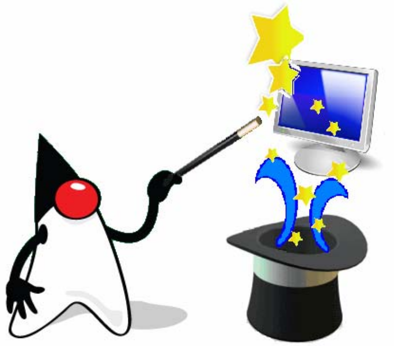


"Everything Java™": JPC, a Fast x86 PC Emulator

Rhys Newman and Chris Dennis

Dept of Physics, Oxford University
http://www-jpc.physics.ox.ac.uk

TS-13820

JPC: Towards a Native Speed x86 PC Emulator in a Pure Java™ Environment

Architectural decisions and implementation tricks needed to get a fast x86 PC emulation within a pure Java Virtual Machine (JVM™).

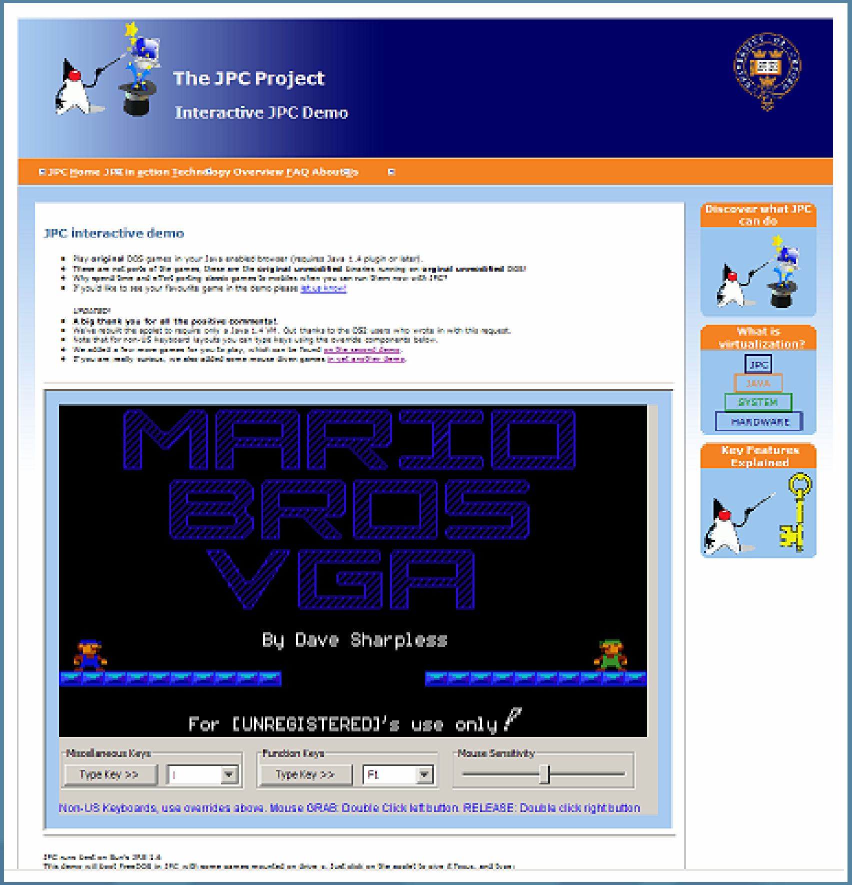
The terms “Java Virtual Machine” and “JVM” mean a Virtual Machine for the Java™ platform.

A Demonstration to Get Started…

- JPC emulates the hardware of an x86 PC using pure Java technology
- You only need a Java 2 platform
  - Currently Java platform v.1.4 compatible (could be back-ported even more if required)
  - No native code, no Java Native Interface (JNI™)
- You can use JPC to execute any x86 code inside a JVM implementation, including operating systems

DEMO

JPC running classic DOS games within a Java technology-based applet in a web page


The JPC Project

Interactive JPC Demo

JPC Home JPC in action Technology Overview FAQ About Us

JPC interactive demo

- Play original DOS games in your Java enabled browser (requires Java 1.4 plugin or later).
- **These are not ports of the games, these are the original unmodified binaries running on original unmodified DOS!**
- Why spend time and effort porting classic games to mobiles when you can run them now with JPC?
- If you'd like to see your favourite game in the demo please let us know!

UPDATED!

- **A big thank you for all the positive comments!**
- We've rebuilt the applet to require only a Java 1.4 VM. Our thanks to the OSX users who wrote in with this request.
- Note that for non-US keyboard layouts you can type keys using the override components below.
- We added a few more games for you to play, which can be found on the second demo.
- If you are really curious, we also added some mouse driven games in yet another demo.

MARIO BROS VGA

By Dave Sharpless

For [UNREGISTERED]'s use only

Miscellaneous Keys: Type Key >> | Function Keys: Type Key >> F1 | Mouse Sensitivity

Non-US Keyboards, use overrides above. Mouse GRAB: Double Click left button. RELEASE: Double click right button

[illegible]

JPC Project Goals

- Bring the power of virtualization into the realm of Java technology
- Project goal **(B)**
  - Get Linux/Windows booting and running standard software without unacceptable speed penalty
- Project status **(A)**
  - Runs DOS and some Linux at up to 10% real time speed
- This talk outlines how we've got to **A**, and also how we plan to get to **B**

Agenda

Why Did We Start JPC?

The Journey So Far

What Can You Do With JPC Now?

Future Developments

“Everything Java”?

