High Performance Java™ Technology

I'll share some of the performance tricks I used to implement a fast PlayStation emulator in the Java programming language, and talk about some cool Java technology stuff I got to mess with in the process

Agenda

Introduction

Enough Already, Let's See It!

Performance Tricks

Cool Stuff—Java HotSpot™ VM for R3000?

Q&A and Another Demo

Agenda

Introduction

Enough Already, Let's See It!

Performance Tricks

Cool Stuff—Java HotSpot VM for R3000?

Q&A and Another Demo

Technical Requirements

- Sony PlayStation Specs
  - 32 bit RISC CPU @ 33.9MHz
  - Geometry co-processor
    - 500k lighted triangles per sec
  - Graphics co-processor @ 33.9MHz
    - 360000 triangles per sec
    - Thousands of 2d sprites with rotation/scaling
    - Alpha transparency and Gouraud shading
    - Resolutions up to 640x512 at 30fps
  - Decompression co-processor for video
  - 24 channel sound @ 44kHz

Architecture Goals

- Object oriented
  - “Machine” should be assembled from loosely-coupled component classes representing:
    - Physical hardware
    - Processor instructions
    - Memory mapped code/data
    - Internal emulator components e.g. byte code generators
- Written entirely in the Java language
  - If it physically can be done entirely in the Java language, then do so
  - Implemented with clear maintainable code

How Did It Turn Out?

- Emulation machine is assembled from arbitrary components
- Except: address space and some execution flow internals
- Uses well known connection points
- Entirely Java technnology (you do need to use a Java Native Interface (JNI) based media component to run directly off CD)
- Code is clear(ish)

DEMO

The Emulator in Action!

Agenda

Introduction

Enough Already, Let's See It!

Performance Tricks

Cool Stuff—Java HotSpot VM for R3000?

Q&A and Another Demo

Handler Functions

```
// Instruction decoding in the interpreter:
//   Calling different handler functions
//   for different op-codes

// simplified instruction interface
interface Instruction {
   public void execute(int opCode);
}

class CPU {
   // simplified interpreter loop
   public void execute() {
      while (true) {
         int opCode = memory[ip++];
         instructions[opCode&0x3f].execute(opCode);
      }
   }
}
```

