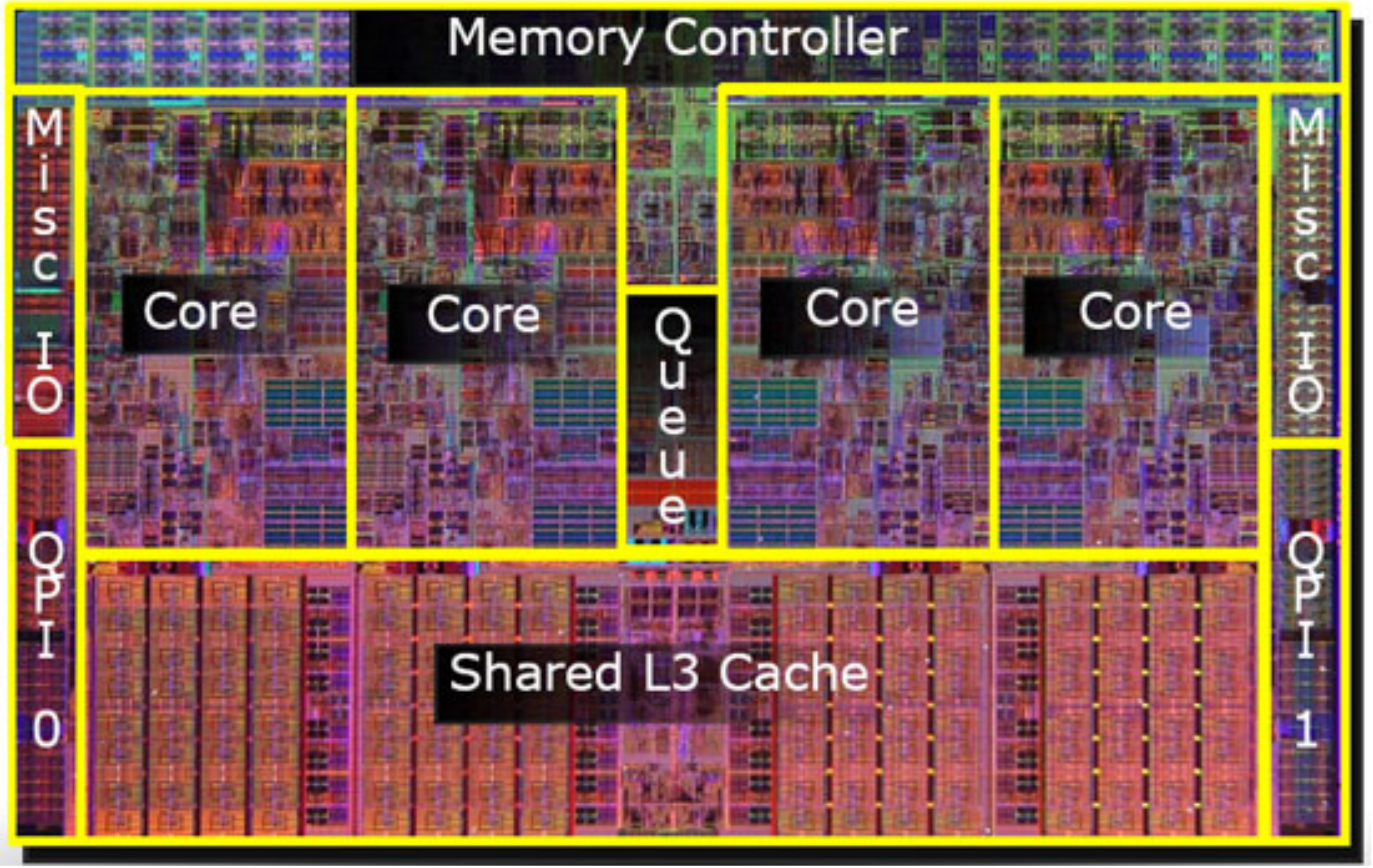
Martijn Verburg - CTO @ jClarity

(**@karianna**)

