UNIVERSITY OF Southampton
School of Electronics and Computer Science

Issues in Distributed Computing

Denis A Nicole

2010-11-15

Scope

- This talk is *not* about Distributed Computing. That topic addresses other issues:
  - Unreliable communications/systems
  - Trust and trust boundaries
  - Hetrogeneous systems, although these will creep in…
- Today, we are discussing Parallel Computing

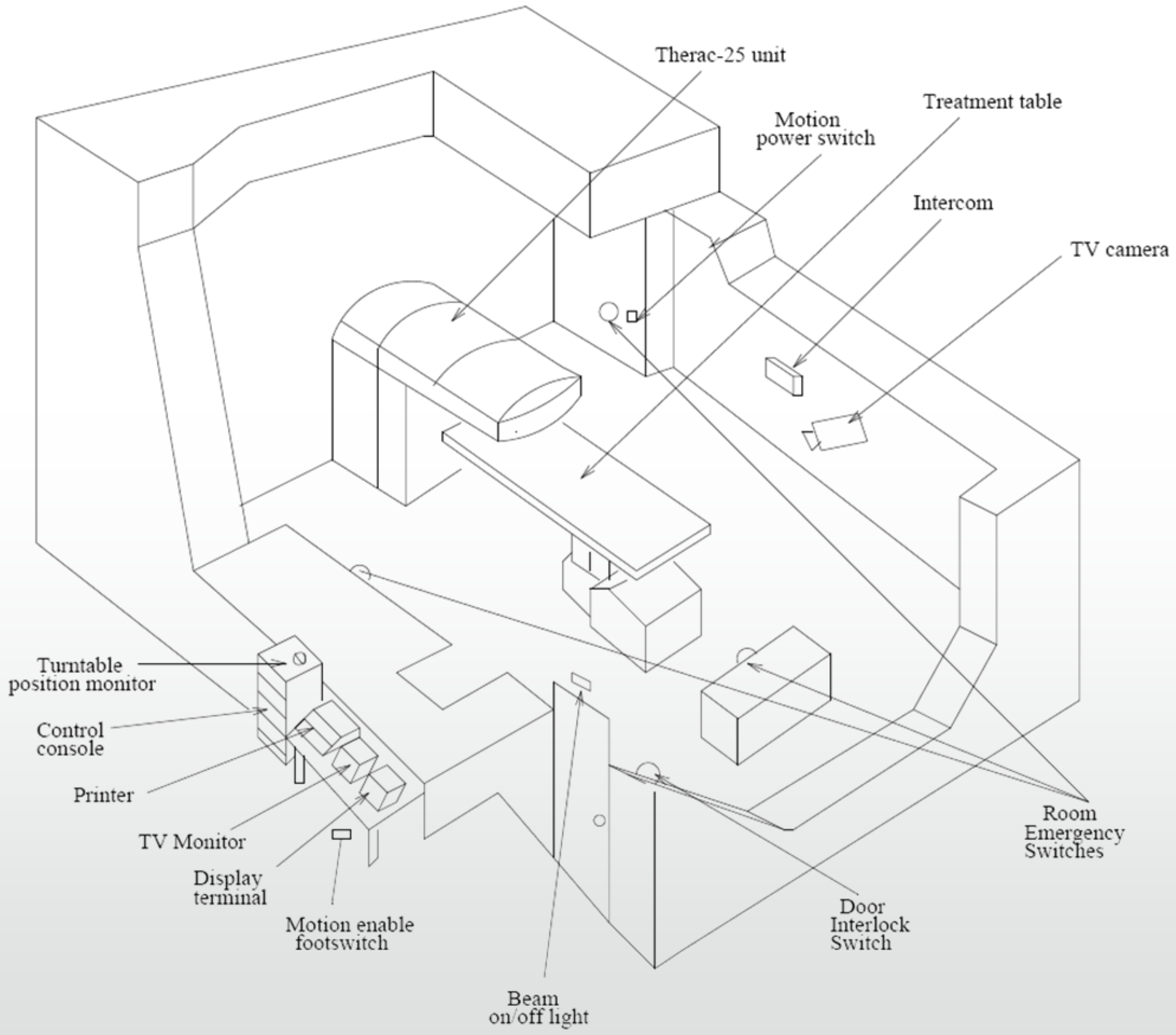
  [No references, just keywords. Use Google]

A sad story

- You throw together (pair program) a fast new concurrent system.
  - It goes fast
  - It works during testing (you can’t type very fast)
  - The customer loves it (on time and on cost)

- You go back to U Waterloo(?) to finish your degree.

Later you read about your system…



http://sunnyday.mit.edu/papers/therac.pdf

UNIVERSITY OF Southampton
School of Electronics and Computer Science

It just gets worse

1. You need enough locks to prevent uncontrolled data sharing. This causes death.

   You are now one step ahead. You may not sell any product, but you won't have to explain your code to a coroner, as it probably does not do anything.

2. You might now get deadlock. Use a standard technique (eg breaking symmetry/ breaking cycles or preventing resource filling) or work something out and test your idea by model checking.

   You now have code that works. Marketing can probably sell it. Of course, there's every probability that your carefully locked code runs slower than a sequential solution.

UNIVERSITY OF Southampton
School of Electronics and Computer Science

Historic approaches

- Programming languages were fashionable in the ’80s

“Embrace our new world and the advantages of parallelism will flow automatically.”

