Speeding up Type-specific Instrumentation for the Analysis of Complex Systems

Andrea Rosà, Walter Binder

Università della Svizzera italiana (USI), Faculty of Informatics, Lugano, Switzerland

ICECCS 2017
November 8th, 2017
Fukuoka, Japan

Dynamic Analysis

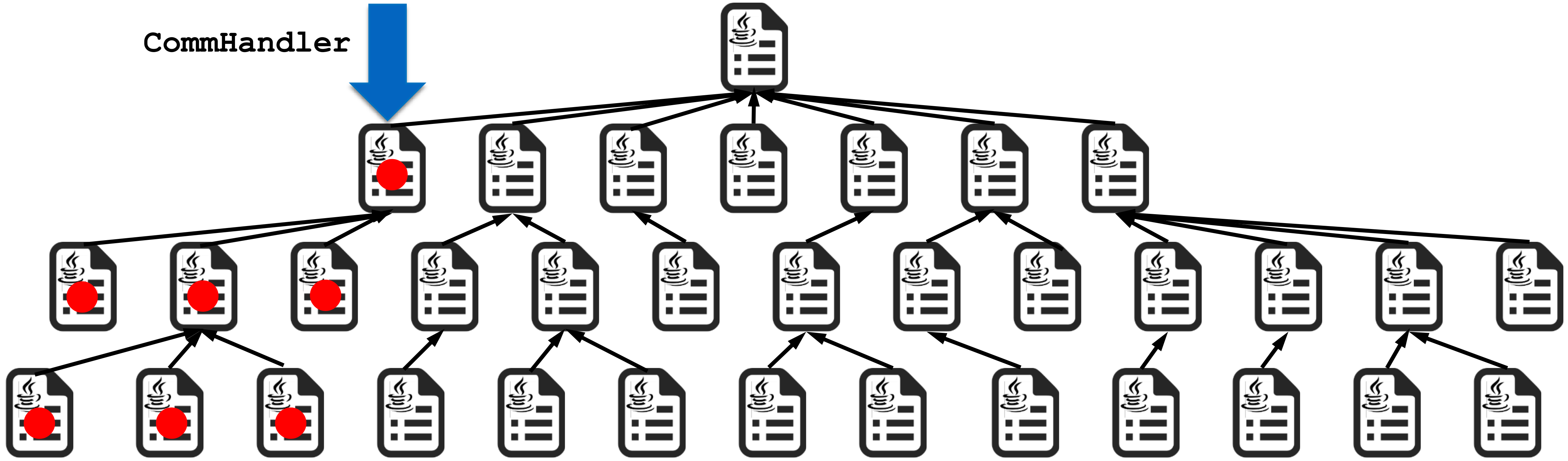
- Dynamic analysis is fundamental for complex systems
  - Enables performance analysis and improvements
- Desired dynamic analysis:
  - Efficient: limited slowdown
  - Complete: profile all relevant components

Dynamic Analysis

- Our focus: **type-specific** dynamic analyses **on the JVM**
  - Often rely on bytecode instrumentation
  - Instrumentation may rely on **reflective supertype information (RSI)**
    - Information on all direct and indirect supertypes
    - Provided by methods of `java.lang.Class`
      - `public Class<? super T> getSuperclass()`
      - `public Class<?>[] getInterfaces()`
      - Recursively applied
- RSI may not be available, resulting in inefficient analyses

