High-level Abstractions for Instrumentation-based Dynamic Program Analysis

Walter Binder
Dynamic Analysis Group
Faculty of Informatics
University of Lugano

walter.binder@usi.ch
http://dag.inf.usi.ch/
Outline

• Introduction and state of the art
• DiSL: a domain-specific language for instrumentation
• Under the hood: the weaver
• Case study
• Evaluation
Dynamic program analysis (DPA)

- Observe relevant events at runtime
- Appropriate where static analysis is impossible or too expensive
  - need for precise performance metrics
  - polymorphism
  - reflection
  - native code
  - dynamically generated or downloaded code
Examples

• Profiling
  – execution time
  – calling context
  – contention
  – hardware performance counters

• Debugging
  – memory leaks
  – data races
  – deadlocks

• Testing

• Program understanding, reverse engineering
Implementation techniques for DPA
State of the art

expressiveness

BTrace

Aspect-Oriented Programming (AOP)

BCEL

Soot

Javassist

ASM

JVMTI (native agent)

tool development effort
Low-level bytecode instrumentation

• Commonly used for building dynamic analysis tools
• Drawbacks
  – error-prone
  – time-consuming
  – expensive
  – resulting tools are difficult to maintain and to extend
Prevailing approaches to bytecode instrumentation

• Native code agents (JVMPi, JVMTI)
  – limited portability
  – lack of native code instrumentation libraries

• Instrumentation in Java
  – limited coverage
  – many low-level bytecode engineering libraries
    (e.g., BCEL, ASM, Javassist, Soot, ShrikeBT, BIT, JOIE, Serp, ...)

Implementation techniques for DPA
State of the art

Aspect-Oriented Programming (AOP)

BTrace

Javassist
Soot
BCEL
ASM
JVMTI (native agent)
AllocAnalysis with AOP

Example: profiling allocated objects

```java
aspect AllocAnalysis {
    after() returning(Object o) : call(*.new(..)) {
        System.out.println("New object allocated: " +
                        System.identityHashCode(o));
    }
}
```
public void instrument(Class<?> clazz) throws ClassNotFoundExceptiion, IOException {
    JavaClass jc = Repository.lookupClass(clazz);
    String className = jc.getClassName();
    ClassGen cg = new ClassGen(jc);
    ConstantPoolGen cgpg = cg.getConstantPool();
    int idx = cgpg.addString("New object allocated: ");

    InstructionFactory factory = new InstructionFactory(cg);

    for(Method m : cg.getMethods()) {
        MethodGen mg = new MethodGen(m, className, cgpg);
        InstructionList il = mg.getInstructionList();
        il(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);
                }
            }
        }
        mg.setMaxLocals();
        mg.setMaxStack();
        cg.replaceMethod(m, mg.getMethod());
    }
    return cg.getClass().getBytes();
}
Aspect-oriented programming (AOP)
AspectJ terminology

- **Join points** are specific execution points
  - read/write field
  - call/execute method
  - call/execute constructor
  - exception handler
  - ...

AspectJ terminology

• **Pointcuts** intercept specific join points

```java
aspect AllocAnalysis {
  pointcut allCalls() :
      call(*.new(..)) && !within(AllocAnalysis);

  after() returning(Object o) : allCalls() {
      System.out.println("New object allocated: "+
                     System.identityHashCode(o));
  }
}
```
AspectJ terminology

- *Advice* is the code executed before/after/around each join point intercepted by a pointcut.

```java
aspect AllocAnalysis {
    pointcut allCalls() :
        call(*.new(..)) && !within(AlocAnalyis);

    after() returning(Object o) : allCalls() {
        System.out.println("New object allocated: " +
                         System.identityHashCode(o));
    }
}
```
AspectJ terminology

• *Aspects* are class-like elements declaring *advice*

```java
aspect AllocAnalysis {
    pointcut allCalls() :
        call(*.new(..)) && !within(AllocAnalyis);

    after() returning(Object o) : allCalls() {
        System.out.println("New object allocated: "+
            System.identityHashCode(o));
    }
}
```
Dynamic program analysis with AOP