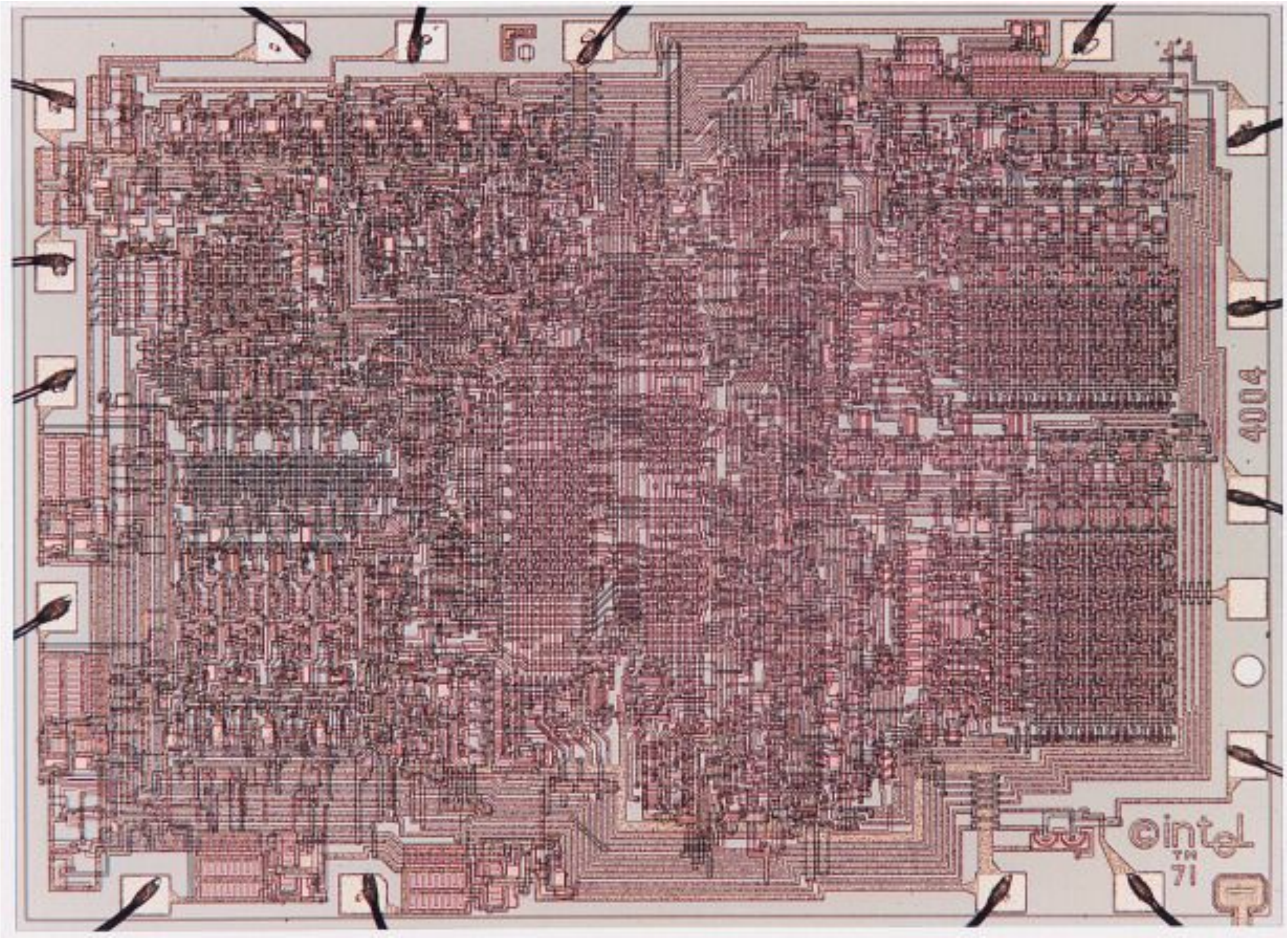
This guy is coming for you



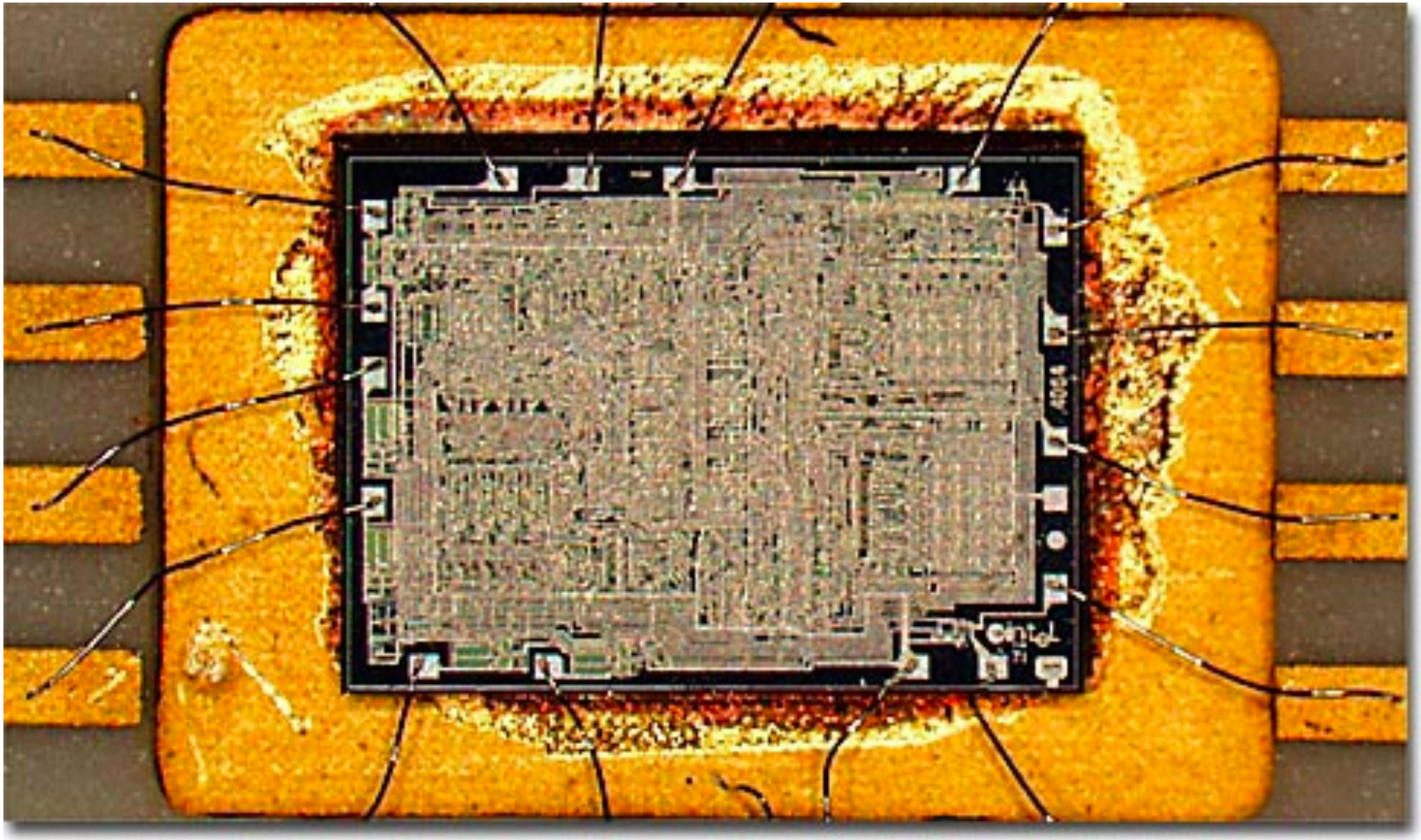
The developer version of the T-800



Do you know what this is?



A relic of a simpler era!!



Intel 4004 - 1971

108khz-740khz

10μm die (1/10th of human hair)

16 pin DIP

2,300 transistors

shared address and data bus

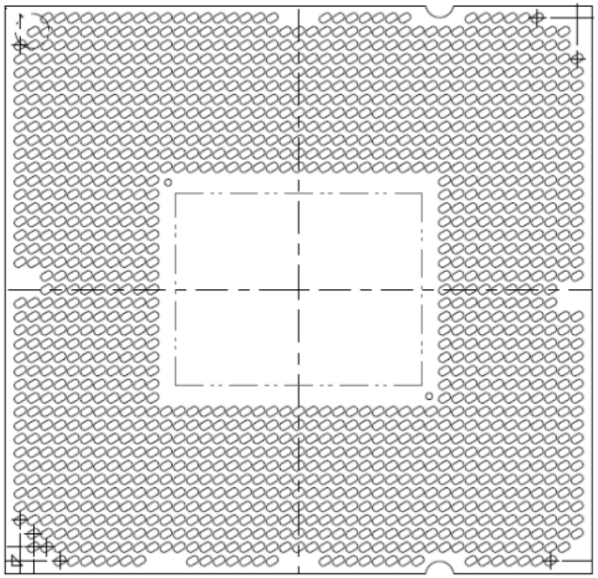
46300 or 92600 instructions / second

Instruction cycle of 10.8 or 21.6 μs (8 clock cycles / instruction cycle)

4 bit bus (12 bit addressing, 8 bit instruction, 4 bit word)

46 instructions (41-8 bit, 5-16 bit), described in a 9 page doc!!!

Intel i7 - 2008



3.3Ghz (4200x)

2nm die (5000x smaller)

64 bit instructions

1370 landings

774 million transistors (~336,000x)

50000x less expensive, 350,000x faster, 5000x

31 stage instruction pipeline (instructions retired @ 1 / clock)

- adjacent and nearly adjacent instructions run at the same time

packaging doc is 102 pages

MSR documentation is well over 7000 pages

So we have lots of challenges!

- **Back to basics**
- **Challenge:** Hardware Threads
- **Challenge:** Virtualisation
- **Challenge:** The JVM
- **Challenge:** Java
- **Conclusion**