So we tried

- Dataflow: parallel subexpression evaluation
  - Manchester Dataflow
  - SISAL: single assignment language
- Functional: parallel argument evaluation
  - Transputer-based Alice hardware and (eager) Hope
  - Lazy languages: Miranda (David Turner), Orwell, Haskell
    - Leading to Bluespec and other current activity
- Logic
  - Parlog: AND parallel Prolog

UNIVERSITY OF Southampton
School of Electronics and Computer Science

And tried…

- Parallel Objects
  - Doom: hardware (too-slow processor coupled to vector units with too-weak memory addressing)
  - Pool: inheritance can interact badly with parallelism. (Inheritance Anomaly)
- Parallel processes
  - Transputers and now the XMOS XC-1.
  - occam: static compile-time concurrency, leading to XC.
  - MPI is still the preferred way of coding big science

occam has an algebra

-
  ```
  CHAN c:
  PAR
    c ! E                     v := e
    c ? v
  ```
  ≡

-
  ```
  CHAN c:
  SEQ
    c ! 0                     STOP
    VAR v:
    c ? v
  ```
  ≡

-
  ```
  PAR                         ALT
    a ? x                       a ? x
    b ? y                         b ? y
                                b ? y
                                  a ? x
  ```
  ≡

These identities hold partly because of the CSP basis but also because of the careful design of the syntax and semantics.

The laws of occam programming, Theoretical Computer Science **60** (1988), 177-229

MPI

UNIVERSITY OF Southampton
School of Electronics and Computer Science

```
#include "mpi.h"
#include <stdio.h>
#define HOST_ID 0
int main(int argc, char* argv) {
    int myid, numprocs; int i, n; int tag = 23; /* arbitrary value */ MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs); MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    if (myid == HOST_ID)  { /* HOST */
           int reply;
           MPI_Status status;
           n = 4;
           printf("Host Process about to broadcast\n");
           MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
           printf("Broadcast %d\n", n);
           for (i=1;i<numprocs; i++) {
                    printf("Host receiving reply %d\n", i);
                    MPI_Recv(&reply, 1, MPI_INT, MPI_ANY_SOURCE, tag, MPI_COMM_WORLD, &status);
                    printf("Received %d from process %d\n", reply, status.MPI_SOURCE); }; }
    else { /* WORKER */
           int reply;
           MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
           reply = n + myid; MPI_Send(&reply, 1, MPI_INT, 0, tag, MPI_COMM_WORLD); };
    MPI_Finalize(); }
```

Of course, Real Programmers would use FORTRAN.

SIMD and MIMD

UNIVERSITY OF Southampton
School of Electronics and Computer Science

- Remember Harvard architectures?
  - eg, Microchip PIC: separate data and program memory

- There was once a time when
  1. Instruction fetch/decode was expensive
  2. Bus communication was (or might be) cheap

- Single Instruction Multiple Data was born
  - ICL DAP, CM1 etc. Useful for radar, vision, search.

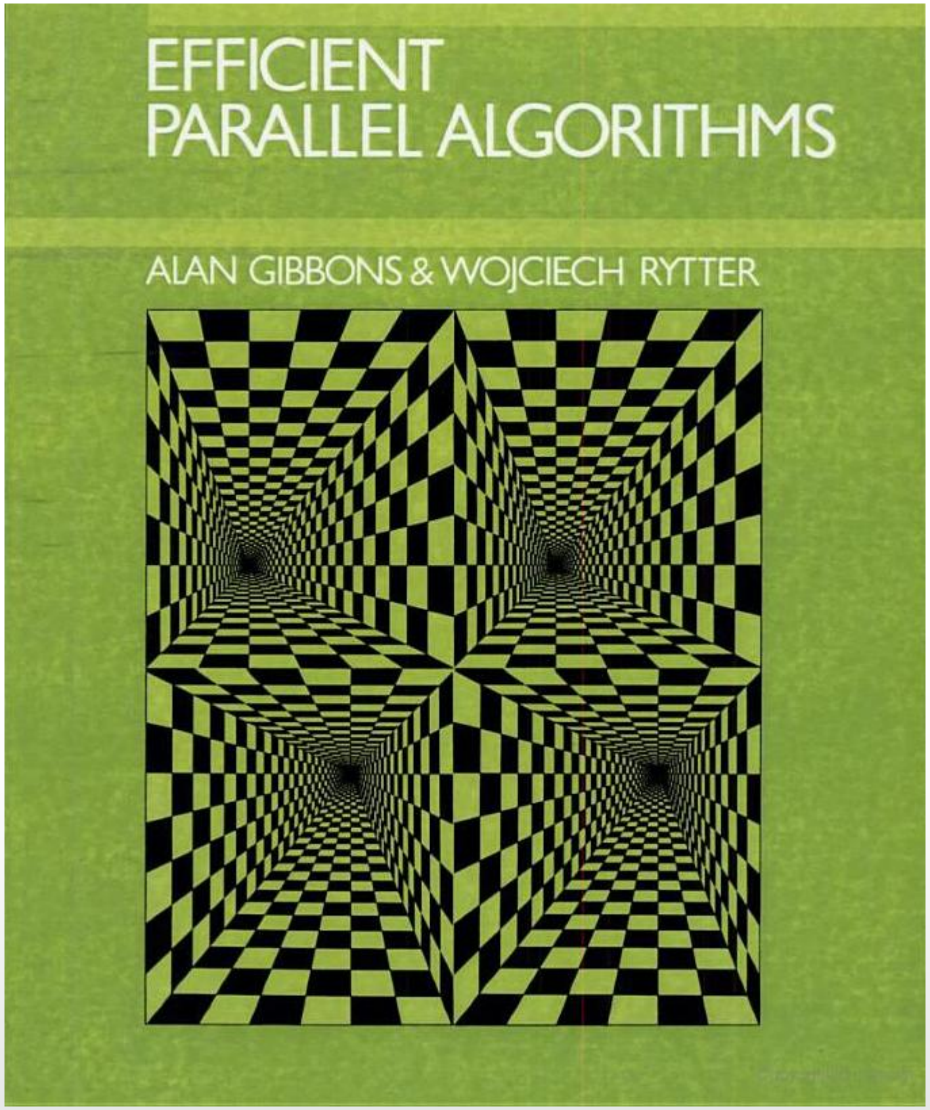
- Actually, a lot of MPI Fortran looks rather like this.

