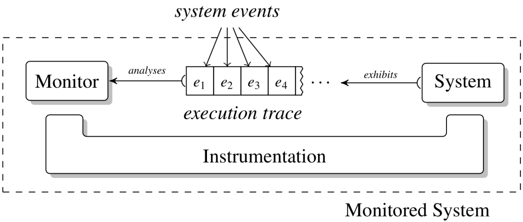
BISM: Bytecode-Level Instrumentation for Software Monitoring

Chukri Soueidi Ali Kassem Yliès Falcone

Univ. Grenoble Alpes, Inria

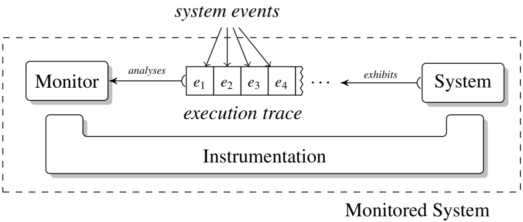
JAIF 20, Grenoble
September 24, 2020

Instrumentation is essential in Software Monitoring



- Allows us to *extract* and *abstract* system events

Instrumentation is essential in Software Monitoring

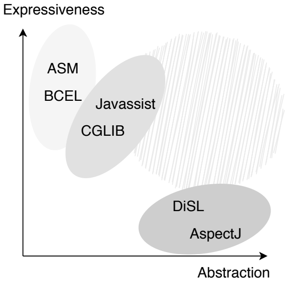


- Allows us to *extract* and *abstract* system events
- *Granularity level* of extracted events may range from:
  - Coarse (e.g., a method call)
  - Fine-grained (e.g., a jump in the control flow)

Instrumentation Tools

Several general-purpose tools for instrumenting Java programs varying in:

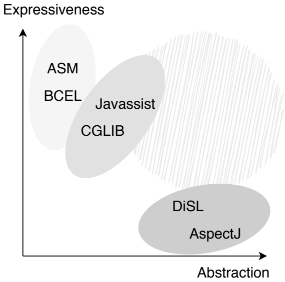
- Levels of *expressiveness*
- Levels of *abstraction*



Instrumentation Tools

Several general-purpose tools for instrumenting Java programs varying in:

- Levels of *expressiveness*
- Levels of *abstraction*



	BCEL	ASM	Javassist	CGLIB	DiSL	AspectJ
Bytecode Visibility	✓	✓	✓	✓	✓	✗
Allows bytecode insertion	✓	✓	✓	✓	✗	✗
No bytecode proficiency	✗	✗	✗	✗	✓	✓
High level of abstraction	✗	✗	✗	✗	✓	✓
Follows AOP paradigm	✗	✗	✗	✗	✓	✓

Motivating Example

- Extract fine-grained eT(..) and eF(..) events after conditional jumps
- Can we automate instrumentation?

```
g_authenticated = BOOL_FALSE;
if(g_ptc > 0) {

    int cmp = byteArrayCompare(...);
    if(cmp == BOOL_TRUE) {

        g_ptc = 3;
        g_authenticated = BOOL_TRUE;
    }
    else {

        g_ptc--;
    }
}
```

Original code

```
g_authenticated = BOOL_FALSE;
if(g_ptc > 0) {
    eT(1, ">", g_ptc, 0);
    int cmp = byteArrayCompare(...);
    if(cmp == BOOL_TRUE) {
        eT(2, "==", cmp, BOOL_TRUE);
        g_ptc = 3;
        g_authenticated = BOOL_TRUE;
    }
    else {
        eF(2, "==", cmp, BOOL_TRUE);
        g_ptc--;
    }
} else
    eF(1, ">", g_ptc, 0);
```

Instrumented code

Motivating Example

- Extract fine-grained eT(..) and eF(..) events after conditional jumps
- Can we automate instrumentation?
  Yes, possible in DiSL but not with AspectJ

```
g_authenticated = BOOL_FALSE;
if(g_ptc > 0) {

    int cmp = byteArrayCompare(...);
    if(cmp == BOOL_TRUE) {

        g_ptc = 3;
        g_authenticated = BOOL_TRUE;
    }
    else {

        g_ptc--;
    }
}
```

Original code

```
g_authenticated = BOOL_FALSE;
if(g_ptc > 0) {
    eT(1, ">", g_ptc, 0);
    int cmp = byteArrayCompare(...);
    if(cmp == BOOL_TRUE) {
        eT(2, "==", cmp, BOOL_TRUE);
        g_ptc = 3;
        g_authenticated = BOOL_TRUE;
    }
    else {
        eF(2, "==", cmp, BOOL_TRUE);
        g_ptc--;
    }
} else
    eF(1, ">", g_ptc, 0);
```

Instrumented code

Another Motivating Example

- Duplicate all if statements and report inconsistency
- Can we automate instrumentation?

```
// a simple if statement
if(g_ptc > 0) {

    int cmp = compare(...);

}
```