• With AOP, instrumentations can be expressed at a higher abstraction level
  – does not require detailed knowledge of JVM, bytecode, and class-file format
  – reduced development time and cost
  – rapid development of extensible tools

• Are mainstream AOP frameworks suited for DPA?
Limitations of many AOP frameworks

• Missing join points
  – basic blocks of code
  – individual bytecodes

• No user-defined static analysis at weave time
  – to statically pre-compute some metrics
  – for advanced scoping strategies

• Inefficient data-passing between advice

• Resulting tools often suffer from high overhead

• Methods in Java class library cannot be woven
Aspect-based DPA Tools

- Senseo
  - ICSM’09
- ReCrash
  - SciCo’10
- Racer
  - GPCE’10
- ParCCT
  - PPPJ’09

Frameworks

- MAJOR
  - PPPJ’08
- MAJOR2
  - AOSD’11
- HotWave
  - GPCE’09
- @J
  - VMIL’09

Instrumentation Components

- Full Meth Coverage
  - PPPJ’07
- Calling Ctxt Reification
  - AOSD’09
- Inter Adv Comm
  - WOSP’10
- Deferred Methods
  - AOSD’10

- BCEL, ASM
- AspectJ

Any Standard JVM
From AOP to a DSL

Find new case study

Use general-purpose AOP framework

Integrate new features in the AOP framework

Design a new DSL for DPA

- Effective and efficient DPA tool created
- Insufficient expressiveness or inefficiency
- Basic weaver functionalities
- Advanced weaver features
- retry
- requirements
Implementation techniques for DPA

- BCEL
- BTrace
- JVMTI (native agent)
- Javassist
- Soot
- ASM
- Aspect-Oriented Programming (AOP)

Expressiveness vs. tool development effort
DiSL: DSL for instrumentation

- AOP-based DSL for rapid development of DPA tools
- Annotation syntax similar to AspectJ
- High-level support for advanced optimizations
- Flexible and efficient weaver
- Compatible with standard compilers and JVMs

- Assumption: DPA inserts monitoring code that does not alter program execution
DiSL language features

- Open join point model
- Access to static context
- Access to dynamic context
- Argument processors
- Synthetic local variables
- Thread-local variables
- Guards and scope
Open join point model

• Every region of bytecode can be a join point
• DiSL provides an extensible library of join point markers for join point selection
• Included markers
  – method body (%BodyMarker)
  – basic block
    (%BasicBlockMarker, PreciseBasicBlockMarker)
  – individual bytecode (%BytecodeMarker)
  – exception handler (%CatchClauseMarker)
Snippets

• DiSL programs contain code snippets
  – correspond to advice in AOP
  – are instantiated, composed, and inlined by the weaver at marked join points
  – can access any static and dynamic context information
  – may process method arguments in a user-defined way
Snippets

- Static methods with void return type
- Defined by annotations
  - @Before
  - @After
  - @AfterReturning
  - @AfterThrowing
- Must not throw exceptions
- Allow developer to precisely control the weaving order
public class SampleInstr {

    @Before(marker = BodyMarker.class)
    static void onMethodEntry() {
        ... // snippet body
    }

    @AfterReturning(marker = BytecodeMarker.class,
                    args = "newarray, anewarray, multianewarray")
    static void afterArrayAllocation() {
        ... // snippet body
    }
}
Weaving order

```java
@Before(...) , order = 0

@Before(...) , order = 1

@After(...) , order = 0

@After(...) , order = 1
```

Snippet A

Snippet B

Snippet C

Snippet D

Begin

Inlined snippets

Join point shadow

Inlined snippets

End

Method body
Static context

• Most DPA tools access static context information
• DiSL can compute custom static information at weave-time and store results in the constant pool
• Predefined static contexts
  – MethodStaticContext
  – BasicBlockStaticContext
  – BytecodeStaticContext
  – DataFlowStaticContext
Static context

