Instrumentation and Evaluation for Dynamic Program Analysis

http://d3s.mff.cuni.cz

Lukáš Marek
Introduction

- **Taint analysis**
  - Observes propagation of values from an initial set of variables (e.g. input) through the application

- **Execution time profiling**
  - Measures the execution time of methods

- **Data race detection**
  - Monitors field accesses in multi-threaded applications

- **Object lifetime analysis**
  - Tracks manipulations with object references
Dynamic Program Analysis

Application code

Observation part

Analysis code

Evaluation part

Probe

Event dispatch (2)

Resume execution (4)

Event evaluation (3)
Dynamic Program Analysis - Challenges

- Application observation
  - Flexibility
    - Observation of arbitrary application behavior
  - Performance
    - Application may contain big amount of inserted probes

- Event evaluation
  - Isolation of the evaluation
  - Reduced overhead of the isolation
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>avrora</td>
<td>5.11</td>
<td>30.96</td>
<td>12.61</td>
<td>2.46</td>
</tr>
<tr>
<td>batik</td>
<td>1.28</td>
<td>2.70</td>
<td>1.78</td>
<td>1.52</td>
</tr>
<tr>
<td>eclipse</td>
<td>16.16</td>
<td>152.92</td>
<td>70.73</td>
<td>2.16</td>
</tr>
<tr>
<td>fop</td>
<td>0.35</td>
<td>3.36</td>
<td>1.68</td>
<td>2</td>
</tr>
<tr>
<td>h2</td>
<td>5.84</td>
<td>63.25</td>
<td>25.27</td>
<td>2.5</td>
</tr>
<tr>
<td>jython</td>
<td>2.67</td>
<td>5.70</td>
<td>3.89</td>
<td>1.47</td>
</tr>
<tr>
<td>luindex</td>
<td>0.90</td>
<td>7.06</td>
<td>2.71</td>
<td>2.61</td>
</tr>
<tr>
<td>lusearch</td>
<td>1.98</td>
<td>13.09</td>
<td>5.49</td>
<td>2.38</td>
</tr>
<tr>
<td>pmd</td>
<td>2.05</td>
<td>10.09</td>
<td>5.10</td>
<td>1.98</td>
</tr>
<tr>
<td>sunflow</td>
<td>3.45</td>
<td>57.24</td>
<td>21.44</td>
<td>2.67</td>
</tr>
<tr>
<td>tomcat</td>
<td>1.97</td>
<td>4.46</td>
<td>3.16</td>
<td>1.41</td>
</tr>
<tr>
<td>tradebeans</td>
<td>5.56</td>
<td>71.48</td>
<td>30.76</td>
<td>2.32</td>
</tr>
<tr>
<td>tradesoap</td>
<td>6.77</td>
<td>25.40</td>
<td>12.80</td>
<td>1.98</td>
</tr>
<tr>
<td>xalan</td>
<td>1.11</td>
<td>20.39</td>
<td>8.15</td>
<td>2.5</td>
</tr>
</tbody>
</table>
Motivation – isolation

- Depending on what is observed, the problems in the observed application include:
  - Deadlocks
  - State corruption
  - Infinite recursion
  - JVM crashes
  - Unwanted additional events
  - Missing events
## Motivation – isolation

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Uninstrumented</th>
<th>In-process FIA</th>
<th>ShadowVM FIA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>GC young/old</td>
<td>GC young/old</td>
<td>GC young/old</td>
</tr>
<tr>
<td></td>
<td>Allocated memory</td>
<td>Allocated memory</td>
<td>Allocated memory</td>
</tr>
<tr>
<td>avrora</td>
<td>1/1</td>
<td>65906</td>
<td>218/1</td>
</tr>
<tr>
<td>batik</td>
<td>1/1</td>
<td>122865</td>
<td>10/1</td>
</tr>
<tr>
<td>fop</td>
<td>1/1</td>
<td>62600</td>
<td>4/1</td>
</tr>
<tr>
<td>h2</td>
<td>3/1</td>
<td>956686</td>
<td>92/1</td>
</tr>
<tr>
<td>jython</td>
<td>1/1</td>
<td>172068</td>
<td>73/1</td>
</tr>
<tr>
<td>lуindex</td>
<td>1/1</td>
<td>32029</td>
<td>2/1</td>
</tr>
<tr>
<td>lusearch</td>
<td>4/1</td>
<td>729793</td>
<td>14/1</td>
</tr>
<tr>
<td>pmd</td>
<td>1/1</td>
<td>35912</td>
<td>2/1</td>
</tr>
<tr>
<td>sunflow</td>
<td>2/1</td>
<td>306649</td>
<td>22/1</td>
</tr>
<tr>
<td>xalan</td>
<td>1/1</td>
<td>190011</td>
<td>11/1</td>
</tr>
</tbody>
</table>