Original code

```
// duplicating the if statement
if(g_ptc > 0) {
    if(g_ptc > 0){
        int cmp = compare(...);
    }
    else {
        Report Attack!
    }
}
else{
    if(g_ptc >0 ) {
        Report Attack!
    }
}
```

Instrumented code

Another Motivating Example

- Duplicate all if statements and report inconsistency
- Can we automate instrumentation?
  Yes, but neither with AspectJ nor DiSL[1]

```
// a simple if statement
if(g_ptc > 0) {

    int cmp = compare(...);

}
```

Original code

```
// duplicating the if statement
if(g_ptc > 0) {
    if(g_ptc > 0){
        int cmp = compare(...);
    }
    else {
        Report Attack!
    }
}
else{
    if(g_ptc >0 ) {
        Report Attack!
    }
}
```

Instrumented code

[1] A workaround can have the same effect (but with a lot more instrumented code) as we will see in the experiments.

Contribution: The BISM tool

BISM is a **lightweight** Java **bytecode** instrumentation tool that features an **expressive** and **high-level** instrumentation language

Contribution: The BISM tool

BISM is a **lightweight** Java **bytecode** instrumentation tool that features an **expressive** and **high-level** instrumentation language

- Follows the **aspect-oriented programming** (AOP) paradigm

Contribution: The BISM tool

BISM is a **lightweight** Java **bytecode** instrumentation tool that features an **expressive** and **high-level** instrumentation language

- Follows the **aspect-oriented programming** (AOP) paradigm
- Features a **control-flow-aware** instrumentation language

Contribution: The BISM tool

BISM is a **lightweight** Java **bytecode** instrumentation tool that features an **expressive** and **high-level** instrumentation language

- Follows the **aspect-oriented programming** (AOP) paradigm
- Features a **control-flow-aware** instrumentation language
- Built-in **Transformers** (instrumentation classes)

Contribution: The BISM tool

BISM is a **lightweight** Java **bytecode** instrumentation tool that features an **expressive** and **high-level** instrumentation language

- Follows the **aspect-oriented programming** (AOP) paradigm
- Features a **control-flow-aware** instrumentation language
- Built-in **Transformers** (instrumentation classes)
- Better **performance** than DiSL and AspectJ demonstrated in:
  - Two security scenarios with
    - Advanced Encryption Standard
    - Financial Transactions System
  - Classic RV scenario with DaCapo benchmarks

Outline

Design Goals & Features

- **Convenient instrumentation mechanism**
  - *Modularity* in separate instrumentation classes that follow AOP paradigm
  - Granularity in joinpoints ranging from bytecode instruction to methods

Design Goals & Features

- **Convenient instrumentation mechanism**
  - *Modularity* in separate instrumentation classes that follow AOP paradigm
  - Granularity in joinpoints ranging from bytecode instruction to methods
- **Access to program context**
  - Complete *static information* for instruction, basic-block, method, and class
  - Dynamic context objects (local variables, stack values, method args ...)
  - New local variables within methods

Design Goals & Features

- **Convenient instrumentation mechanism**
  - *Modularity* in separate instrumentation classes that follow AOP paradigm
  - Granularity in joinpoints ranging from bytecode instruction to methods
- **Access to program context**
  - Complete *static information* for instruction, basic-block, method, and class
  - Dynamic context objects (local variables, stack values, method args ...)
  - New local variables within methods
- **Control-flow context**
  - Out-of-the-box *CFG information* in static context objects
  - *Conditional jumps* joinpoints (OnTrueBranch, OnFalseBranch)
  - *Visualizer* for CFGs

Design Goals & Features

- **Convenient instrumentation mechanism**
  - *Modularity* in separate instrumentation classes that follow AOP paradigm
  - Granularity in joinpoints ranging from bytecode instruction to methods
- **Access to program context**
  - Complete *static information* for instruction, basic-block, method, and class
  - Dynamic context objects (local variables, stack values, method args ...)
  - New local variables within methods
- **Control-flow context**
  - Out-of-the-box *CFG information* in static context objects
  - *Conditional jumps* joinpoints (OnTrueBranch, OnFalseBranch)
  - *Visualizer* for CFGs
- **Compatibility with ASM**
  - Bytecode *instruction insertion* using ASM syntax
  - Preserve ASM object structure

Design Goals & Features

- **Convenient instrumentation mechanism**
  - *Modularity* in separate instrumentation classes that follow AOP paradigm
  - Granularity in joinpoints ranging from bytecode instruction to methods
- **Access to program context**
  - Complete *static information* for instruction, basic-block, method, and class
  - Dynamic context objects (local variables, stack values, method args ...)
  - New local variables within methods
- **Control-flow context**
  - Out-of-the-box *CFG information* in static context objects
  - *Conditional jumps* joinpoints (OnTrueBranch, OnFalseBranch)
  - *Visualizer* for CFGs
- **Compatibility with ASM**
  - Bytecode *instruction insertion* using ASM syntax
  - Preserve ASM object structure
- **Two Instrumentation modes**
  - *Build-time* for static instrumentation
  - *Load-time* as a Java agent