(1) find used static contexts
(2) invoke
(3) use reified info
(4) insert result
(5) inline

Static contexts

Class

Method

Marked region 1

Marked region 2

Marked region n

Class

Method

Shadow 1

Current shadow

Shadow n

Snippet

Reified information

(1) find used static contexts
Dynamic context

- DiSL provides an interface to
  - access local variables
  - inspect the operand stack

- The weaver translates calls to the API methods into bytecode sequences to retrieve the desired values in the inlined code

- The developer must know where to access the data

```java
public interface DynamicContext {
    <T> T getLocalVariableValue(int index, Class<T> valueType);
    <T> T getStackValue(int distance, Class<T> valueType);
}
```
Dynamic context

- Example: profiling write access to object arrays

```java
public class ArrayAccessAnalysis {
    @Before(marker = BytecodeMarker.class, args = "aastore")
    static void beforeArrayStore(DynamicContext dc) {
        Object array = dc.getStackValue(2, Object.class);
        int index = dc.getStackValue(1, int.class);
        Object stored = dc.getStackValue(0, Object.class);
        Analysis.process(array, index, stored); // not shown
    }
}
```
Argument processors

• AOP frameworks usually expose method arguments by wrapping them in an object array

• DiSL provides argument processors
  – reflective mechanism to process arguments by type
  – allows users to capture only specific argument types
  – supports both method arguments and callsite arguments
Argument processors

foo(int i1, Object o, int i2)

(1) process

IntArgsProcessor

static void intProc(int val, ArgumentContext ac, ...) {
    ...
}

(2) expand

ac.getPosition() == 0
val = i1

ac.getPosition() == 2
val = i2

(3) inline

@Before(...)
static void args(ArgumentProcessorContext ap)

... ap.apply(IntArgsProcessor.class, ArgumentProcessorMode.METHOD_ARGS);
...

Snippet
Synthetic local variables

• AspectJ lacks an efficient mechanism for passing data between advice woven in the same method body

• DiSL supports **synthetic local variables**
  – accessed through static fields annotated with @SyntheticLocal
  – mapped to local variables in woven methods
  – require snippet inlining
Synthetic local variables
Synthetic local variables

Code

Instrumentation

Woven code

@SyntheticLocal
data

data = ...

local_data = ...

.. = local_data
Example: TimeAnalysis (AspectJ)

```java
public aspect TimeAnalysis {
    Object around() : (call(* *(..)) || call(*.new(..))) &&
        !within(TimeAnalysis) {
            long start = System.nanoTime();
            try {
                return proceed(); // proceed with the execution
            } finally {
                Analysis.logExec(thisJoinPoint, System.nanoTime() – start);
            }
        }
}
```

- Passing `start` from `before` to `after` method call
- Problem: `around` creates wrapper methods
  - violates current hotswapping constraints
  - may not work in `java.lang.Object` (e.g., on Oracle’s HotSpot VM)
  - may break stack introspection in certain methods of the Java class library
Example: TimeAnalysis (DiSL)

```java
public class TimeAnalysis {
    @SyntheticLocal
    static long start;

    @Before(marker = BytecodeMarker.class, args = "invoke*")
    static void beforeCalls() {
        start = System.nanoTime();
    }

    @After(marker = BytecodeMarker.class, args = "invoke*")
    static void afterCalls(MethodStaticContext msc) {
        long elapsed = System.nanoTime() - start;
        Analysis.logExec(msc.thisMethodFullName(), elapsed);
    }
}
```

- **start** is shared between all snippets woven in the same method body
Thread-local variables

- DiSL supports efficient **thread-local variables**
  - accessed through static fields annotated with `@ThreadLocal`
  - mapped to added instance fields in `java.lang.Thread`
  - initialized to default value of their type
Guards and scope

• Two complementary mechanisms for restricting the application of snippets
  – guards: based on weave-time evaluation of conditionals
  – scope: based on method signature matching