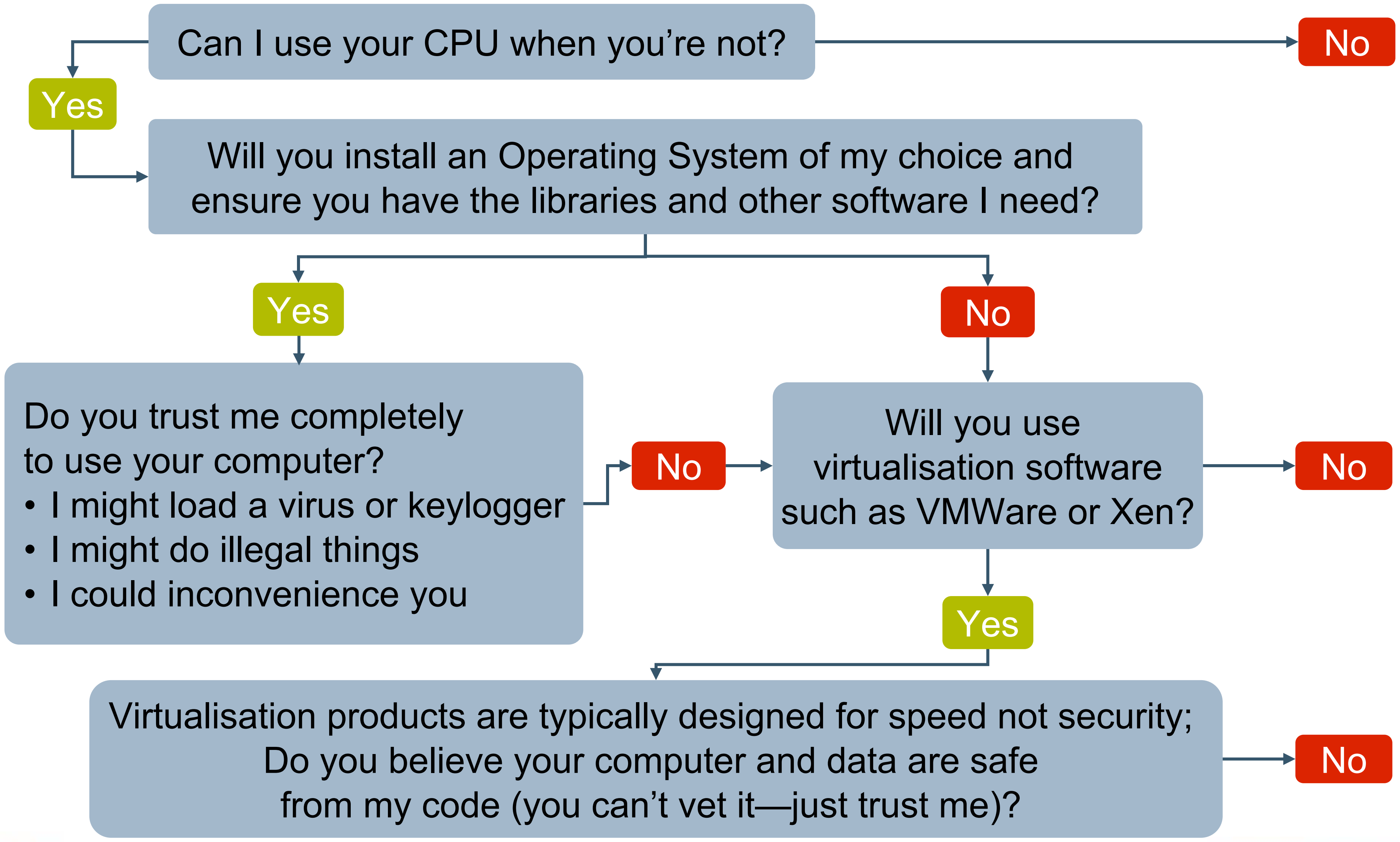
Agenda

- Why Did We Start JPC?
  - Why the Need?
  - What Could Be the Benefits?
- The Journey So Far
- What Can You Do With JPC Now?
- Future Developments
- “Everything Java”?

Why JPC?

- JPC has been developed in the department of Physics at Oxford University—Why?
- Particle Physics needs **loads** of computer time…
  - LHC alone needs 100,000 1GHz PCs
- Too expensive to buy—Use the idle time of existing machines
  - We have developed Nereus, a Java middleware technology offering bytecode-only execution on idle computers
- **But** current physics analysis software:
  - Is Linux based
  - Assumes complete access to machine (root access)
  - Is not commercial/production quality

A Sequence Diagram With a Potential CPU Donor



Can I use your CPU when you're not?
No
Yes
Will you install an Operating System of my choice and ensure you have the libraries and other software I need?
Yes
No
Do you trust me completely to use your computer?
• I might load a virus or keylogger
• I might do illegal things
• I could inconvenience you
No
Will you use virtualisation software such as VMWare or Xen?
No
Yes
Virtualisation products are typically designed for speed not security; Do you believe your computer and data are safe from my code (you can't vet it—just trust me)?
No

We Need a Virtual x86 PC With Unparalleled Security…

- We need a virtual x86 PC in which **anything** can run and **could not possibly escape** into the real machine
  - Security of equal strength on all operating systems
  - Cannot test for possible security holes/instabilities caused by interactions with other installed software
  - Must have a system immune to our own programmatic errors, e.g., buffer overruns, pointers left looking at “free” memory
- So build an emulator using Java technology!
  - The JVM software insulates us from testing on all platforms, OS patches, and other software combinations
  - Inherent security of the JVM software (array bounds checks, inbuilt memory manager, etc.) provides an independent security layer
  - Users can use their own trusted JVM implementation and then use our pure Java code to run the potentially dangerous x86 code

Added Benefits of JPC

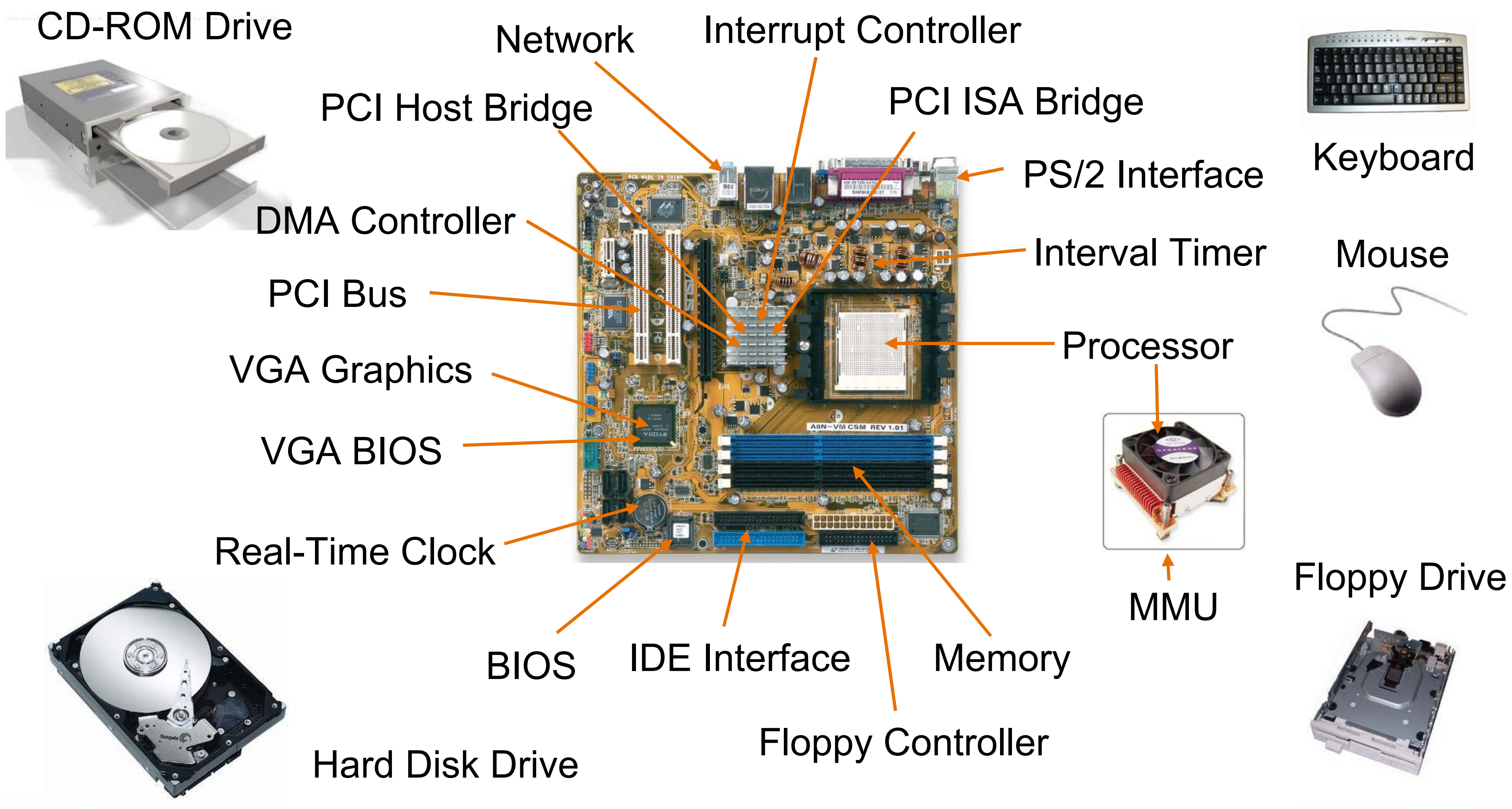
- With a pure Java technology emulator more than just x86 hardware would be available for use
  - Still no need to alter the software
- If JPC gets close to native speed then there is no longer a lock-in to specific hardware…
  - All your software, including favourite OS, can move to the best solution $/Ghz or Ghz/Watt
  - Barriers to adopting new technology are much lower as the required hardware is now virtual
    - Disk Image in a data centre, CPU/Screen elsewhere
    - Disk and CPU at the office, Screen on your mobile phone
- **Potential benefits of JPC go far beyond those of a secure x86 execution environment for sharing idle resources**

