Everything Java™ Technology... but Better and Faster: The Evolution of JPC

Rhys Newman
Dept Physics, Oxford University and GSA Capital
Chris Dennis
Department of Physics Oxford University

TS-5180

Java

Sun microsystems

DEMO: JPC In Action...

- Run DOS and Linux virtual machines inside a Java Virtual Machine
  - Try Mario Bros, Pac Man and more classics ... all from within an applet in a browser!
  - Visit http://www-jpc.physics.ox.ac.uk

DEMO: Want to see more...?

- Latest JPC implements the x86 *Virtual8086 mode* which lets it run.....

JPC is an *Emulator* not a *Virtualizer*

- A virtualizer
  - Uses hardware support to run *guest* software (including an OS) in isolation to the real *host* operating system
  - Will only work when guest software (or OS) is built for the actual hardware being used (x86 on x86, not x86 on Sparc).
  - Can achieve near native speeds for the guest software

- An emulator
  - Models all hardware in software
  - No explicit dependency on underlying hardware – very flexible
  - Can suffer significant speed penalty

DEMO: Emulation can be *very* flexible...

- JPC can boot unmodified DOS on a mobile phone
  - Native x86 OS software on an ARM11 system

How Does JPC Do it?

- We read the technical specifications of the x86 PC hardware
- We implement the data flow and structures in Java application environment to mimic these
- We correct these implementations where the specs are wrong!
  - Loads of low-level debugging at “virtual” assembly level
  - We use existing (open source) examples to check out how things work
- Simple so far…..?

The hardware inside an x86 PC:

CD-ROM Drive

Network

Interrupt Controller

PCI Host Bridge

PCI ISA Bridge

PS/2 Interface

DMA Controller

Interval Timer

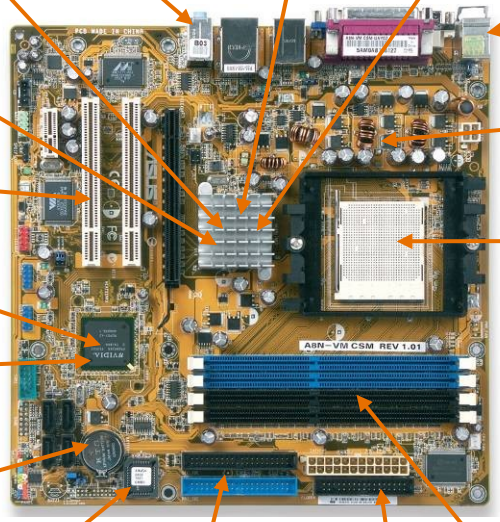
PCI Bus

Processor

VGA Graphics

VGA BIOS

Real Time Clock



MMU

Keyboard

Mouse

Floppy Drive

Hard Disk Drive

BIOS

IDE Interface

Memory

Floppy Controller

Emulation Speed

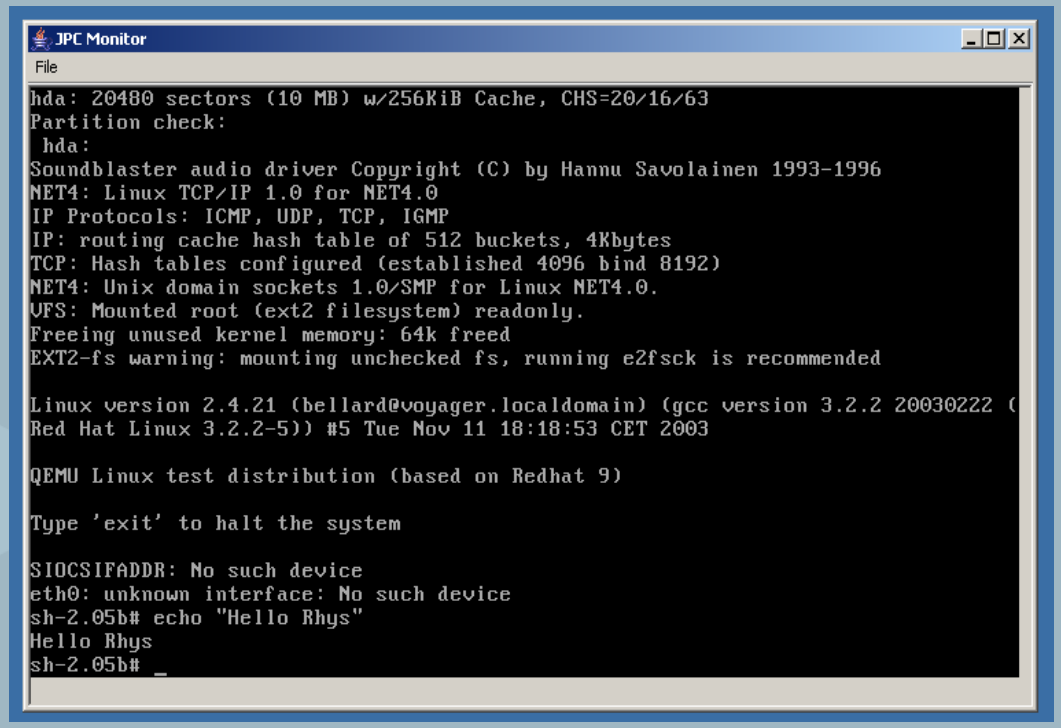
- Doing this the simple naïve way results in a really slow emulator
  - Suffers the double penalty of emulation and JVM overhead (resulting from speed implications of structured OO design)
  - For a much simpler x86 emulator, which is just like this see

DIOSCURI

- But JPC aims to be much faster…
  - Exploit the fact that x86 code is not often changed once loaded
  - Decode once and run many (ratio of 1:100000 is typical)
  - Build classes on the fly and load into the Java Virtual Machine (JVM™ machine) using a runtime classloader – let hotspot go for it….

