DiSL: A Domain-specific Language for Efficient and Comprehensive Dynamic Program Analysis

Project led by Prof. Walter Binder

University of Lugano, Switzerland
walter.binder@usi.ch

The DiSL team

- Walter Binder[1]
- Petr Tuma[2]
- Lubomir Bulej[2]
- Lukas Marek[2]
- Yudi Zheng[1,3]
- Danilo Ansaloni[1]
- Aibek Sarimbekov[1]

[1] University of Lugano, Switzerland
[2] Charles University, Czech Republic
[3] Shanghai Jiao Tong University, China

Dynamic program analysis (DPA)

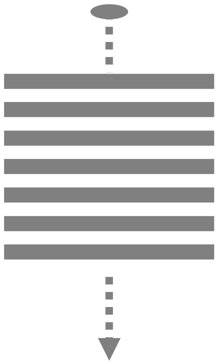
DPA tools

- observe runtime behavior of programs
- often based on code instrumentation
- do not alter the execution of the observed program

Examples

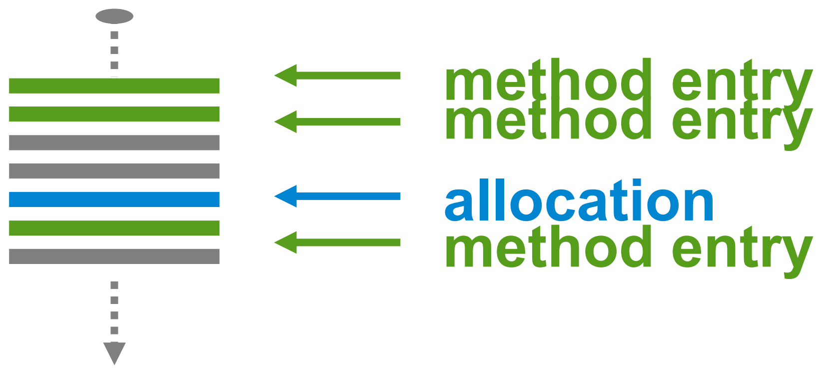
- profilers: execution time, contention, object allocation
- debuggers: memory leaks, data races

Dynamic program analysis (DPA)
Example



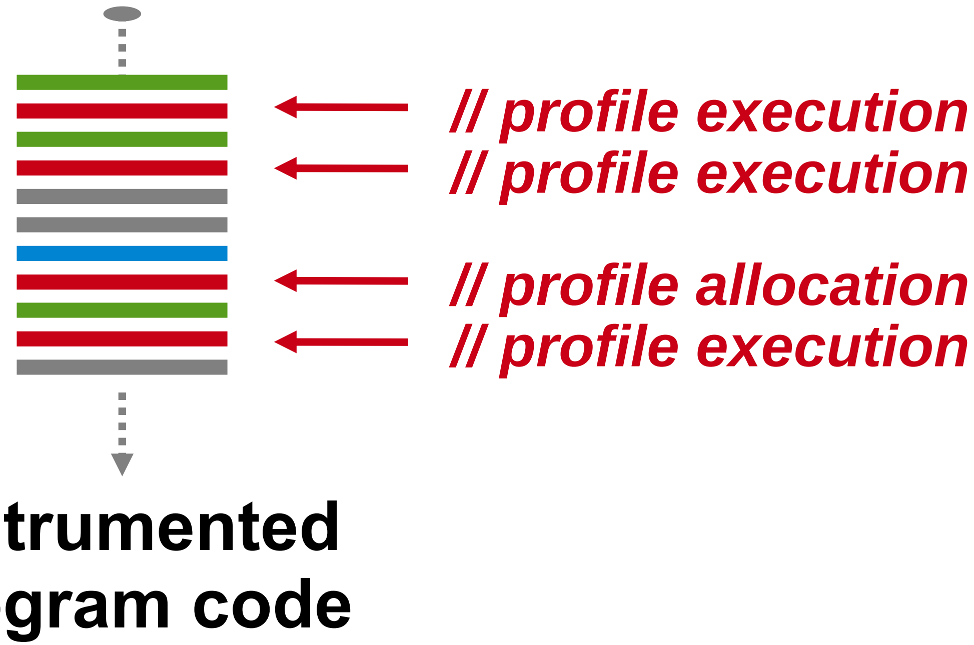
program code

Dynamic program analysis (DPA)
Example



program code

Dynamic program analysis (DPA)
Example



The goal of our research

Improve

- flexibility
- efficiency

of dynamic program analysis tools

The goal of our research

Improve

- flexibility

→ high-level abstractions for rapid specification of custom instrumentations

- efficiency

of dynamic program analysis tools

The goal of our research

Improve

- flexibility

→ high-level abstractions for rapid specification of custom instrumentations

- efficiency

→ advanced optimization techniques for improving runtime performance

of dynamic program analysis tools

The goal of our research

Improve

- flexibility

→ high-level abstractions for rapid specification of custom instrumentations

- efficiency

→ advanced optimization techniques for improving runtime performance

of dynamic program analysis tools

We focus on languages that compile to Java bytecode

Recent publications on the topic

On DiSL

- *DiSL: A Domain-Specific Language for Bytecode Instrumentation* [AOSD'12]
- *Turbo DiSL: Partial Evaluation for High-level Bytecode Instrumentation* [TOOLS'12]
- Java Bytecode Instrumentation Made Easy: The DiSL Framework for Dynamic Program Analysis [APLAS'12]

Using DiSL

- *Accelerating Dynamic Program Analysis on Multicores* [CGO'12]
- *new Scala() instanceof Java: A Comparison of the Memory Behaviour of Java and Scala Programs* [ISMM'12]
- *Challenges for Refinement and Composition of Instrumentations: Position Paper* [SC'12]

Outline

The need for a new DSL

- motivation and state of the art

DiSL: a DSL for instrumentations

- language constructs
- code examples

Evaluation

- source code metrics
- performance

Turbo: a partial evaluator for DiSL

Implementation techniques for DPA
Low-level bytecode instrumentation

Low-level bytecode instrumentation libraries

- commonly used for building DPA tools

Advantages

- high flexibility
- it is possible to minimize overhead

Implementation techniques for DPA
Low-level bytecode instrumentation

Low-level bytecode instrumentation libraries

- commonly used for building DPA tools

Advantages

- high flexibility
- it is possible to minimize overhead

Drawbacks

- error-prone
- high development effort
- resulting tools are difficult to maintain and to extend

Implementation techniques for DPA
Low-level bytecode instrumentation

Object allocation analysis with BCEL

```
public void instrument(Class<?> clazz) throws ClassNotFoundException, IOException {
    JavaClass jc = Repository.lookupClass(clazz);
    String className = jc.getClassName();
    ClassGen cg = new ClassGen(jc);
    ConstantPoolGen cpg = cg.getConstantPool();
    int idx = cpg.addString("New object allocated: ");

    InstructionFactory factory = new InstructionFactory(cg);

    for(Method m : cg.getMethods()) {
        MethodGen mg = new MethodGen(m, className, cpg);
        InstructionList il = mg.getInstructionList();
        if(il != null) {
            int pendingNew = 0;
            for(InstructionHandle ih : il.getInstructionHandles()) {
                Instruction i = ih.getInstruction();
                if(i.getOpcode() == 187) { //NEW
                    InstructionList newIL = new InstructionList();
                    newIL.append(new DUP());
                    newIL.append(new ASTORE(getFreeIdx(++pendingNew)));
                    il.append(ih, newIL);
                }
                else if(i.getOpcode() == 183) { //INVOKESPECIAL
                    if(((INVOKESPECIAL)i).getMethodName(cpg).equals("<init>")) {
                        if(pendingNew > 0) {
                            InstructionList newIL = new InstructionList();
                            newIL.append(factory.createGetStatic("java/lang/System", "out", Type.getType(java.io.PrintStream.class)));
                            newIL.append(factory.createNew((ObjectType)Type.getType(java.lang.StringBuilder.class)));
                            newIL.append(new DUP());
                            newIL.append(new LDC(idx));
                            newIL.append(factory.createInvoke("java.lang.StringBuilder", "<init>", Type.VOID, new Type[] { Type.STRING }, Constants.INVOKESPECIAL));
                            newIL.append(new ALOAD(getStoredIdx(pendingNew--)));
                            newIL.append(factory.createInvoke("java.lang.System", "identityHashCode", Type.INT, new Type[] { Type.OBJECT}, Constants.INVOKESTATIC));
                            newIL.append(factory.createInvoke("java.lang.StringBuilder", "append", Type.getType(java.lang.StringBuilder.class), new Type[] { Type.INT }, Constants.INVOKEVIRTUAL));
                            newIL.append(factory.createInvoke("java.lang.StringBuilder", "toString", Type.STRING, Type.NO_ARGS, Constants.INVOKEVIRTUAL));
                            newIL.append(factory.createInvoke("java.io.PrintStream", "println", Type.VOID, new Type[] { Type.STRING }, Constants.INVOKEVIRTUAL));

                            il.append(ih, newIL);
                        }
                    }
                }
            }
        }
        mg.setMaxLocals();
        mg.setMaxStack();
        cg.replaceMethod(m, mg.getMethod());
    }

    return cg.getJavaClass().getBytes();
}
```