Handler Functions

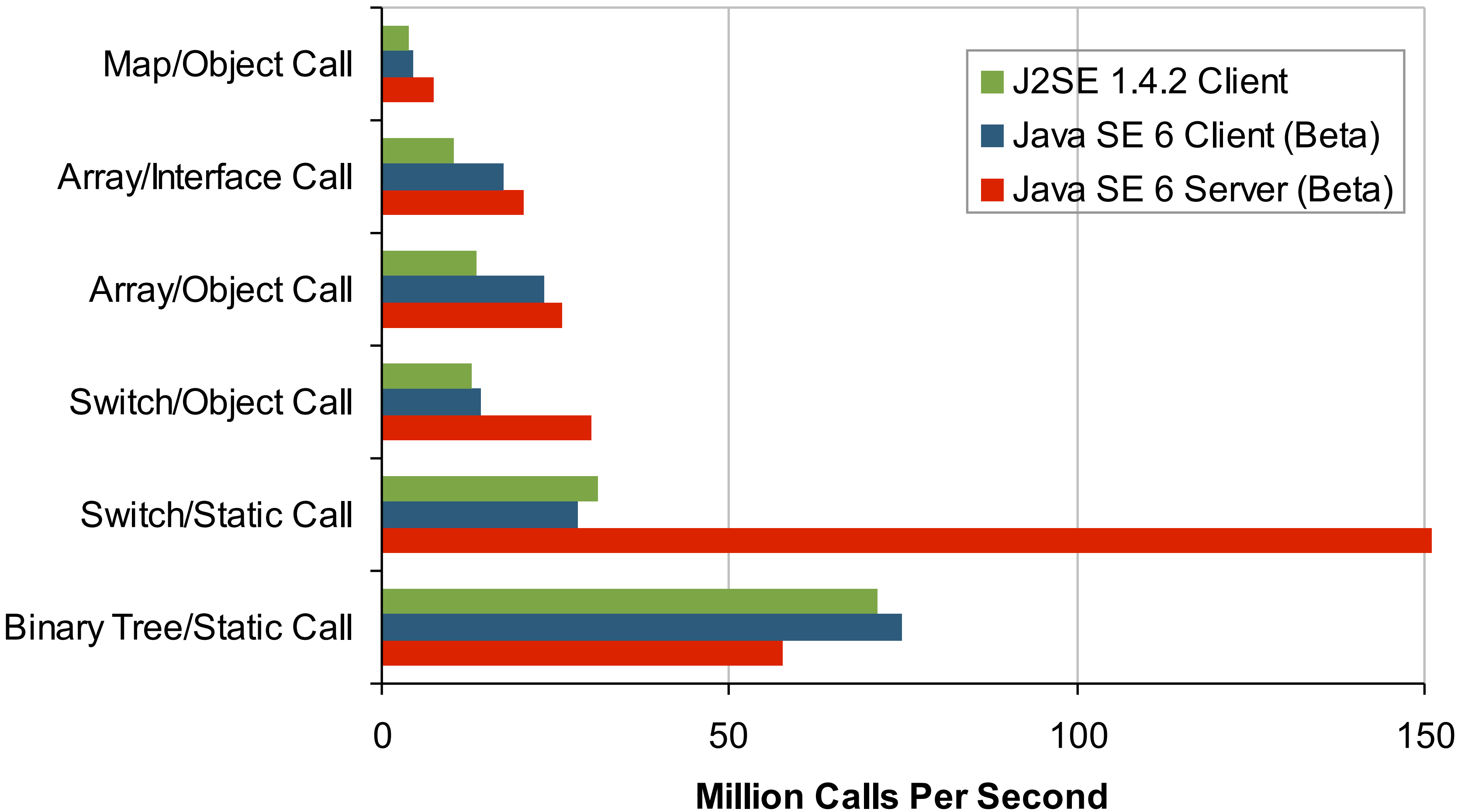
```
// Memory Mapped I/O
//   similar, but address range is big and sparsely filled

// Possible Solution 1: Map
Handler handler =
           (Handler)handlerMap.get(new Integer(address));
handler.write(data);

// Possible Solution 2: Switch with non static method
switch (address) {
   case 20:  handler20.write(data);  break;
   case 100: handler100.write(data); break;
}

// Possible Solution 3: Switch with static method
switch (address) {
   case 20:  Handler20.write(data);  break;
   case 100: Handler100.write(data); break;
}
```

Handler Functions Test



Source: Average of 5 runs after warm up on my Windows XP laptop

Handler Functions Summary

- Picked method for fastest execution
  - Binary Tree of “if” statements
  - Calls to a static member function of a particular implementation class
- Still needed run time configurability
  - Instructions/Handlers registered in Array/Map during start up
  - Utility class uses BCEL to build optimal method to dispatch calls

Costly IFs

```
// Generic class which handles 4 possible
// rendering combinations
public class Renderer {
    public void render( boolean alpha, boolean paletted) {
        // simplified pixel loop
        for_all_pixels {
            int color;
            if (!paletted)
                color = texture[src];
            else
                color = palette[texture[src]&0xf];
            if (alpha)
                color = alpha*color + (1-alpha)*background;
            screen[dest] = color;
        }
    }
}
```

Costly IFs

```
// Specialized class which handles just one combination
// no-alpha, no-palette
public class NoAlphaNoPaletteRenderer {
    public void render() {
        // simplified pixel loop
        for_all_pixels {
            screen[dest] = texture[src];
        }
    }
}
```