DIABOLICAL DEVELOPER

This talk might hurt

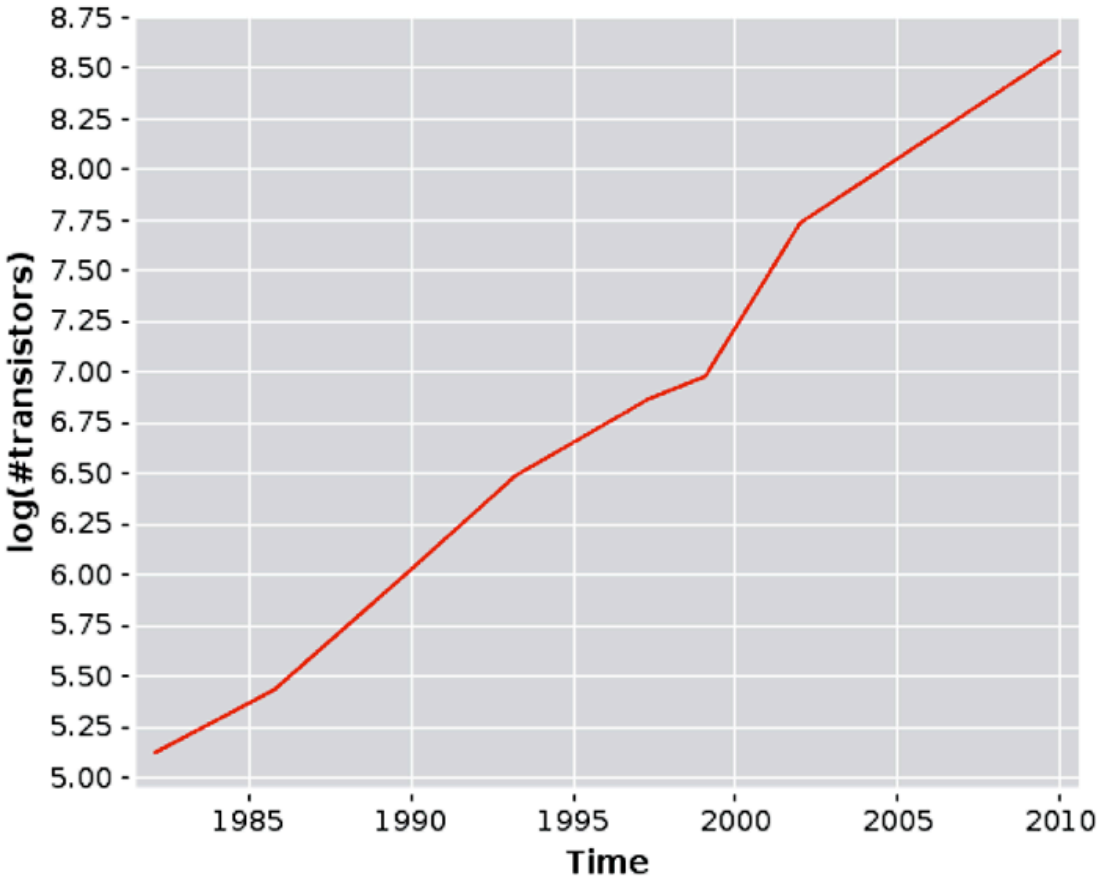
T-800 don't care

Back to Basics

- Moore's Law
- Little's Law
- Amdahl's Law
- It's the hardware stupid!
- What can I do?

Moore's Law

- It's about transistors not clocks



jClarity

Little's Law

$$\lambda = 1 / \mu$$

alternatively

average # of attendees =
arrivals / h * length of stay

=> 500 = 1000 * 0.5

Little's Law

$$\lambda = 1 / \mu$$

Throughput = 1 / Service Time

alternatively

average # of attendees = arrivals / h * length of stay => 500 = 1000 * 0.5

Amdahl's Law

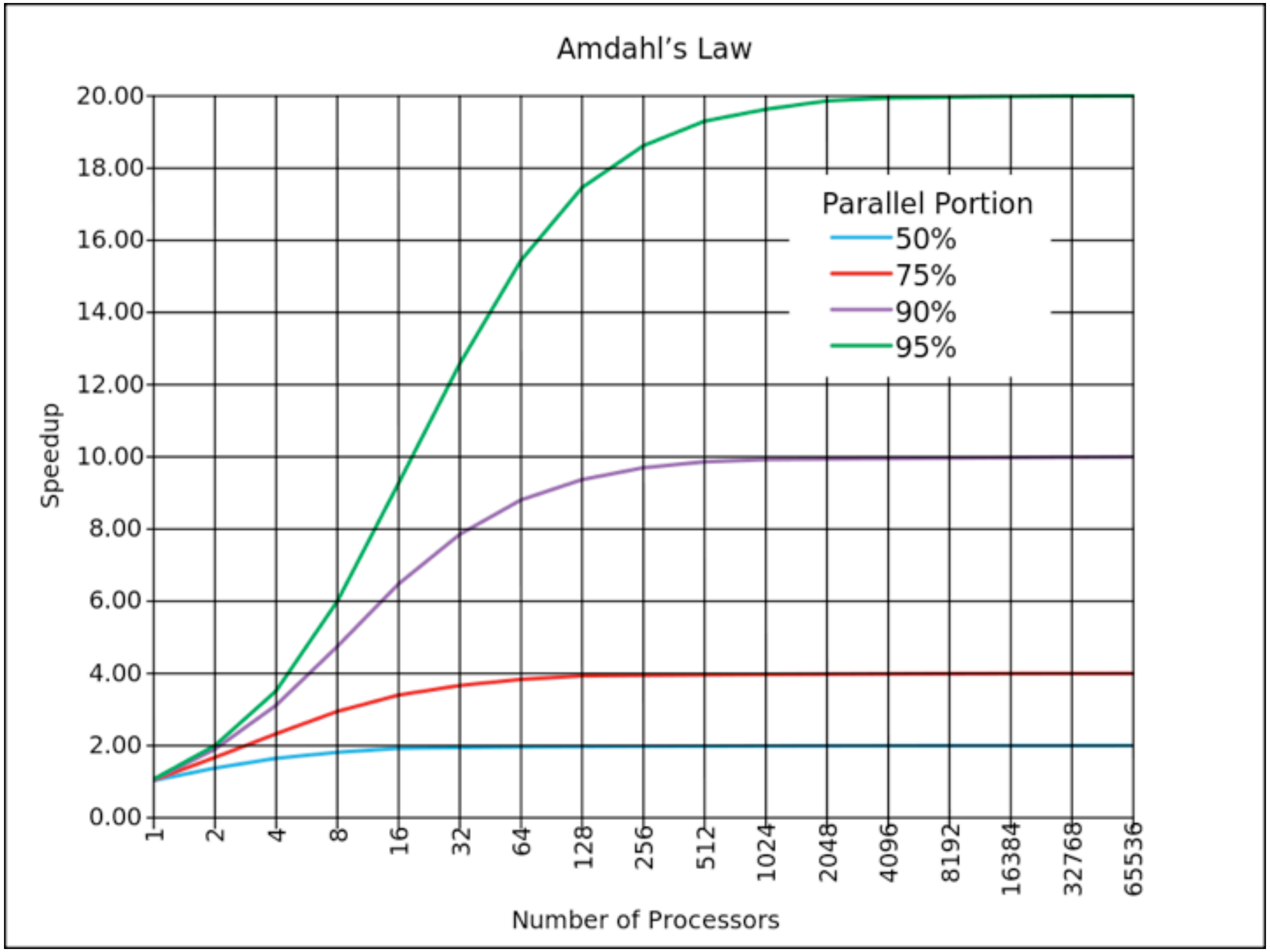
$$\frac{1}{(1-P)+\frac{P}{S}}$$

P = proportion
S = speedup

e.g.

