A Method Specialisation and Virtualised Execution Environment for Java

Andrew M Cheadle
Anthony J Field
Johan T Nyström-Persson

Imperial College London
public class Incrementor {
    int value = 0;
    public void addOne() {
        value++;
    }
    ...
}

Partial: Java
Optional: C++, C#
Complete: Message dispatch in Smalltalk
Idea

- Virtualise *everything*
- Specialise anything
- Opposite of conventional wisdom for optimizing dynamic languages
- Buy back resulting overhead by exploiting the indirection
Exploiting Virtualisation

```java
incrementor.addOne();
```

Method table

- addOne
- addTwo
- ...

(method body)

Method table 2

- addOne
- addTwo
- ...

Special action
Applications

- Object Proxying
- Orthogonal persistence
- Distributed JVM: ANU’s dJVM, IBM’s cJVM
- Profiling
- Instrumentation
- Fickle objects
- Read barrier for incremental garbage collector
This Paper

- Framework for full virtualisation and method specialisation.
- Optimization is allowed to proceed using guarded inlining of specialisations.
- Case Study - Elimination of redundant incremental garbage collector read barriers
- Evaluation of both framework and dynamic dispatching read barrier overheads
Class Transform Toolkit (CTTk)

- Our framework for virtualisation and specialisation
- Uses Jakarta BCEL to modify and generate Java code
- Pipelined code transformations
public interface Transform {
    public TransformClass transform(TransformClass transformClass);
}

public class TransformClass {
    public void setJavaClass(JavaClass javaClass) { ... }
    public JavaClass getJavaClass() { ... }
    public void setSpecialisedMethods(Method[][] specialisedMethods) {
        ... }
    public Method[][] getSpecialisedMethods() { ... }
}
VMT switching API

VM_ObjectModel.restoreTIB(Object obj);

VM_ObjectModel.hijackTIB(Object obj, int tibSpec);
Virtualisation

- Effectively beanification - Java Bean contract
- Achieve full virtualisation through code transformation
- Create access methods for each field
- Replace each direct field access with a call to an access method
- Remove redundant access methods
public class Incrementor {
    private int value = 0;

    int _g_value_Incrementor_I() {
        return value;
    }

    void _s_value_Incrementor_I(int field1) {
        value = field1;
    }

    public void addOne() {
        _s_value_Incrementor_I(_g_value_Incrementor_I() + 1);
    }
}
The Type Information Block

- The TIB is an Object array
- Full type description
- Contains method table!
TIB specialisations

Type
TIB array ptr
Current TIB idx
Primary TIB ptr
Superclass ids
Implements trits
Array element TIB
Type cache
iTABLES
Indirect IMT
... Interface slot i
... Virtual method i
... Object[]

Default TIB TIB 1 ... TIB n
Object[]
tibArray

Type
TIB array ptr
Current TIB idx
Primary TIB ptr
Superclass ids
Implements trits
Array element TIB
Type cache
iTABLES
Indirect IMT
... Interface slot i
... Virtual method i
... Object[]
TIB n
public class Incrementor {
    int value = 0;
    public void addOne() {
        value++;
    }
    public void addTwo() {
        addOne();
        addOne();
    }
}

public class Incrementor {
    int value = 0;
    public void addOne() {
        value++;
    }
    public void addTwo() {
        // inlined call site
        value++;
        // inlined call site
        value++;
    }
}
 Inline guards

- Inlining is a crucial optimization
- Call site is provably *monomorphic*: inline directly
- Call site is *polymorphic*: perform *guarded inlining*
- Typically one of: `ig_method_test`, `ig_class_test`, `ig_patch_point`
- We add the `ig_tib_test` and allow for an arbitrary number of guards (we use at most 2)
Inline Guards in Practice

If any of the guard tests fail, the method will be invoked as normal.
Case Study: Read Barrier for GC

- Use CTTk to create the $IncrGCTransform$
- Transform creates specialised methods: 1 normal TIB, one specialised
- Specialised methods maintain the to-space invariant; they scavenge the object, then restore the TIB and invoke the method as normal
- The read barrier action is thus inserted at no extra cost, only where needed
Read Barrier for GC - Summary

a. GC off

Heap

Code
b. GC on - copy all objects into TO-space

TO-space

FROM-space

Code'