Costly IFs

```
// Using JAVAC to specialize the class for us

public class NoAlphaNoPaletteRenderer {
   public static final boolean alpha = false;
   public static final boolean paletted = false;

   public void render() {
      // simplified pixel loop
      for_all_pixels {
         int color;
         if (!paletted)
            color = texture[src];
         else
            color = palette[texture[src]&0xf];
         if (alpha)
            color = alpha*color + (1-alpha)*background;
         screen[dest] = color;
      }
   }
}
```

Costly IFs

```
// Another generic class which handles all 4 combinations

public class TemplateRenderer {
   public static final boolean alpha = isAlpha();
   public static final boolean paletted = isPalette();

   public void render() {
      // simplified pixel loop
      for_all_pixels {
         int color;
         if (!paletted)
            color = texture[src];
         else
            color = palette[texture[src]&0xf];
         if (alpha)
            color = alpha*color + (1-alpha)*background;
         screen[dest] = color;
      }
   }
}
```

Costly IFs

```
// as compiled by HotSpot if alpha and palette
// are set to false during static initialization
public class TemplateRenderer {
    public static final boolean alpha = getFalse();
    public static final boolean paletted = getFalse();

    public void render() {
        // simplified pixel loop
        for_all_pixels {
            int color;
            if (!paletted)
                color = texture[src];
            else
                color = palette[texture[src]&0xf];
            if (alpha)
                color = alpha*color + (1-alpha)*background;
            screen[dest] = color;
        }
    }
}
```

Costly IFs

- Using code generation
  - I clone and rename my generic class, and change the bytecode for the static member initializers
- Without using code generation
  - Set static final variables based on immutable configuration properties
  - Factory alternate implementations of the same class in separate class loaders
  - Or…

Costly IFs

```
// hybrid case!

public static final boolean xIsMutable = …;
public static final boolean xInitialValue = …;
boolean xValue = xInitialValue;

public boolean getX () {
   return xIsMutable ? xValue : xInitialValue;
}

if xIsMutable == true, simplifies to xValue;

if xIsMutable == false, simplifies to xInitialValue,
   and hence statically either true or false,

In either case it will likely be inlined
```

1ms Timer Resolution

- I want to…
  - Have a main CPU thread
  - Have a separate background thread for asynchronous hardware
- Which means I need to…
  - Measure time to 1ms accuracy
  - Add callbacks at arbitrary but accurate frequencies
- But how entirely in the Java language?
  - On some platforms System.currentTimeMillis() has poor resolution
  - And Thread.sleep()?

Poor Man's 1ms Resolution Timer

```
// TimeKeeper thread running at Thread.MAX_PRIORITY
while (true) {
    // repeated calls to Thread.sleep(1) actually
    // keep the delay accurate on Windows!
    Thread.sleep(1);
    synchronized (this) {
        // provide rough estimate of time (can be behind)
        time++;
        // schedule notification to event worker thread
        // running at Thread.NORM_PRIORITY+2
        if (time>nextScheduledEventTime) {
            notify();
        }
    }
}
```

Agenda

Introduction

Enough Already, Let’s See It!

Performance Tricks

Cool Stuff—Java HotSpot VM for R3000?

Q&A and Another Demo

Two-Stage Compiler

- Converts units of R3000 code into Java classes
- 1st stage
  - Used in preference to interpreter
  - Simple translation of code
  - Gathers data to help second stage
- 2nd stage
  - Used for “hot” methods
  - Does flow analysis and constant propagation
  - Uses information from the first stage for some key optimizations

What Is a Code Unit?