RSI during Instrumentation

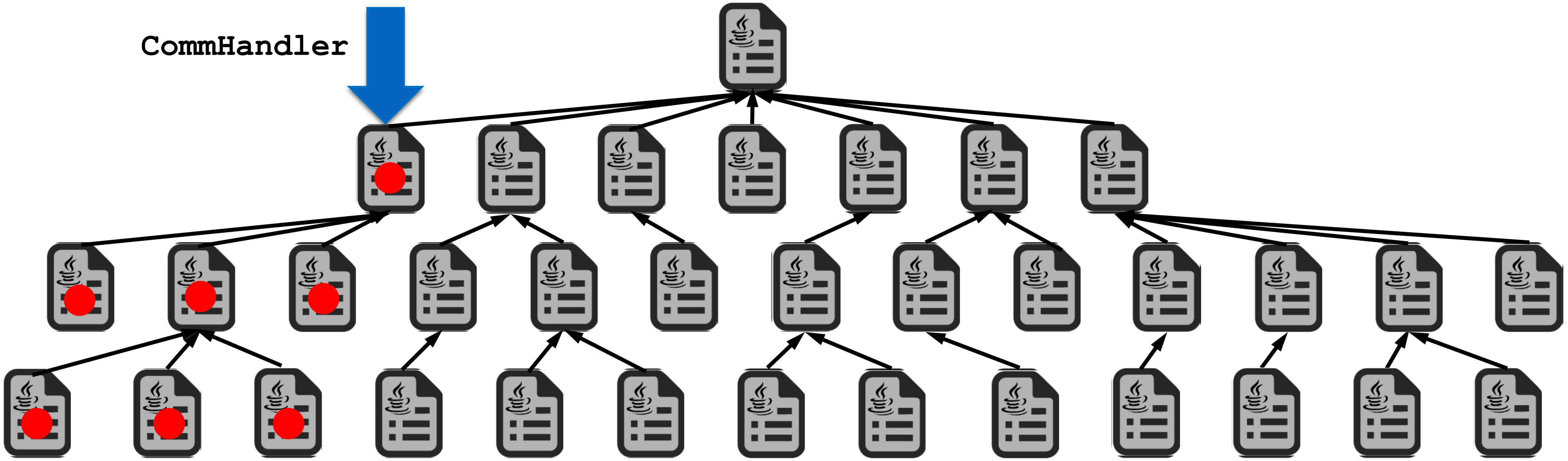
- Example: type-specific analysis targeting communication handlers



- **Without** RSI:

RSI during Instrumentation

- Example: type-specific analysis targeting communication handlers

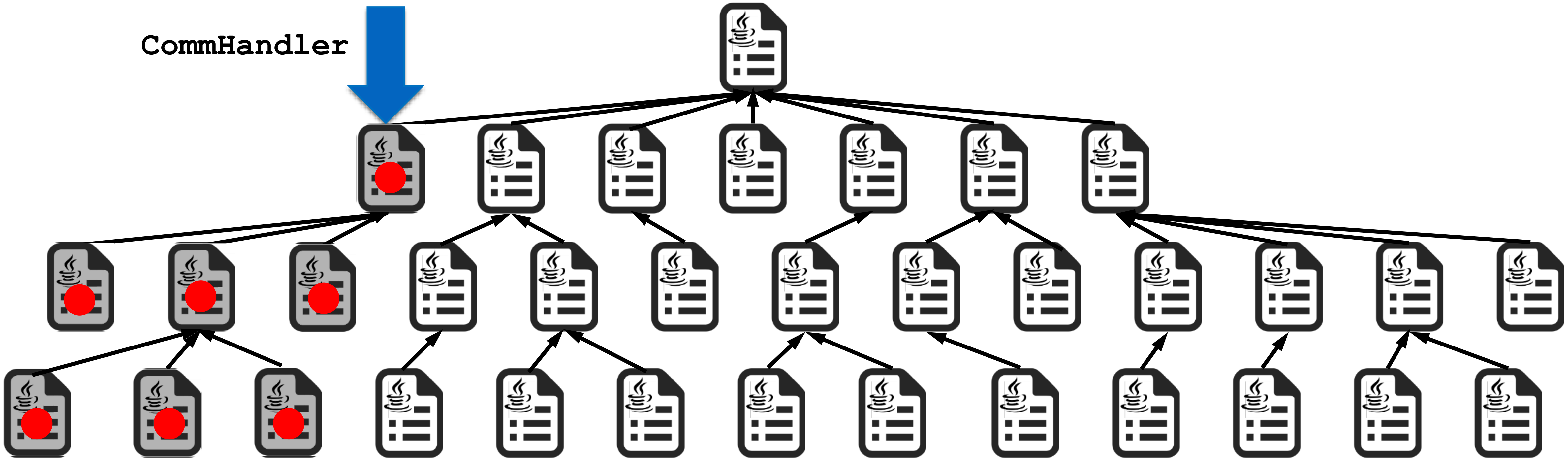


- Without RSI:
  - Many classes instrumented
  - Increased overhead

```
if (this instanceof CommHandler){
    // Do something
}
```