Size of the Task

Complexity of building an x86 Emulator

- x86 CPU Instruction Set is large and complex
  - Opcodes lengths vary from 1 to 15 bytes
  - Up to 4 prefixes which modify the action of basic instructions
  - 6 memory segments, 3 operand sizes, 4½ system operating modes
  - 30 years of legacy hardware design still supported
- Graphics Card Implementation
  - VGA/CGA/EGA, varying resolutions, and palliated modes
  - Over 10 pixel packing formats for a basic card, plus text modes
- IDE Devices
  - Reading ISO images for floppy, HDD, and CD-ROM
  - Viewing a directory on the underlying system as a disk drive for simple input/output to/from virtual computer
- Memory Management Unit
  - Complex paging mechanism with permissions (read/write/user/supervisor)
- Debugging All of the Above!
  - Standard tools of limited use

The Hardware Inside an x86 PC:



Major Issue—SPEED!

- Typically emulators sacrifice speed for simplicity
  - For legacy hardware, modern hardware is so fast this is often not a concern
- Java technology often criticised as being slow compared to C/C++
  - Was once true, but modern JVM implementations don’t have this problem (e.g., Java HotSpot™ VM)
- So, just how fast could an emulator in Java technology be?

Agenda

- Why Did We Start JPC?
- **The Journey So Far**
  - **Why We’re Not Insane**
  - **First Prototype (0.1% Speed)**
  - **Tricks to Get Where We Are Now**
- What Can You Do With JPC Now?
- Future Developments
- “Everything Java”?

The “Toy” CPU test

- Our “Toy” processor has 13 instructions, 2 registers, and 128 bytes of RAM
- A test program:

```
for (int i=0; i<10; i++)
    for (int j=0; j<50; j++)
        memory[51+j] += 4;
```

- Native code compiled with GCC; 100,000 runs of this on a P4 3Ghz (1GB RAM)
  - With no optimisation flags takes **360**ms
  - Turning GCC optimisations on reduces this to **86**ms

Toy Emulator #1

- Use a member variable to hold a 128 byte array to represent the memory
- Use member variables to hold the IP, and registers
- Execute each instruction by reading the byte from “memory” where the IP is pointing, decoding the action in a switch and continue…
- Times:
  - **7250**ms 1.6 Windows Client VM
  - **8030**ms 1.6 Windows Server VM

```
while (true)
{
    switch (memory[ip])
    {
        case LOAD_SP:
           sp = memory[ip+1];
           ip += 2; break;
        case LOAD:
           eax = memory[sp];
           ip++; break;
        case LOAD_EAX:
           eax = memory[ip+1];
           ip += 2; break;
        case ADD:
            eax += ebx;
            ip++; break;
        case MUL:
            eax *= ebx;
            ip++; break;
        case STORE:
            memory[sp] = eax;
            ip++; break;
        case JMP:
            ip = memory[ip+1]; break;
        case BNE:
            if (eax != ebx)
                ip = arg;
            else
                ip++;
            break;
            ....................
```

Toy Emulator #2

- Eliminate the “lookup dispatch” loop by writing the algorithm out in toy assembly code
- Effectively emulating a toy JIT
- Times:
  - **7250 → 1340**ms (Client)
  - **8030 → 710**ms (Server)

```
sp = 49;
ip += 2;
eax = 0;
ip += 2;
memory[sp] = eax;
ip++;

while (true)
{
    sp = 50;
    ip += 2;
    eax = 1;
    ip += 2;
    memory[sp] = eax;
    ip++;

    while (true)
    {
        eax = memory[sp];
        ip++;

        ............
```

Toy Emulator #3

- Remove the interleaved ip++ instructions (may permit better reordering)
- Increment the ip only on exit from the "function" and where the IP value is used
- Times:
  - **1340 → 690**ms (Client)
  - **710 → 250**ms (Server)

```
sp = 49;
eax = 0;
memory[sp] = eax;

while (true)
{
    sp = 50;
    eax = 1;
    memory[sp] = eax;

    while (true)
    {
        eax = memory[sp];

        ............
```

Toy Emulator #4

- Use local variables to hold register values
- Use a local reference to hold the byte[] reference to memory
- Times:
  - **690 → 220**ms (Client)
  - **250 → 330**ms (Server)

```
byte eax, ebx, sp, ip;
eax = ebx = sp = ip = 0;
byte[] memory = new byte[128];

sp = 49;
eax = 0;
memory[sp] = eax;

while (true)
{
    sp = 50;
    eax = 1;
    memory[sp] = eax;

    while (true)
    {
        eax = memory[sp];

        ............
```

Toy Emulator #5

- Assume aggressive optimisations to recognise the loops in the toy assembly code
- Also eliminate unnecessary movement of data into/out of registers

```
byte eax, ebx, sp, ip;
eax = ebx = sp = ip = 0;
byte[] memory = new byte[128];

for (int i=0; i<10; i++)
{
    memory[50] = 1;
    sp = 50;

    for (int j=0; j<50; j++)
        memory[50+j] += 4;
}
```

- Times **(Native Code: 360 or 86ms)**
  - **7250 → 1340 → 690 → 220 → 170**ms (Client)
  - **8030 → 710 → 250 → 330 → 80**ms (Server)

Conclusions From the Toy Tests

- Not having a "lookup-dispatch" loop gives a factor of ~ 10 speed improvement
- Not incrementing the instruction pointer after every instruction yields ~ 2x
- Using local variables in the VM rather than member variables of classes: ~ 3x (at least in the client VM!)
- Using other tricks to eliminate unnecessary intermediate data movements, etc. can still gain ~ 3x improvement
- **If all of the above are applied, the emulator is at least as fast as the native C compiled code**
- **If the native code is not compiled with optimisations, then the emulator can be faster**

The First JPC Prototype

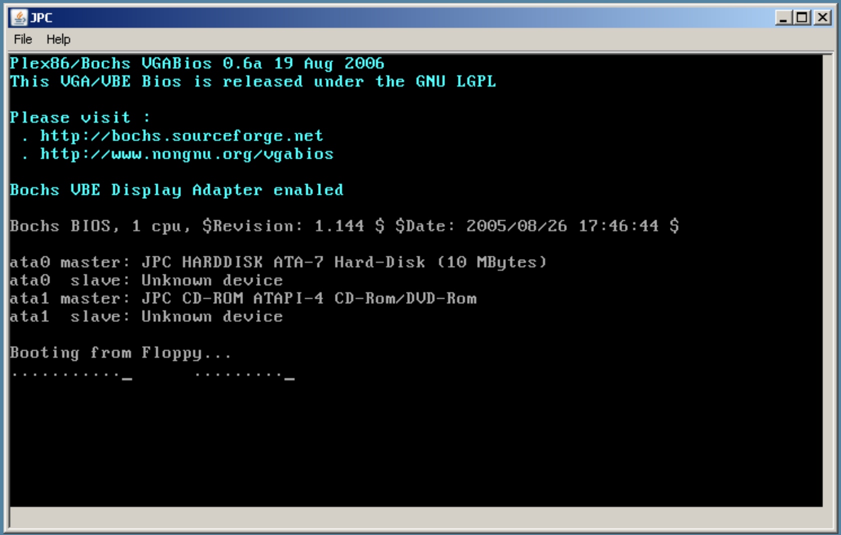
- All major hardware components implemented but not bug free…
- VGA screen updates consume 50% of total time
- Overall performance 0.1% native speed
- Equivalent to toy #1:
  - Lookup dispatch—No compilation
  - Follows OO design principles
    - Object created for each x86 instruction
    - x86 decoder needs resetting between decodes
    - Loads of object churn (simpler to implement and debug)
  - IP incremented after each instruction

The x86 PC Memory Is 4GB Long!