Implementation techniques for DPA
Aspect-oriented programming (AOP)

AOP allows users to insert fragments of code at identifiable execution points

- method invocations
- method bodies
- field accesses
- …

Implementation techniques for DPA
Aspect-oriented programming (AOP)

Object allocation analysis with AspectJ

```
aspect AllocAnalysis {
  after() returning(Object o) : call(*.new(..)) {
    System.out.println("New object allocated: "
                       + System.identityHashCode(o));
  }
}
```

Implementation techniques for DPA
Prevailing AOP frameworks

Advantages

- instrumentations are specified at a high abstraction level
  - does not require detailed knowledge of the Java bytecode
  - allows rapid development of custom DPA tools

Implementation techniques for DPA
Prevailing AOP frameworks

Advantages

- instrumentations are specified at a high abstraction level
  - does not require detailed knowledge of the Java bytecode
  - allows rapid development of custom DPA tools

Drawbacks

- limited flexibility
  - cannot instrument basic blocks of code and individual bytecodes
- possibly high runtime overhead
  - no user-defined static analysis at instrumentation time
  - inefficient access to static and dynamic context information

A new DSL for DPA

Design goals

- allow high-level specification of DPA tools
  - compact tools, easy to develop and to extend

→ **similar to AOP languages**

- maximize flexibility
  - instrument also basic blocks of code and individual bytecodes
- maximize efficiency
  - support for static analysis at instrumentation time
  - efficient access to static and dynamic context information

→ **similar to low-level instrumentation libraries**

DiSL at a glance

Design choices

- AOP-inspired DSL for rapid development of DPA tools
  - annotation syntax similar to AspectJ
  - high-level support for advanced optimizations
- language hosted in Java
  - compatible with standard compilers and JVMs
- support for comprehensive analysis
  - any method with a bytecode representation can be instrumented

DiSL at a glance

Language constructs

- markers and snippets
- static and dynamic context
- synthetic local vaiables and thread-local variables
- scope restrictions and guards

Motivating example

- DPA tool for hotspot detection
  - count number of executed basic-blocks of code and bytecodes
  - count allocated objects

DiSL markers

Each bytecode region can be instrumented

- DiSL provides an extensible library of bytecode ***markers*** to select custom regions of bytecodes
- included markers
  - method body
  - basic block
  - individual bytecode
  - exception handler

DiSL snippets

DiSL classes specify code ***snippets***

- inlined at marked code regions
- marked by annotations
  - @Before
  - @After
  - @AfterReturning
  - @AfterThrowing
- allow precise control over the insertion order
- can access any static and dynamic context information
- must not throw any exception

DiSL markers and snippets
Example

```java
public class DiSLClass {

    @Before(marker = BasicBlockMarker.class)
    public static void onBB() {
        Profile.profileBB(); // count number of exec basic blocks of code
    }

    @AfterReturning(marker = BytecodeMarker.class, args = "new")
    public static void onAlloc() {
        Profile.profileAlloc(); // count number of allocated objects
    }

}
```

Accessing context information

Challenges

- most DPA tools frequently access context information
  - access to such information must be efficient
  - collection of custom context information must be supported
- insertion of extra fields should be avoided
  - inserted fields could be visible through reflection, potentially interfering with the execution of the observed program
  - class redefinition does not support field insertion

Static context information

DiSL can compute custom static information during instrumentation, storing results in the constant pool

- no need to add extra fields
- predefined static contexts
  - MethodStaticContext
  - BasicBlockStaticContext
  - BytecodeStaticContext
  - DataFlowStaticContext
- support for custom static contexts
- allow static analysis at instrumentation time

Static context information
Example

```
public class DiSLClass {

    @Before(marker = BasicBlockMarker.class)
    public static void onBB(MethodStaticContext msc,
                            BasicBlockStaticContext bbsc) {
        Profile.profileBB(
            msc.thisMethodFullName(),   // full method name
            bbsc.getBBIndex(),          // basic block index (int value)
            bbsc.getBBSize()            // bytecodes in the BB (int value)
        );
    }

}
```

Dynamic context information

DiSL provides constructs to

- access local variables
- inspect the operand stack

Calls to the dynamic context API inline bytecode sequences to retrieve the desired values

- the developer must know how to access the data
- may require custom static context information

Dynamic context information
Example

```
public class DiSLClass {

    @AfterReturning(marker = BytecodeMarker.class, args = "new”)
    public static void onAlloc(DynamicContext dc) {
        // access allocated object
        Object allocObj = dc.getStackValue(0, Object.class);
        Profile.profileAlloc(allocObj); // profile allocated object
    }

}
```

Scope and guards

Two complementary mechanisms for restricting the application of snippets

- ***scope***: based on method signature matching
- ***guards***: based on static evaluation of conditionals

No need for expensive runtime checks

Scope and guards
Example

```
public class DiSLClass {
  @Before(marker = BasicBlockMarker.class,
          scope = “TargetClass.*”,        // constrain instrumentation
          guard = LoopGuard.class)       // constrain instrumentation
  public static void onBB(BasicBlockStaticContext bbsc) {
    Profile.profileBytecodes(bbsc.getBBSize());
  }
}

public class LoopGuard {
  @GuardMethod
  public static boolean isApplicable(BasicBlockStaticContext bbsc) {
    return bbsc.isFirstOfLoop(); // instrument only first BBs of loops
  }
}
```

Synthetic local variables

An efficient mechanism for passing data between snippets inserted into the same method body

- accessed through static fields annotated with @SyntheticLocal
- mapped to local variables in instrumented methods
- require snippet inlining

Synthetic local variables
Example

```
public class DiSLClass {

    @SyntheticLocal static long sizeSL;

    @Before(marker = BasicBlockMarker.class)
    public static void onBB(BasicBlockStaticContext bbsc) {
        sizeSL += bbsc.getBBSize();
    }

    @After(marker = BodyMarker.class)
    public static void onMethodCompletion() {
        Profile.profileBytecodes(sizeSL);
    }

}
```

Thread-local variables

DiSL supports efficient thread-local variables

- accessed through fields annotated with @ThreadLocal
- mapped to added instance fields in java.lang.Thread

Thread-local variables
Example

```
public class DiSLClass {

    @ThreadLocal static Profile profileTL;

    @Before(marker = BodyMarker.class, order = 1)
    public static void onMethodEntry() {
        if (profileTL == null) profileTL = new Profile();
    }

    @Before(marker = BasicBlockMarker.class, order = 0)
    public static void onBB(BasicBlockStaticContext bbsc) {
        profileTL.profileBytecodes(bbsc.getBBSize());
    }

}
```

Evaluation
Case study: Senseo

DPA tool for code comprehension and profiling

- originally implemented in AspectJ
- *Exploiting Dynamic Information in IDEs Improves Speed and Correctness of Software Maintenance Tasks* [TSE'12]
  D. Röthlisberger, M. Härry, W. Binder, P. Moret,
  D. Ansaloni, A. Villazón, and O. Nierstrasz

Evaluation
Case study: Senseo

Collects calling-context-sensitive dynamic metrics

- number of method executions
- number of allocated objects
- runtime types of arguments and return values

We recasted Senseo in DiSL

- improved runtime performance and coverage
- improved granularity → basic block metrics

Evaluation
Case study: Senseo

Observed workloads

- DaCapo 9.12 benchmarks

Execution environment

- 3.0 GHz Intel Core 2 Quad Q9650 with 8 GB RAM
- Ubuntu 10.04 64-bit
- 1.6.0_30 Hotspot Server VM (64-bit) with 7 GB heap size