- Starts at the target address of any call, or any jump to a dynamic address
- Includes all instructions which can be determined to be reached; i.e. it stops at a branch to another dynamic address.
- It turns out that this is often a single C function

Stage 1 Unit Class

```
public class _1XXXXXXXX implements Executable {
   // holder of runtime state for this code unit
   public static CodeUnit unit;

   // static method to execute the unit
   public static int s(int retAddr, boolean jump){
      // 1) forward to stage 2 if this is hot
      if (unit.useStage2())
         return _2XXXXXXXX.s(retAddr, jump);
      // 2) simple state machine
      if (unit.count>0) unit.count--;
      else              unit.countComplete();
      // 3) implementation of R3000 code (omitted)
   }
}
```

Code Units Call Each-other Directly

```
0x80103020 addiu r2, r0, #4 ; load register 2 with 4
0x80103024 jal 0x80104088   ; call function at 80104088
                            ; saving return address in r31
0x80103028 nop              ; delay slot
0x8010302c …                ; next instruction

public class _180103020 implements Executable {
   public static int s(int retAddr, boolean jump) {
      // preceding code omitted
      Compiler.reg_2  = 4;
      Compiler.reg_31 = 0x8010302c;
      _180104088.s(0x8010302c,false);
      // following code omitted
   }
}
```

R3000 Calls Are Java Language Calls

```
public class _180103020 implements Executable {
    * @param retAddr the expected return address of
    *                the current R3000 frame
    * @param jump    true we're here by jump not call
    * @return        the next execution address
    public static int s(int retAddr, boolean jump) {
        // (omitted all but the last instruction)

        // code for "jr r31" (basically "return")
        int target = Compiler.reg_31;
        while (true) {
            if (target == retAddr || jump)
                return target;
            else
                target = Compiler.jump( target, retAddr);
        }
    }
}
```

Compiler Architecture

- CPU execution thread (normal priority)
  - Runs interpreter loop, calls into stage 1 classes for any JAL
  - ClassLoader does stage 1 compilation as necessary
  - Schedules for background stage 2 compilation any code units which have become “hot”
- Background compilation thread 1 (low priority)
  - Does stage 1 compilation
- Background compilation thread 2 (low priority)
  - Does stage 2 compilation
- Stage 1 compilation
  - Includes scheduling for background compilation any referenced (by JAL) but currently missing stage 1 classes

Stage 2 Unit Class

```
// 80131000 lui r2, 0x8001      ; load r2 with 0x80010000
// 80131004 lw  r2, r2[0x1234] ; load r2 from 0x80011234

public class _280131000 implements Executable {
   public static int s(int retAddr, boolean jump)
   {
      if (replaced) return _380131000(retAddr, jump);
      // (stage 1 version is)
      // Compiler.reg_2 = 0x80010000
      // Compiler.reg_2 =
      //            AddressSpace.read32(Compiler.reg_2);
      Compiler.reg_2 = AddressSpace.ram[0x11234/4];
      // remaining R3000 code omitted
   }
}
```

ArrayIndexOutOfBoundsException Is Our Friend!

```
// 80131004 lw  r2, r6[0]

public class _2XXXXXXXX implements Executable {
   public static int s(int retAddr, boolean jump)
   {
      if (replaced) return _3XXXXXXXX(retAddr, jump);
      // (stage 1 version is)
      // AddressSpace.tagRead(0x80131004,Compiler.reg_6);
      // Compiler.reg_2 =
      //       AddressSpace.read32(Compiler.reg_6);
      Compiler.reg_2 =
          AddressSpace.ram[(Compiler.reg_6&RAM_MASK)/4];
      // remaining R3000 code omitted
   }
}
```

Other Interesting Tidbits

- Oops—R3000 code in RAM is over-writable!
  - We have to throw away our class loader on instruction cache flush
- There are bugs in the R3000 code too!
- The compiler is just a component too—you can replace it if you like
- We detect and avoid busy-wait, so we have time for our background threads, and don't hog the CPU

Q&A and Another Demo

Summary

- Java technology is fast enough to run a PlayStation emulator on modern hardware
- Byte-code generation is cool, but you can do a bunch of stuff without it
- You don't have to sacrifice code maintainability
- I plan to open source, so people can start adding stuff (e.g., Java 3D™ API), SPU rewrite with latest Java Sound API, etc.