Danger
c. Reference accesses object
public class Incrementor {
    int value = 0;

    public void addOne[0]() {
        _s_value_Incrementor_I(_g_value_Incrementor_I() + 1);
    }

    public void addOne[1]() {
        VM_ObjectModel.restoreTIB(this);
        MMIk.scavenge(this);
        addOne(); // will invoke addOne[0]
    }
}

Concurrency

- Our framework ensures that no thread switchpoints are being emitted while we switch between one TIB and another.
- However, we make no guarantee about the order in which methods or specialisations of methods will be executed. Make sure that their code is thread safe.
- Users must define and enforce a policy by themselves.
Evaluation

- Evaluate read barrier overheads using upper and lower bounds
- Upper bound — all objects scavenged via the mutator using method specialisations
- Lower bound — the collector scavenges all objects, the mutator none
- DaCapo and Spec JVM98 benchmarks run to convergence
### CTTk Overheads

#### CTTk overheads on execution time

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Original(s)</th>
<th>Virtual (% o/head)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Init</td>
<td>Conv</td>
</tr>
<tr>
<td>_201_compress</td>
<td>6.17</td>
<td>5.02</td>
</tr>
<tr>
<td>_202_jess</td>
<td>3.54</td>
<td>2.83</td>
</tr>
<tr>
<td>_209_db</td>
<td>13.72</td>
<td>13.40</td>
</tr>
<tr>
<td>_213_javac</td>
<td>8.83</td>
<td>5.87</td>
</tr>
<tr>
<td>_222_mpegaudio</td>
<td>6.68</td>
<td>3.72</td>
</tr>
<tr>
<td>_227_mtrt</td>
<td>5.08</td>
<td>2.53</td>
</tr>
<tr>
<td>antlr</td>
<td>5.57</td>
<td>4.46</td>
</tr>
<tr>
<td>bloat</td>
<td>15.02</td>
<td>11.51</td>
</tr>
<tr>
<td>fop</td>
<td>5.33</td>
<td>3.21</td>
</tr>
<tr>
<td>luindex</td>
<td>19.36</td>
<td>15.40</td>
</tr>
<tr>
<td>pmd</td>
<td>15.14</td>
<td>9.78</td>
</tr>
<tr>
<td><strong>Min</strong></td>
<td>3.54</td>
<td>2.83</td>
</tr>
<tr>
<td><strong>Max</strong></td>
<td>19.36</td>
<td>13.4</td>
</tr>
<tr>
<td><strong>Geo. mean</strong></td>
<td>8.23</td>
<td>5.82</td>
</tr>
</tbody>
</table>
### Read Barrier Overheads

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Implicit (Lower)</th>
<th>Implicit (Upper)</th>
<th>Explicit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Init</td>
<td>Conv</td>
<td>Init</td>
</tr>
<tr>
<td>_201_compress</td>
<td>42.79%</td>
<td>8.37%</td>
<td>42.86%</td>
</tr>
<tr>
<td>_202_jess</td>
<td>90.89%</td>
<td>2.95%</td>
<td>91.11%</td>
</tr>
<tr>
<td>_209_db</td>
<td>12.10%</td>
<td>2.86%</td>
<td>1.12%</td>
</tr>
<tr>
<td>_213_javac</td>
<td>59.52%</td>
<td>23.27%</td>
<td>64.35%</td>
</tr>
<tr>
<td>_222_mpegaudio</td>
<td>28.62%</td>
<td>5.66%</td>
<td>28.70%</td>
</tr>
<tr>
<td>_227_mtrt</td>
<td>42.79%</td>
<td>8.21%</td>
<td>31.34%</td>
</tr>
<tr>
<td>antrr</td>
<td>328.17%</td>
<td>5.62%</td>
<td>328.17%</td>
</tr>
<tr>
<td>bloat[*]</td>
<td>165.79%</td>
<td>12.27%</td>
<td>165.79%</td>
</tr>
<tr>
<td>fop*[i]</td>
<td>711.83%</td>
<td>4.41%</td>
<td>711.83%</td>
</tr>
<tr>
<td>luindex*[i]</td>
<td>273.46%</td>
<td>21.49%</td>
<td>273.46%</td>
</tr>
<tr>
<td>pmd*[i]</td>
<td>360.70%</td>
<td>21.0%</td>
<td>360.70%</td>
</tr>
<tr>
<td>Min</td>
<td>12.10%</td>
<td>2.86%</td>
<td>1.12%</td>
</tr>
<tr>
<td>Max</td>
<td>711.83%</td>
<td>23.27%</td>
<td>711.83%</td>
</tr>
<tr>
<td>Geo. mean</td>
<td>102.49%</td>
<td>8.15%</td>
<td>80.87%</td>
</tr>
</tbody>
</table>