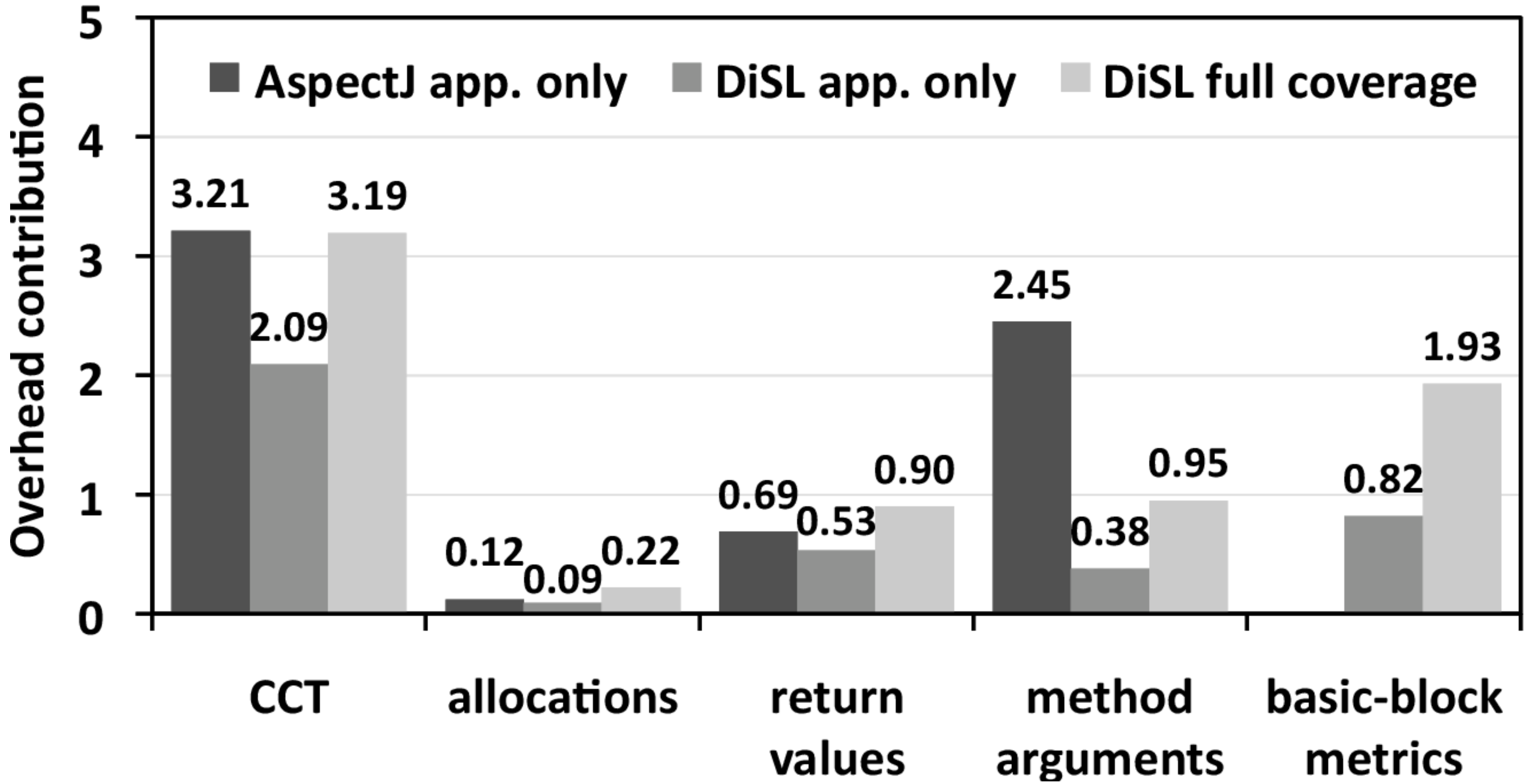
Senseo
Performance evaluation

	SenseoAJ	SenseoDiSL		Senseo2	
	app. only	app. only	full cov.	app. only	full cov.
Instr. time [s]	54.97	43.28	134.17	65.61	174.73

Three implementations

- **SenseoAJ**: original implementation, based on AspectJ
- **SenseoDiSL**: equivalent to SenseoAJ, based on DiSL
- **Senseo2**: based on DiSL, additionally profiles basic-block metrics (not possible with AspectJ)

Senseo
Performance evaluation



- baseline: uninstrumented benchmarks

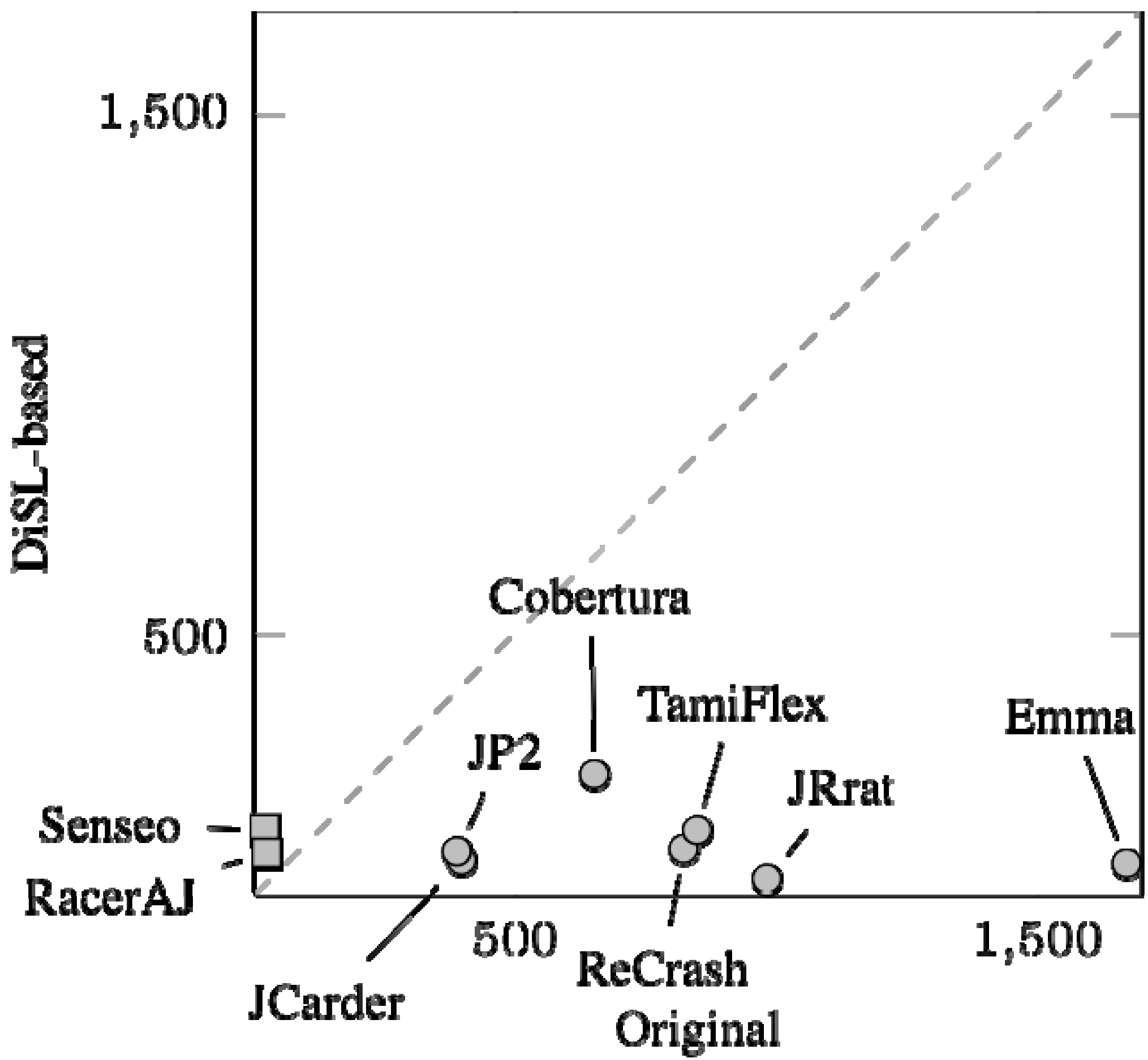
Senseo
Tool development effort

	SenseoDiSL	SenseoAJ	SenseoASM
Physical lines-of-code	74	44	489
Logical lines-of-code	44	19	338

