The Virtualized Virtual Machine:
The Next Generation of Virtual Machine Technology

David F Bacon
IBM T.J. Watson Research Center

November 5, 2003
The Virtual Machine

Virtual Machine Definition

Scheduler
  Lock Mgr  Object Model

Garbage Collector

Compiler
  Stack Layout
  Data Layout

Instruction Set Architecture

Object Model

Garbage Collector

Compiler

Instruction Set Architecture
The Virtualized Virtual Machine

Instruction Set Architecture

Virtual Machine Definition

Scheduler
- Lock Mgr
- Object Model
- Garbage Collector

Compiler
- Stack Layout
- Data Layout

Instruction Set Architecture
Virtualizing the Garbage Collector

Virtual Machine Definition

Lock Mgr  Object Model  Compiler

Scheduler

Garbage Collector

Instruction Set Architecture

Object Model

Data Layout

Stack Layout
Virtualizing the Garbage Collector

- Existing VMs support multiple collectors (static)
- Can make selection dynamic [Soman et al ’04]
  - Can switch during execution
  - Based on profiling
- Can generate new “algorithms” [Bacon et al ’04]
  - Generate dynamically
  - Respond to program characteristics
    - Mutation rate
    - Survival rate
    - Available memory and power
    - Heap topology
Single Heap Collectors

- Tracing (1960)
- Pure Reference Counting (1960)
- Partial Tracing (new)
- Deferred Reference Counting (1976)
Generational Collectors


“Redundant Reference Counting” (new)  “Inferior Reference Counting” (new)
Combining Design and Implementation Frameworks

- Generating new algorithms easily
  - Current state of the art: new collector in 30 minutes

- Mechanically?
  - Leverage frameworks

- Online?
  - Just In Time Garbage Collector generation

- Adaptively?
  - “Invalidate” collector and regenerate
Virtualizing the Object Model

Virtual Machine Definition

Lock Mgr
Object Model
Garbage Collector
Compiler

Instruction Set Architecture

Scheduler
Stack Layout
Data Layout
Virtualizing the Object Model
Many Small Objects of a Single Type

<table>
<thead>
<tr>
<th>PAGE</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>f1</td>
<td>class pointer</td>
</tr>
<tr>
<td>f1</td>
<td>GC bits</td>
</tr>
<tr>
<td>f1</td>
<td>hash codes</td>
</tr>
<tr>
<td>f1</td>
<td>locks</td>
</tr>
<tr>
<td>f2</td>
<td></td>
</tr>
<tr>
<td>f1</td>
<td></td>
</tr>
<tr>
<td>f1</td>
<td></td>
</tr>
<tr>
<td>f1</td>
<td></td>
</tr>
<tr>
<td>f1</td>
<td></td>
</tr>
<tr>
<td>f1</td>
<td></td>
</tr>
<tr>
<td>f1</td>
<td></td>
</tr>
<tr>
<td>f2</td>
<td></td>
</tr>
<tr>
<td>f2</td>
<td></td>
</tr>
<tr>
<td>f2</td>
<td></td>
</tr>
<tr>
<td>f2</td>
<td></td>
</tr>
<tr>
<td>f2</td>
<td></td>
</tr>
<tr>
<td>f2</td>
<td></td>
</tr>
<tr>
<td>f2</td>
<td></td>
</tr>
<tr>
<td>f2</td>
<td></td>
</tr>
<tr>
<td>f2</td>
<td></td>
</tr>
<tr>
<td>f2</td>
<td></td>
</tr>
<tr>
<td>f2</td>
<td></td>
</tr>
<tr>
<td>f2</td>
<td></td>
</tr>
<tr>
<td>f2</td>
<td></td>
</tr>
<tr>
<td>f2</td>
<td></td>
</tr>
<tr>
<td>f2</td>
<td></td>
</tr>
<tr>
<td>f2</td>
<td></td>
</tr>
<tr>
<td>f2</td>
<td></td>
</tr>
<tr>
<td>f2</td>
<td></td>
</tr>
<tr>
<td>f2</td>
<td></td>
</tr>
</tbody>
</table>
Virtualizing the Data Layout

Virtual Machine Definition

Scheduler
- Lock Mgr
- Garbage Collector

Compiler
- Object Model
- Stack Layout
- Data Layout

Instruction Set Architecture
Objects of Varying Size: char[ ]