RSI during Instrumentation

- Example: type-specific analysis targeting communication handlers

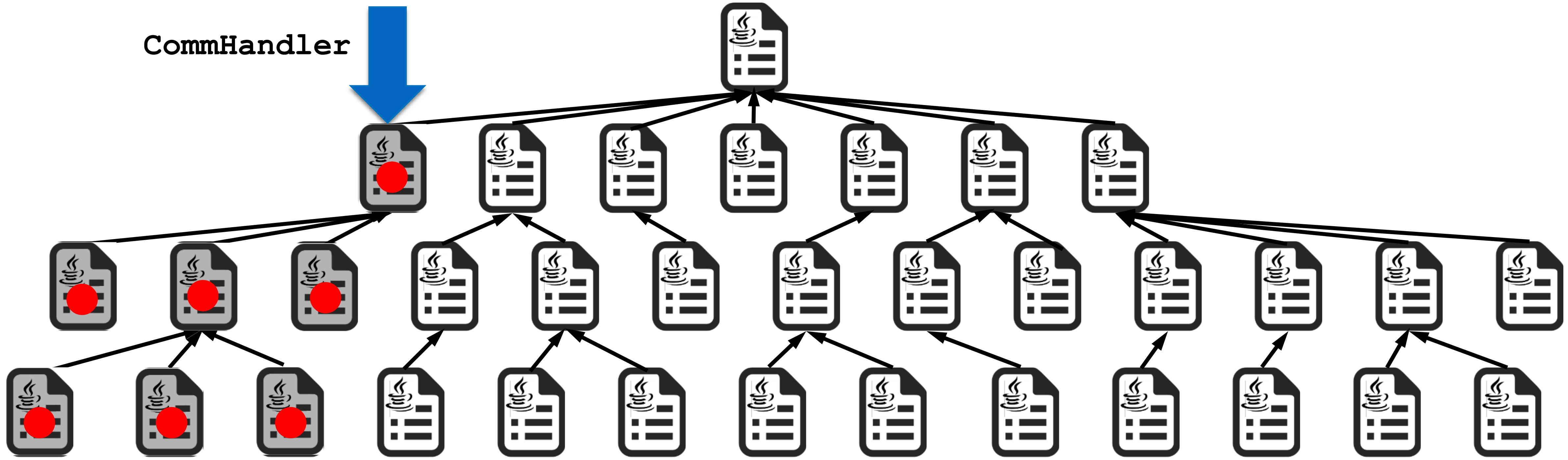


- With RSI:
  - Less classes instrumented
  - Less overhead

```
~~if (this instanceof CommHandler){~~
    // Do something
~~}~~
```

RSI during Instrumentation

- Example: type-specific analysis targeting communication handlers



- With RSI:
  - Less classes instrumented
  - Less overhead

Availability of RSI
vs
Full code coverage

Efficiency
vs
Completeness

RSI during Instrumentation

Instrumentation	Complete RSI available?	Full code coverage?
Compile-time (e.g., AspectJ [1] compile-time weaver)	✗	✗
Load-time in-process (e.g., AspectJ [1] load-time weaver)	✓	✗
Load-time out-of-process (e.g., DiSL [2])	✗	✓
Our approach (DiSL [2])	✓	✓

[1] Kiczales et al. *An overview of AspectJ*. ECOOP'01.

[2] Marek et al. DiSL: *A Domain-specific Language for Bytecode Instrumentation*. AOSD'12.