Within MIMD

- There are a lot of different memory models
  - Shared nothing (message passing)
  - Symmetric (SMP)
  - Partially shared
  - Hierarchical (NUMA)
  - Hybrid
- and lots of data coherence models

Theoretical approaches

- The PRAM: magical shared memory
  - A large set of processors, running their own codes but sharing a common address space and running in lockstep.
  - Came in varieties: EREW, CREW etc. which made different decisions about simultaneous accesses to a single address
- Many algorithms depended at compile-time on problem size.


EFFICIENT PARALLEL ALGORITHMS

ALAN GIBBONS & WOJCIECH RYTTER

Memory networks

- PRAM is unbuildable. Accessing big (size N) memory requires:
  - Money: **O(N)** for the memory and processors
  - Money: **O(N ln N)** for a routing network
  - Time: **O(ln N)** to get the messages through an uncongested network (and potentially *much* longer if congested).
- While we are at it, lets add intelligence to the network.
  - Merge read requests
  - Fetch-and-add: rank assignment, prefix sum
  - Fetch-and-swap: list builder
  - Reduce: for convergence tests etc.

Valiant (BSP) to the rescue

UNIVERSITY OF Southampton
School of Electronics and Computer Science

- Memory delay can be hidden by O(ln N) excess parallelism.
- Memory bandwidth is an absolute constraint.
- This approach was used by Burton-Smith in the HEP and Tera machines.
- Now split into three camps:
  - Aggressive fine grained (Cray MTA)
  - Fine-grained but few threads (XMOS, META)
  - Opportunistic (Intel HyperThreading)
- Was popular on conventional architectures at Oxford.

The end of history

- Why did almost none of this catch on?
  - Poor backward compatibility: new languages
  - Academic-led: no real industrial pull
  - Often static parallelism: difficult to use in, eg, an operating system or a web application
  - Much of it didn't work very well.
- But the Imagination META is a success in mixed DSP+UI
- the alternative for these applications is the heterogeneous OMAP (ARM+VLIW etc) which, as it happens contains Imagination's PowerVR technology.

UNIVERSITY OF Southampton
School of Electronics and Computer Science

PURE Sensia Network audio player / DAB / FM clock radio

A more recent try: transactional memory

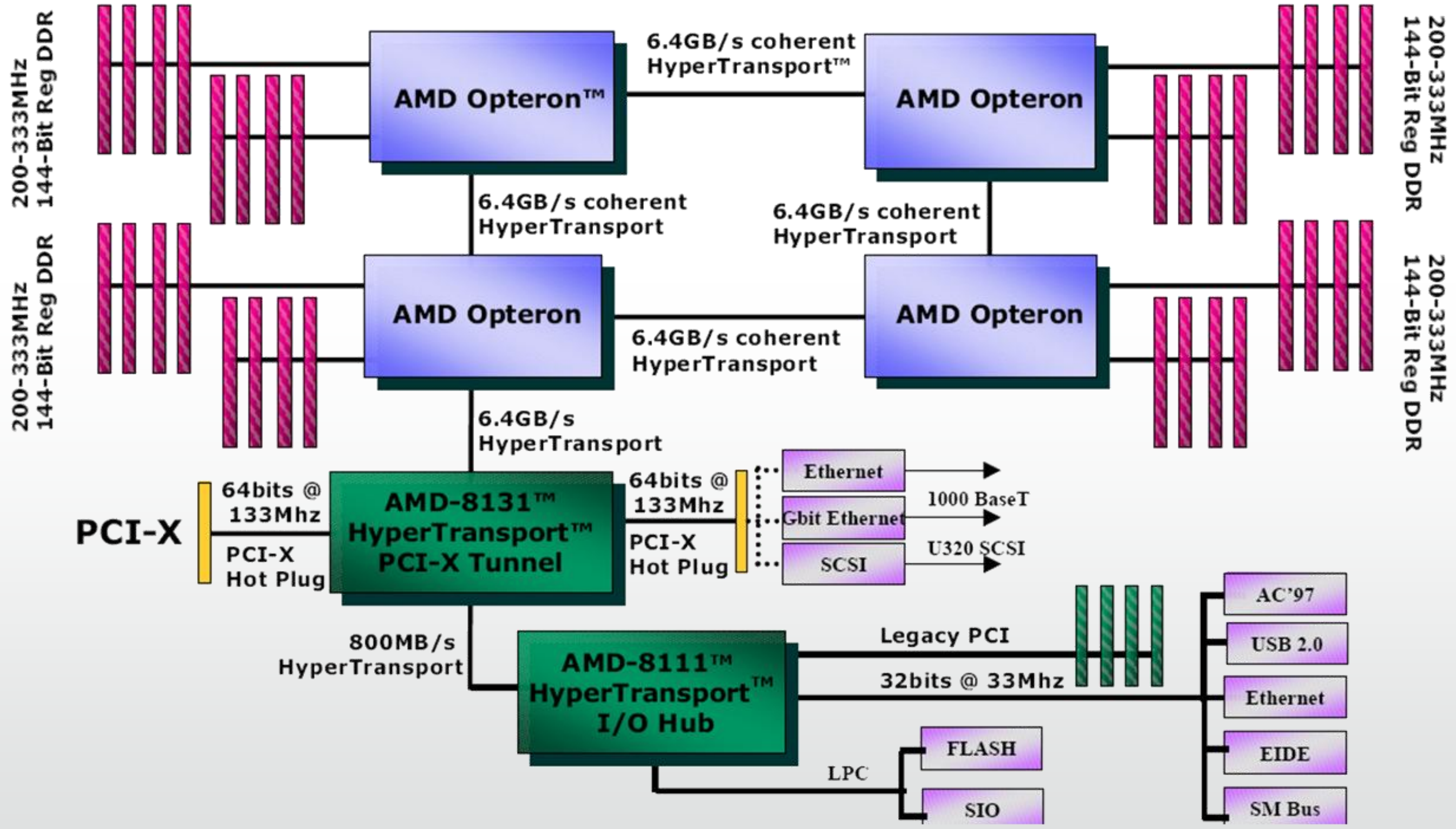
- Hardware (Rock) and Java (DSTM2) support from Sun.
- A sequence of memory operations either executes completely (commits) or has no effect (aborts).
- Works well if uncontended...but then so do a lot of techniques!
- Can be implemented in HW, SW or hybrid.
- Needs careful hybrid optimisation for contended case.