Outline

Joinpoints

Identify specific bytecode locations in target program and capture their execution

- **Instruction Joinpoints**
  - `BeforeInstruction`: *before* bytecode instruction
  - `AfterInstruction`: *after* bytecode instruction
  - `BeforeMethodCall`: *after loading* all needed values on stack
  - `AfterMethodCall`: *before storing* return value

Joinpoints

Identify specific bytecode locations in target program and capture their execution

- **Instruction Joinpoints**
  - `BeforeInstruction`: *before* bytecode instruction
  - `AfterInstruction`: *after* bytecode instruction
  - `BeforeMethodCall`: *after loading* all needed values on stack
  - `AfterMethodCall`: *before storing* return value
- **Basic Block Joinpoints**
  - `OnBasicBlockEnter`: at *entry* of basic block
  - `OnBasicBlockExit`: *after last instruction* of basic block
  - `OnTrueBranchEnter`: *after conditional jump* on True evaluation
  - `OnFalseBranchEnter`: *after conditional jump* on False evaluation

Joinpoints

Identify specific bytecode locations in target program and capture their execution

- **Instruction Joinpoints**
  - `BeforeInstruction`: *before* bytecode instruction
  - `AfterInstruction`: *after* bytecode instruction
  - `BeforeMethodCall`: *after loading* all needed values on stack
  - `AfterMethodCall`: *before storing* return value
- **Basic Block Joinpoints**
  - `OnBasicBlockEnter`: at *entry* of basic block
  - `OnBasicBlockExit`: *after last instruction* of basic block
  - `OnTrueBranchEnter`: *after conditional jump* on True evaluation
  - `OnFalseBranchEnter`: *after conditional jump* on False evaluation
- **Method Joinpoints**
  - `OnMethodEnter`: on *method entry* block
  - `OnMethodExit`: on all *exit blocks* of method before return or throw

Static Context

Provide relevant static information at joinpoints

- **Instruction**
  - `index,opcode,next,previous,isConditionalJump()...`
  - `node,getBasicValueFrame(),getSourceValueFrame()`
  - `methodName,basicBlock,className`
- **MethodCall**
  - `methodOwner,methodName,currentClassName`
  - `node,instruction`

Static Context

Provide relevant static information at joinpoints

- **Instruction**
  - `index,opcode,next,previous,isConditionalJump()...`
  - `node,getBasicValueFrame(),getSourceValueFrame()`
  - `methodName,basicBlock,className`
- **MethodCall**
  - `methodOwner,methodName,currentClassName`
  - `node,instruction`
- **BasicBlock**
  - `id,index,blockType,size,getFirstInstruction()...`
  - `getSuccessorBlocks(),getPredecessorBlocks()`
  - `getTrueBranch(),getFalseBranch()`
  - `method`

Static Context

Provide relevant static information at joinpoints

- **Instruction**
  - `index,opcode,next,previous,isConditionalJump()...`
  - `node,getBasicValueFrame(),getSourceValueFrame()`
  - `methodName,basicBlock,className`
- **MethodCall**
  - `methodOwner,methodName,currentClassName`
  - `node,instruction`
- **BasicBlock**
  - `id,index,blockType,size,getFirstInstruction()...`
  - `getSuccessorBlocks(),getPredecessorBlocks()`
  - `getTrueBranch(),getFalseBranch()`
  - `method`
- **Method**
  - `name,getNumberofBlocks(),getEntryBlock(),getExitBlocks()`
  - `node,classContext`
- **Class**: `name,node`

Example – BasicBlock Execution

Intercepting basic block executions with BISM

```
public class BasicBlockTransformer extends Transformer {

    @Override
    public void onBasicBlockEnter(BasicBlock bb){
        String blockId =
              bb.method.className+"."+bb.method.name+"."+bb.id;

        print("Entered block:" + blockId)
    }

    @Override
    public void onBasicBlockExit(BasicBlock bb)
        String blockId =
              bb.method.className+"."+bb.method.name+"."+bb.id;

        print("Exited block:" + blockId)
    }
}
```

Dynamic Context

Provide access to dynamic values possibly only known during execution

- **Common**
  - `getThis()`
  - `getLocalVariable(int index)`
  - `getStackValue(int index)`
  - `getInstanceField(String name)`
  - `getStaticField(String name)`

Dynamic Context

Provide access to dynamic values possibly only known during execution

- **Common**
  - `getThis()`
  - `getLocalVariable(int index)`
  - `getStackValue(int index)`
  - `getInstanceField(String name)`
  - `getStaticField(String name)`
- **Method related**
  - `getMethodArgs(int index)`
  - `getMethodReceiver()`
  - `getMethodResult()`