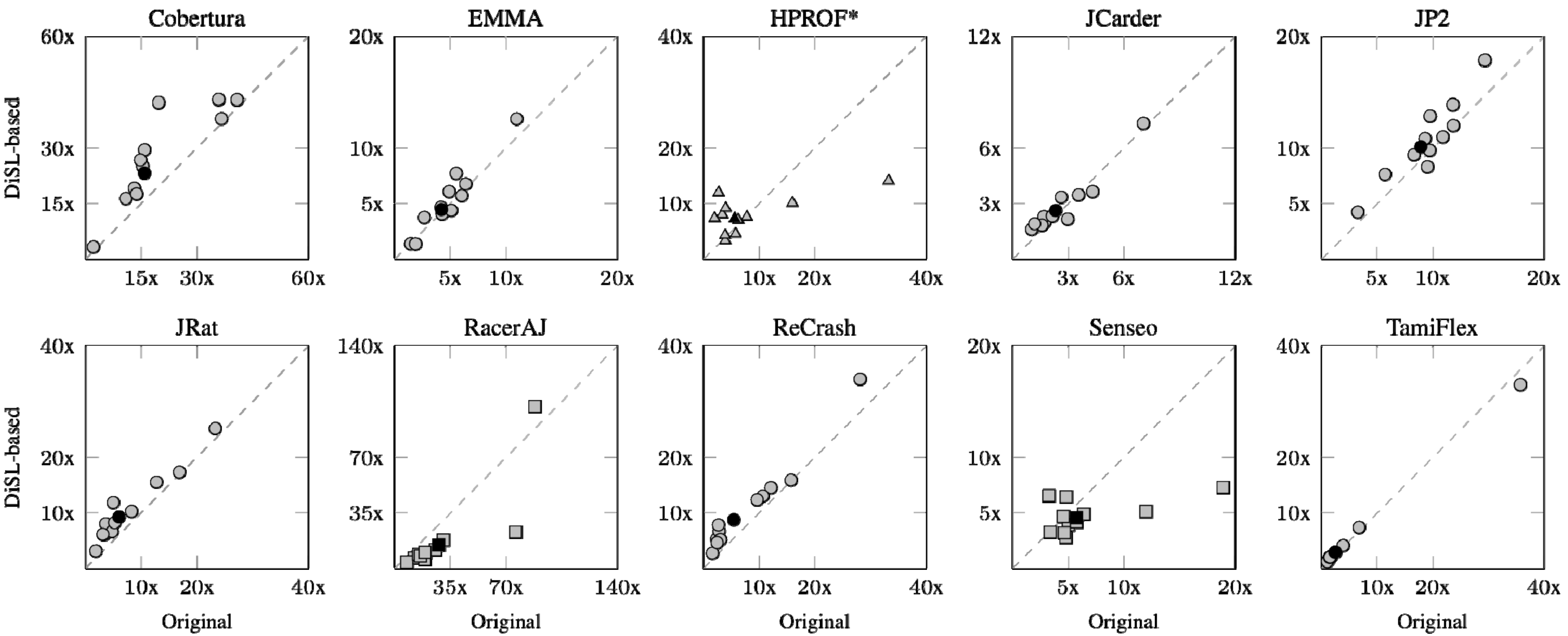
Three implementations

- **SenseoAJ**: original implementation, based on AspectJ
- **SenseoDiSL**: equivalent to SenseoAJ, based on DiSL
- **SenseoASM**: equivalent to SenseoAJ, based on ASM

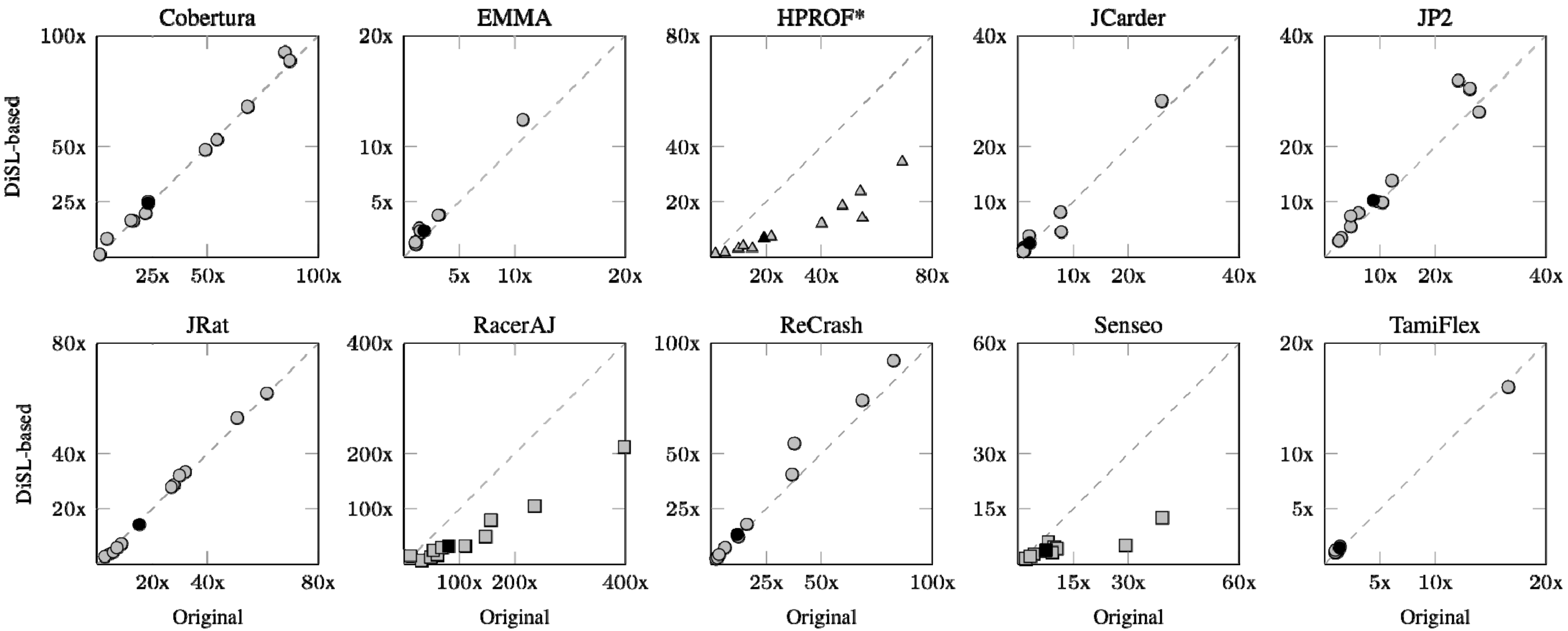
Recasted tools: logical lines of code



Recasted tools: startup performance



Recasted tools: steady-state perf.



Limitations of DiSL

Expressions to be evaluated at instrumentation time must be wrapped in separate classes

- static context analyzers
- guards

Case study #2

Execution trace profiling

- profile the execution of basic blocks of code, creating a custom basic block ID
- profile class initialization

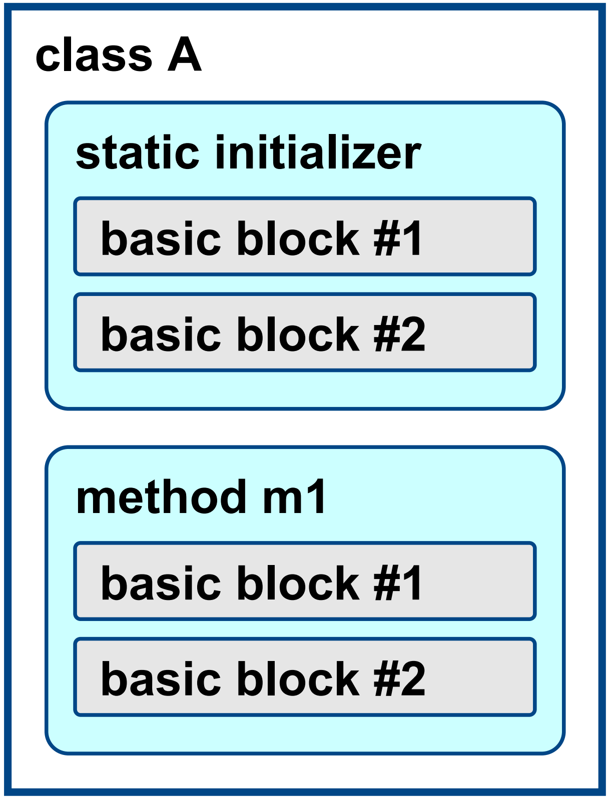
Execution trace profiling
Naïve implementation

```
public class ExecutionTraceProfiler {

  @Before(marker = BasicBlockMarker.class)
  static void onBB(BasicBlockStaticContext bbsc, MethodStaticContext msc) {

    if (bbsc.getBBindex() == 0 && msc.thisMethodName().equals("<clinit>")) {
      ... /* profile class initialization */
    }

    String bbID = msc.thisMethodFullName() + ":" + String.valueOf(bbsc.getBBindex());
    ... /* profile basic block entry */
  }

}
```