• Avoid expensive runtime checks
Guards and scope

```java
public class ArgumentAnalysis {
    @Before(marker = BodyMarker.class, guard = MethodReturnsRef.class)
    static void onMethodEntry() {
        ... // inlined only if the method returns an object
    }
}

public class MethodReturnsRef {
    @GuardMethod
    static boolean evalGuard(ReturnTypeStaticContext rtsc) {
        return !rtsc.isPrimitive();
    }
}
```
Wrapping up

• Weaving with complete method coverage
• Framework overview
• Example: Senseo
• Evaluation
• Conclusion
Full method coverage

• DPA tools often require instrumentation of all executed methods
  – application classes
  – dynamically generated code
  – Java class library

• Weaving in the Java class library is difficult
  – introduced class dependencies may crash the JVM upon bootstrapping
  – infinite recursions when snippets invoke woven methods in the Java class library
Full method coverage

- DPA tools often require instrumentation of all executed methods
  - application classes
  - dynamically generated code
  - Java class library

- Weaving in the Java class library is difficult
  - introduced class dependencies may crash the JVM upon bootstrapping
  - infinite recursions when snippets invoke woven methods in the Java class library
Weaving only application code

application code

woven application code

inline snippet

Java class library

original code
Weaving with full method coverage

application code

woven application code

Java class library

woven Java class library

inlined snippet

inlined snippet
Weaving with full method coverage

- application code
  - woven application code
  - inlined snippet
- Java class library
  - woven Java class library
  - inlined snippet

INFINITE RECURSION!
⇒ JVM CRASH!
Weaving with full method coverage

execute original if (bypass == true)

execute original if (bypass == true) OR during bootstrap

temporarily bypass woven code (bypass = true)

inlined snippet

inlined snippet
Weaver

Snippet parsing
(1)

Scope matching
(3)

Shadow marking
(4)

Static context evaluation
(5)

Processor selection
(6)

Weaving
(7)

Target class
(2)

Woven target class
(8)

Load-time instrumentation framework
Example: Senseo

• DPA tool for code comprehension and profiling
  – originally implemented in AspectJ

• 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
  – improve runtime performance
  – improve coverage
  – improve granularity: basic block metrics
void f() {
    for(int i=0; i<10; i++) {
        h();
        g(i+1);
    }
}

void g(int i) {
    for(int j=0; j<i; j++) {
        h();
    }
}

void h() {return;}

Calling context tree (CCT)
public class Senseo {
    @ThreadLocal
    static CCTNode currentNode;
    @SyntheticLocal
    static CCTNode callerNode;

    @Before(marker = BodyMarker.class, order = 1)
    static void onMethodEntry(MethodStaticContext msc, ArgumentProcessorContext proc) {
        if ((callerNode = currentNode) == null)
            callerNode = CCTNode.getRoot();
        currentNode = callerNode.profileCall(msc.thisMethodFullName());
        proc.apply(ReferenceProc.class, ArgumentProcessorMode.METHOD_ARGS);
    }

    @After(marker = BodyMarker.class, order = 1)
    static void onMethodCompletion() { currentNode = callerNode; }