Dynamic Context

Provide access to dynamic values possibly only known during execution

- **Common**
  - `getThis()`
  - `getLocalVariable(int index)`
  - `getStackValue(int index)`
  - `getInstanceField(String name)`
  - `getStaticField(String name)`
- **Method related**
  - `getMethodArgs(int index)`
  - `getMethodReceiver()`
  - `getMethodResult()`
- **Adding new local variables**
  - `addLocalVariable(Object value)`
  - `updateLocalVariable(LocalVariable, Object value)`

Dynamic Context

Provide access to dynamic values possibly only known during execution

- **Common**
  - `getThis()`
  - `getLocalVariable(int index)`
  - `getStackValue(int index)`
  - `getInstanceField(String name)`
  - `getStaticField(String name)`
- **Method related**
  - `getMethodArgs(int index)`
  - `getMethodReceiver()`
  - `getMethodResult()`
- **Adding new local variables**
  - `addLocalVariable(Object value)`
  - `updateLocalVariable(LocalVariable, Object value)`

`BISM` weaves necessary bytecode to extract values from stack or local variables

Instrumentation Methods

Methods to instrument target program to emit events or modify code

- **Printing**
  - `print(String),print(DynamicValue),...`
  - `printHash(DynamicValue),...`

Instrumentation Methods

Methods to instrument target program to emit events or modify code

- **Printing**
  - `print(String),print(DynamicValue),...`
  - `printHash(DynamicValue),...`
- **Invoking static methods**
  - `new StaticInvocation(..)` object should be constructed
  - `addParameter()` to add primitive or `DynamicValue`
  - `invoke(StaticInvocation)`

Instrumentation Methods

Methods to instrument target program to emit events or modify code

- **Printing**
  - `print(String),print(DynamicValue),...`
  - `printHash(DynamicValue),...`
- **Invoking static methods**
  - `new StaticInvocation(..)` object should be constructed
  - `addParameter()` to add primitive or `DynamicValue`
  - `invoke(StaticInvocation)`
- **Raw bytecode insertion**
  - `insert(AbstractInsnNode ins)`
  - `insert(List<AbstractInsnNode> ins)`

Instrumentation Methods

Methods to instrument target program to emit events or modify code

- **Printing**
  - `print(String)`,`print(DynamicValue)`,...
  - `printHash(DynamicValue)`,...
- **Invoking static methods**
  - `new StaticInvocation(..)` object should be constructed
  - `addParameter()` to add primitive or `DynamicValue`
  - `invoke(StaticInvocation)`
- **Raw bytecode insertion**
  - `insert(AbstractInsnNode ins)`
  - `insert(List<AbstractInsnNode> ins)`

Calls are compiled by `BISM` into bytecode instructions and inlined at needed locations

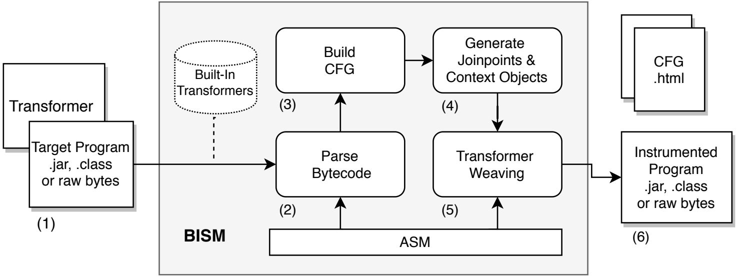
Example – Monitoring a List Iterator

```
public class IteratorTransformer extends Transformer {
    @Override
    public void afterMethodCall(MethodCall mc, MethodCallDynamicContext
         dc){
        if (mc.methodName.equals("iterator") &&
             mc.methodOwner.endsWith("List")) {
            // Access to dynamic data
            DynamicValue list = dc.getMethodTarget(mc);
            DynamicValue iterator = dc.getMethodResult(mc);

            // Invoking a monitor
            StaticInvocation sti =
                new StaticInvocation("IteratorMonitor",
                     "iteratorCreation");
            sti.addParameter(list);
            sti.addParameter(iterator);
            invoke(sti);
        }
    }
}
```

Outline

BISM Implementation

- Open-source implemented in Java (≈ 4,000 LOC and 40 classes)
- Uses ASM for bytecode parsing, analysis, and weaving



Instrumentation process in BISM

Outline

AES (Advanced Encryption Standard)

- Experiment
  - Instrument external AES implementation in build-time mode with BISM and DiSL
  - Deploy inline monitors to report test inversions by duplicating all conditional jumps in successor basic blocks

```
// duplicating the
     if-statements

if(g_ptc > 0) {
    if(g_ptc > 0){
        ...
    }
    else {
        Report Attack!
    }
}
else{
    if(g_ptc > 0 ) {
        Report Attack!
    }
}
```

Needed Instrumentation

AES (Advanced Encryption Standard)