- To simulate a 4GB address space
  - Have a 1MB index array each potentially holding a reference to a 4Kb byte array representing a page
    - The x86 can have 4kb, 2Mb, or 4Mb page sizes, so use the lowest common denominator
  - The index entries are all initialised to null—Don't check rather catch the NullPointerException and allocate pages lazily
- Expensive in memory (4MB used right away) but quick

JPC Version 1

A version which runs *really* slow but almost boots DOS



```
Plex86/Bochs VGABios 0.6a 19 Aug 2006
This VGA/VBE Bios is released under the GNU LGPL

Please visit :
 . http://bochs.sourceforge.net
 . http://www.nongnu.org/vgabios

Bochs VBE Display Adapter enabled

Bochs BIOS, 1 cpu, $Revision: 1.144 $ $Date: 2005/08/26 17:46:44 $

ata0 master: JPC HARDDISK ATA-7 Hard-Disk (10 MBytes)
ata0  slave: Unknown device
ata1 master: JPC CD-ROM ATAPI-4 CD-Rom/DVD-Rom
ata1  slave: Unknown device

Booting from Floppy...
..........._      ........._
```

Towards JPC Version 2

- Instructions occur in blocks (Basic blocks) which contain no branches or jumps and so can be decoded together
- Each decoded instruction is executed 10,000 times on average (measured using JPC)
  - Decode once and cache for later
  - Can spend a **lot** of time optimising blocks as the benefit shows
- Instruction pointer incremented only at the end of a CodeBlock

Codeblock Caching

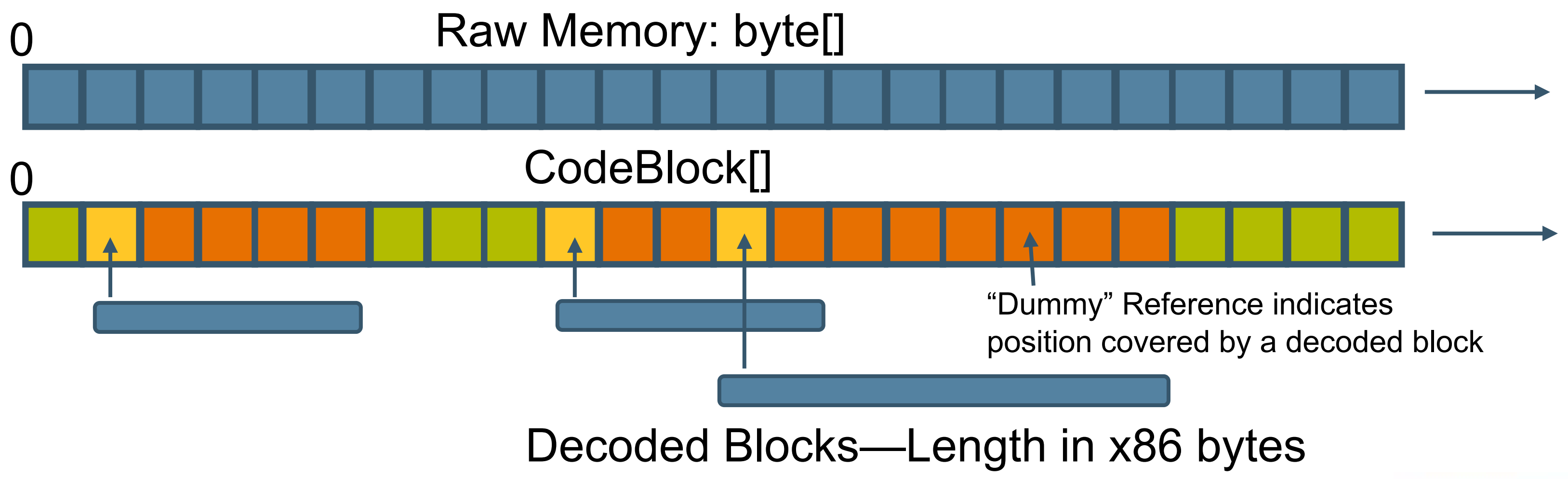
- The main program loop in JPC is very simple:

```
while (true)
{
    int ip = processor.getEIP();
        CodeBlock block = memory.getBlockAt(ip);
    block.execute(processor);
}
```

- The memory object is asked for a block with an address argument
  - Thus memory should cache the blocks using address as a key
  - Minimal cost for lookup, and also for failed lookup
- Use shadow arrays of CodeBlocks which contain references to previously decoded blocks at a particular address
- Cached CodeBlocks contain raw x86 instructions “decoded” into RISC-like JPC instructions
  - Still “lookup-dispatch” but on a smaller set
  - RISC instructions allow combinations to be optimised
  - Can eliminate unnecessary flag calculations (simply delete them)

Codeblock Memory

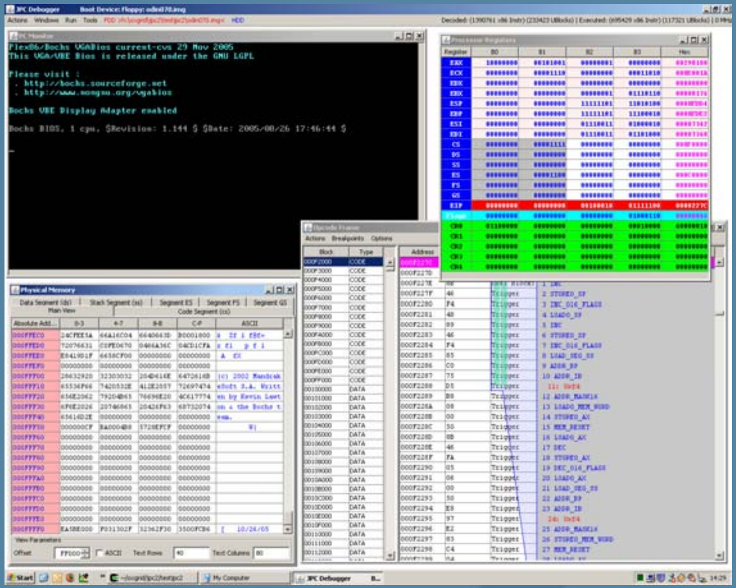
- Need to detect when the raw memory which contributed to a codeblock has been changed and invalidate the codeblock (forces redecode)
- Self-modifying code does happen
  - An original x86 allows this anywhere
  - Modern CPUs do not encourage this!
- Codeblock caching makes this work with minimal overhead:


0
Raw Memory: byte[]
0
CodeBlock[]
"Dummy" Reference indicates position covered by a decoded block
Decoded Blocks—Length in x86 bytes

JPC Debugger

A graphical tool built to expose the inner workings of the x86 PC

Again, pure Java technology and fully integrated with JPC



```
Plex86/Bochs VGABios current-cvs 29 Nov 2005
This VGA/VBE Bios is released under the GNU LGPL

Please visit :
 . http://bochs.sourceforge.net
 . http://www.nongnu.org/vgabios

Bochs VBE Display Adapter enabled

Bochs BIOS, 1 cpu, $Revision: 1.144 $ $Date: 2005/08/26 17:46:44 $
```

JPC Version 2

A version which boots DOS, plays games about 10x faster than version 1

Towards Version 3

- Eliminate lookup-dispatch by compiling the codeblocks into class files
    - Toy tests show this should be a 10x improvement
- Use a custom ClassLoader to load tailor made classes into the Java Virtual Machine at runtime—But:
    - How fast is compilation?
    - How good is the compiler?
    - How fast can these be loaded?
    - Which classes to compile—Try to detect the "hot" spots?