UNIVERSITY OF Southampton
School of Electronics and Computer Science

Typical Modern Hardware

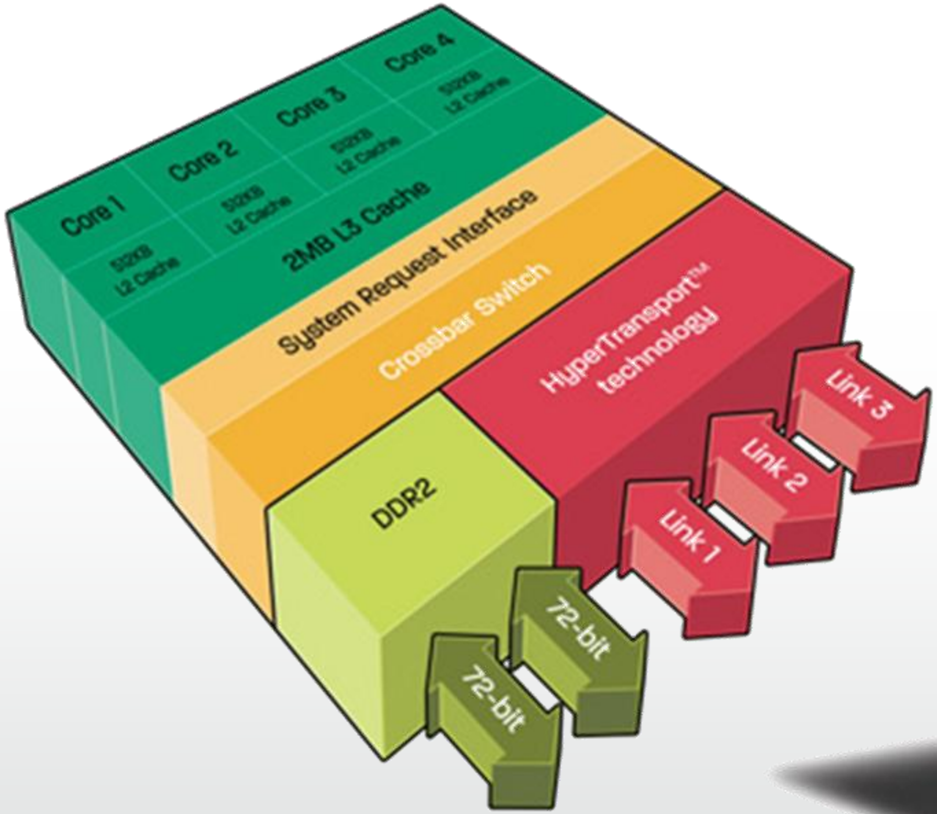
- Not much parallel *architecture*.
- Just implements what the hardware designer can easily provide; an awkward mix of (hierarchical) shared memory and message passing, often with poor synchronisation.
- What about HW/SW co-design?
  - Cray have abandoned FPGA boards.
  - Bluespec is Haskel
  - SystemVerilog has OO features
  - SystemC is widely used for high-level simulation.