- Experiment
  - Instrument external AES implementation in build-time mode with BISM and DiSL
  - Deploy inline monitors to report test inversions by duplicating all conditional jumps in successor basic blocks
- Measurement
  - Execution time and used memory on different input files
  - Median of 100 runs for each input file
- Experimental Setup
  - Java JDK 8u181 with 4 GB maximum heap
  - Intel Core i72.2 GHz, 16 GB RAM
  - MacOS Catalina v10.15.5 64-bit

```
// duplicating the
     if-statements

if(g_ptc > 0) {
    if(g_ptc > 0){
        ...
    }
    else {
        Report Attack!
    }
}
else{
    if(g_ptc > 0 ) {
        Report Attack!
    }
}
```

Needed Instrumentation

AES (Advanced Encryption Standard)

- Experiment
  - Instrument external AES implementation in build-time mode with BISM and DiSL
  - Deploy inline monitors to report test inversions by duplicating all conditional jumps in successor basic blocks
- Measurement
  - Execution time and used memory on different input files
  - Median of 100 runs for each input file
- Experimental Setup
  - Java JDK 8u181 with 4 GB maximum heap
  - Intel Core i72.2 GHz, 16 GB RAM
  - MacOS Catalina v10.15.5 64-bit

```
// duplicating the
     if-statements

if(g_ptc > 0) {
    if(g_ptc > 0){
        ...
    }
    else {
        Report Attack!
    }
}
else{
    if(g_ptc > 0 ) {
        Report Attack!
    }
}
```

Needed Instrumentation

Inlining the monitor cannot be done with AspectJ

AES – DiSL Implementation

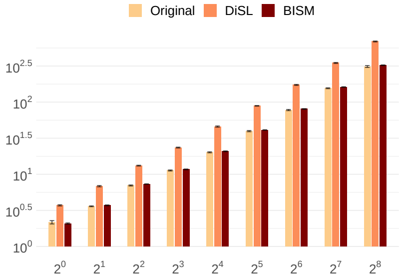
```
@SyntheticLocal
static int target = -1; //similarly op1,op2,opcode

@Before(marker = BytecodeMarker.class, args = "IF_ICMPGT,IF_ICMPLE..")
static void onIntegerCMP(InstructionStaticContextFull bcsc ..) {
  opcode = bcsc.getOpcode();
  op2 = dc.getStackValue(0, int.class);
  op1 = dc.getStackValue(1, int.class);
  target = bcsc.getJumpTargetLabel();
}

@Before(marker = BasicBlockMarker.class)
public static void afterbb(InstructionFullStaticContext bcsc ..) {
  boolean tt = target == ((int) bcsc.getIndexOfRegionStart());
  boolean attack = false;
  switch (opcode) {
      case 164: // IF_ICMPLE(164),
          if (((int) op1 <= (int) op2) != tt) {
              attack = true;
          }
          break;
      ...
```

AES – BISM Implementation

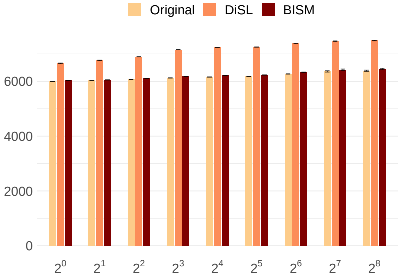
```
@Override
public void beforeInstruction(Instruction ins ..) {
    if (ins.isConditionalJump()) {
        if (ins.stackOperandsCountIfConditionalJump() == 1) {
            insert(new InsnNode(Opcodes.DUP));
        } // else use DUP2

@Override
public void onTrueBranchEnter(BasicBlock bb ..) {
    LabelNode l_1 = new LabelNode();

    insert(new JumpInsnNode(bb.getLastRealInstruction().opcode, l_1));
    print("Test Inversion Attack!");
    insert(l_1);
}
```

AES Results – BISM vs. DiSL

Input File (KB)	2^0	2^1	2^2	2^3	2^4	2^5	2^6	2^7	2^8
Events (M)	0.92	1.82	3.65	7.34	14.94	29.53	58.50	117.24	233.10



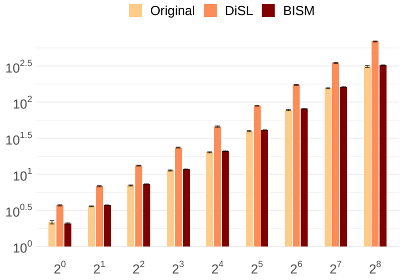
Runtime (ms)



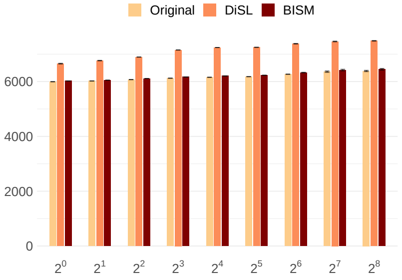
Memory footprint (KB)

AES Results – BISM vs. DiSL

Input File (KB)	2^0	2^1	2^2	2^3	2^4	2^5	2^6	2^7	2^8
Events (M)	0.92	1.82	3.65	7.34	14.94	29.53	58.50	117.24	233.10