    //...
}
Example: Senseo

```java
public class Senseo {
    @ThreadLocal
    static CCTNode currentNode;
    @SyntheticLocal
    static CCTNode callerNode;

    @ArgumentProcessor
    public class ReferenceProc {
        static void objProc(Object obj, ArgumentContext ac) {
            Senseo.currentNode.profileArgument(ac.getPosition(), obj);
        }
    }

    @Before(marker = BodyMarker.class, order = 1)
    static void onMethodEntry(MethodStaticContext msc, ArgumentProcessorContext proc) {
        if ((callerNode = currentNode) == null)
            callerNode = CCTNode.getRoot();
        currentNode = callerNode.profileCall(msc.thisMethodFullName());
        proc.apply(ReferenceProc.class, ArgumentProcessorMode.METHOD_ARGS);
    }

    @After(marker = BodyMarker.class, order = 1)
    static void onMethodCompletion() { currentNode = callerNode; }
    // ...
}
```
Example: Senseo

```java
public class Senseo {
    @ThreadLocal
    static CCTNode currentNode;
    //...
    @AfterReturning(marker = BodyMarker.class, order = 0,
                      guard = MethodReturnsRef.class)
    static void onReturnRef(DynamicContext dc) {
        Object obj = dc.getStackValue(0, Object.class);
        currentNode.profileReturnReturn(obj);
    }

    @AfterReturning(marker = BytecodeMarker.class, order = 0,
                     args = "new, newarray, anewarray, multianewarray")
    static void onAllocation() {
        currentNode.profileAllocation();
    }
}
```
public class Senseo {
    @ThreadLocal
    static CCTNode currentNode;
    //...
    @Before(marker = BasicBlockMarker.class, order = 0)
    static void onBasicBlock(BasicBlockStaticContext bbsc) {
        currentNode.profileBB(bbsc.getBBIndex());
    }
}
Overhead

Measurement settings
• geo. mean { for each DaCapo “bach” benchmark: median(15 runs) }
• Intel Core2 Quad Q9650 (3.0 GHz, 8GB RAM), Ubuntu GNU/Linux 10.04 64-bit
• Oracle HotSpot 1.6.0_27 64-bit JVM, AspectJ 1.6.11
## Intercepted join points

<table>
<thead>
<tr>
<th>Category</th>
<th>Application only</th>
<th>Full coverage</th>
<th>Difference [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>method bodies</td>
<td>5.60E+09</td>
<td>8.84E+09</td>
<td>57.75</td>
</tr>
<tr>
<td>methods returning a reference</td>
<td>1.14E+09</td>
<td>2.00E+09</td>
<td>76.11</td>
</tr>
<tr>
<td>object and array allocations</td>
<td>1.76E+08</td>
<td>3.44E+08</td>
<td>95.28</td>
</tr>
<tr>
<td>basic-blocks of code</td>
<td>2.21E+10</td>
<td>3.34E+10</td>
<td>51.26</td>
</tr>
</tbody>
</table>

Total number of intercepted join points for a single iteration of the whole DaCapo suite
## Weave time performance

<table>
<thead>
<tr>
<th></th>
<th>AspectJ</th>
<th></th>
<th>DiSL</th>
<th></th>
<th>DiSL</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>no basic-block metrics</td>
<td>app. only</td>
<td>no basic-block metrics</td>
<td>app. only</td>
<td>with basic-block metrics</td>
<td>full coverage</td>
</tr>
<tr>
<td>Weave time [s]</td>
<td>54.97</td>
<td></td>
<td>43.28</td>
<td>134.17</td>
<td>65.61</td>
<td>174.73</td>
</tr>
<tr>
<td>Latency [s]</td>
<td>53.86</td>
<td></td>
<td>155.25</td>
<td></td>
<td>75.06</td>
<td>213.71</td>
</tr>
</tbody>
</table>

Total weave time and latency for a single iteration of the whole DaCapo suite
Ongoing research

• Partial evaluator for instantiated snippets ("Turbo")
  – simplifies programming model (less guards)
  – increases weave time
  – can reduce runtime overhead

• Modularization
  – reusable analysis units
  – contracts for analysis units
  – composition of analysis units

• Dynamic deployment/removal of analysis units within an executing system
Conclusion

• Building DPA tools with AOP techniques helps reduce development effort and eases extension

• Current general-purpose AOP frameworks lack some essential features for DPA

• DiSL: a DSL for instrumentation
  – open join point model
  – flexible weaver
  – support for
    • efficient access to context information
    • guards evaluated at weave-time
Get more information!


**DiSL: A Domain-specific Language for Bytecode Instrumentation.**
In: Proceedings of the 11\textsuperscript{th} International Conference on Aspect-Oriented Software Development (AOSD), 2012.

- [http://disl.origo.ethz.ch/](http://disl.origo.ethz.ch/)