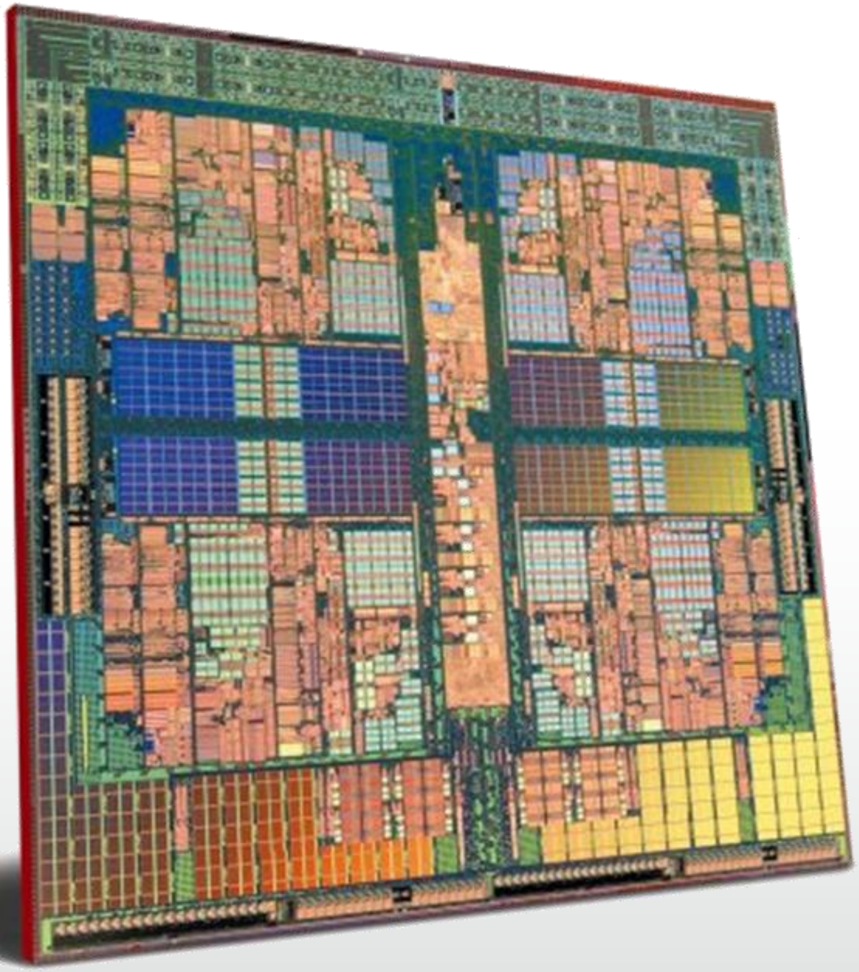
AMD and Intel Multi-core


200-333MHz 144-Bit Reg DDR
AMD Opteron™
6.4GB/s coherent HyperTransport™
AMD Opteron
200-333MHz 144-Bit Reg DDR
6.4GB/s coherent HyperTransport
6.4GB/s coherent HyperTransport
200-333MHz 144-Bit Reg DDR
AMD Opteron
6.4GB/s coherent HyperTransport
AMD Opteron
200-333MHz 144-Bit Reg DDR
6.4GB/s HyperTransport
PCI-X
64bits @ 133Mhz
PCI-X Hot Plug
AMD-8131™ HyperTransport™ PCI-X Tunnel
64bits @ 133Mhz
PCI-X Hot Plug
Ethernet
Gbit Ethernet
SCSI
1000 BaseT
U320 SCSI
800MB/s HyperTransport
AMD-8111™ HyperTransport™ I/O Hub
Legacy PCI
32bits @ 33Mhz
LPC
FLASH
SIO
AC'97
USB 2.0
Ethernet
EIDE
SM Bus

UNIVERSITY OF Southampton
School of Electronics and Computer Science

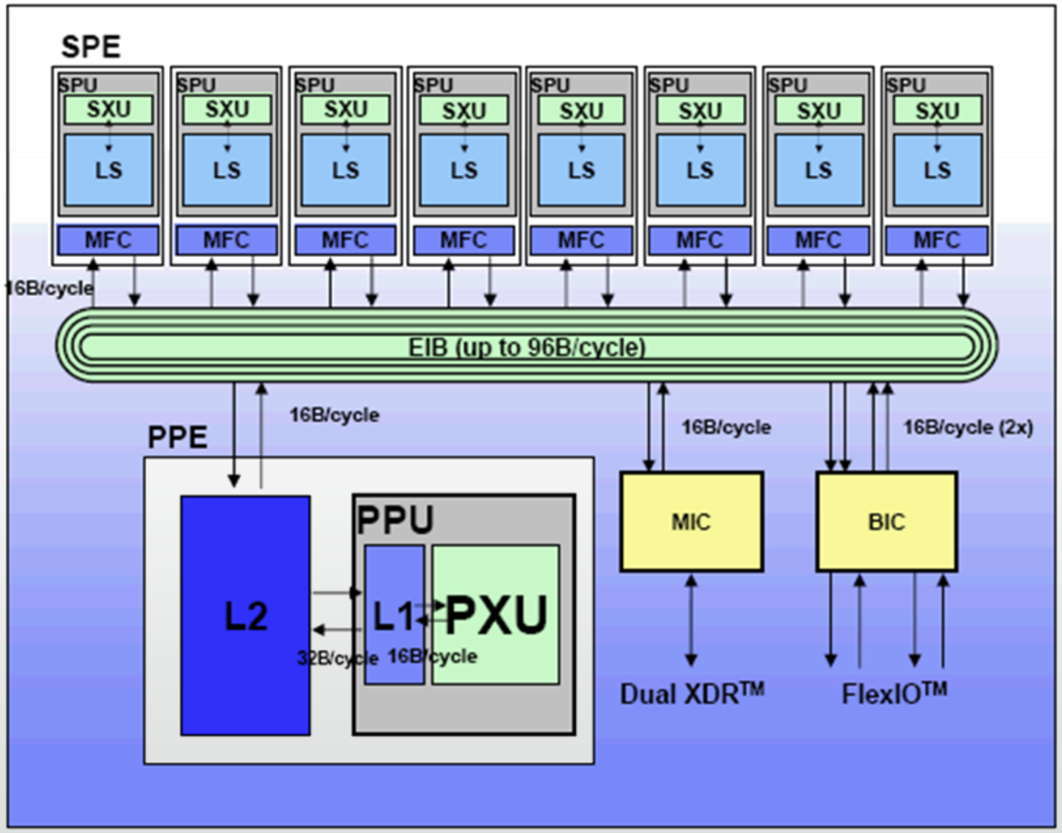
and inside each chip





IBM/Sony/Toshiba Cell

- Cell is a heterogeneous chip multiprocessor that consists of an IBM 64-bit Power Architecture™ core, augmented with eight specialized co-processors based on a novel single-instruction multiple-data (SIMD) architecture called Synergistic Processor Unit (SPU), which is for data-intensive processing, like that found in cryptography, media and scientific applications.



