Overhead Comparison of OpenTelemetry, inspectIT and Kieker

David Georg Reichelt\textsuperscript{1}  Stefan Kühne\textsuperscript{1}  Wilhelm Hasselbring\textsuperscript{2}

\textsuperscript{1}Universität Leipzig, Universitätsrechenzentrum, Forschung und Entwicklung

\textsuperscript{2}Christian-Albrechts-Universität zu Kiel, Software Engineering Group

9th of November 2021
Why Overhead Comparison?

- Performance measurement creates overhead
  - Instrumentation
  - Measurement
  - Serialization
  ⇒ Should be as low as possible
  ⇒ Replicable measurement provided by MooBench

- Overhead is relevant
  - ... for comparison of monitoring tools
    ⇒ this work
  - ... for further reduction of monitoring overhead for continuous root cause analysis in Peass
    ⇒ ongoing research
Outline

1. MooBench
2. Monitoring Frameworks
3. Overhead Comparison
4. Summary
Performance Measurement in JVM

- **Warmup**
  - Measure Start
  - Execute Workload
  - Measure End

- **Measurement**
  - Measure Start
  - Execute Workload
  - Measure End

- **VM-Start**

- **Repetition**
  - Measurement Iterations

- **VMs**
Performance Measurement in JVM

- Non-deterministic effects influence performance
  - Just-in-Time-Compilation
  - Garbage collection
  - Memory fragmentation
  - ...

- Measurement process
  - Warmup iterations
  - Measurement iterations
  - Repetition inside VMs
  - Analysis of values by statistical test, e.g. T-Test
MooBench (Measurement Process)

Additional parametrisation by
- SLEEP_TIME (Sleep time between VM starts, so system can cooldown)
MooBench (Variants)

- Baseline
- Regular instrumentation
- Deactivated monitoring
- Different monitoring configurations (e.g. different serialization)
Prior Work on MooBench

- Continuous measurement (Waller, Ehmke and Hasselbring, 2015)
- Testing of replicability (Knoche and Eichelberger, 2017; Knoche and Eichelberger, 2018)
- Effects of multithreading (Waller and Hasselbring, 2015)
Monitoring Frameworks
OpenTelemetry

- “Ubiquitous“ telemetry ⇒ Support of a many languages
- Supports variety of frameworks itself
- Different exporters (Zipkin, Prometheus, Jaeger)
- Instrumentation through javaagent
- Configuration through command line, yaml file, ...
inspectIT

- “Zero-configuration” Java agent for performance collection
- Supports variety of frameworks by usage of OpenCensus
- Different exporters (Zipkin, Prometheus, Jaeger)

- Configuration (through command line, yaml file, ...)
  - Scopes define measured methods
  - Rules define measurement metrics
  - Actions define processing on extracted data
Measurement with MooBench

SUT

Workload

Kieker Agent

OpenTelemetry Agent

inspectIT Agent

TCP Receiver

Prometheus

Zipkin

Hard Disc Logging
Overhead Comparison
Setup

- OpenJDK 11.0.11

- Hardware
  - For replicability to older data: Raspberry Pi 4
  - Current desktop: i7-4770 CPU @ 3.40GHz with 16 GB RAM, running Ubuntu 20.04

- Workload sizes
  - Call tree depth 10 (default) for all configurations
  - Exponential growing call tree depth for TCP export
## Call Tree Depth 10 (Kieker)

<table>
<thead>
<tr>
<th>Variant</th>
<th>Raspberry Pi</th>
<th>i7-4770</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>95 % CI</td>
<td>σ</td>
</tr>
<tr>
<td>Baseline</td>
<td>[1.5;1.5]</td>
<td>0.1</td>
</tr>
<tr>
<td><strong>Kieker</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Deactivated Probe</td>
<td>[4.1;4.1]</td>
<td>7.5</td>
</tr>
<tr>
<td>DumpWriter</td>
<td>[51.9;52.0]</td>
<td>14.6</td>
</tr>
<tr>
<td>Logging (Text)</td>
<td>[743.3;799.4]</td>
<td>14315.8</td>
</tr>
<tr>
<td>Logging (Binary)</td>
<td>[59.8;87.8]</td>
<td>7149.4</td>
</tr>
<tr>
<td>TCP</td>
<td>[45.6;45.7]</td>
<td>14.6</td>
</tr>
</tbody>
</table>

**Tabelle:** Measurement Results for Kieker (in μs)
Call Tree Depth 10 (Kieker)

- Also deactivated probe has noticeable overhead
- Regular text logging is very inefficient
- Fastest configuration for local processing: Binary Logging
## Call Tree Depth 10 (default)

<table>
<thead>
<tr>
<th>Variant</th>
<th>Deactivated</th>
<th>OpenTelemetry</th>
<th>Zipkin</th>
<th>Prometheus</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Pi 4 CI</strong></td>
<td>[26.8;26.9]</td>
<td>[53.4;53.6]</td>
<td>[44.4;44.5]</td>
<td></td>
</tr>
<tr>
<td><strong>σ</strong></td>
<td>20.4</td>
<td>46.7</td>
<td>25.2</td>
<td></td>
</tr>
<tr>
<td><strong>i7-4770 CI</strong></td>
<td>[4.9;5.0]</td>
<td>[6.8;6.9]</td>
<td>[6.9;6.9]</td>
<td></td>
</tr>
<tr>
<td><strong>σ</strong></td>
<td>4.1</td>
<td>8.5</td>
<td>4.9</td>
<td></td>
</tr>
<tr>
<td><strong>inspectIT</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Pi 4 CI</strong></td>
<td>[9.9;9.9]</td>
<td>[97.2;97.8]</td>
<td>[32.3;32.4]</td>
<td></td>
</tr>
<tr>
<td><strong>σ</strong></td>
<td>10.5</td>
<td>149.6</td>
<td>16.6</td>
<td></td>
</tr>
<tr>
<td><strong>i7-4770 CI</strong></td>
<td>[1.3;1.4]</td>
<td>[10.9;11.2]</td>
<td>[4.0;4.0]</td>
<td></td>
</tr>
<tr>
<td><strong>σ</strong></td>
<td>8.2</td>
<td>57.4</td>
<td>4.1</td>
<td></td>
</tr>
</tbody>
</table>

**Tabelle:** Measurement Results for OpenTelemetry and inspectIT (in μs)
OpenTelemetry has lower overhead for Zipkin trace (spans) export

inspectIT has lower overhead for metrics export and deactivated probe

Deactivated pobe overheads are significantly higher than in Kieker
**Growing Call Tree Depth**

![Graph](image)

**Overview of Method Execution Durations**

- Baseline
- Kieker (TCP)
- inspectIT (Zipkin)
- OpenTelemetry (Zipkin)

**Abbildung**: Overhead evolution with growing call tree depth
Growing Call Tree Depth

- Different writer configurations are not comparable
- Kieker currently does not support aggregated metrics export
- Only full trace export to Zipkin / TCP export comparable
Summary
Summary

- Monitoring overhead needs to be as low as possible
- MooBench compares Monitoring overhead of different frameworks and monitoring configurations
- MooBench was extended to support OpenTelemetry and inspectIT
- Measurement of traces by OpenTelemetry and inspectIT is slower than with Kieker
Outlook

- Benchmarking with more complex tree structure
- Comparison of overhead for different frameworks, e.g. Jersey
- Comparison of accuracy (How well does root cause analysis algorithm \( X \) perform?)