Manipulating Managed Execution
Runtimes to support Self-Healing Systems

Rean Griffith‡, Gail Kaiser‡

Presented by Rean Griffith
rg2023@cs.columbia.edu
‡ - Programming Systems Lab (PSL) Columbia University
Introduction
Overview

• Motivation
• Managed Execution Model
• System Architecture
• How it works
• Performing a repair
• Performance
• Conclusions & Future work
Motivation

• Managed execution environments e.g. JVM, CLR provide a number of application services that enhance the robustness of software systems, BUT...

• They do not currently provide services to allow applications to perform consistency checks or repairs of their components

• Managed execution environments intercept everything applications running on top of them attempt to do. Surely we can leverage this
Managed Execution Model

1. Application/Module Load
2. Class Load
3. Method Invoke
4. JIT Compile (if necessary)
5. Function Enter
6. Function Exit

// Other code
SampleClass s = new SampleClass();
s.doSomethingUseful();
// More code

Execute
Find member
doSomethingUseful()
in memory

Method body JIT Compiled?
No
Do JIT-Compile of Bytecode

Yes
Jump to JIT-Compiled native assembly version of the method
# Runtime Support Required

<table>
<thead>
<tr>
<th>Facility</th>
<th>CLR v1.1</th>
<th>JVM v5.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>The ability to receive notifications about current execution stage</td>
<td>Profiler API</td>
<td>JVMTI (no JIT)</td>
</tr>
<tr>
<td>The ability to obtain information (metadata) about the application,</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>types, methods etc. from profiler</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The ability to make controlled changes or extensions to metadata</td>
<td>Fine-grained (full)</td>
<td>Coarse-grained (partial)</td>
</tr>
<tr>
<td>e.g. new function bodies, new type, type::method references</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The ability to have some control over the JIT-Compilation process</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>
System Architecture

![Diagram]

Figure 2: Prototype architecture diagram
Our Prototype’s Model of Operation

1. **Application/Module Load**
   - Augment type metadata, e.g. define new method stubs as repair-engine hooks

2. **Class Load**

3. **Method Invoke**
   - Fill in method stubs, edit/replace method body, insert jumps into repair engine, undo changes

4. **JIT Compile (if necessary)**

5. **Function Enter**
   - RepairEngine::RepairMe(this)
   - Repair/Consistency check

6. **Function Exit**
Phase I – Preparing Shadow Methods

- At module load time but before type definition installed
  - Extend type metadata by defining with new methods which will be used to allow a repair engine to interact with instances of this type
Phase II – Creating Shadow Methods

- At first JIT-Compilation
  - Define the body of the shadow method and re-wire some things under-the-hood

```
SampleMethod( args )
  <room for prolog>
  push args
  call _SampleMethod( args )
  <room for epilog>
  return value/void
```
Performing a Repair

- Augment the wrapper to insert a jump into a repair engine at the *control point(s)* before and/or after a shadow method call

SampleMethod( args )
  RepairEngine::RepairMe(this)
push args
call _SampleMethod( args )
  RepairEngine::RepairMe(this)
return value/void
Performance – No Repairs Active

**Figure 7:** Overheads when no repair active
Overheads on the Managed Execution Cycle

<table>
<thead>
<tr>
<th>Module Name</th>
<th>SciMark.exe</th>
</tr>
</thead>
<tbody>
<tr>
<td>Module Load time (ms)</td>
<td>0.0230229</td>
</tr>
<tr>
<td>Module bind time (ms)</td>
<td>0.374817</td>
</tr>
<tr>
<td># shadows prepared</td>
<td>2</td>
</tr>
<tr>
<td>Total shadow prepare time (ms)</td>
<td>0.196664</td>
</tr>
<tr>
<td>Average shadow prepare time (ms)</td>
<td>0.0983317</td>
</tr>
<tr>
<td>Bind time - shadow prepare time (ms)</td>
<td>0.178153</td>
</tr>
</tbody>
</table>

Table 1: Overheads of preparing shadows

<table>
<thead>
<tr>
<th>Method name</th>
<th>SOR::execute</th>
</tr>
</thead>
<tbody>
<tr>
<td>First JIT time (ms)</td>
<td>13.7202</td>
</tr>
<tr>
<td># shadows created</td>
<td>1</td>
</tr>
<tr>
<td>Total shadow create time (ms)</td>
<td>13.3576</td>
</tr>
<tr>
<td>Average shadow create time (ms)</td>
<td>13.3576</td>
</tr>
<tr>
<td>First Jit time - shadow create time (ms)</td>
<td>0.3626</td>
</tr>
</tbody>
</table>

Table 2: Overheads of creating shadows

<table>
<thead>
<tr>
<th>Function ID</th>
<th>Wrapper Method SOR::execute</th>
<th>Shadow Method SOR::execute</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enter/Leave count</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>JIT Count</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td># shadows created</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Create shadow (ms)</td>
<td>11.1834</td>
<td>n/a</td>
</tr>
<tr>
<td>Total Invoke time (ms)</td>
<td>6273.27</td>
<td>6272.31</td>
</tr>
<tr>
<td>Total JIT time (ms)</td>
<td>2.9622</td>
<td>0.90244</td>
</tr>
<tr>
<td>Total method time (ms)</td>
<td>6287.4156</td>
<td>6273.21244</td>
</tr>
</tbody>
</table>

Table 3: Execution overheads on SciMark2.SOR::execute
Contributions

• Framework for dynamically attaching/detaching a repair engine to/from a target system executing in a managed execution environment
• Prototype which targets the Common Language Runtime (CLR) and supports this dynamic attach/detach capability with low runtime overhead (~5%)
Limitations

• Repairs can be scheduled but they depend on the execution flow of the application to be effected
  – Deepak Gupta et al. prove that it is un-decidable to automatically determine that “now” is the right time for a repair
  – Programmer-knowledge is needed to identify “safe” control-points at which repairs could be performed
  – The “safe” control points may be difficult to identify and may impact the kind of repair action possible

• Primarily applicable to managed execution environments
  – Increased metadata availability/accessibility
  – Security sandboxes restrict the permissions of injected bytecode to the permissions granted to the original application
Conclusions & Future Work

• Despite being primarily applicable to managed execution environments, these techniques may help us “Watch the Watchers”
  – the management infrastructure we are building is likely to be written in managed code (Java, C#) running in the JVM, CLR mainly because these environments provide application services that enhance the robustness of managed applications

• On the to-do list:
  – Continue working on the prototype for the JVM so we can compare the performance and generalize the runtime support requirements listed earlier
  – Do a real case study to see what issues we run into with respect to identifying and leveraging “safe” control points, the implications of architectural style
Comments, Questions, Queries

Thank You

Contact: rg2023@cs.columbia.edu
Extra slides
Managed execution environments e.g. JVM, CLR provide a number of application services that enhance the robustness of software systems BUT...

- They do not currently provide services to allow applications to perform consistency checks or repairs of their components.
- Managed execution environments intercept everything applications running on top of them attempt to do. Surely we can leverage this.
Our Prototype’s Model of Operation

Execution Runtime

1. Application/Module Load
2. Class Load
3. Method Invoke
4. JIT Compile (if necessary)
5. Function Enter
6. Function Exit

Augment type metadata e.g. define new method stubs as repair-engine hooks

Fill in method stubs, edit/replace method body, insert jumps into repair engine, undo changes

Metadata extensions e.g. add references to new modules, types and methods

RepairEngine::RepairMe(this)

Repair/Consistency check

JIT-Control API e.g. request method re-JIT