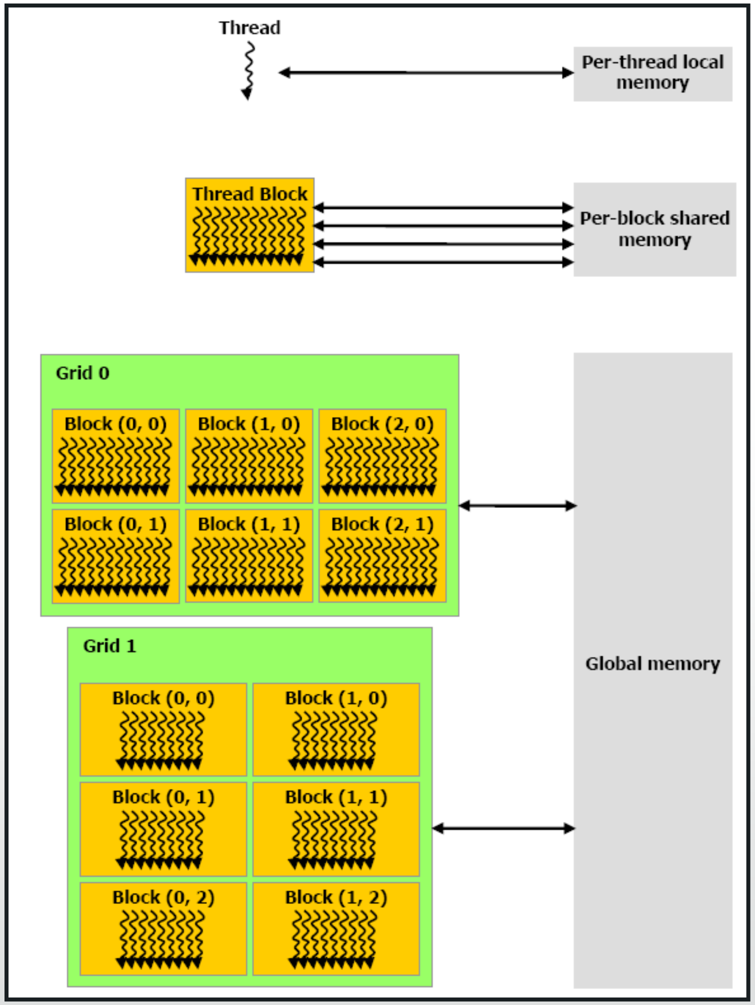
Source: M. Gschwind et al., Hot Chips-17, August 2005

Nvidia CUDA GPUs

- Terrifyingly complex:

    CUDA threads may access data from multiple memory spaces during their execution as illustrated by Figure 2-2. Each thread has a private local memory. Each thread block has a shared memory visible to all threads of the block and with the same lifetime as the block. Finally, all threads have access to the same global memory.

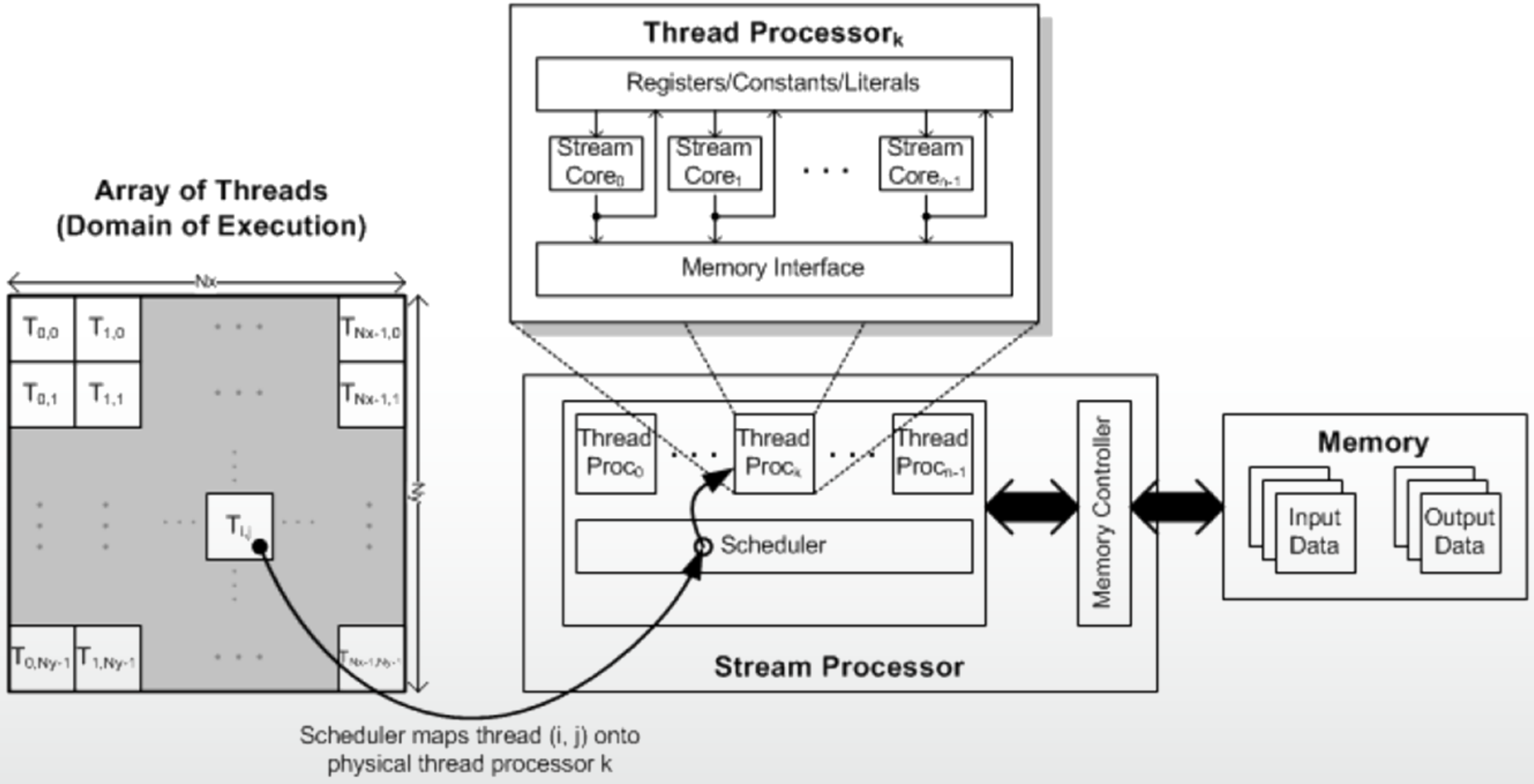
    There are also two additional read-only memory spaces accessible by all threads: the constant and texture memory spaces. The global, constant, and texture memory spaces are optimized for different memory usages. Texture memory also offers different addressing modes, as well as data filtering, for some specific data formats.