DEMO: Dynamic Compilation

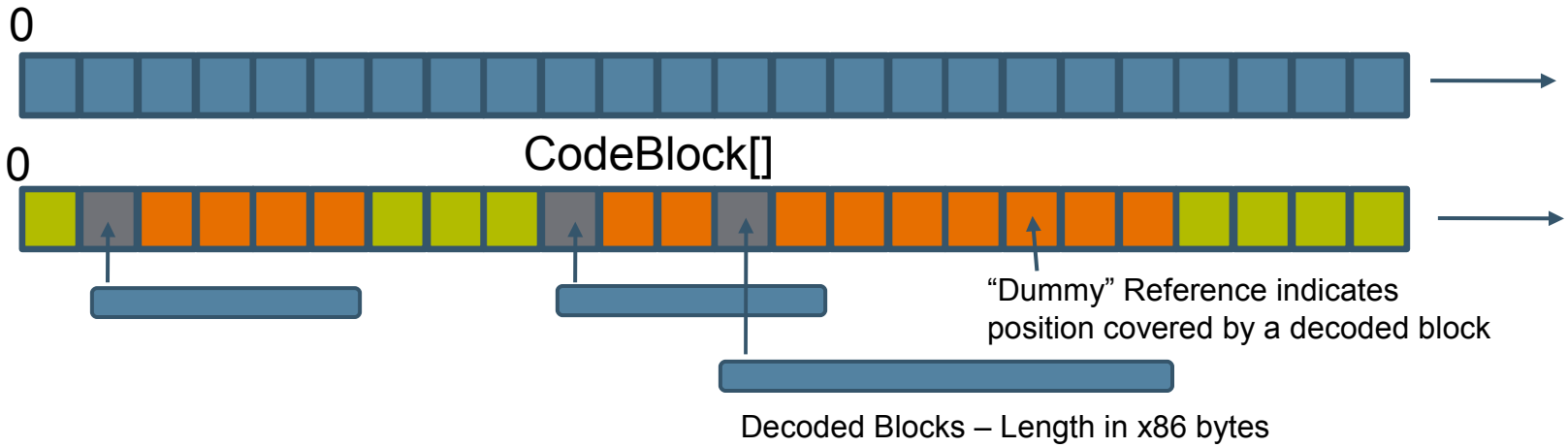
The only way to run a modern 32bit OS like Linux!



…and we get hardware level checkpointing for free!

How to get the speed

- Instruction pointer jumps to memory address
- Decode x86 instructions up to next jump
- Place object in shadow CodeBlock array which executes the instructions
  - This object can simply interpret x86 in a basic manner
  - Group of x86 can be bunched up, examined and a class file built on the fly to imitate the instructions – *it's our own bytecode compiler*
  - Class file then loaded via a special classloader ready for fast execution next time



The 2007 JavaOne Event Compiler

- Let's Be Honest
  - The compiler shown at the 2007 JavaOne conference was dumb
- It needed rewriting…
  - We need smaller classes…
  - We need shorter startup code for each block….
  - We need more efficient code
    - Dead/redundant code elimination
    - Other standard compiler optimisations
  - We need to chain blocks together….
- Gentlemen, we can rebuild him. We have the technology

We Form a Directed Acyclic Graph

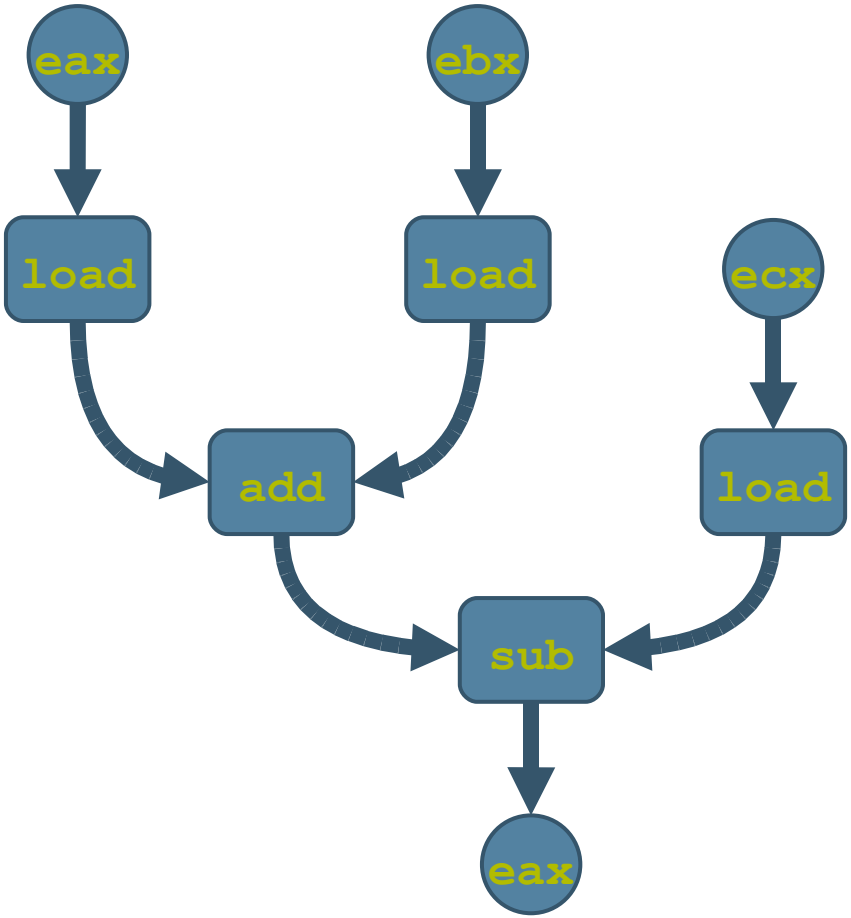
```
load reg0, eax;
load reg1, ebx;

add reg0, reg1;

load reg1, ecx;
sub reg0, reg1;

store eax;
```

Then rinse and repeat for every affected variable.