Towards Version 3

- Java Development Kit (JDK™) 1.6 includes a Java Compiler tool to compile source code at runtime
  - Initial JPC compiler used this—“Source Compiler”
  - Developed our own bytecode compiler
  - Optimised this to reduce object churn

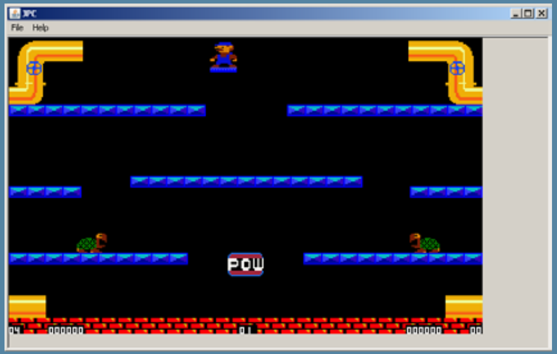
Codeblock Length (x86 instructions)	Java Source Compiler	Bytecode Compiler	Lean Bytecode Compiler
2	44.4	2.8	**2.8**
10	50.5	3.9	**3.6**
100	111.5	14.2	**9.8**

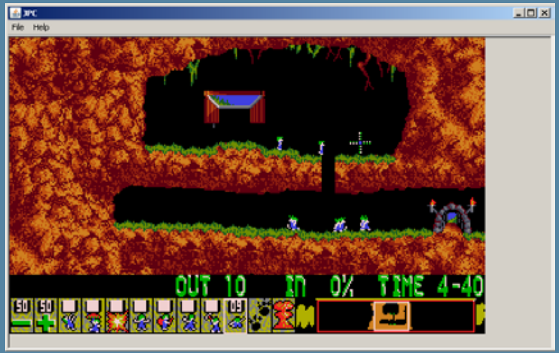
Time in Milliseconds

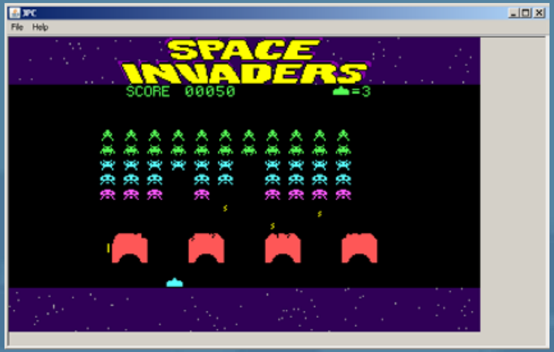
JPC Version 3

A version which boots DOS, plays games about 5–10x faster than version 2

Uses bytecode compiler







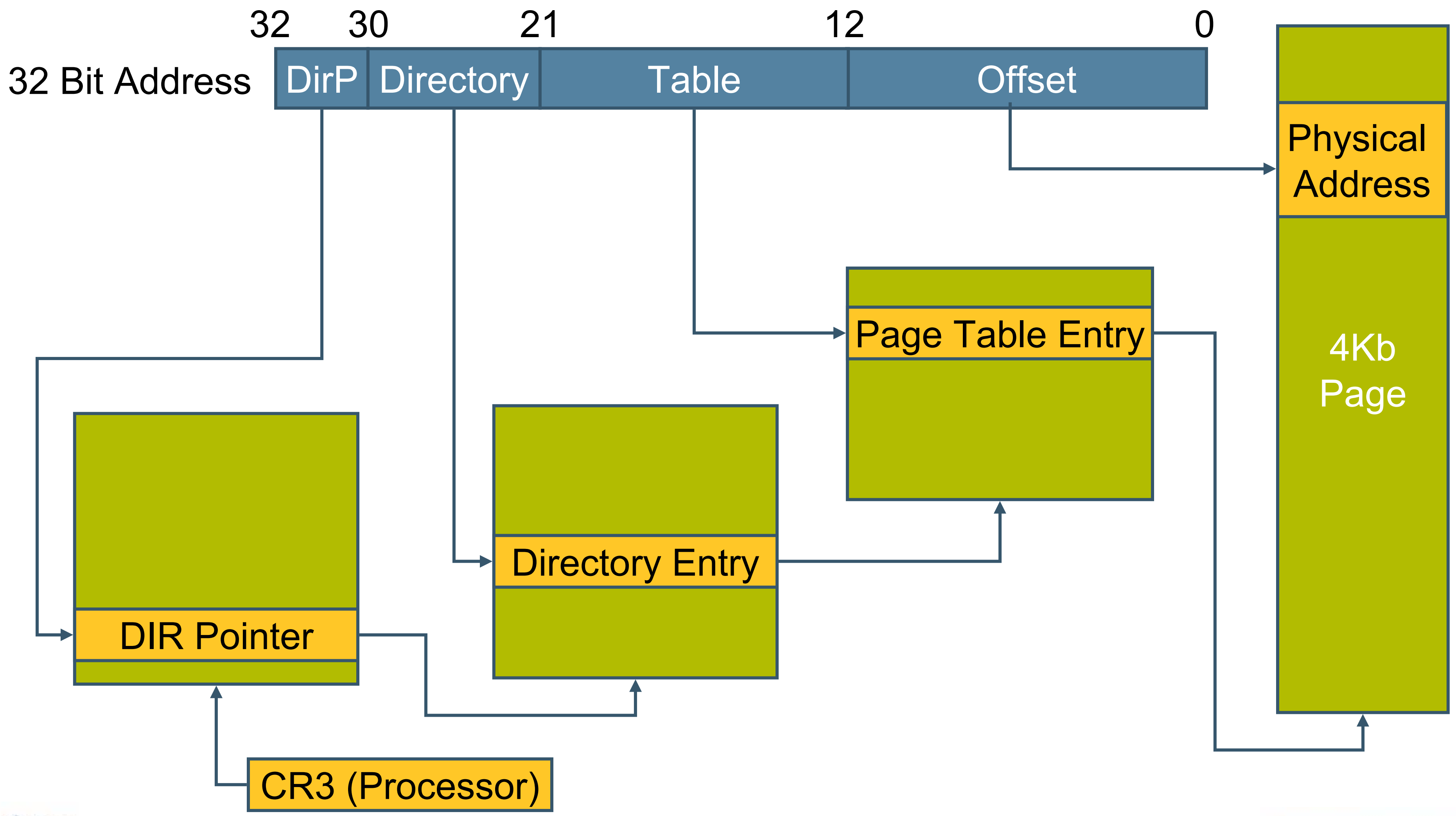
Towards Version 4

- A number of compromises taken in version 3 to increase speed at the expense of memory
  - Need a limited memory solution
  - Need a fast solution where the compiler is unavailable
    - Java 2 Platform, Micro Edition (J2ME™ platform)
- Need to move beyond real mode (DOS) to implement the complex memory model provided by Protected Mode (needed for Linux and Windows)

Protected Mode Memory Model

- In protected mode each memory page has read/write permissions for each process enforced by the Memory Management Unit (MMU)
  - If the OS chooses, it can use the paging system to give each process a 4GB address with complete isolation from other processes
  - Both Windows and Linux need protected mode
- Achieved through special control registers of the processor (CR0 and CR3)