Execution trace profiling
Naïve implementation

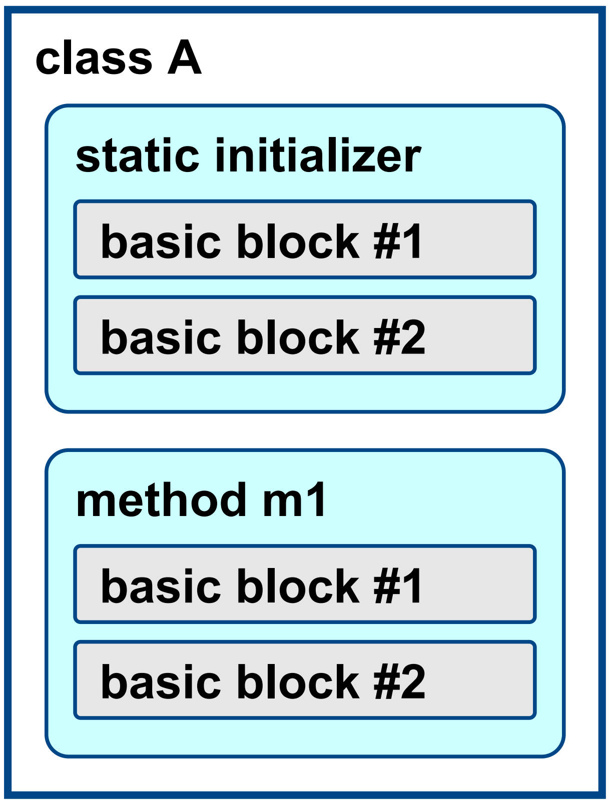
```
public class ExecutionTraceProfiler {

  @Before(marker = BasicBlockMarker.class)
  static void onBB(BasicBlockStaticContext bbsc, MethodStaticContext msc) {

    if (bbsc.getBBindex() == 0 && msc.thisMethodName().equals("<clinit>")) {
      ... /* profile class initialization */
    }

    String bbID = msc.thisMethodFullName() + ":" + String.valueOf(bbsc.getBBindex());
    ... /* profile basic block entry */
  }

}
```

Execution trace profiling
Naïve implementation



observed code

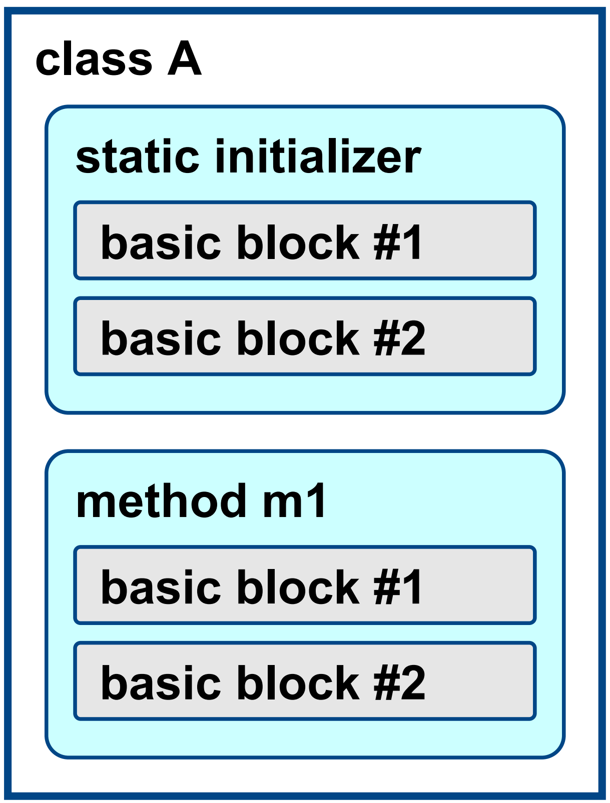
Execution trace profiling
Naïve implementation



observed code

```
public class ExecutionTraceProfiler {
  @Before(marker = BasicBlockMarker.class)
  static void onBB(BasicBlockStaticContext bbsc, MethodStaticContext msc) {
    if (bbsc.getBBindex() == 0 && msc.thisMethodName().equals("<clinit>")) {
      ... /* profile class initialization */
    }
    String bbID = msc.thisMethodFullName() + ":" + String.valueOf(bbsc.getBBindex());
    ... /* profile basic block entry */
  }
}
```

Execution trace profiling
Naïve implementation



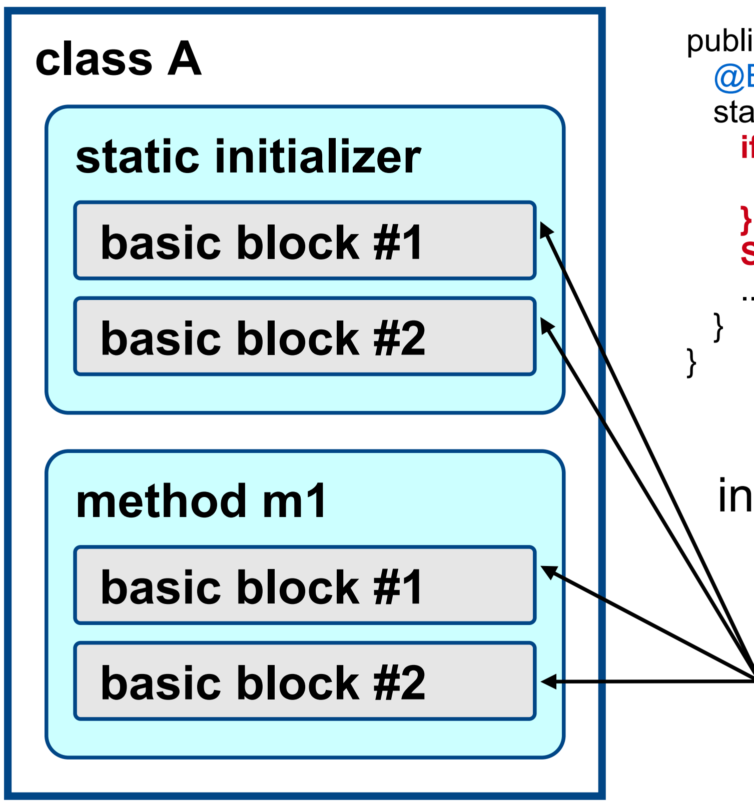
observed code

```
public class ExecutionTraceProfiler {
  @Before(marker = BasicBlockMarker.class)
  static void onBB(BasicBlockStaticContext bbsc, MethodStaticContext msc) {
    if (bbsc.getBBindex() == 0 && msc.thisMethodName().equals("<clinit>")) {
      ... /* profile class initialization */
    }
    String bbID = msc.thisMethodFullName() + ":" + String.valueOf(bbsc.getBBindex());
    ... /* profile basic block entry */
  }
}
```

instantiation

```
if($bbIndex$ == 0 && $methodName$.equals(“<clinit>”))
   … /* profile class initialization */
String bbID = $methodName$ + “:” + $bbIndex$;
… /* profile basic block entry */
```

Execution trace profiling
Naïve implementation



observed code

```
public class ExecutionTraceProfiler {
  @Before(marker = BasicBlockMarker.class)
  static void onBB(BasicBlockStaticContext bbsc, MethodStaticContext msc) {
    if (bbsc.getBBindex() == 0 && msc.thisMethodName().equals("<clinit>")) {
      ... /* profile class initialization */
    }
    String bbID = msc.thisMethodFullName() + ":" + String.valueOf(bbsc.getBBindex());
    ... /* profile basic block entry */
  }
}
```

instantiation

inlining

```
if($bbIndex$ == 0 && $methodName$.equals("<clinit>"))
   ... /* profile class initialization */
String bbID = $methodName$ + ":" + $bbIndex$;
... /* profile basic block entry */
```