RSI during Instrumentation

Instrumentation	Complete RSI available?	Full code coverage?	Classloader namespaces?
Compile-time (e.g., AspectJ [1] compile-time weaver)	✗	✗	✗
Load-time in-process (e.g., AspectJ [1] load-time weaver)	✓	✗	✗
Load-time out-of-process (e.g., DiSL [2])	✗	✓	✗
Our approach (DiSL [2])	✓	✓	✓

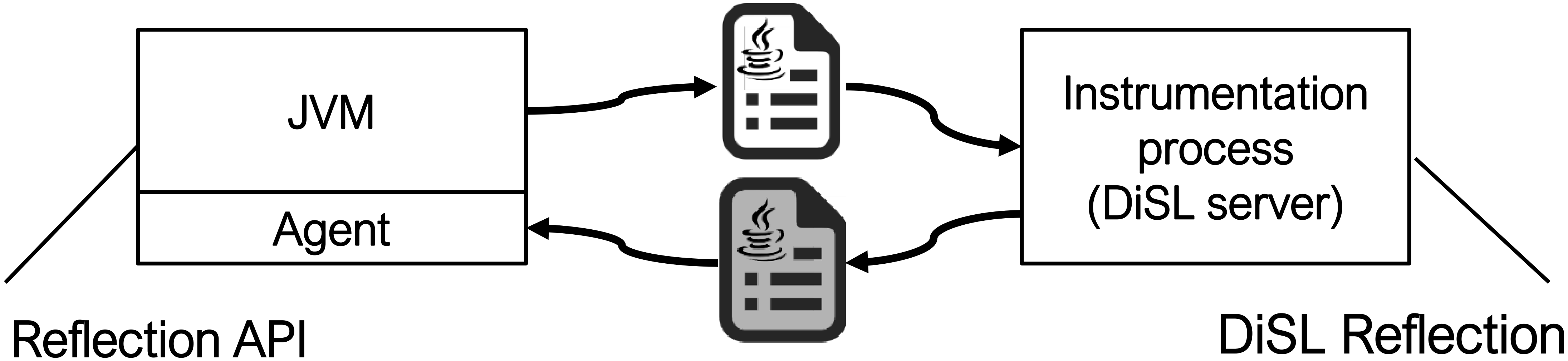
- Challenges:
  - Dynamic class loading
  - Classloader namespaces

Contribution

- We present an extension of DiSL [2] that **accurately reifies complete RSI** in a **separate instrumentation process**
  - Reification of classloader namespaces
  - Complete RSI available upon instrumentation of any class
  - Works with full code coverage

[2] Marek et al. DiSL: *A Domain-specific Language for Bytecode Instrumentation*. AOSD’12.

DiSL Reflection API



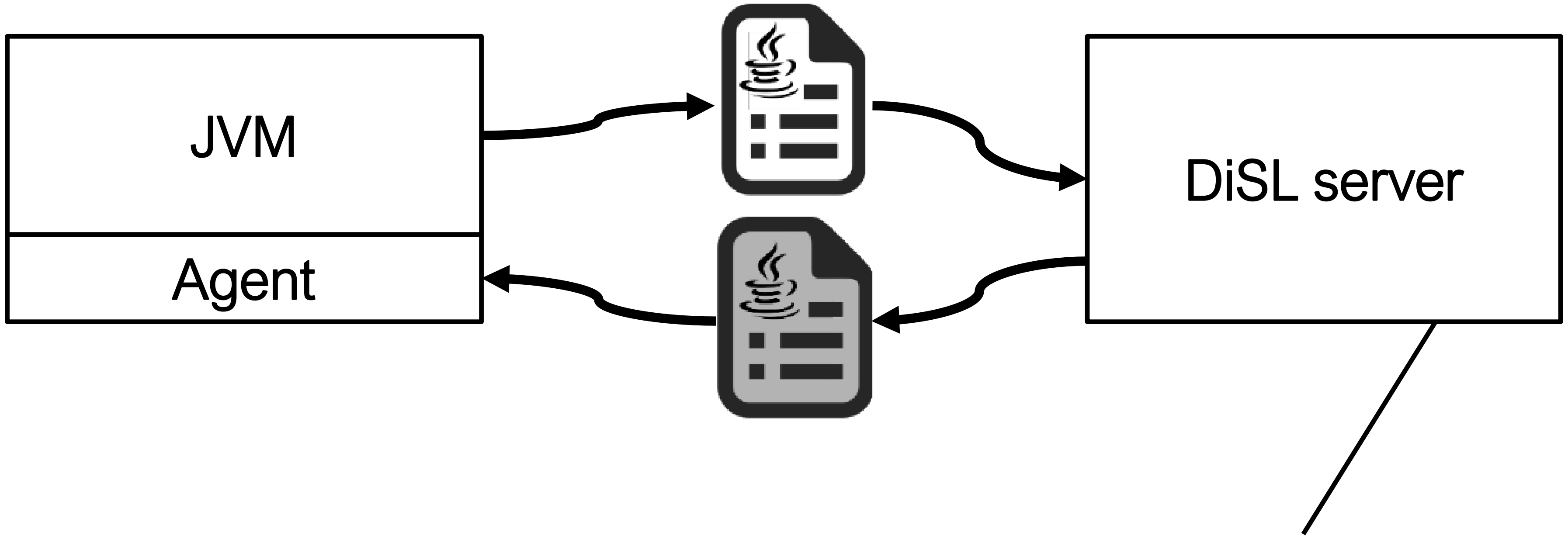
Java Reflection API
`Class, Classloader`

