TRACERJD: GENERIC TRACE-BASED DYNAMIC DEPENDENCE ANALYSIS WITH FINE-GRAINED LOGGING

Haipeng Cai and Raul Santelices
Department of Computer Science and Engineering
University of Notre Dame

SANER 2015

Supported by ONR Award N000141410037
Context

Program-dependence Analysis

Testing/Debugging
Reverse Engineering
Performance optimization
Evolution

Program
Context

Dynamic Dependence Analysis

Testing/Debugging
Reverse Engineering
Performance optimization
Evolution

Inputs
Program
Some tools monitor high-level system states (e.g., network traffic, resource usage, \ldots)

- DTrace
- Valgrind

Some reports coarse-level runtime conditions (function invocations, exception trace, \ldots)

- JTracer
- JavaTracer

Others are applicable to specific tasks only (dynamic slicing, execution reduction, \ldots)

- JavaSlicer
- Code-Investigator
Only few techniques available capture fine-grained source-level dynamic dependence information that supports a variety of dependence-based applications.

No tool supports.
Approach

TracerJD

Offers **generic/common** fine-grained dynamic dependence information to support various applications

Many specific tools can be built upon it
TracerJD

Application tools

- Dynamic slicer
- Performance profiler

Dynamic dependence querying subroutines

Hierarchical trace indexing

Structured logging of execution events

- Method calls/returns
- Statement occurrence
- Variable definitions/uses
Performance

- Instrumentation time

<table>
<thead>
<tr>
<th>Schedule1</th>
<th>NanoXML</th>
<th>XML-security</th>
<th>JMeter</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.7</td>
<td>15.1</td>
<td>143.3</td>
<td>324.9</td>
</tr>
</tbody>
</table>
Performance

- Runtime slowdown

![Bar chart showing factors of runtime slowdown for different tools: Schedule 1 (12x), NanoXML (13x), XML-security (10x), and JMeter (2x).]
# Performance

## Storage costs

<table>
<thead>
<tr>
<th>Schedule 1</th>
<th>NanoXML</th>
<th>XML-security</th>
<th>JMeter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total trace size (MB)</td>
<td>125.3</td>
<td>36.1</td>
<td>44.6</td>
</tr>
<tr>
<td>Factor of code size growth</td>
<td>2.0</td>
<td>2.1</td>
<td>3.2</td>
</tr>
</tbody>
</table>

The table above shows the total trace size (MB) and the factor of code size growth for different benchmarks. The chart visually represents the same data.
Use scenarios

- Scenario 1: dynamic slicing

- **TracerJD** - *dynamic dependence querying subroutines*
Scenario 2: performance profiling

- Set of statements
- Execution-time report
- Compute time elapses between instances
- Statement instances
- TracerJD - dynamic dependence querying subroutines
- Timing flag
Conclusions

- A framework that provides generic dynamic dependence information to support various applications
- An effective trace indexing scheme that enables efficient dynamic-dependence querying
- Two example client analyses that offer readily utilities and demonstrate the flexibility of building diverse applications
Acknowledgements

Office of Naval Research for funding

All of you for time and attention
PLEASE ASK QUESTIONS