Execution trace profiling
Naïve implementation

class A

- **static initializer**
  - **basic block #1**
  - **basic block #2**
- **method m1**
  - **basic block #1**
  - **basic block #2**

observed code

```
if(0 == 0 && "<clinit>".equals("<clinit>"))
   … /* profile class initialization */
String bbID = "<clinit>" + ":" + 0;
... /* profile basic block entry */
```

```
if(1 == 0 && "<clinit>".equals("<clinit>"))
   … /* profile class initialization */
String bbID = "<clinit>" + ":" + 1;
... /* profile basic block entry */
```

```
if(0 == 0 && "m1".equals("<clinit>"))
   … /* profile class initialization */
String bbID = "m1" + ":" + 0;
... /* profile basic block entry */
```

```
if(1 == 0 && "m1".equals("<clinit>"))
   … /* profile class initialization */
String bbID = "m1" + ":" + 1;
... /* profile basic block entry */
```

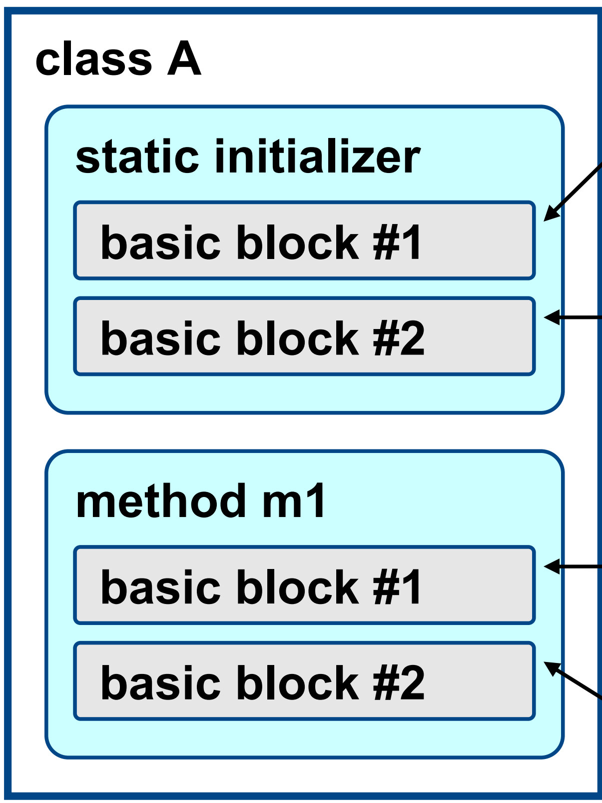
Execution trace profiling
DiSL-optimized

```
public class ExecutionTraceProfiler {
  @Before(marker = BasicBlockMarker.class)
  static void onBB(CustomBasicBlockStaticContext cbbsc) {
    String bbID = cbbsc.thisBBID();
    ... /* profile basic block entry */
  }
  ...
}

...

public class CustomBasicBlockStaticContext extends BasicBlockStaticContext {
  public String thisBBID() {
    String methodFullName = staticContextData.getClassNode().name
                            + "." + staticContextData.getMethodNode().name;
    return methodFullName + ":" + String.valueOf(getBBindex());
  }
}
```

Execution trace profiling
DiSL-optimized

```
public class ExecutionTraceProfiler {
  ...

  @Before(marker = BasicBlockMarker.class, guard = ClassInitGuard.class)
  static void onClassInit() {
    ... /* profile class initialization */
  }
}

public class ClassInitGuard {
  @GuardMethod
  static boolean evalGuard(BasicBlockStaticContext bbsc, MethodStaticContext msc) {
    return (bbsc.getBBindex() == 0) && msc.thisMethodName().equals("<clinit>");
  }
}
```

Execution trace profiling
Naïve implementation



observed code

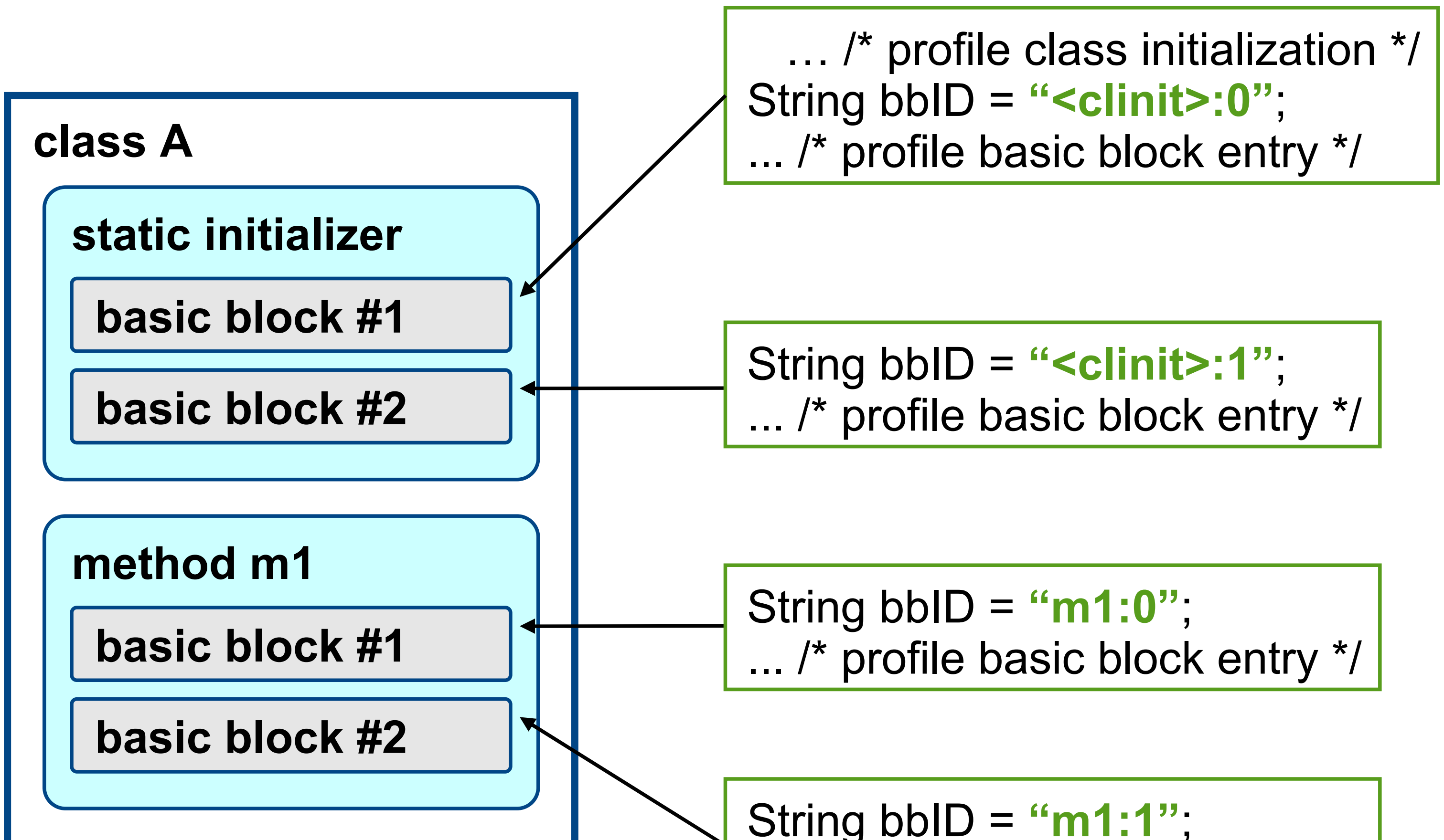
```
if(0 == 0 && "<clinit>".equals("<clinit>"))
   … /* profile class initialization */
String bbID = "<clinit>" + ":" + 0;
... /* profile basic block entry */
```

```
if(1 == 0 && "<clinit>".equals("<clinit>"))
   … /* profile class initialization */
String bbID = "<clinit>" + ":" + 1;
... /* profile basic block entry */
```

```
if(0 == 0 && "m1".equals("<clinit>"))
   … /* profile class initialization */
String bbID = "m1" + ":" + 0;
... /* profile basic block entry */
```

```
if(1 == 0 && "m1".equals("<clinit>"))
   … /* profile class initialization */
String bbID = "m1" + ":" + 1;
... /* profile basic block entry */
```