P = 0.3 means 30% of the alg can be sped up

S = 2 twice as fast



It's the Hardware stupid!

- **Software often thinks there are no limitations**
  - If you're Turing complete you can do anything!
- **The reality is that its fighting over finite resources**
  - You remember hardware right?
- **These all have finite capacities and throughputs**
  - CPU
  - RAM
  - Disk
  - Network
  - Bus

Mechanical Sympathy

Mechanical Sympathy

-

"Hardware and Software working together"

— Martin Thompson

What can I do?

- **Don't panic**
- **Learn the laws**
  - Have them up next to your monitor
- **Code the laws**
  - See them in action!
- **Amdahl's law says "don't serialise!"**
  - With no serialisation, no need to worry abut Little's law!
- **Understand what hardware you have**
  - Understand capacities and throughputs
  - Learn how to measure, not guess those numbers

Still with me?

Good.

Hardware Threads

- Threads are non deterministic
- Single threaded programs
- Multi core programs
- What can I do?
- Protect your code

Threads are non-deterministic

"Threads seem to be a small step from sequential computation"

"In fact, they represent a huge step."

— The Problem with Threads, Edward A. Lee, UC Berkeley, 2006

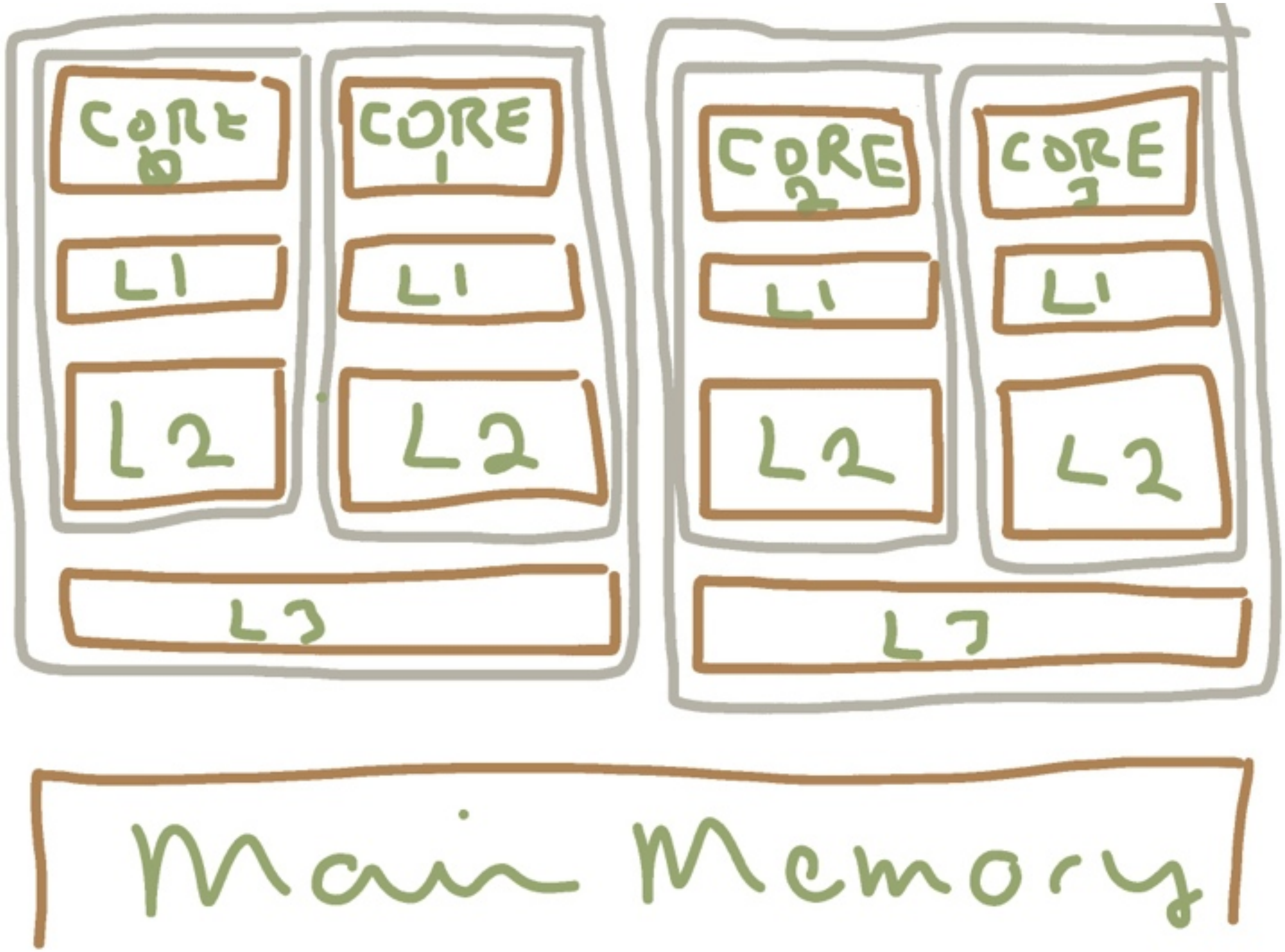
In the beginning there was *fork*

- **It was a copy operation**
  - No common memory space
  - So we had to work hard to share state between processes