Linear Address Translation

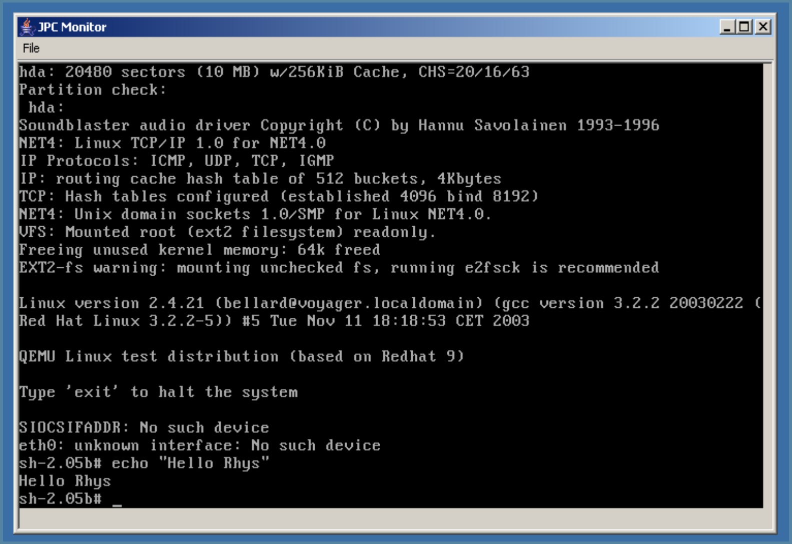


Performance Implications

- In theory all memory access is via this 4 stage process
  - The directory and page entries also have permission flags (R/W/U/S) which need to be obeyed
- If JPC emulated this…
- Even real processors cache the translation of this lookup process in translation look-aside buffers
  - JPC uses one array for each of 4 possible modes: RU, RS, WU, WS
  - Select the mode and look up the memory at an address
    - A null entry means a full lookup needs to be done
    - If access is permitted then the memory is simply returned from the array
    - Illegal modes cache "memory" pages which throw exceptions when accessed
  - Cache is flushed when certain key instructions are executed (INVLPG, setting CR0 or CR3)
- There is minimal overall cost with this strategy, so JPC's protected mode memory access is practically the same speed as real mode

JPC Version 4

A version which boots 32-bit Linux Protected Mode—Not yet using the compiler



```
hda: 20480 sectors (10 MB) w/256KiB Cache, CHS=20/16/63
Partition check:
 hda:
Soundblaster audio driver Copyright (C) by Hannu Savolainen 1993-1996
NET4: Linux TCP/IP 1.0 for NET4.0
IP Protocols: ICMP, UDP, TCP, IGMP
IP: routing cache hash table of 512 buckets, 4Kbytes
TCP: Hash tables configured (established 4096 bind 8192)
NET4: Unix domain sockets 1.0/SMP for Linux NET4.0.
VFS: Mounted root (ext2 filesystem) readonly.
Freeing unused kernel memory: 64k freed
EXT2-fs warning: mounting unchecked fs, running e2fsck is recommended

Linux version 2.4.21 (bellard@voyager.localdomain) (gcc version 3.2.2 20030222 (
Red Hat Linux 3.2.2-5)) #5 Tue Nov 11 18:18:53 CET 2003

QEMU Linux test distribution (based on Redhat 9)

Type 'exit' to halt the system

SIOCSIFADDR: No such device
eth0: unknown interface: No such device
sh-2.05b# echo "Hello Rhys"
Hello Rhys
sh-2.05b# _
```

Agenda

Why Did We Start JPC?

The Journey So Far

What Can You Do With JPC Now?

- **Current Applications—Even at Current Speed**

Future Developments

“Everything Java”?

Current Applications

- Abandonware
  - Running DOS games has proven to be very popular!
- Deploying DOS software in a flexible (browser) way
  - Request for Novell client in JPC
  - DOS Point-Of-Sale and other older systems—Work fine, but modern systems increasingly have issues running them
    - C: not always accessible
    - Shouldn't play around with autoexec.bat, setup.ini…
    - 64 bit windows does not support 16-bit applications
- Archiving Legacy Software and Data
  - Libraries who wish to keep multimedia CDs and other software in original form (and usable like that forever)
- Quarantine
  - Even the nastiest viruses are safe inside JPC, and they could be studied in great detail using the JPC debugger

Agenda

Why Did We Start JPC?

The Journey So Far

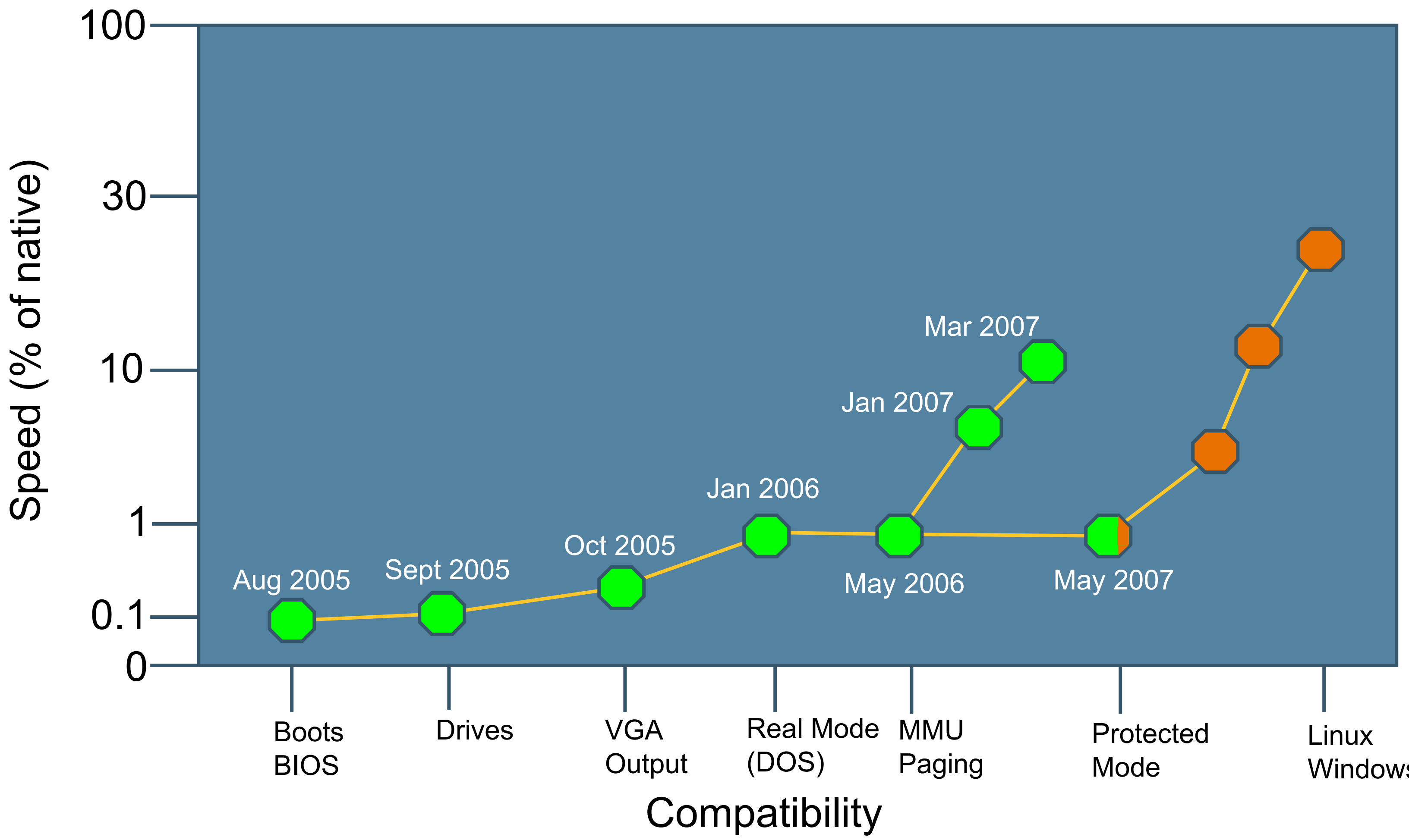
What Can You Do With JPC Now?