Execution trace profiling
DiSL-optimized



observed code

Turbo: a partial evaluator for DiSL

Y. Zheng[1], D. Ansaloni[2], L. Marek[3], A. Sewe[4], W. Binder[2], A. Villazon[5], P. Tuma[3], Z. Qi[1], M. Mezini[4]

Turbo DiSL: Partial Evaluation for High-level Bytecode Instrumentation

[1] Shanghai Jiao Tong University, China
[2] University of Lugano, Switzerland
[3] Charles University, Czech Republic
[4] Technische Universität Darmstadt, Germany
[5] Universidad Privada Boliviana, Bolivia

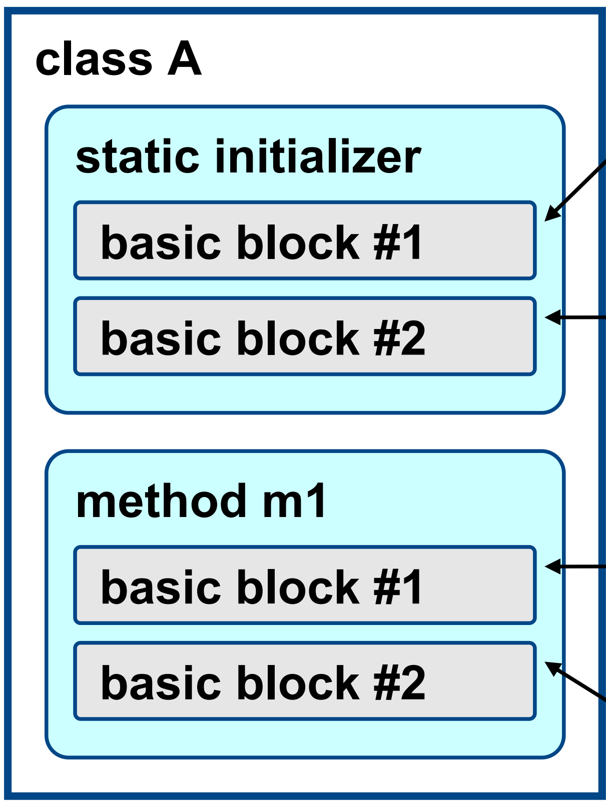
TOOLS'12

Turbo: a partial evaluator for DiSL

Turbo optimizes inlined snippets, performing

- symbolic execution with constant propagation
- conditional reduction
- dead code removal

Execution trace profiling
Naïve implementation



observed code

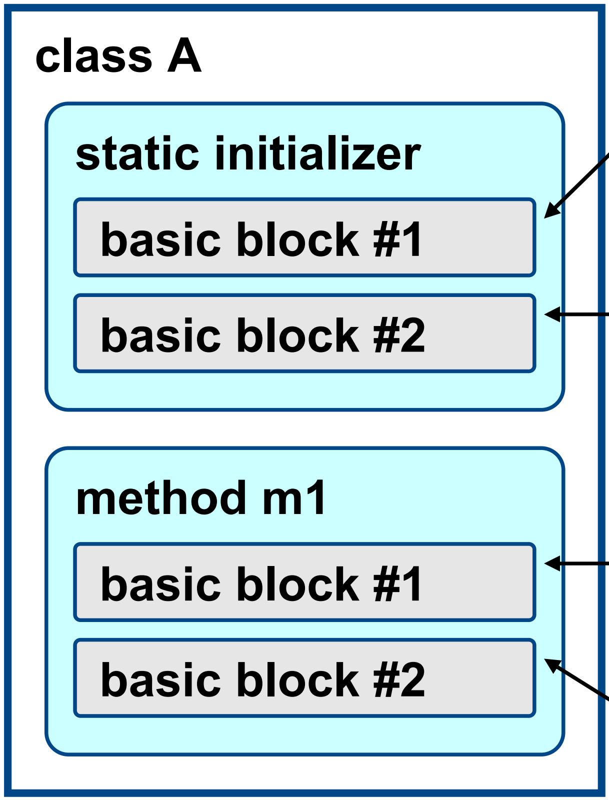
```
if(0 == 0 && "<clinit>".equals("<clinit>"))
   … /* profile class initialization */
String bbID = "<clinit>" + ":" + 0;
... /* profile basic block entry */
```

```
if(1 == 0 && "<clinit>".equals("<clinit>"))
   … /* profile class initialization */
String bbID = "<clinit>" + ":" + 1;
... /* profile basic block entry */
```

```
if(0 == 0 && "m1".equals("<clinit>"))
   … /* profile class initialization */
String bbID = "m1" + ":" + 0;
... /* profile basic block entry */
```

```
if(1 == 0 && "m1".equals("<clinit>"))
   … /* profile class initialization */
String bbID = "m1" + ":" + 1;
... /* profile basic block entry */
```

Turbo: a partial evaluator for DiSL
Partial evaluation: constant propagation



observed code

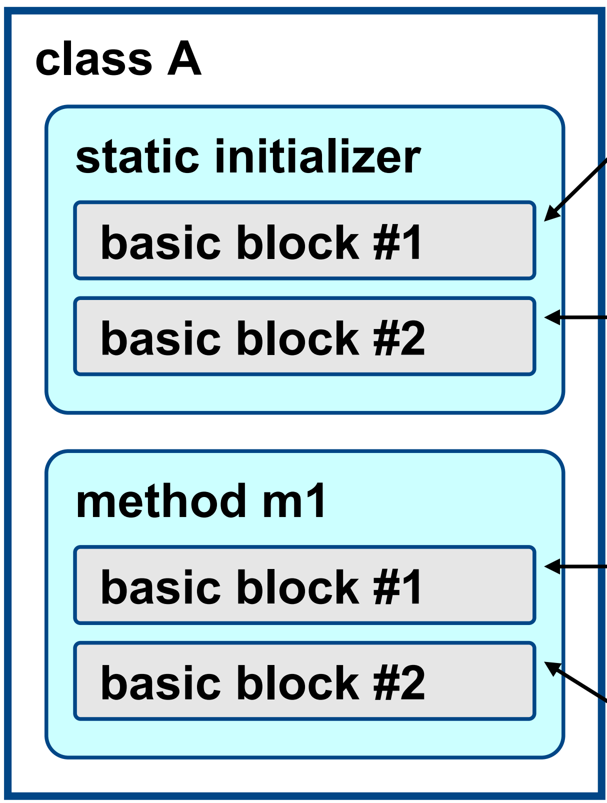
```
if(0 == 0 && "<clinit>".equals("<clinit>"))
   … /* profile class initialization */
String bbID = "<clinit>" + ":" + 0;
... /* profile basic block entry */
```

```
if(1 == 0 && "<clinit>".equals("<clinit>"))
   … /* profile class initialization */
String bbID = "<clinit>" + ":" + 1;
... /* profile basic block entry */
```

```
if(0 == 0 && "m1".equals("<clinit>"))
   … /* profile class initialization */
String bbID = "m1" + ":" + 0;
... /* profile basic block entry */
```

```
if(1 == 0 && "m1".equals("<clinit>"))
   … /* profile class initialization */
String bbID = "m1" + ":" + 1;
... /* profile basic block entry */
```

Turbo: a partial evaluator for DiSL
Partial evaluation: constant propagation



observed code

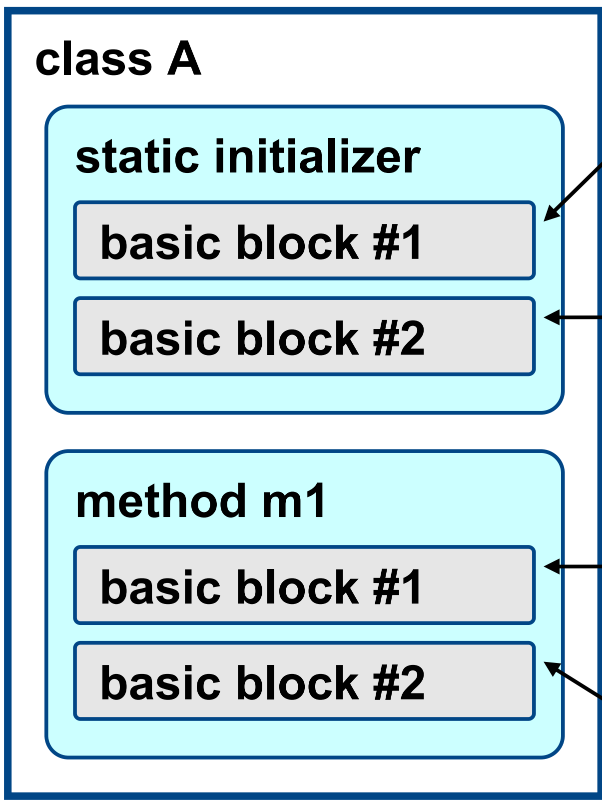
```
if(TRUE && TRUE)
   … /* profile class initialization */
String bbID = "<clinit>:" + "0";
... /* profile basic block entry */
```

```
if(FALSE && TRUE)
   … /* profile class initialization */
String bbID = "<clinit>:" + "1";
... /* profile basic block entry */
```

```
if(TRUE && FALSE)
   … /* profile class initialization */
String bbID = "m1:"+ "0";
... /* profile basic block entry */
```

```
if(FALSE && FALSE)
   … /* profile class initialization */
String bbID = "m1:" + "1";
... /* profile basic block entry */
```