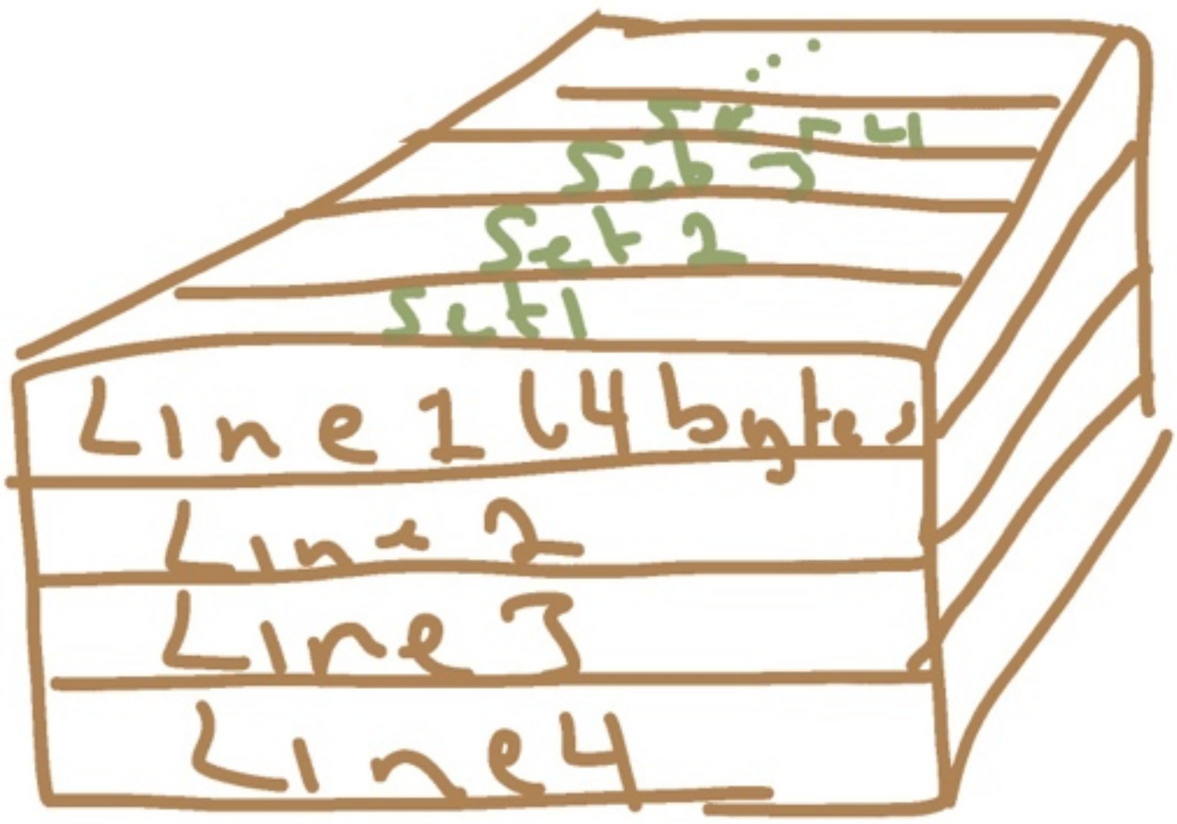
- **Everything was local to us**
  - We had no race conditions
  - Hardware caching was transparent, we didn't care
  - And we could cache anything at anytime without fear!

- **Life was simpler**
  - Ordering was guaranteed

In case you've forgotten



In case you've forgotten



Then we had multi-thread / single core

- **`new Thread()` is like a light-weight fork**
  - Memory is now shared
  - You don't work hard to share state
  - Instead you work hard to guarantee order
- **Guaranteeing order is a much harder problem**
  - We started with `volatile` and `synchronized`
  - They were fence builders, to help guarantee order
  - It made earlier processors do more work
- **Hardware, O/S & JVM guys were playing leap frog!**

Now we have multiple hardware threads

- **`new Thread()` is still a light-weight fork**
  - Memory is still shared
  - Now we have to work hard to share state due to caching
  - We still have to work hard to guarantee order
- **Guaranteed order is now a performance hit**
  - `volatile` and `synchronized` force draining and refreshing of caches
  - Some support to get around this (**CAS**)
  - Some more support to get around this (**NUMA**)
- **Hardware, O/S & JVM guys are still playing leap frog!**

What can I do?

"We protect code with the hope of protecting data."

— Gil Tene, Azul Systems, 2008

Protect your data

- **You can use a bunch of different techniques**
  - `synchronized`
  - `volatile`
  - atomics, e.g. `AtomicInteger`
  - Explicit Java 5 Locks, e.g. `ReentrantLock`

- **But these all have a performance hit**

- **Lets see examples of these in action**

The following source code can be read later

Don't worry! there is a point to this

Running Naked

```
      private int counter;
public void execute( int threadCount) {
      init();
      Thread[] threads = new Thread[ threadCount];
      for ( int i = 0; i < threads.length; i++) {
           threads[i] = new Thread( new Runnable() {
              public void run() {
                   long iterations = 0;
                   try {
                       while ( running) {
                            counter++;
                            counter--;
                            iterations++;
                       }
                   } finally {
                       totalIterations += iterations;
                   }
              }
          });
      }
    ...
```

Using volatile

```
private volatile int counter;
public void execute( int threadCount) {
    init();
    Thread[] threads = new Thread[ threadCount];
    for ( int i = 0; i < threads.length; i++) {
        threads[i] = new Thread( new Runnable() {
            public void run() {
                long iterations = 0;
                try {
                    while ( running) {
                        counter++;
                        counter--;
                        iterations++;
                    }
                } finally {
                    totalIterations += iterations;
                }
            }
        });
    }
  ...
```