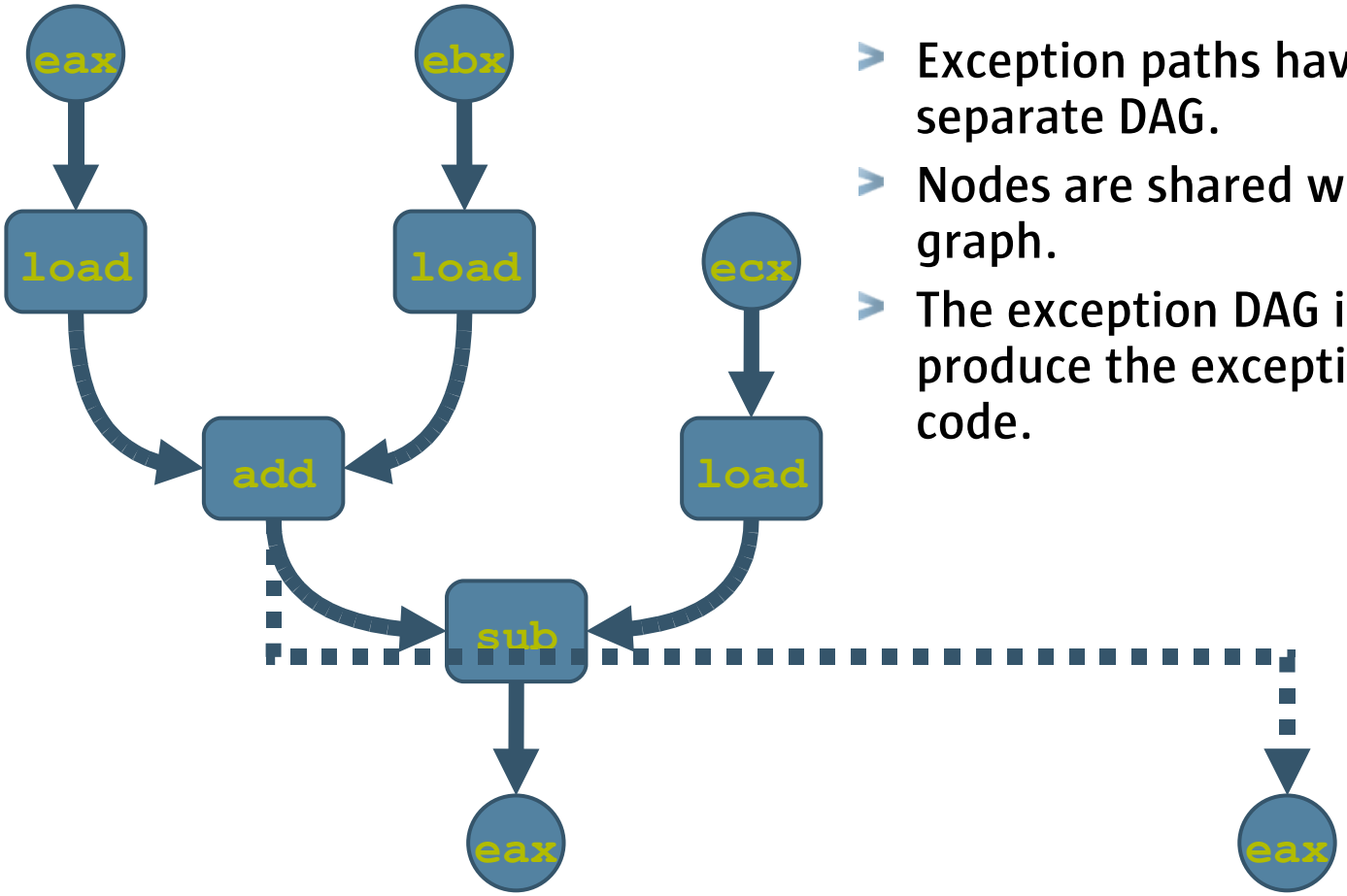
Then We Generate The Code…

- Code generation from a DAG is simply performed as a depth first traversal of the graph.
- For optimal traversal we should always follow the deepest path at each junction.
  - In a register based machine this optimally reduces the amount of ‘register juggling’
  - In a stack based machine it will give us the minimal stack depth usage.
- Naturally reject unused code – unused code is orphaned from the graph’s final sink
- By tracking multiple node visits we can also eliminate duplicate code.

Damn Those Exceptions...

- In real mode, this scheme works fine as processor (hardware level) exceptions are rare and serious
- In protected mode, the MMU paging system means *every* memory access potentially throws a page fault
  - Not all exceptions are bad, they're just exceptional!
  - In modern machines, this is common
  - After the page fault is handled (memory swapped back in), execution must continue from where it left off
- If this happens within a compiled block of bytecode:
  - Map processor exceptions to Java exceptions and use the JVM exception handling mechanism

Again, again!
Yet more Directed Acyclic Graphs



- Exception paths have their own separate DAG.
- Nodes are shared with the main graph.
- The exception DAG is traversed to produce the exception handler code.

Compiling in Protected Mode

- Use the Java class files exception table to catch page faults
  - The memory object throws a Page Fault exception as required
  - The exception table is populated with special code during compilation to save out processor/memory state up to the point of the fault
  - The fault is then handled as normal
  - The instruction pointer returns to the required location – now part way through the codeblock being executed previously
  - A new codeblock is constructed for that location – most likely JPC will simply interpret the x86 code to the end of the block
  - No problem, next time around the entire block will probably execute in compiled mode as desired!

Better than he was before.