Future Developments

- **Developments Still to Do**
- **Where to Get Extra Speed—How to Build on Our Current Knowledge to get Towards Full Speed**

“Everything Java”?

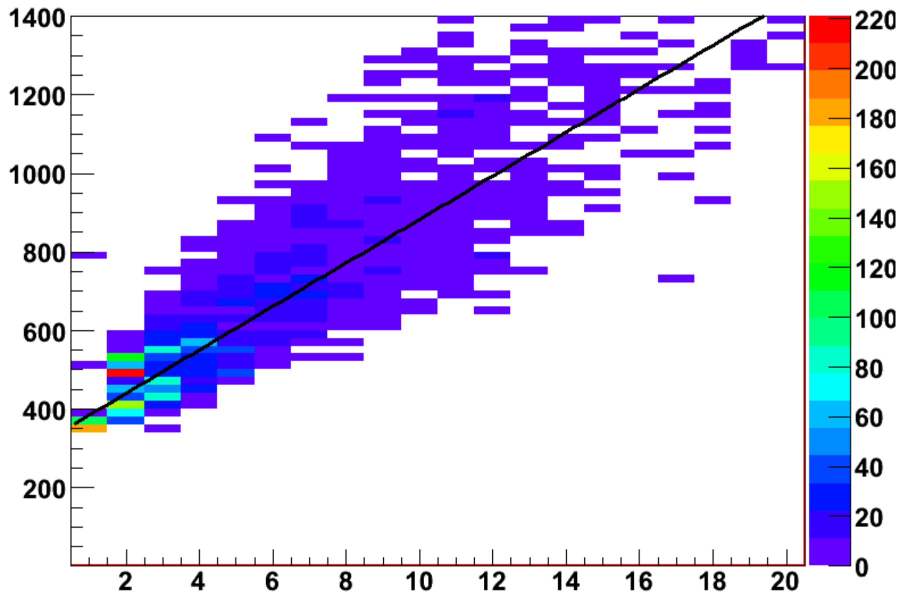
Summary of JPC Progress



Compiler Improvements

- The compiler currently generates very inefficient code
  - A library of bytecode fragments for each x86 opcode
  - Compiler simply appends them together
- Initial tests show a better version will be beneficial
  - Not actually obvious given Java HotSpot virtual machine is present trying to spot its own optimizations

Compiled Method Length (bytecodes)

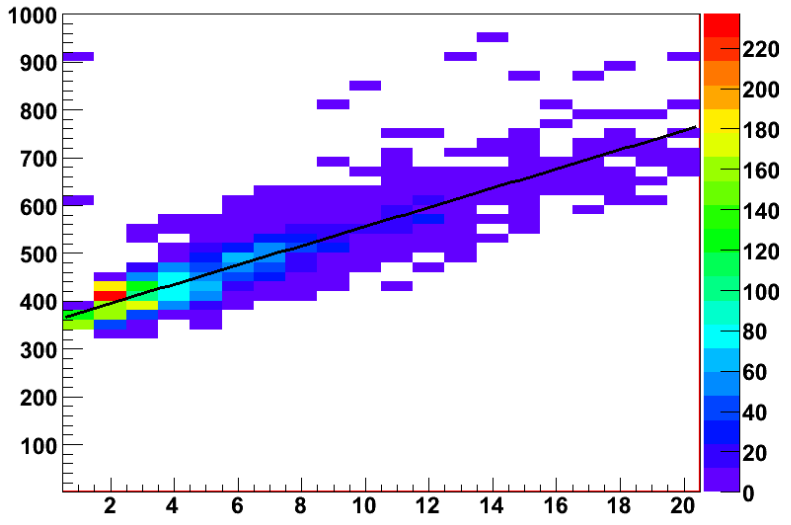


Emulated Block Size (x86 Opcodes)

Compiler Improvements

- SUN released Java HotSpot VM source code as part of the open source Java technology initiative last November
  - We now have a debug build of Java HotSpot VM which can reveal what actual x86 code it produces for any of our codeblock classes
  - For every emulated x86, Java HotSpot VM produces ~20
  - Before even 1 emulated instruction, Java HotSpot VM produces ~350
  - **Plenty of scope here for improvements:**
    - **Static-Single-Assignment form**
    - **Peephole optimisations**
    - **Dead code elimination**

Actual x86 Instructions



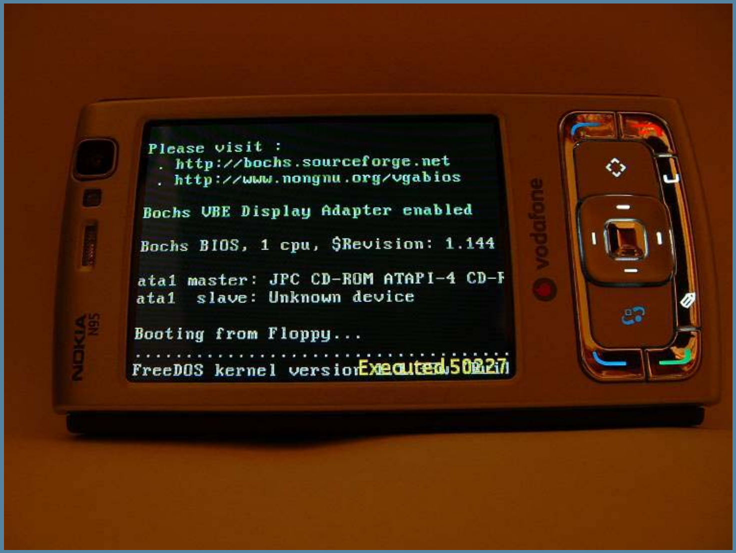
Code Block Length (x86 Opcodes)

Agenda

- Why Did We Start JPC?
- The Journey So Far
- What Can You Do With JPC Now?
- Future Developments
- **“Everything Java”?**
  - **Implications of This Technology**

Mobile JPC

JPC Running DOS
on a Mobile Phone

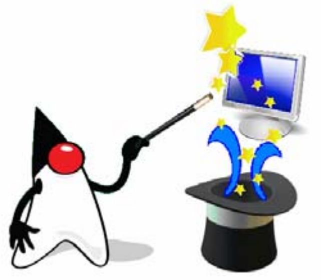


```
Please visit :
 . http://bochs.sourceforge.net
 . http://www.nongnu.org/vgabios

Bochs VBE Display Adapter enabled

Bochs BIOS, 1 cpu, $Revision: 1.144

ata1 master: JPC CD-ROM ATAPI-4 CD-R
ata1  slave: Unknown device

Booting from Floppy...
.......................................
FreeDOS kernel versioExecuted 50227
```

Where Could JPC Go?

- At 10% Real mode we have niche applications
- At 10% Protected mode we can run Linux and possibly Windows
- At 30% Protected mode we pass economic break-even for CPU rental
  - Costs the same to rent an idle CPU for 3 hours than to suffer the depreciation cost of a dedicated machine for 1 hour
  - Point at which the original intention (a computer grid) becomes compelling
- We have Nereus, pure Java grid middleware ready to harness the idle time just for this purpose
  - See www-nereus.physics.ox.ac.uk
- 50% is realistic: real-time compilation, profiling and intelligent caching for fast code execution has come a long way and is well understood
  - The JVM software has improved its execution speed by a factor of 10 over 10 years (now 95% or more compared to native speed)
  - With this experience, and the Java HotSpot VM source code now available… who knows?