Using synchronized

```
private int counter;
private final Object lock = new Object();
public void execute( int threadCount) {
     init();
     Thread[] threads = new Thread[ threadCount];
     for ( int i = 0; i < threads.length; i++) {
          threads[i] = new Thread( new Runnable() {
              public void run() {
                   long iterations = 0;
                   try {
                        while ( running) {
                             synchronized (lock) {
                                  counter++;
                                  counter--;
                             }
                             iterations++;
                        }
                   } finally {
                        totalIterations += iterations;
                   }
              }
          });
     }
   ...
```

Using fully explicit Locks

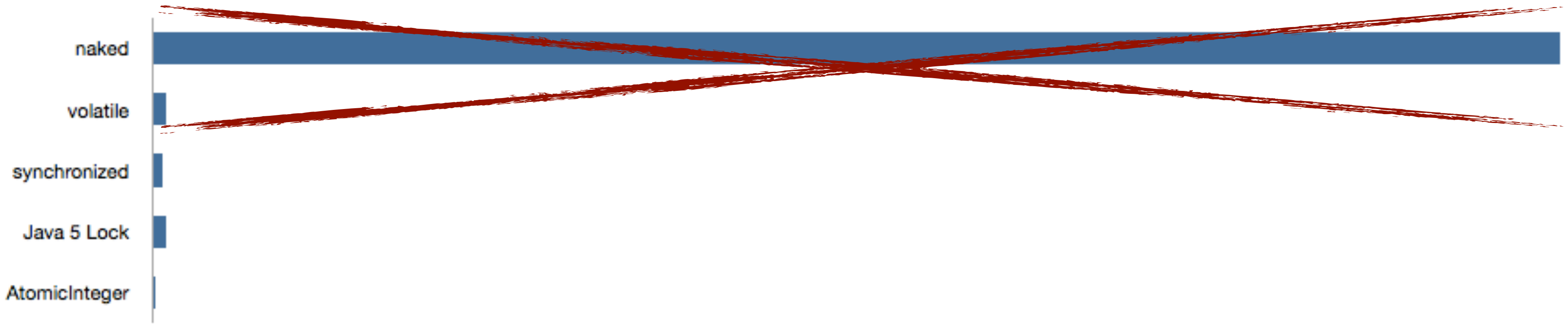
```
private int counter;
private final ReentrantReadWriteLock lock =
            new ReentrantReadWriteLock();
public void execute( int threadCount) {
     init();
     Thread[] threads = new Thread[ threadCount];
     for ( int i = 0; i < threads.length; i++) {
          threads[i] = new Thread( new Runnable() {
              public void run() {
                   long iterations = 0;
                   try {
                        while ( running) {
                             try {
                                  lock.writeLock.lock();
                                  counter++;
                                  counter--;
                             } finally {
                                  lock.writeLock.unlock();
                             }
                             iterations++;
                        }
                   } finally {
                        totalIterations += iterations;
                   }
   ...
```

Using AtomicInteger

```
private AtomicInteger counter = new AtomicInteger(0);
public void execute( int threadCount) {
    init();
    Thread[] threads = new Thread[ threadCount];
    for ( int i = 0; i < threads.length; i++) {
        threads[i] = new Thread( new Runnable() {
            public void run() {
                long iterations = 0;
                try {
                    while ( running) {
                        counter.getAndIncrement();
                        counter.getAndDecrement();
                        iterations++;
                    }
                } finally {
                    totalIterations += iterations;
                }
  ...
```

A small benchmark comparison

A small benchmark comparison



Locking kills performance!

Challenge: Virtualisation

- Better utilisation of hardware?
- You can't virtualise into more hardware
- What can I do?

Virtualisation

"Why do we do it?"

— Martijn Verburg and Kirk Pepperdine, JAX London 2012

Better utilisation of hardware?

- **Could be utopia because we waste hardware?**
  - Most hardware is idle
  - Load averages are far less than 10% on many systems

- **But why are our systems under utilised?**
  - Often because we can't feed them (especially CPU)
  - Throughput in a system is often limited by a single resource

- **People have forgotten to ask the question**
  - Sadly first principles are forgotten

T-800 is incapable of seeing your process

"A Process is defined as a locus of control, an instruction sequence and an abstraction entity which moves through the instructions of a procedure, as the procedure is executed by a processor"

— Jack B Dennis, Earl C Van Horn, 1965

Diabolical Developer

Processes don't exist as far as hardware is concerned

You can't virtualise into more hardware!

- **Throughput is often limited by a single resource**
  - That bottleneck can starve everybody else!

- **Going from most problematic to most problematic**
  - **Network, Disk, RAM, CPU**
  - Throughput overwhelms the wire
  - NAS means more pressure on network
  - CPU/RAM speed gap growing (8% per year)
  - Many thread scheduling issues (including cache!)
  - The list goes on and on and on..........

- **VMs sharing hardware causes resource conflict**
  - You need more than core affinity

What can I do?

- **Build your financial and capacity use cases**
  - TCO for virtual vs bare metal
  - What does your growth curve look like?
- **CPU Pinning trick / Processor Affinity**
- **Memory Pinning trick**
- **Move back to bare metal!**
  - It's OK - really!

Final Thought

"There are many reasons to move to the cloud - performance isn't necessarily one of them."

— Kirk Pepperdine - random rant - 2010

Challenge: The JVM

- WORA
- The cost of a strong memory model
- GC scalability
- GPU
- File Systems
- What can I do?