Default Representation: Unicode

Compressed Representation: Bytes

Big Character Look-aside Table (SW or HW)
Virtualizing the Hardware

Virtual Machine Definition

Lock Mgr  Object Model

Scheduler

Garbage Collector

Compiler

Stack Layout

Data Layout

Instruction Set Architecture

Instruction Set Architecture
Kava: Enabling Dynamic Hardware

- Combine Java abstraction with VHDL precision
- Eliminate primitive types (int, boolean, float)
  - Instead define class `int`, `float`, etc. in `java.lang`
- Object-orientation down to a single bit!
- Allows user-defined primitives
  - Efficient
  - Flexible

- Hardware flexibility
  - Expose (MMX)
  - Compile (float)
  - JIT (point3D)
Object-Oriented Bits

```java
enum bit {
    zero, one;

    public bit ~ this { return not[this]; }
    public bit this & bit that { return and[this][that]; }

    public bit sum(bit a, bit b) { return sum[this][a][b]; }

    static final bit not[bit] = {one, zero};
    static final bit and[bit][bit] = {{zero, zero}, {zero, one}};
    static final bit sum[bit][bit][bit] = {{{zero, one}, {one, zero}},
                                           {{one, zero}, {zero, one}}};

    ....
```
Primitive Types: unsigned byte

```c
enum byteindex { b0, b1, b2, b3, b4, b5, b6, b7 };

final value ubyte {
    private bit data[byteindex];
    …
}
```

```
ubyte data
```

```
bit[ ]
b0
b1
b2
b3
b4
b5
b6
b7
```

```
bit 1
```

```
bit 0
```
Implementing Comparison

```java
public boolean this < ubyte that {
    for each (byteindex x: byteindex.last...byteindex.first)
        if (data[x] != that.data[x])
            return data[x] < that.data[x];
    return false;
}
```
protected float add(float that) {
    if (this.isNaN() || that.isNaN())
        return propagateNaN(that);

    if (this.isInfinity()) {
        if (that.isInfinity() && this.sgn != that.sgn)
            return NaN;
        else
            return this;
    }

    if (this.isZero() && that.isZero()) {
        if (this.sgn == sign.minus && that.sgn == sign.minus)
            return NEGATIVE_ZERO;
        else
            return POSITIVE_ZERO;
    }

    uint thisman = this.mantissa.uintValue() << ComputeShift;
    uint thatman = that.mantissa.uintValue() << ComputeShift;

    if (this.exponent == that.exponent && this.exponent.denormalized())
        return new float(this.sgn,
            this.mantissa.carry(that.mantissa) == bit.zero ?
            exponent.zero : exponent.one, this.mantissa + that.mantissa);

    thisman = thisman.setbit(ImplicitBit);
    thatman = thatman.setbit(ImplicitBit);

    return roundAndPack(this.sgn, this.exponent, 
        (thisman + thatman) << wordindex.bit1);
}

fexponent exp;

if (this.exponent > that.exponent) {
    fexponent diff = this.exponent - that.exponent;

    if (that.denormalized())
        diff--;  
    else
        thatman = thatman.setbit(ImplicitBit);

    thatman = shiftAndJam(thatman, diff);
    exp = this.exponent;
}
else {
    fexponent diff = that.exponent - this.exponent;

    if (this.denormalized())
        diff--;  
    else
        thisman = thisman.setbit(ImplicitBit);

    thisman = shiftAndJam(thisman, diff);
    exp = that.exponent;
}

return roundAndPack(this.sgn, exp, 
    (thisman + thatman) << wordindex.bit1);
What the Programmer Sees
What the Machine Sees

point

x: ubyte

\[
\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
\end{array}
\]

y: ubyte

\[
\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
\end{array}
\]

point(15,15)  

0000111100001111  

(16-bit halfword)
JITing the Instruction Set

- Interpret
- Compile
- Optimize
- Re-program hardware

- Programmable Functional Units
  - Shift, mask, fill
  - Modify/add ALU operations
Next Level: Programmable Fabrics

- Fundamental property of hardware: finite
- Imagine a CPU with an attached FPGA

“Compiling” a critical loop:
- No external dependencies
- Strip mine (finite bounds)
- Feed into FPGA compiler
- Reprogram FPGA fabric
- Re-JIT code to use new fabric operations
Conclusions

- Virtual machine interfaces allow flexibility
- But so far, we only exploit one dimension
- Everything below the interface is malleable
- Huge opportunity
  - Innovation
  - Performance