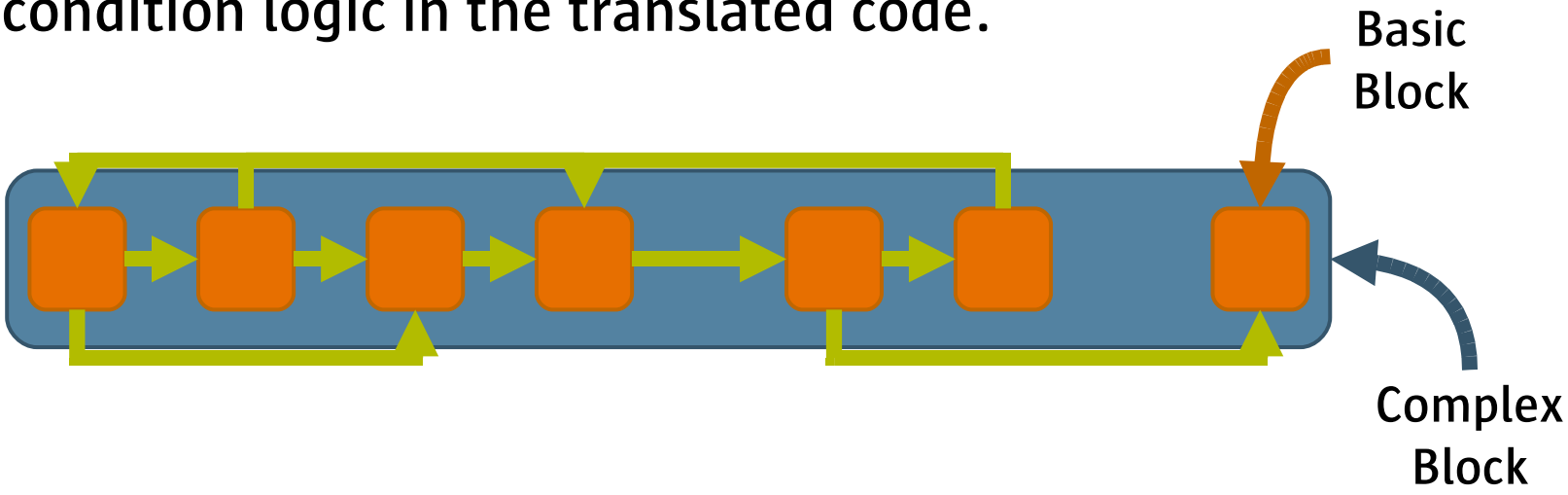
Better, stronger, faster.

- We can now compile blocks in all modes (Real, Protected, Virtual 8086...)
- With full exception handling.
- Redundant code removal.
- Duplicate code removal.

- But this is not the end...

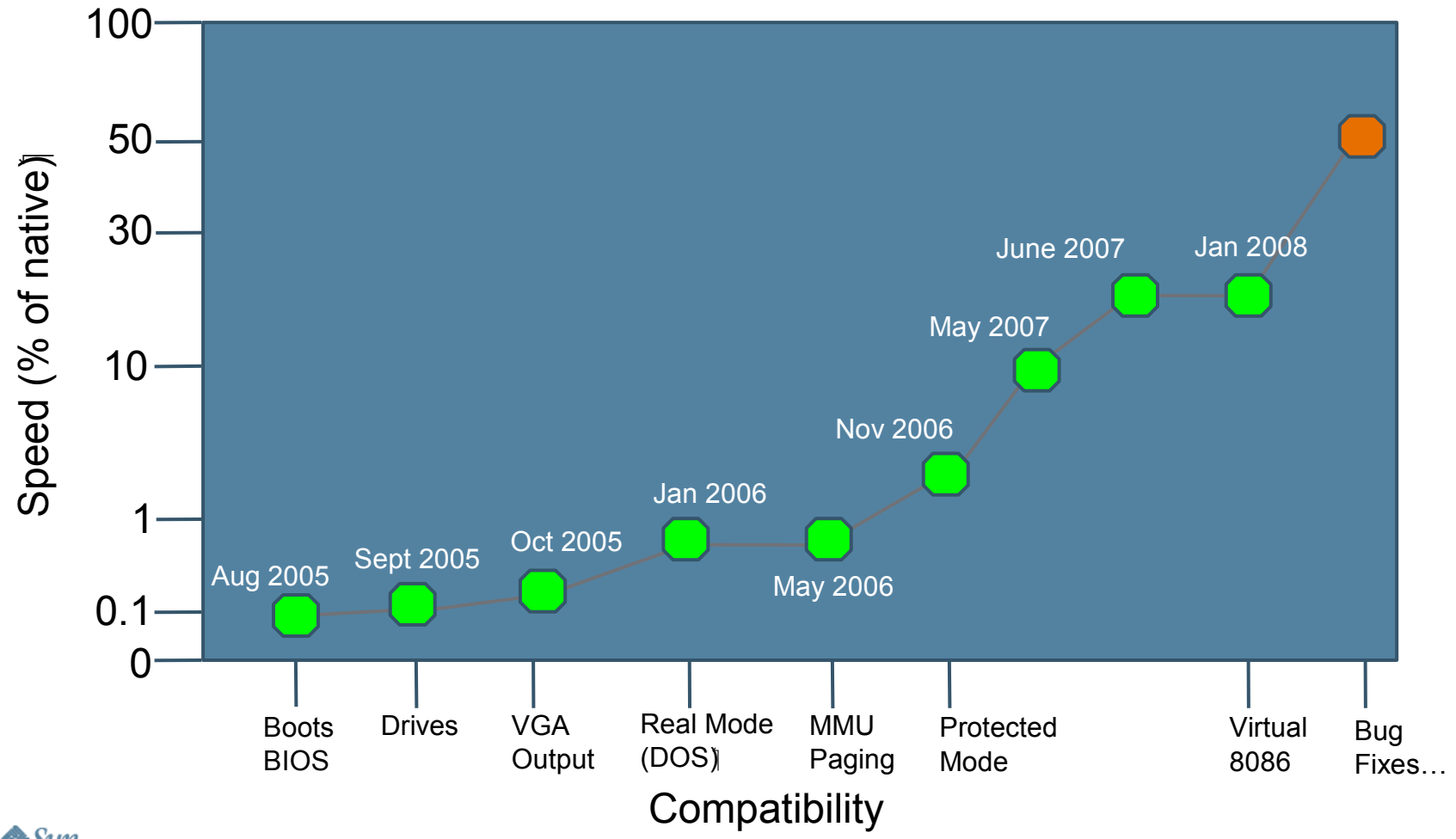
Compilers:
The Next Generation – Invocation Chaining

- Chain invocation of basic blocks by stitching together with appropriate control flow logic.
- Allows the JVM to gain knowledge of the higher level loop and condition logic in the translated code.



- JITed environments like Java Hotspot™ performance engine then have far more scope for optimisation – SPEED!