Diabolical Developer

Brian Goetz et al?

You've got a lot of work to do

WORA costs

- **CPU Model differences**
  - When are where are you allowed to cache?
  - When and where are you allowed to reorder?
  - AMD vs Intel vs Sparc vs ARM vs .....

- **File system differences**
  - O/S Level support

- **Display Devices - Argh!**
  - Impossible to keep up with the consumer trends
  - Aspect ratios, resolution, colour depth etc

- **Power**
  - From unlimited to extremely limited

The cost of the strong memory model

- **The JVM is ultra conservative**
  - It rightly ensures correctness over performance
- **Locks enforce correctness**
  - But in a very pessimistic way, which is expensive
- **Locks delineate regions of serialisation**
  - Serialisation?
  - Uh oh! Remember Little's and Amdahl's laws?

The visibility mismatch

- **Unit of visibility on a CPU != that in the JVM**
  - CPU - Cache Line
  - JVM - Where the memory fences in the instruction pipeline are positioned
- **False sharing is an example of this visibility mismatch**
  - `volatile` creates a fence which flushes the cache
- **We think that `volatile` is a modifier on a field**
  - In reality it's a modifier on a cache line
- **It can make other fields *accidentally volatile***

GC Scalability

- **GC cost is dominated by the # of live objects in heap**
  - Larger Heap ~= more live objects
- ***Mark* identifies live objects**
  - Takes longer with more objects
- ***Sweep* deals with live objects**
  - *Compaction* only deals with live objects
  - *Evacuation* only deals with live objects
- **G1, Zing, and Balance are no different!**
  - They still have to mark and sweep live objects
  - *Mark* for Zing is more efficient

GPU

- **Can't utilise easily today**
  - Java does not have normalised access to the GPU
  - The GPU does not have access to Java heap
- **Today - third party libraries**
  - That translate byte code to GPU RISC instructions
  - Bundle the data and the instructions together
  - Then push data and instructions through the GPU
- **Project Sumatra on its way**
  - http://openjdk.java.net/projects/sumatra/
  - Aparapi backs onto OpenCL
- **CPU and GPU memory to converge?**

What can I do?

- **Not using `java.util.concurrent`?**
  - You're probably doing it wrong
- **Avoid locks where possible!**
  - They make it difficult to run processes concurrently
  - **But** don't forget integrity!
- **Use `Unsafe` to implement lockless concurrency**
  - **Dangerous!!**
  - Allows non-blocking / wait free implementations
  - Gives us access to the pointer system
  - Gives us access to the CAS instruction
- **Cliff Click's awesome lib**
  - http://sourceforge.net/projects/high-scale-lib/

A reminder of Unsafe

What can I do?

- **Workaround GC with SLAB allocators**
  - Hiding the objects from the GC in native memory
  - It's a slippery slope to writing your own GC
    - We're looking at you Cassandra!
- **Workaround GC using near cache**
  - Storing the objects in another JVM
- **Peter Lawrey's thread affinity OSS project**
  - https://github.com/peter-lawrey/Java-Thread-Affinity
- **Join Adopt OpenJDK**
  - http://www.java.net/projects/adoptopenjdk

Challenge: Java

- `java.lang.Thread` is low level
- File systems
- Functional and Parallel
- What can I do?

`java.lang.Thread` is low level

- **Has low level co-ordination**
  - `wait()`
  - `notify()`, `notifyAll()`
- **Also has dangerous operations**
  - `stop()` unlocks all the monitors - leaving data in an inconsistent state
- **`java.util.concurrent` helps**
  - But managing your threads is still manual

Functional and Parallel

- **State of play today**
  - Manual parallelism
  - Very difficult to get right
- **Java 8 will solve much of this**
  - Lambdas
  - Internal iteration as opposed to external iteration
  - means you can multi-thread over a collection
- **JSR-166y - parallel collections**
  - Doug Lea et al on the concurrency-interest list
  - Collections libraries will be enhanced
  - `.parallel()`

File Systems

- **Java 7 to the rescue!**
  - Move to it as soon as you can

- **It was difficult to do asynchronous I/O**
  - You would get stalled on large file interaction
  - You would get stalled on multi-casting

- **There was no native file system support**
  - NIO bought in pipe and selector support
  - NIO.2 bought in symbolic links support

What can I do?

- **Read Brian Goetz's book**
  - Java Concurrency in Practice
- **Move to `java.util.concurrent`**
  - People smarter than us have thought about this
  - Use `Runnables`, `Callables` and `ExecutorServices`
- **Use thread safe collections**
  - `ConcurrentHashMap`
  - `CopyOnWriteArrayList`
- **Move to Java 7**
  - NIO.2 gives you asynchronous I/O!
  - Fork and Join helps with Amdahls law

Conclusion

- **Learn about hardware again**
  - Check your capacity and throughput
- **Virtualise with caution**
  - It can work for and against you
- **The JVM needs to evolve and so do you**
  - OpenJDK is adapting, hopefully fast enough!
- **Learn to use parallelism in code**
  - Challenge each other in peer reviews!

Don't become extinct

Be like Sarah

Acknowledgements

- The jClarity Team
- Gil Tene - Azul Systems
- Warner Bros pictures
- Lionsgate Films
- Dan Hardiker - Adaptivist
- Trisha Gee - 10gen
- James Gough - Stackthread

jClarity

http://www.jclarity.com

Martijn Verburg (**@karianna**)