Runtime (ms)



Memory footprint (KB)

Less execution time and memory overhead than DiSL for all input file sizes

Financial Transaction System

- Experiment
  - Instrument an abstraction of a Financial Transaction System in build-time mode, with BISM and DiSL, to emit events begin(i) and end(i)
  - Deploy outline monitor to report arbitrary jumps

```
// emit events begin(i)
     & end(i)

// Basic Block i
begin(i)
first instruction
    ...
last instruction
end(i)

// begin(i) & end(i) are
     function calls
```

Needed Instrumentation

Financial Transaction System

- Experiment
  - Instrument an abstraction of a Financial Transaction System in build-time mode, with BISM and DiSL, to emit events begin(i) and end(i)
  - Deploy outline monitor to report arbitrary jumps
- Measurement
  - Execution time and used memory on two scenarios
  - Median of 100 runs for each scenario

```
// emit events begin(i)
    & end(i)

// Basic Block i
begin(i)
first instruction
    ...
last instruction
end(i)

// begin(i) & end(i) are
    function calls
```

Needed Instrumentation

Financial Transaction System

- Experiment
  - Instrument an abstraction of a Financial Transaction System in build-time mode, with BISM and DiSL, to emit events begin(i) and end(i)
  - Deploy outline monitor to report arbitrary jumps
- Measurement
  - Execution time and used memory on two scenarios
  - Median of 100 runs for each scenario
- Experimental Setup
  - Java JDK 8u181 with 4 GB maximum heap
  - Intel Core i72.2 GHz, 16 GB RAM
  - MacOS Catalina v10.15.5 64-bit

```
// emit events begin(i)
    & end(i)

// Basic Block i
begin(i)
first instruction
    ...
last instruction
end(i)

// begin(i) & end(i) are
    function calls
```

Needed Instrumentation

Transactions – DiSL Implementation

```
public class BasicBlockExtendedContext extends BasicBlockStaticContext {
  public String uniqueBlockId() {
      MethodNode method = staticContextData.getMethodNode();
      String uniqueId = staticContextData.getClassNode().name
           +"."+method.name+"."+method.signature+"."+ this.getIndex ();
      return uniqueId;
  }
}
```

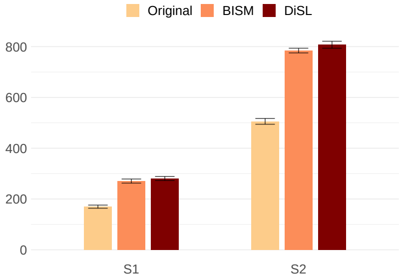
```
@Before(marker = BasicBlockMarker.class,scope ="transaction.*.*")
public static void beforebbb( BasicBlockExtendedContext bc,
     MethodStaticContext mc ) {
    transaction.MonitorAJ.begin(bc.uniqueBlockId());
}

@After(marker = BasicBlockMarker.class,scope ="transaction.*.*" )
public static void afterbbb( BasicBlockExtendedContext bc,
     MethodStaticContext mc) {
    transaction.MonitorAJ.end(bc.uniqueBlockId());
}
```

Transactions – BISM Implementation

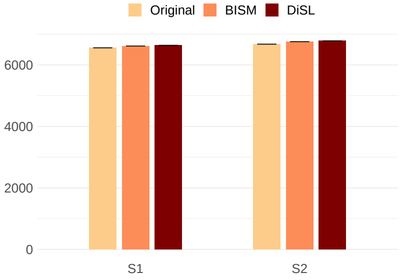
```
@Override
public void onBasicBlockEnter(BasicBlock bb){
    String uniqueBlockId = bb.method.className+"."+bb.method.name+"."
      +bb.method.methodNode.signature+"."+bb.index;

    StaticInvocation sti =
            new StaticInvocation(monitor, "begin");
    sti.addParameter(uniqueBlockId);
    invoke(sti);
}

@Override
public void onBasicBlockExit(BasicBlock bb){
    String uniqueBlockId = bb.method.className+"."+bb.method.name+"."
      +bb.method.methodNode.signature+"."+bb.index;

      StaticInvocation sti =
            new StaticInvocation(monitor, "end");
    sti.addParameter(uniqueBlockId);
    invoke(sti);
}
```

Transactions Results – BISM vs. DiSL

Scenario	S1	S2
Events (M)	679	2307



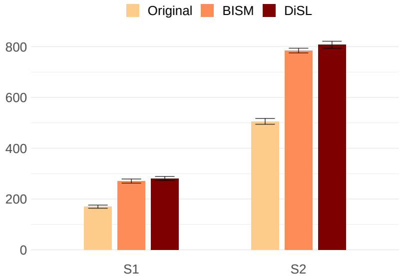
Runtime (ms)