Summary of JPC Progress

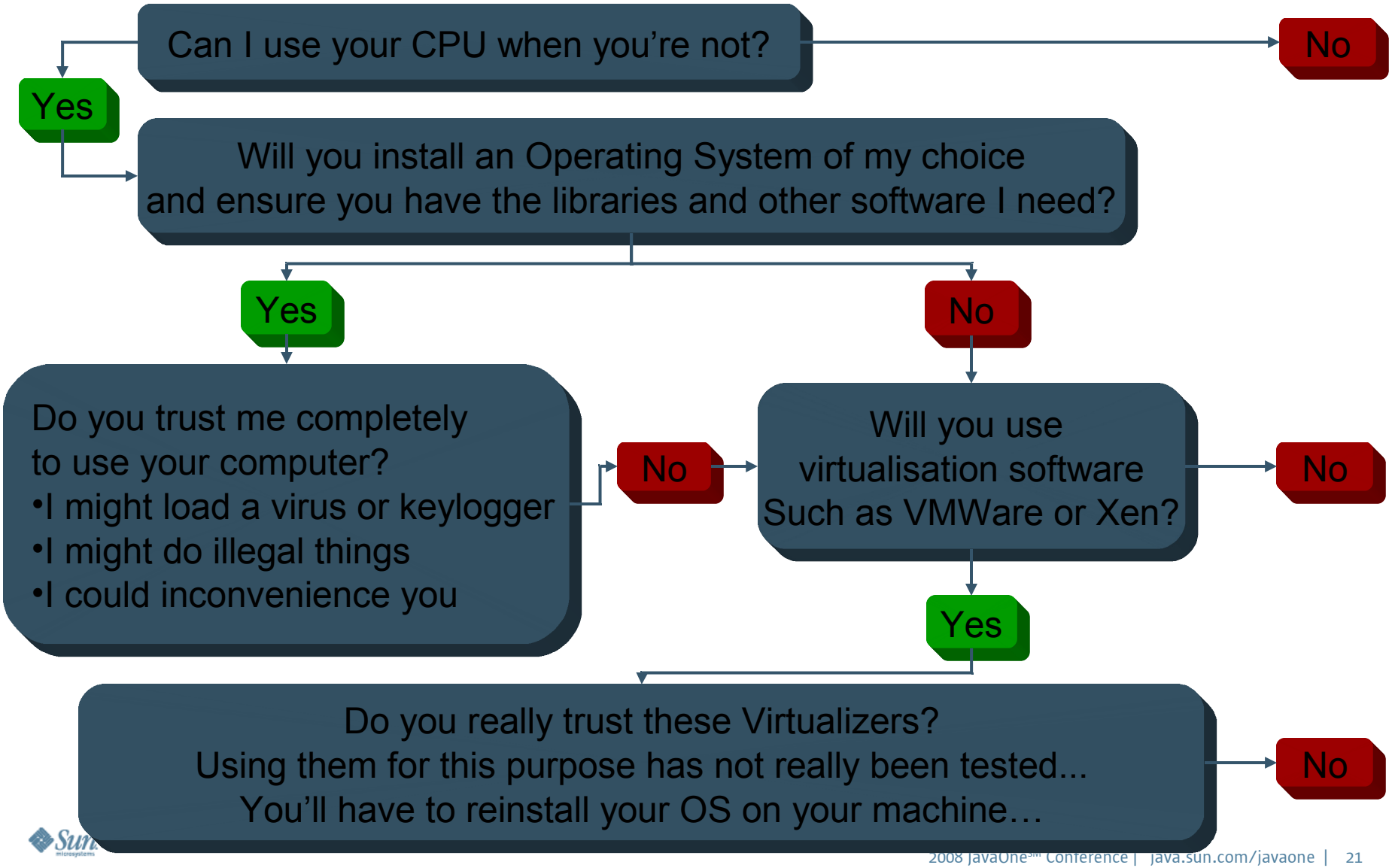


But what is JPC *really* for?

- If you need to isolate guest software from the host, use a virtualiser and get virtually no speed penalty
  - Virtualisation is well understood and can be done for free
  - Major vendors are increasingly supporting virtualisation – both a the hardware level (AMD Pacifica, Intel VT) and at the vendor level

- **But** what if you need so many machines you can't possibly buy them or find a datacentre big enough?
  - There are over 1 billion PCs in the world
  - If you could "borrow" these when idle you could get an unimaginably large amount of CPU power
  - There *are* projects which need this level of processing….

A Typical Conversation with a Donor….



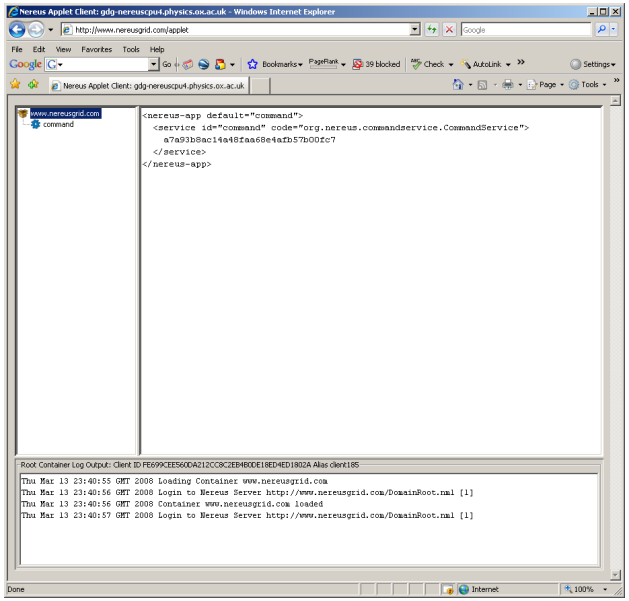
Security for Massively Parallel Computations