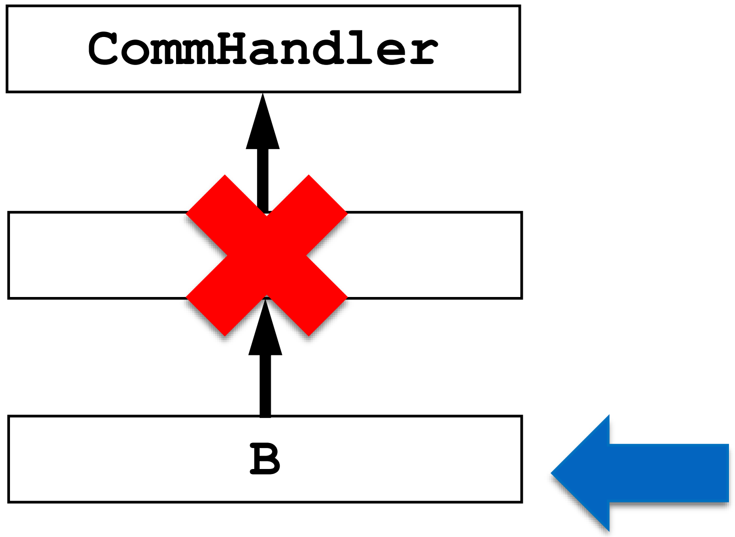
DiSL Reflection API
`DiSLClass, DiSLClassloader`

```
public interface DiSLClassLoader {
    long getID();
    DiSLClass forName(String fullyQualifiedClassName);
    boolean isBootstrapLoader();
    boolean isApplicationLoader();
    ...
}

public interface DiSLClass {
    String getName();
    DiSLClassLoader getClassLoader();
    DiSLClass getSuperclass();
    Stream<DiSLClass> getInterfaces();
    Stream<DiSLClass> getSupertypes();    // All direct and indirect supertypes
    ...
}
```