Conclusion

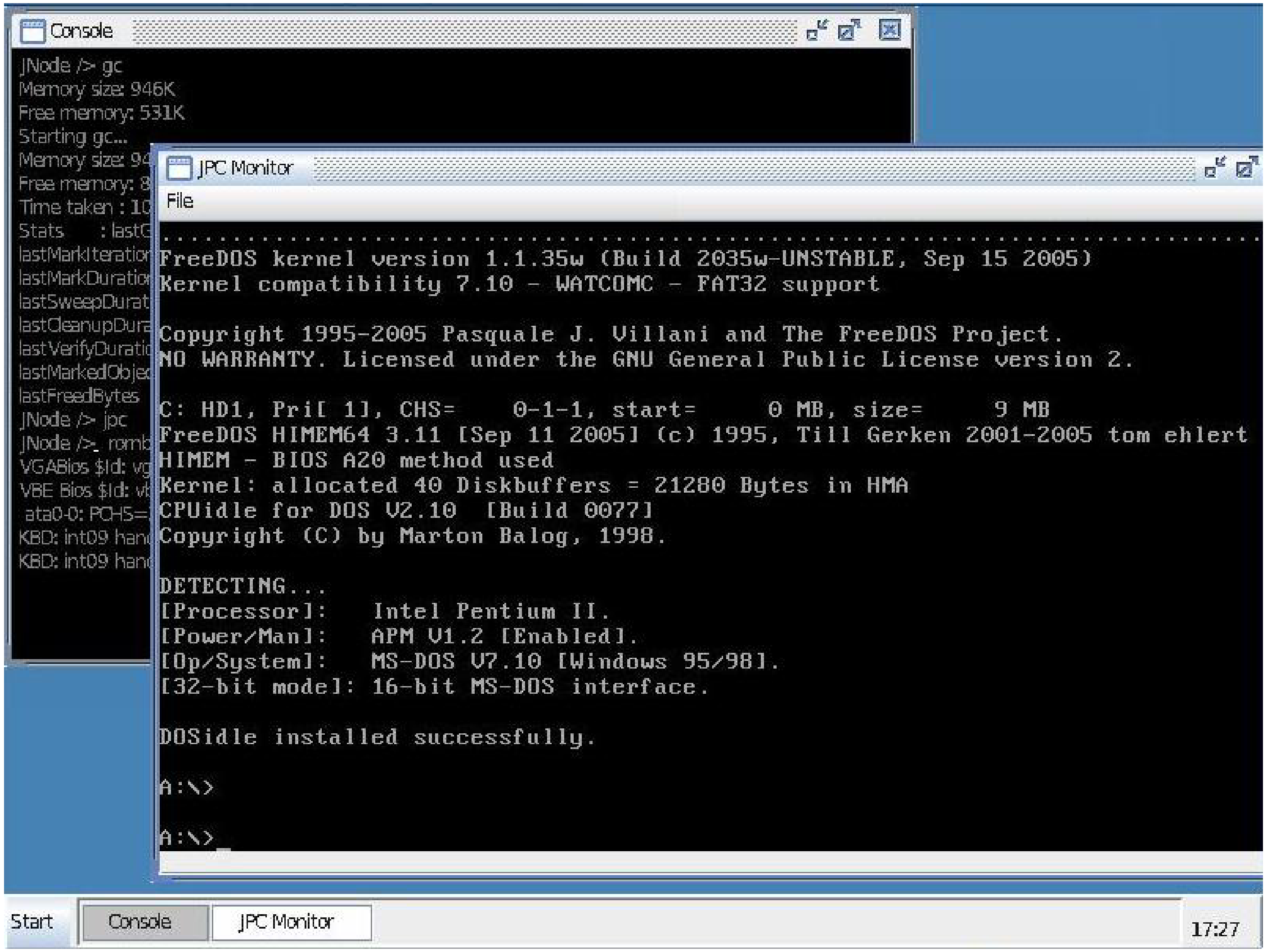


Although only recently developed, JPC is already one of the fastest x86 emulators available.

If we can continue on our current trajectory and realise some of the potential still locked away, we believe JPC could bring practical, secure, and flexible x86 execution to the Java platform.

Leveraging the security and hardware-independence of the Java Virtual Machine could enable JPC to revolutionize the way information technology is structured and delivered.

Final Thought…JPC on JNode



```
JNode /> gc
Memory size: 946K
Free memory: 531K
Starting gc...
Memory size: 94
Free memory: 8
Time taken : 10
Stats      : lastG
lastMarkIteration
lastMarkDuration
lastSweepDurat
lastCleanupDura
lastVerifyDuratio
lastMarkedObjec
lastFreedBytes
JNode /> jpc
JNode />_ romb
VGABios $Id: vg
VBE Bios $Id: vb
 ata0-0: PCHS=
KBD: int09 hand
KBD: int09 hand
```

```
FreeDOS kernel version 1.1.35w (Build 2035w-UNSTABLE, Sep 15 2005)
Kernel compatibility 7.10 - WATCOMC - FAT32 support

Copyright 1995-2005 Pasquale J. Villani and The FreeDOS Project.
NO WARRANTY. Licensed under the GNU General Public License version 2.

C: HD1, Pri[ 1], CHS=    0-1-1, start=     0 MB, size=     9 MB
FreeDOS HIMEM64 3.11 [Sep 11 2005] (c) 1995, Till Gerken 2001-2005 tom ehlert
HIMEM - BIOS A20 method used
Kernel: allocated 40 Diskbuffers = 21280 Bytes in HMA
CPUidle for DOS V2.10  [Build 0077]
Copyright (C) by Marton Balog, 1998.

DETECTING...
[Processor]:   Intel Pentium II.
[Power/Man]:   APM V1.2 [Enabled].
[Op/System]:   MS-DOS V7.10 [Windows 95/98].
[32-bit mode]: 16-bit MS-DOS interface.

DOSidle installed successfully.

A:\>

A:\>_
```

See: http://www.jnode.org/

STOP THE PRESSES!

This morning Oxford University granted permission to release JPC source code under GPL.

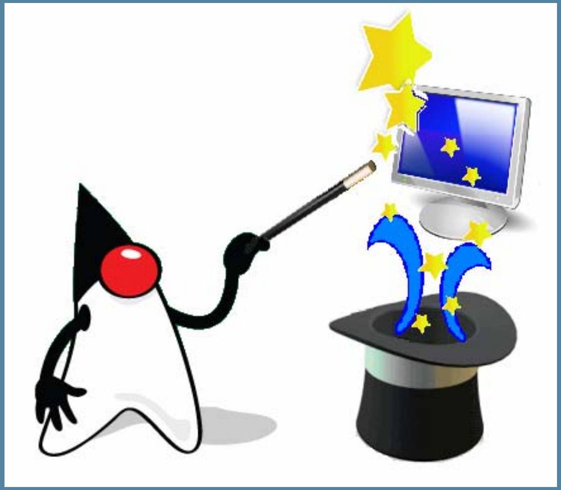
You can download this source code at:

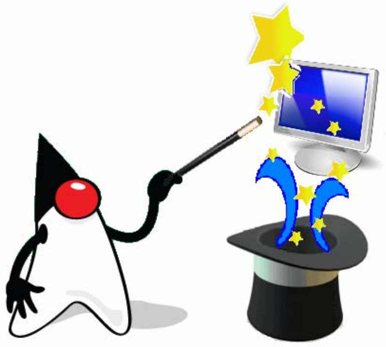
http://www-jpc.physics.ox.ac.uk

Q&A

Rhys Newman

Chris Dennis





“Everything Java™”: JPC, a Fast x86 PC Emulator

Rhys Newman and Chris Dennis

Dept of Physics, Oxford University
http://www-jpc.physics.ox.ac.uk

TS-13820