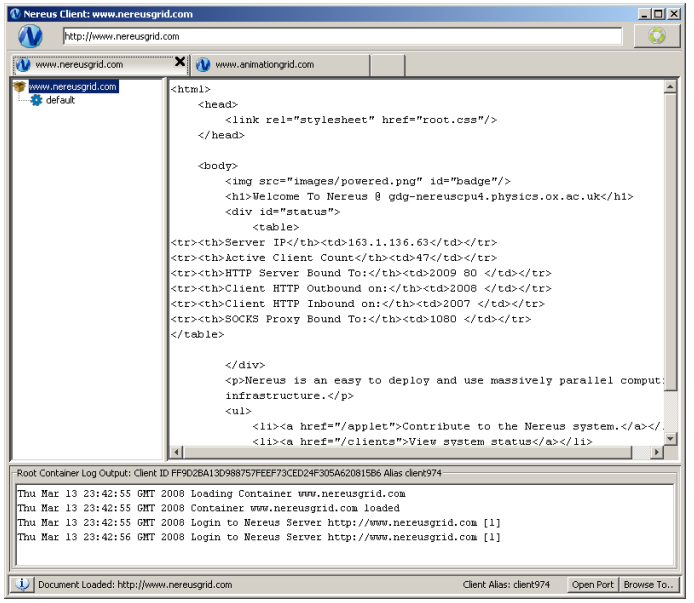
- To harness all these computers you need *absolutely bulletproof* security
  - You can't vet the code before it executes to ensure it isn't stealing private data or doing other naughty things
- You need to stay inside a Sandbox, just like the Applet sandbox
  - has run unvetted code on people's machines for over 10 years
- So you need JPC and a distribution management system for Java technology-based code…

Enter Nereus…..

What is Nereus?

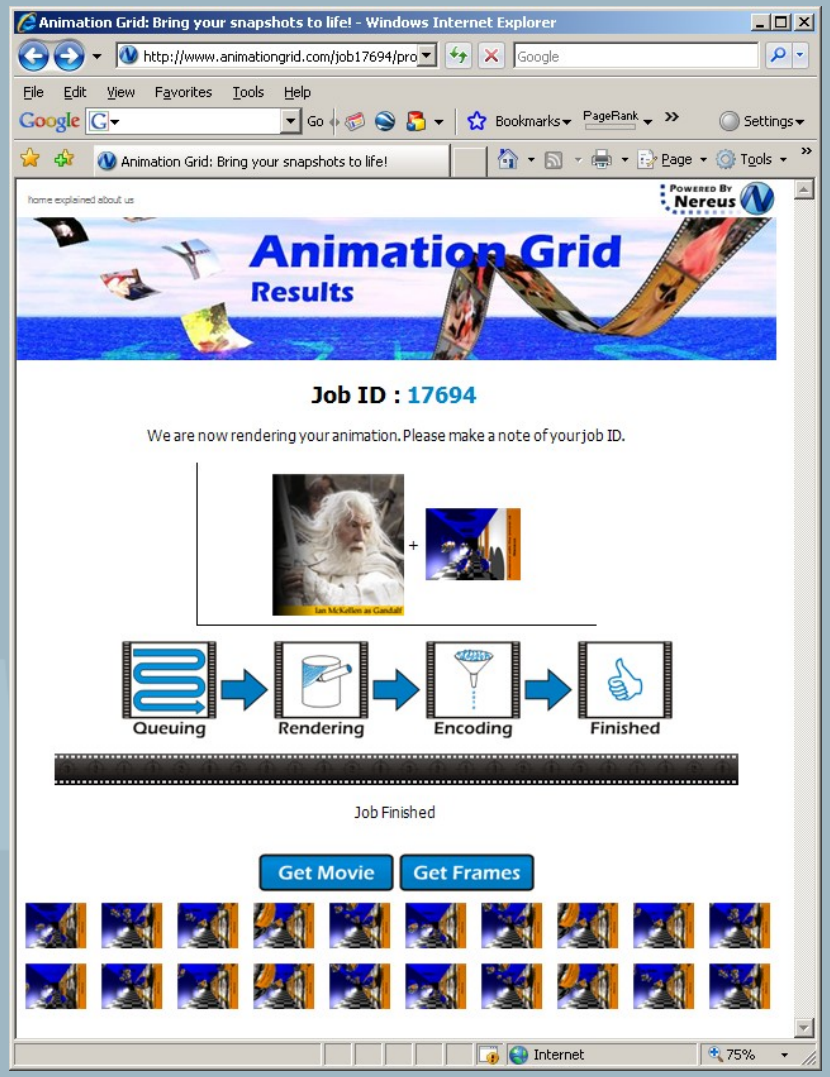
- Nereus is an open source pure Java technology infrastructure for MPC
  - Already in use for research, tested on >500 nodes
- To contribute to a Nereus installation either
  - Browse to the installation's web page and click on a link – you'll get the Nereus Applet
  - Download the Nereus Client Application and browse to the installation's root (a normal URL)