Forced Loading of Supertypes



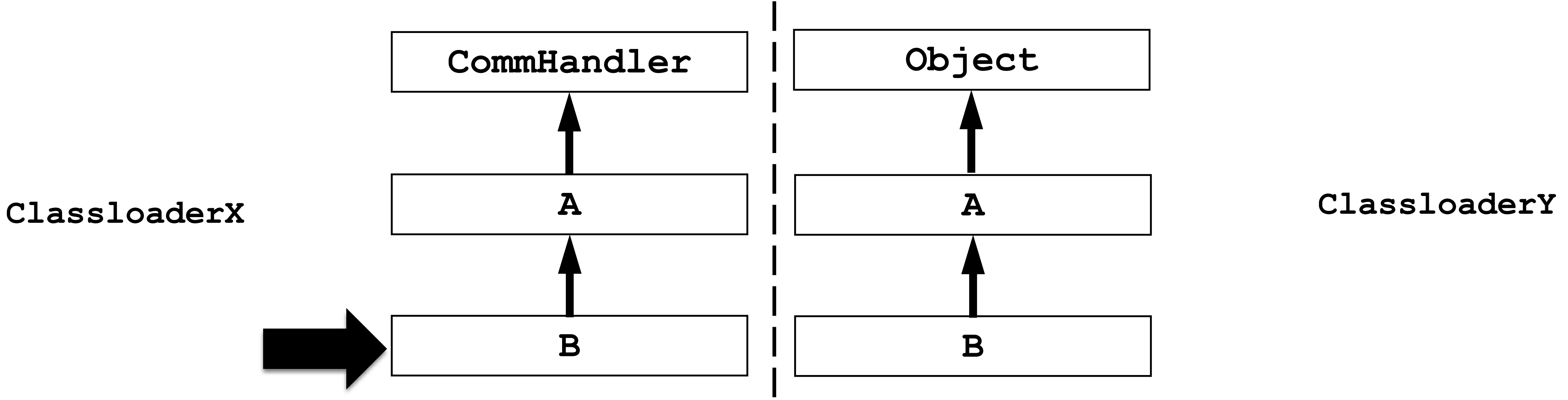


```
public interface DiSLClass {
    String getName();
    DiSLClassLoader getClassLoader();
    DiSLClass getSuperclass();
    Stream<DiSLClass> getInterfaces();
    Stream<DiSLClass> getSupertypes();
    ...
}
```

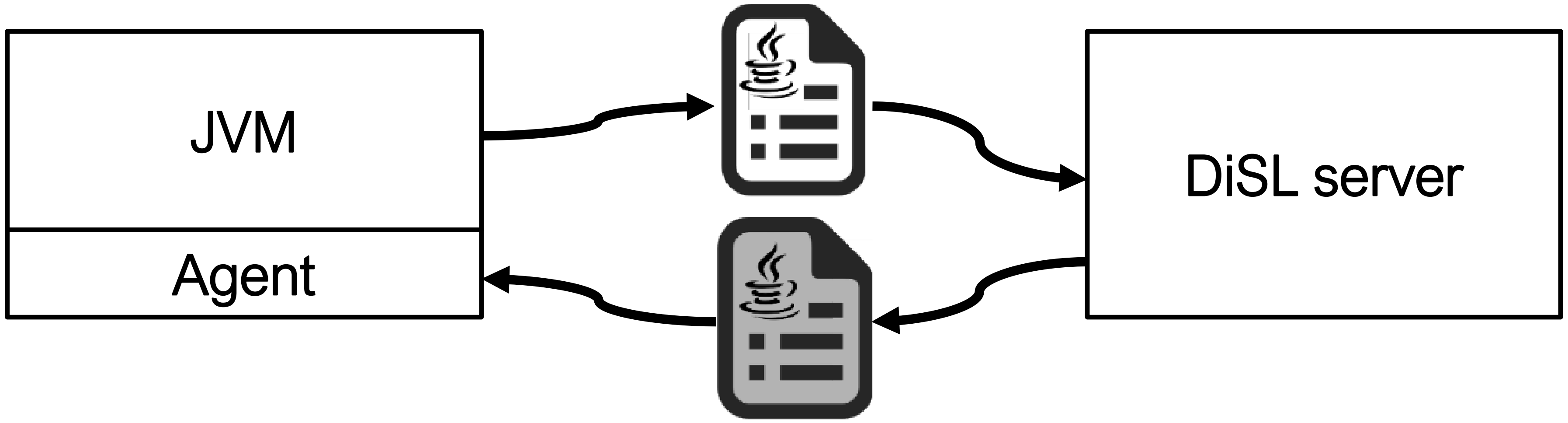
- JVM may load subtypes before supertypes