AMD (ATI) Stream Computing

UNIVERSITY OF Southampton
School of Electronics and Computer Science

- Like Nvidia, only different…

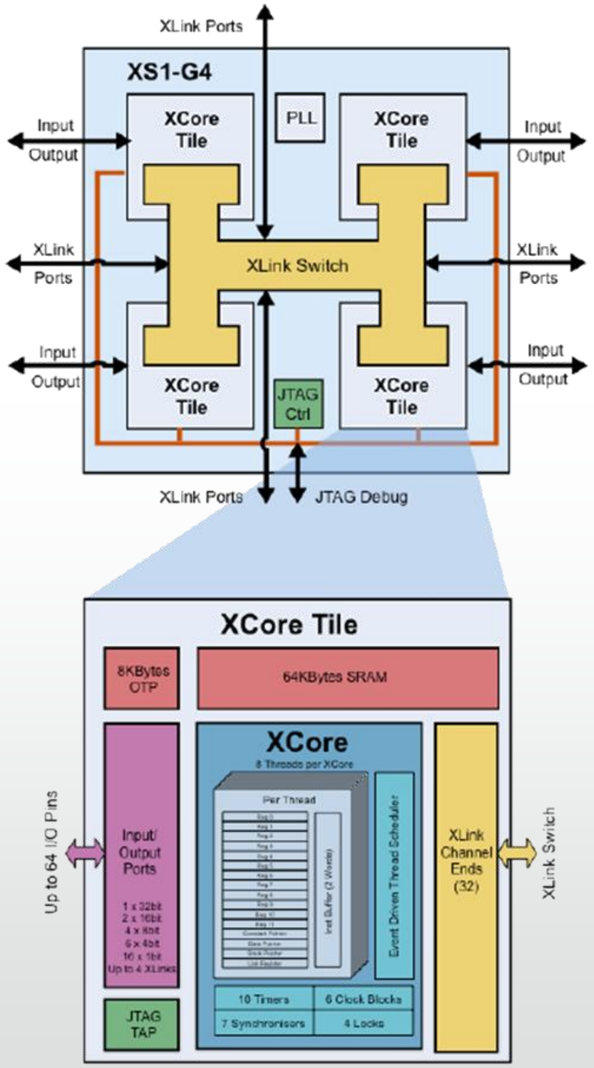


Simplified AMD Stream Computing Programming Model

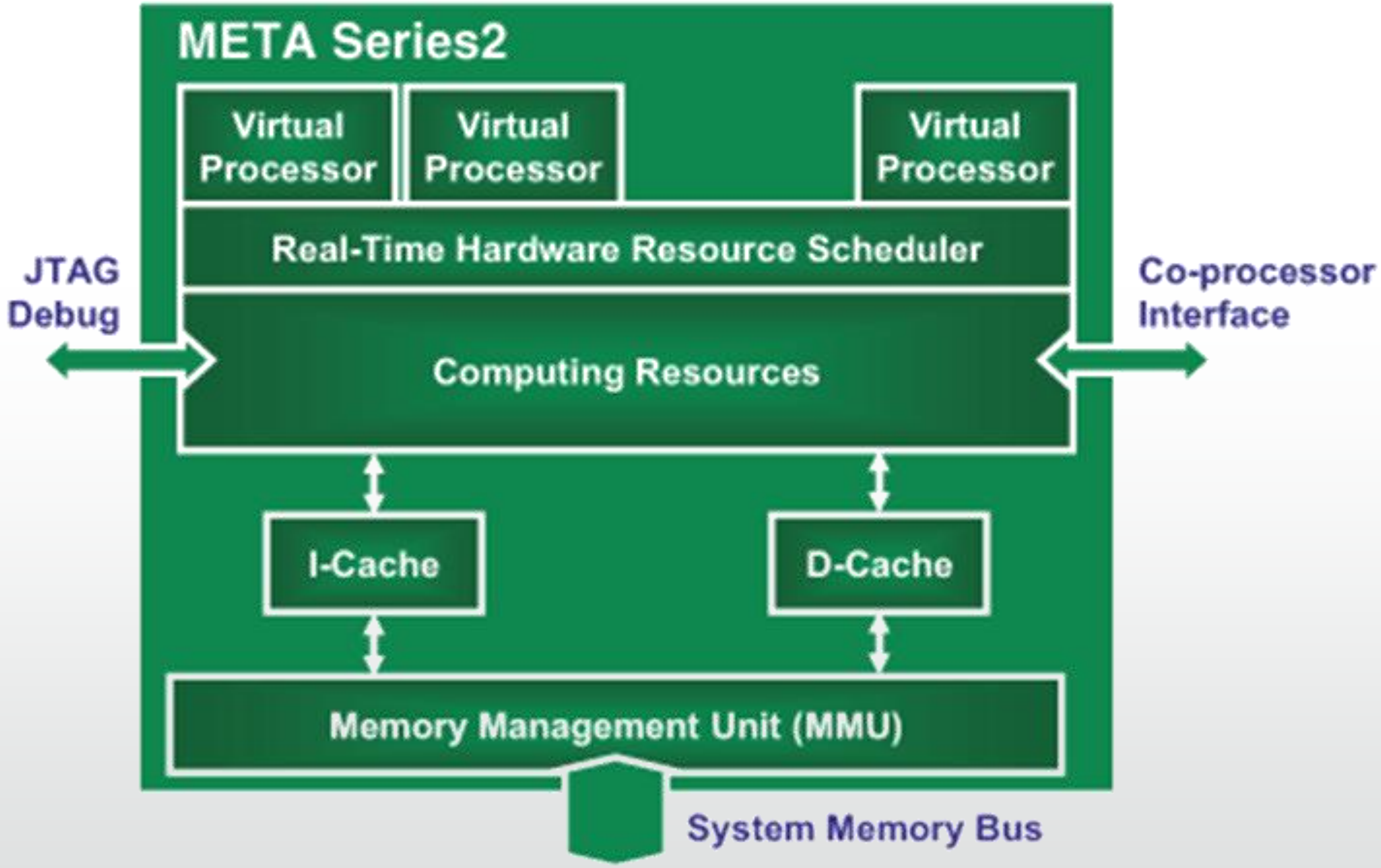
XMOS: Transputers (and Burton-Smith) ride again