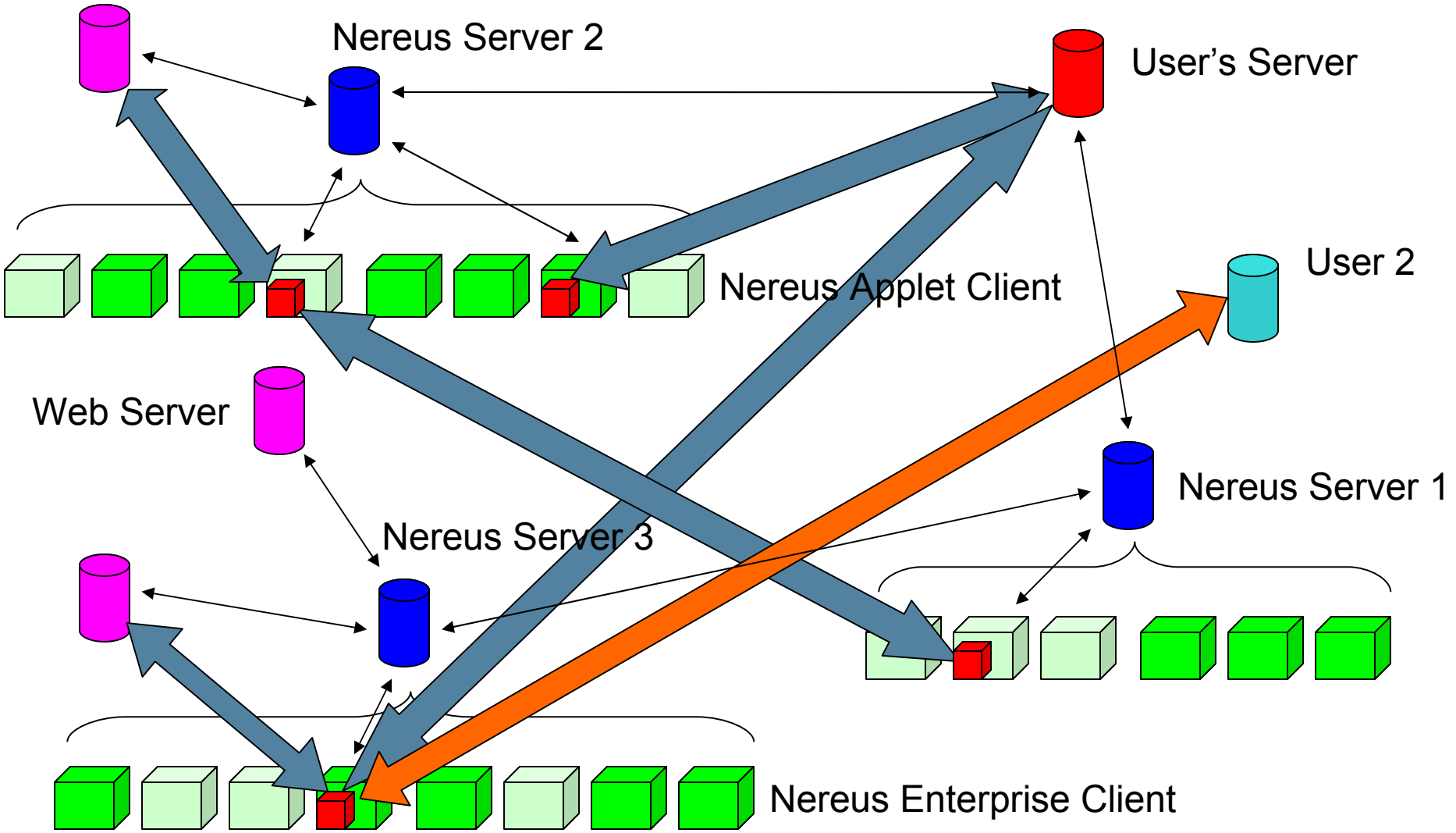


Demo: AnimationGrid

Using “Art of Illusion” Java technology-based renderer to render separate frames on different clients



Nereus System Design:

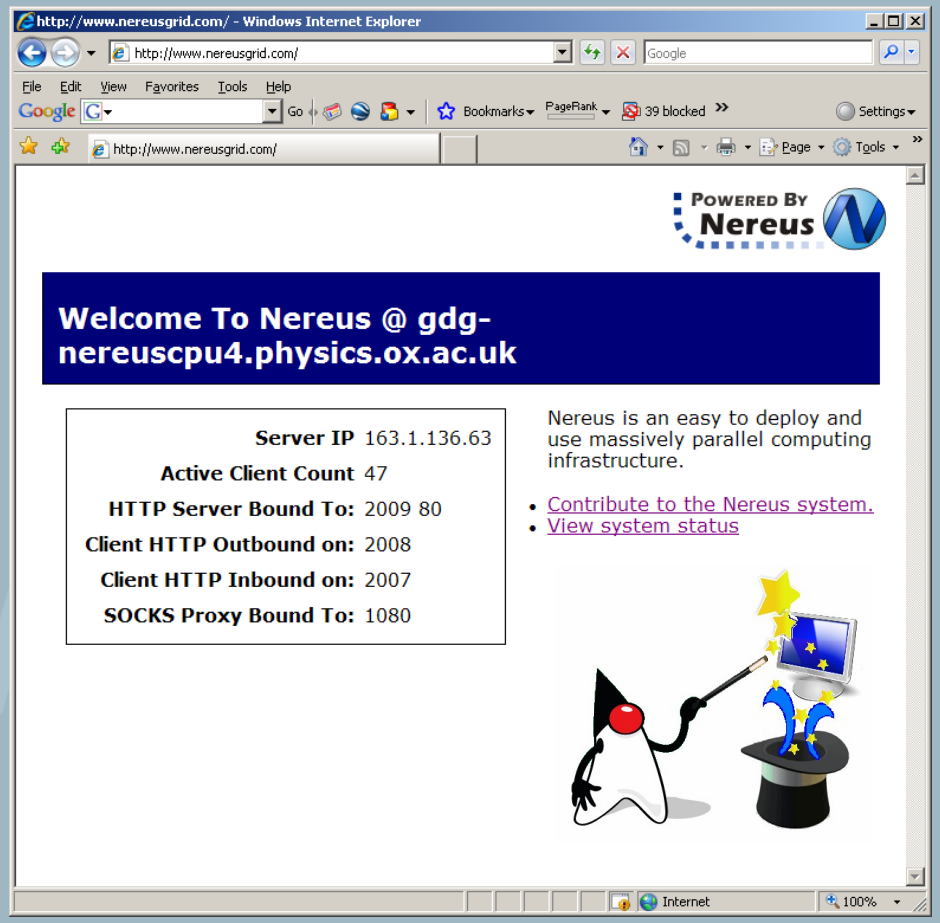


Bringing it all together…

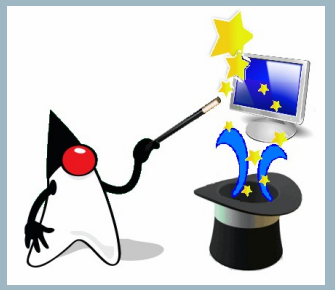
- Use Nereus to manage the distributed resources
- Use JPC to enable arbitrary x86 code (include your own OS) to execute within a *double insulated* sandbox
- Together you get millions of machines able to run your processing
  - On demand
  - Without prior approval processes/vetting procedures
  - Very cheaply – you're renting an existing computer
  - Environmentally friendly – the host machine would probably be on anyway and you're saving energy by not buying more machines in huge datacentres to do your processing!

Demo:
JPC on Nereus

How to remotely instantiate JPC instances on Nereus resources – and run native x86 software in an ad-hoc rented machine!



Demo:
“Can I have a Volunteer?”