Allocated memory in KB
Application observation (event triggering)

Event evaluation
Application observation using DiSL

- An AOP inspired language for high-level instrumentation specification
- Probes inserted using instrumentation
- Designed to be transparent (full control over the inserted probes)
Benefits of DiSL
public static void methodInvocationTrace() {
    long time = System.nanoTime();
    System.out.println(time);
}

- Simple instrumentation specification using Java language
- All code is directly inlined for best performance
@After(marker=BodyMarker.class)
public static void methodInvocationTrace() {
    long time = System.nanoTime();
    System.out.println(time);
}

- Allows to instrument arbitrary execution pattern
@After(marker=BodyMarker.class)
public static void methodInvocationTrace(MethodStaticContext sc) {
    long time = System.nanoTime();
    System.out.println(sc.thisMethodFullName() + " : " + time);
}

- Allows arbitrary static analysis during weave time
- The information is pre-computed and inserted directly into the code
Synthetic variables

```java
@SyntheticLocal
static long entryTime;

@Before(marker=BodyMarker.class)
public static void methodInvocationTrace(MethodStaticContext sc) {
    long entryTime = System.nanoTime();
}

@After(marker=BodyMarker.class)
public static void methodInvocationTrace(MethodStaticContext sc) {
    long time = System.nanoTime();
    System.out.println(sc.thisMethodFullName() + " : " + (time - entryTime));
}
```

- Efficient data passing between snippets
- No additional memory allocation needed
Other benefits of DiSL

- Guards for instrumentation scoping during weave time
- Efficient access to dynamic context information
- Defective probe handlers
Application observation (event triggering)

Event evaluation
Event evaluation using ShadowVM

- Achieves isolation by offloading events out of the analyzed application and evaluates them in different JVM (process)

- Introduces asynchronous event processing and event ordering models to stay performance competitive
Isolation in ShadowVM

- Instrumented base program
- Instrumented Java class library
- Observed JVM
- Native code
- Event API
ShadowVM architecture

- Analysis server
  - User-supplied analysis code
  - Shadow API implementation (+ host JVM)

- Instrumented base program
- Instrumented Java class library

- Observed JVM
- JVMTI agent

- Event API

- Analysis process
- Observed process
Other benefits of ShadowVM

- Event ordering models for improved performance and simplified event evaluation programming
- Life-cycle events for simple resource management
- Shadow API to easily access class hierarchy information
Conclusion

- “Complete” system for dynamic program analysis
- Simple but extensible language for instrumentation specification (probe definition)
- Developer has full control over the inserted code
- Isolation achieved by offloading the analysis out of the context of the observed application


Kell S., Ansaloni D., Binder W., Marek L.: **The JVM is not observable enough (and what to do about it)**, In Proceedings of the 6th workshop on Virtual Machines and Intermediate Languages (VMIL 2012)


DiSL was successfully used in several other research projects.

An ongoing effort at University of Lugano to extend DiSL and ShadowVM to the Android platform.

DiSL was accepted as an OW2 project and sources are available at

http://disl.ow2.org/
Questions – Kieker

- Extensible framework for monitoring and analyzing the runtime behavior of concurrent or distributed software systems
- Monitoring based on predefined probes or custom probes implemented in AspectJ
- Data stored in file or DB
- Kieker is more about analysis of gathered results

- Kieker is another higher-level layer that could use DiSL and ShadowVM as framework for instrumentation and event offloading
Questions – SPASS-meter

- Resource monitoring approach for Java and Android Apps
- Relays on predefined probes
- Monitors: CPU, Memory, File ops., Network ops.
- Processing of events inside of the observed application
- Instrumentation abstraction layer

- DiSL and ShadowVM could be used as under or replace the instrumentation abstraction layer
- ShadowVM may solve isolation problems encountered by SPASS-meter