- Solution: force loading of supertypes

Classloader Namespaces



`<A, ClassloaderX>`
`<B, ClassloaderX>`



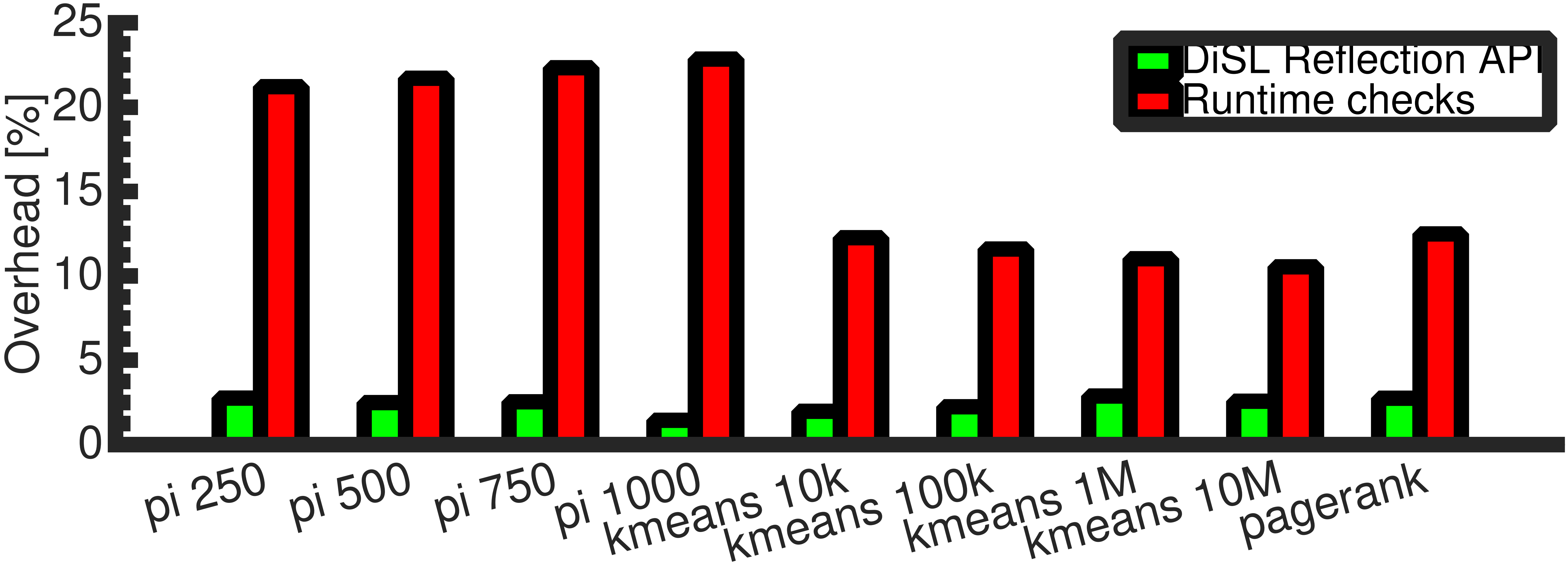
Evaluation

- Task profiling on Apache Spark [3] with *tgp* [4]
- Target workloads:
  - *pi*
  - *kmeans*
  - *pagerank*
- Comparison between:
  - DiSL without Reflection API
  - DiSL with Reflection API

[3] Zaharia et al. *Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing*. NSDI'12.

[4] Rosales et al. *tgp: a Task-Granularity Profiler for the Java Virtual Machine.* APSEC'17.

Evaluation



- High overhead without DiSL Reflection API
  - 10.5%~22.8%

- Reduced overhead with DiSL Reflection API
  - 1.4%~2.9%
  - Max difference: 21.4%

Conclusions

- Extension of DiSL that reifies RSI in a separate instrumentation process
  - Instrumentation process aware of classloader namespaces
  - Accurate and complete RSI available for each loaded class
  - Reconciles complete RSI and full code coverage
- Significant overhead reductions
  in type-specific instrumentations

Thank you for the attention

- DiSL: **`https://disl.ow2.org`**
- Andrea Rosà

  `andrea.rosa@usi.ch`
  `http://www.inf.usi.ch/phd/rosaa`