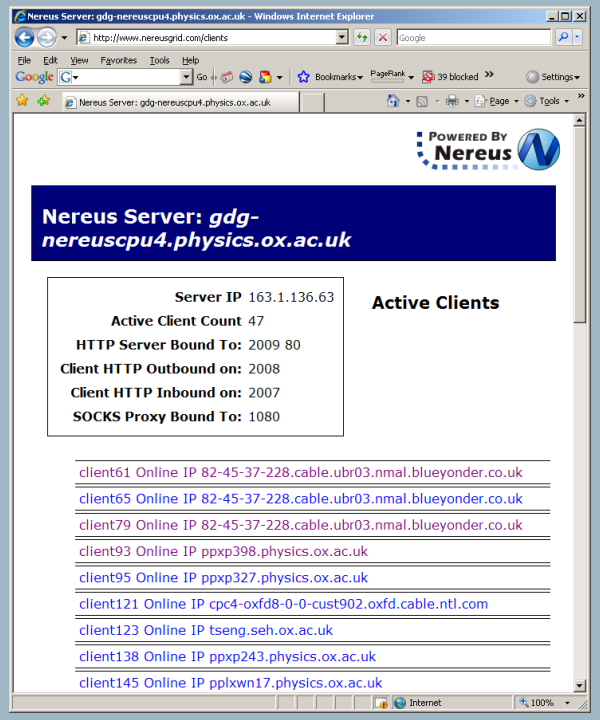


Please open your laptops and browse to http://www.nereusgrid.com/

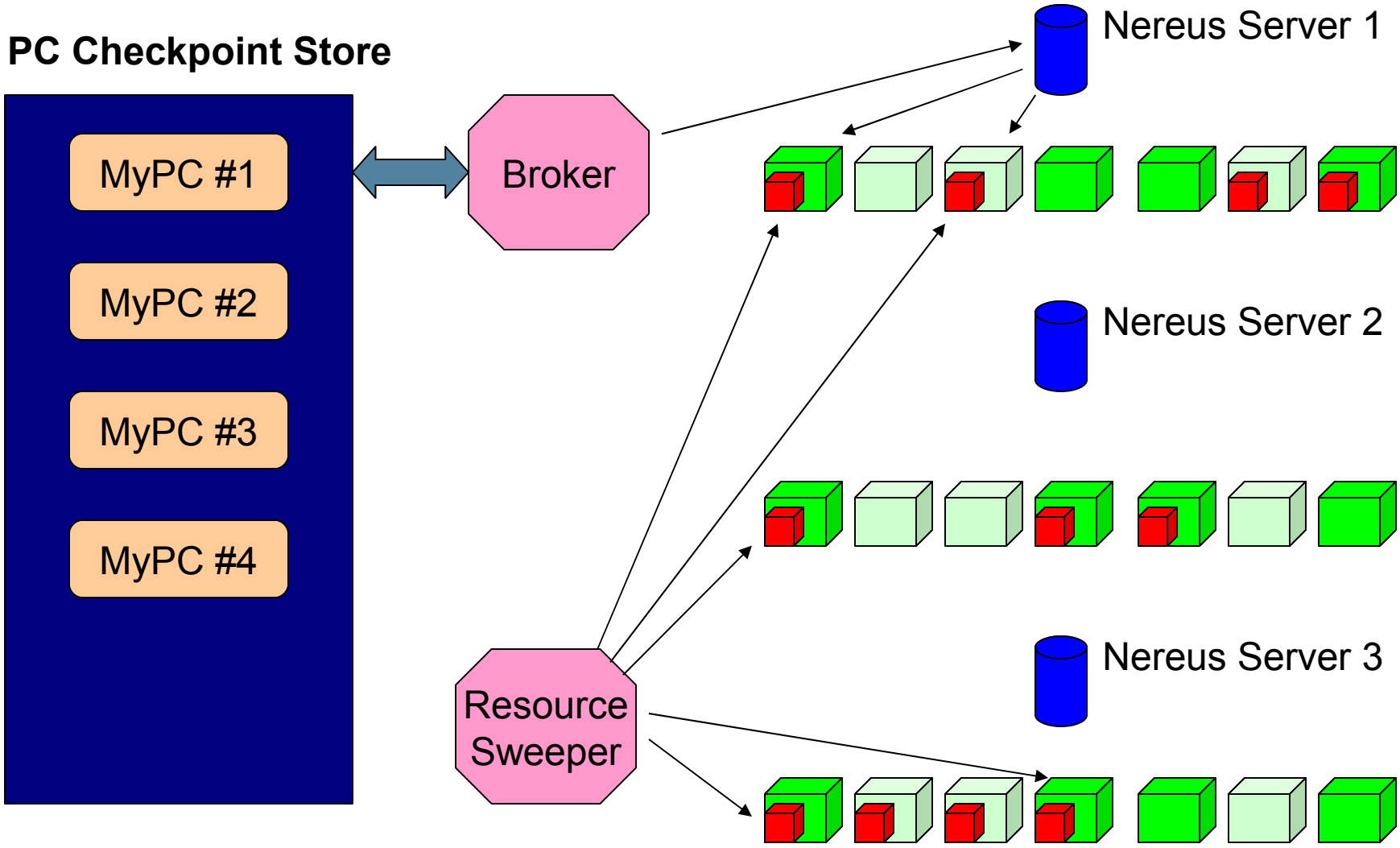
Note your “clientID”

We will now start JPC on your machine (remotely from the podium), and boot Linux inside your JPC/Nereus applet.

We’re in control, of our “machine”, and you retain control of yours!



Global MPC Infrastructure



Conclusions

- With over 1 Billion PCs in the world most of the time, there is a vast pool of CPU power going to waste every day
- JPC, combined with Nereus, offers a potential means to release this power by addressing the critical issue of security
- For MPC, where millions machines are needed, there is no practical alternative to this type of technology
- JPC and Nereus offer this opportunity for the Java platform
- With the next level of compilers JPC should be able to reach 50% native speed – well past the tipping point for this global potential
- Download JPC/Nereus and help us out! All our technology is open source GPLv2

http://www-jpc.physics.ox.ac.uk

http://www-nereus.physics.ox.ac.uk

THANK YOU

Rhys Newman
Dept Physics, Oxford University and GSA Capital
Chris Dennis
Department of Physics Oxford University

TS-5180