UNIVERSITY OF Southampton
School of Electronics and Computer Science

- "Software Defined Silicon" (well, maybe)
- Fixed eight threads per processor (cf Tera)
- Hardware routing switch (cf INMOS C104)
- Initially horrid power consumption
- XC programming language (cf occam, Handel C)



Imagination META



Java

- The Java authors adopted a version of pthreads' mutexes and condition variables. The standard usage is:

```
synchronized foo(…) throws InterruptedException {
        while (! OK_to_proceed) wait();
        /* DO_STUFF */
        notifyAll(); }
```

UNIVERSITY OF Southampton
School of Electronics and Computer Science

Java wait() is truly horrid

- `while` -> `if` results in random failure

```
synchronized foo(…) throws InterruptedException {
        while (! OK_to_proceed) wait();
        /* DO_STUFF */
        notifyAll(); }
```

Java wait() is truly horrid

- `while` -> `if` results in random failure
- What's with `InterruptedException`?

```
synchronized foo(…) throws InterruptedException {
      while (! OK_to_proceed) wait();
      /* DO_STUFF */
      notifyAll(); }
```

UNIVERSITY OF Southampton
School of Electronics and Computer Science

Java wait() is truly horrid

- `while` -> `if` results in random failure
- What's with `InterruptedException`?
- Why is the lock exposed to class users?

```
synchronized foo(…) throws InterruptedException {
        while (! OK_to_proceed) wait();
        /* DO_STUFF */
        notifyAll(); }
```

UNIVERSITY OF Southampton
School of Electronics and Computer Science

Java wait() is truly horrid

- `while` -> `if` results in random failure
- What's with `InterruptedException`?
- Why is the lock exposed to class users?
- How does it scale?

```
synchronized foo(…) throws InterruptedException {
        while (! OK_to_proceed) wait();
        /* DO_STUFF */
        notifyAll(); }
```

UNIVERSITY OF Southampton
School of Electronics and Computer Science

Java wait() is truly horrid

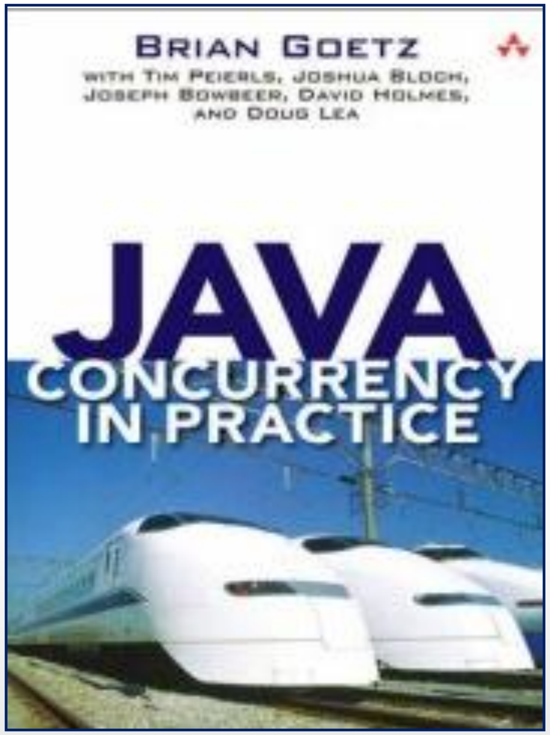
- `while -> if` results in random failure
- What's with `InterruptedException`?
- Why is the lock exposed to class users?
- How does it scale?
- What are the fairness properties?

```
synchronized foo(…) throws InterruptedException {

while (! OK_to_proceed) wait();

/* DO_STUFF */

notifyAll(); }
```

UNIVERSITY OF Southampton
School of Electronics and Computer Science

Java.util.concurrency

- Java 1.5 fix to the major known problems
- A reasonable implementation of a bad model
- Takes advantage of optimistic concurrency primitives


BRIAN GOETZ
WITH TIM PEIERLS, JOSHUA BLOCH, JOSEPH BOWBEER, DAVID HOLMES, AND DOUG LEA
JAVA CONCURRENCY IN PRACTICE

UNIVERSITY OF Southampton
School of Electronics and Computer Science

Old style

```
public final class Counter {
   private long value = 0;
      public synchronized long getValue() {
      return value; }
   public synchronized long increment() {
      return ++value; } }
```

UNIVERSITY OF Southampton
School of Electronics and Computer Science

New style

```
public class NonblockingCounter {
  private AtomicInteger value;
  public int getValue() {
      return value.get(); }
  public int increment() {
      int v;
      do
           { v = value.get(); }
           while (!value.compareAndSet(v, v + 1));
      return v + 1; } }
```

[http://www-128.ibm.com/developerworks/java/library/j-jtp04186/index.html]

Think about the ABA problem: not an issue here

AtomicInteger is fast

- `compareAndSet` is implemented through `sun.misc.Unsafe`
- The JVM has the opportunity to implement this using

  ```
      LOCK CMPXCHG r/m32, r32
  ```

  on Intel 486+ processors
- The *uncontended* case is fast
- What about fairness? Infinite delay is “unlikely”, and there’s always progress.

UNIVERSITY OF Southampton
School of Electronics and Computer Science

Frighten yourself