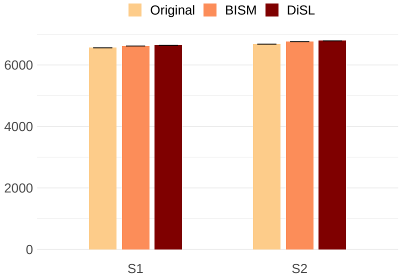
Memory footprint (KB)

Transactions Results – BISM vs. DiSL

Scenario	S1	S2
Events (M)	679	2307



Runtime (ms)



Memory footprint (KB)

Less execution time and memory overhead than DiSL for both scenarios

DaCapo Benchmarks

We compare BISM, DiSL & AspectJ in a general RV scenario

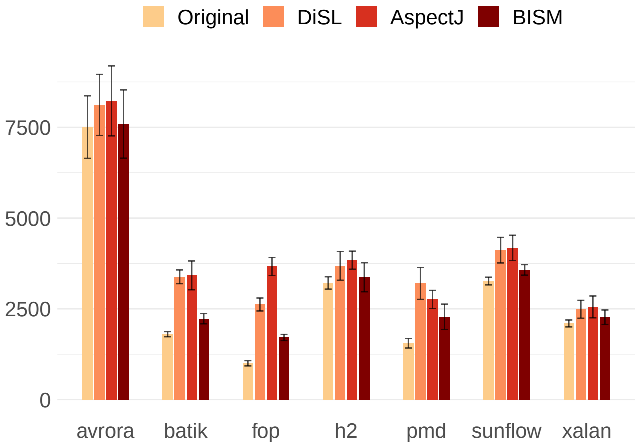
- Experiment
  - Use **HasNext**, **UnSafeIterator** and **SafeSyncMap** properties
  - External monitor library with stub methods to receive extracted objects
  - **DaCapo** suite (dacapo-9.12-bach) targeting only packages specific each benchmark

DaCapo Benchmarks

We compare BISM, DiSL & AspectJ in a general RV scenario

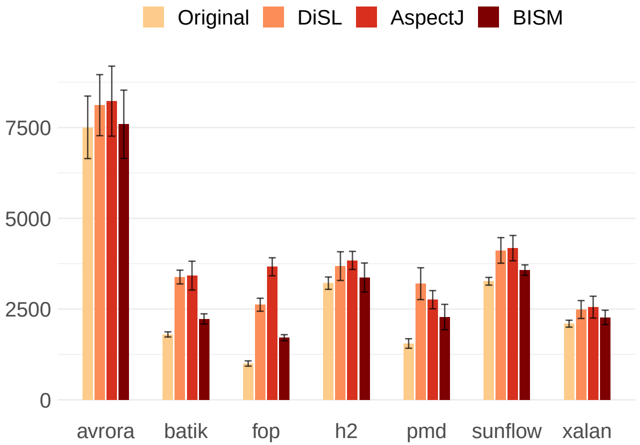
- Experiment
  - Use **HasNext**, **UnSafeIterator** and **SafeSyncMap** properties
  - External monitor library with stub methods to receive extracted objects
  - **DaCapo** suite (dacapo-9.12-bach) targeting only packages specific each benchmark
- Measurement
  - **Two modes**: **Build-time** and **Load-time** instrumentation
  - Median of 100 runs for each benchmark for each mode
- Experimental Setup
  - Java JDK 8u251 with 2 GB maximum heap size
  - Intel Core i9-9980HK (2.4 GHz. 8 GB RAM)
  - Ubuntu 20.04 LTS 64-bit

DaCapo – Load-time Results



Load-time instrumentation runtime (ms).

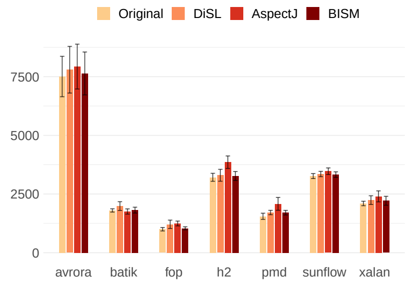
DaCapo – Load-time Results



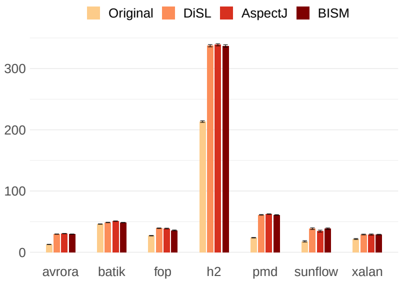
Load-time instrumentation runtime (ms).

BISM shows better performance over DiSL and AspectJ in all benchmarks

DaCapo – Build-time Results

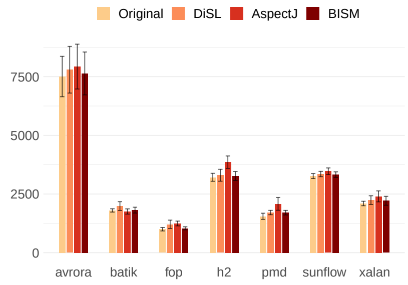


Runtime (ms).