* For the explicit barrier, not all benchmarks can be run to completion. Reported values are based on measurements taken prior to the crashes.
## (De)Virtualisation Counts and (Static) Code Bloat

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Virt&quot;s</th>
<th>Devirt&quot;s</th>
<th>Code Bloat(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>_201_compress</td>
<td>5189</td>
<td>2989</td>
<td>10.1</td>
</tr>
<tr>
<td>_202_jess</td>
<td>5977</td>
<td>3684</td>
<td>13.7</td>
</tr>
<tr>
<td>_209_db</td>
<td>5078</td>
<td>2564</td>
<td>23.2</td>
</tr>
<tr>
<td>_213_javac</td>
<td>9159</td>
<td>2668</td>
<td>11.8</td>
</tr>
<tr>
<td>_222_mpegaudio</td>
<td>6172</td>
<td>3673</td>
<td>32.2</td>
</tr>
<tr>
<td>_227_mtrt</td>
<td>5499</td>
<td>2635</td>
<td>28.1</td>
</tr>
<tr>
<td>antlr</td>
<td>14025</td>
<td>4716</td>
<td>21.2</td>
</tr>
<tr>
<td>bloat</td>
<td>9871</td>
<td>9781</td>
<td>12.1</td>
</tr>
<tr>
<td>fop</td>
<td>9295</td>
<td>3431</td>
<td>24.2</td>
</tr>
<tr>
<td>luindex</td>
<td>6661</td>
<td>2513</td>
<td>18.2</td>
</tr>
<tr>
<td>pmd</td>
<td>13898</td>
<td>2928</td>
<td>26.3</td>
</tr>
</tbody>
</table>
## BCEL Overheads

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>BCEL only (Conv)</th>
</tr>
</thead>
<tbody>
<tr>
<td>_201_compress</td>
<td>7.57%</td>
</tr>
<tr>
<td>_202_jess</td>
<td>2.47%</td>
</tr>
<tr>
<td>_209_db</td>
<td>1.27%</td>
</tr>
<tr>
<td>_213_javac</td>
<td>13.12%</td>
</tr>
<tr>
<td>_222_mpegaudio</td>
<td>2.42%</td>
</tr>
<tr>
<td>_227_mtrt</td>
<td>3.95%</td>
</tr>
<tr>
<td>antlr</td>
<td>4.26%</td>
</tr>
<tr>
<td>bloat</td>
<td>8.95%</td>
</tr>
<tr>
<td>fop</td>
<td>3.12%</td>
</tr>
<tr>
<td>luindex</td>
<td>14.09%</td>
</tr>
<tr>
<td>pmd</td>
<td>14.83%</td>
</tr>
<tr>
<td><strong>Min</strong></td>
<td><strong>1.27%</strong></td>
</tr>
<tr>
<td><strong>Max</strong></td>
<td><strong>14.83%</strong></td>
</tr>
<tr>
<td><strong>Geo. mean</strong></td>
<td><strong>5.19%</strong></td>
</tr>
</tbody>
</table>
Conclusions

- Overhead of full virtualisation is low
- CTTk delivers both ease of use and good performance
- A read barrier for an incremental collector has been constructed with relative ease, and it compares favourably
Work in progress

- Framework is now current with Jikes RVM CVS Head and Java 1.5 -> Spec JBB.
- Currently implementing incremental collectors for Jikes RVM
  - Explicit Baker-style read barrier collector
  - Dynamic dispatching read barrier collector
- Porting the framework to use ASM (more lightweight) instead of BCEL
- DSL (syntactic sugar) to ease transform specification