Turbo: a partial evaluator for DiSL
Partial evaluation: conditional reduction



observed code

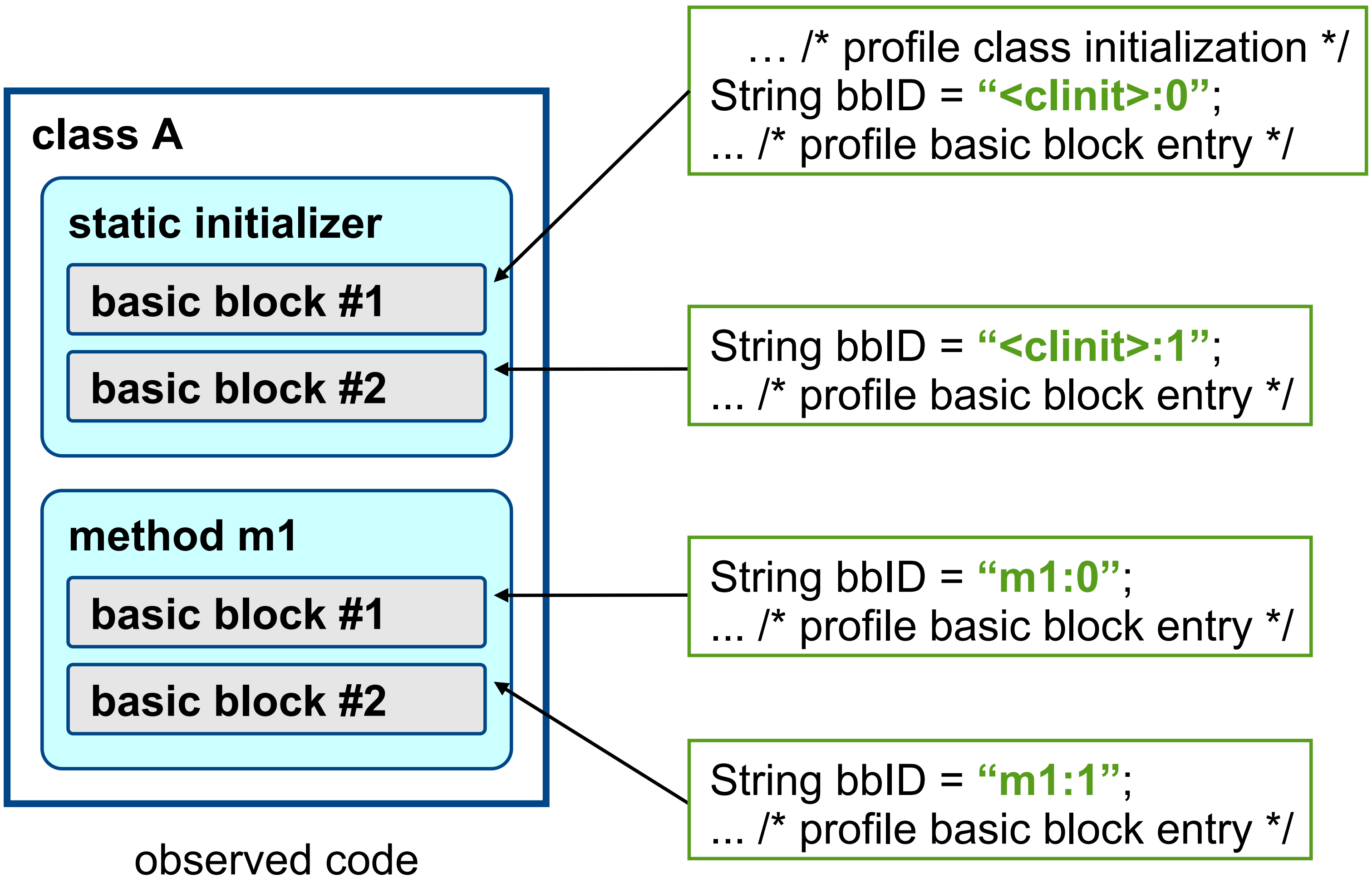
```
if(TRUE)
   … /* profile class initialization */
String bbID = "<clinit>:0";
... /* profile basic block entry */
```

```
if(FALSE)
   … /* profile class initialization */
String bbID = "<clinit>:1";
... /* profile basic block entry */
```

```
if(FALSE)
   … /* profile class initialization */
String bbID = "m1:0";
... /* profile basic block entry */
```

```
if(FALSE)
   … /* profile class initialization */
String bbID = "m1:1";
... /* profile basic block entry */
```

Turbo: a partial evaluator for DiSL
Partial evaluation: optimized code



Turbo
Performance evaluation

Baseline

- DiSL naïve: execution trace profiling in a single snippet

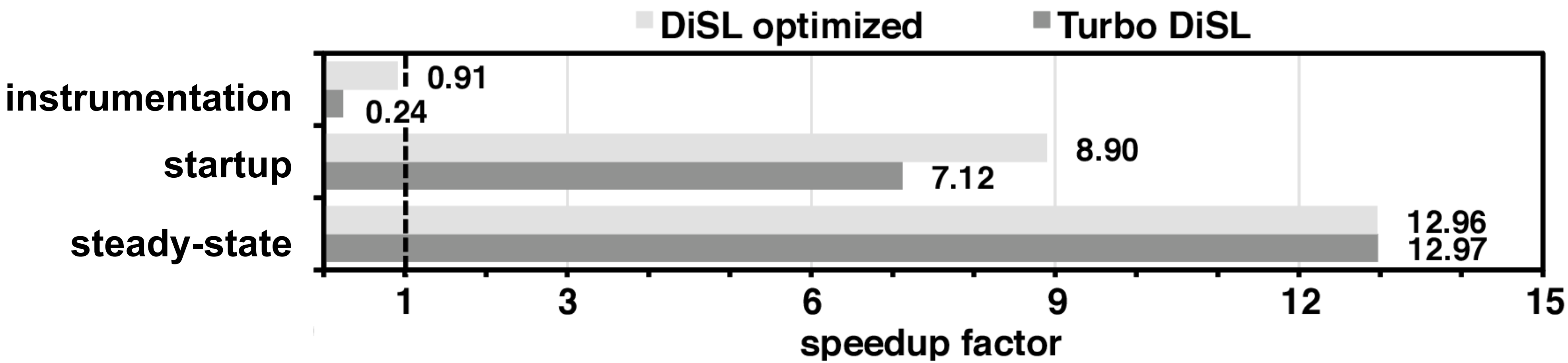
Comparison between

- DiSL optimized: functionally equivalent to DiSL naïve, with a custom static context analyzer and a guard
- Turbo DiSL: same as DiSL naïve, with Turbo

Observed metrics

- instrumentation time
- startup performance
- steady-state performance

Turbo
Performance evaluation



Baseline

- DiSL naïve: execution trace profiling in a single snippet

Observed metrics

- instrumentation time
- startup performance
- steady-state performance

Conclusion

DiSL: a DSL for instrumentation

- allows rapid, high-level specification of custom DPA tools

→ similar to AOP languages

- flexible and efficient instrumentation framework
  → similar to low-level bytecode engineering libraries

Ongoing research

- language constructs to improve modularity and reuse
- declarative, high-level languages that compile to DiSL
- high-level constructs for parallelizing the execution of analysis code
- out-of-process analysis to reduce measurement perturbation

Get DiSL!

http://disl.ow2.org/