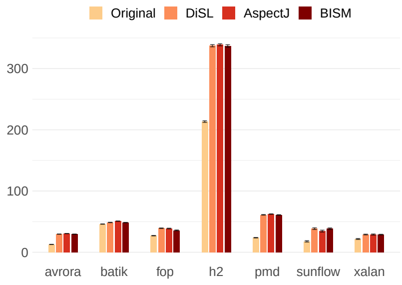


Memory footprint (MB).

DaCapo – Build-time Results



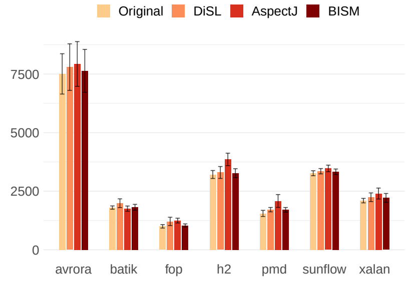
Runtime (ms).



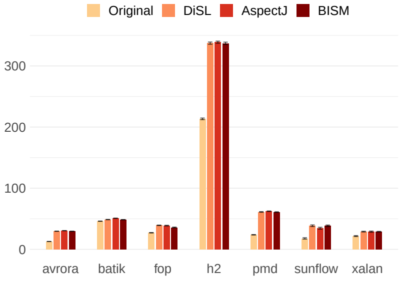
Memory footprint (MB).

- BISM shows less overhead in all benchmarks in execution time, except for batik, where AspectJ emits fewer events

DaCapo – Build-time Results



Runtime (ms).



Memory footprint (MB).

- BISM shows less overhead in all benchmarks in execution time, except for batik, where AspectJ emits fewer events
- BISM also shows less overhead in used-memory footprint, except for sunflow, where AspectJ emits much fewer events

DaCapo – More Results

	Scope	Instr.	BISM Ovh.%	DiSL Ovh.%	AspectJ Ovh.%	Events (M) #	Events (M) AJ
avrora	1,550	35	2.72	5.06	34.24	2.5	2.5
batik	2,689	136	1.81	2.85	9.59	0.5	0.4
fop	1,336	172	1.35	5.16	27.07	1.6	1.5
h2	472	61	1.44	3.75	37.75	28	28
pmd	721	90	2.38	5.03	29.63	6.6	6.3
sunflow	221	8	2.90	7.25	23.19	3.9	2.6
xalan	661	9	1.00	3.00	16.00	1	1

Bytecode size of the instrumented benchmarks applications

DaCapo – More Results

	Scope	Instr.	BISM Ovh.%	DiSL Ovh.%	AspectJ Ovh.%	Events (M) #	AJ
avrora	1,550	35	2.72	5.06	34.24	2.5	2.5
batik	2,689	136	1.81	2.85	9.59	0.5	0.4
fop	1,336	172	1.35	5.16	27.07	1.6	1.5
h2	472	61	1.44	3.75	37.75	28	28
pmd	721	90	2.38	5.03	29.63	6.6	6.3
sunflow	221	8	2.90	7.25	23.19	3.9	2.6
xalan	661	9	1.00	3.00	16.00	1	1

Bytecode size of the instrumented benchmarks applications

BISM incurs less bytecode size overhead for all benchmarks

Outline

Conclusion

- BISM is effective for low-level and control-flow aware instrumentation and can be used for lightweight and expressive runtime verification
- BISM shows better performance than BISM and AspectJ

	Overhead reduction	
	DiSL	AspectJ
Bytecode Performance		
Execution time	45%	82%
Used Memory	5%	8%
Size	56%	91%
Tool performance		
Execution time	64%	68%

Conclusion

- BISM is effective for low-level and control-flow aware instrumentation and can be used for lightweight and expressive runtime verification
- BISM shows better performance than BISM and AspectJ

	Overhead reduction DiSL	Overhead reduction AspectJ
Bytecode Performance		
Execution time	45%	82%
Used Memory	5%	8%
Size	56%	91%
Tool performance		
Execution time	64%	68%

	BCEL	**ASM**	**Javassist**	**CGLIB**	**DiSL**	**AspectJ**	**BISM**
Bytecode Visibility	✓	✓	✓	✓	✓	✗	✓
Allows bytecode insertion	✓	✓	✓	✓	✗	✗	✓
No bytecode proficiency	✗	✗	✗	✗	✓	✓	✓
High level of abstraction	✗	✗	✗	✗	✓	✓	✓
Follows AOP paradigm	✗	✗	✗	✗	✓	✓	✓

The Future of BISM

We plan on adding

- More instrumentation methods like calling non-static external methods
- More built-in Transformers covering classic RV use cases
- Instrumentation directives
- Perform additional static analysis techniques

The Future of BISM

We plan on adding

- More instrumentation methods like calling non-static external methods
- More built-in Transformers covering classic RV use cases
- Instrumentation directives
- Perform additional static analysis techniques

New directions now possible with BISM

- Control-flow-aware RV
- Runtime Enforcement using bytecode insertion capabilities