Dynamic Program Analysis – Reconciling Developer Productivity and Tool Performance

Aibek Sarimbekov[b], Yudi Zheng[b], Danilo Ansaloni[b], Lubomír Bulej[b], Lukáš Marek[a], Walter Binder[b], Petr Tůma[a], Zhengwei Qi[c]

[a]*Charles University, Czech Republic*
[b]*University of Lugano, Switzerland*
[c]*Shanghai Jiao Tong University, China*

Abstract

Dynamic program analysis tools serve many important software engineering tasks such as profiling, debugging, testing, program comprehension, and reverse engineering. Many dynamic analysis tools rely on program instrumentation and are implemented using low-level instrumentation libraries, resulting in tedious and error-prone tool development. Targeting this issue, we have created the Domain-Specific Language for Instrumentation (DiSL), which offers high-level programming abstractions especially designed for instrumentation-based dynamic analysis. When designing DiSL, our goal was to boost the productivity of tool developers targeting the Java Virtual Machine, without impairing the performance of the resulting tools. In this paper we assess whether DiSL meets this goal. First, we perform a controlled experiment to measure tool development time and correctness of the developed tools, comparing DiSL with a prevailing, state-of-the-art instrumentation library. Second, we recast 10 open-source software engineering tools in DiSL and compare source code metrics and performance with the original implementations. Our studies show that DiSL significantly improves developer productivity, enables concise tool implementations, and does not have any negative impact on tool performance.

Keywords:
Dynamic program analysis, bytecode instrumentation, development productivity, controlled experiment

1. Introduction

With the growing complexity of computer software, dynamic program analysis (DPA) has become an invaluable tool for obtaining information about computer programs that is difficult to ascertain from the source code alone. Existing DPA tools aid in a wide range of tasks, including profiling [1], debugging [2, 3, 4], and program comprehension [5, 6].

The implementation of a typical DPA tool usually comprises an analysis part and an instrumentation part. The analysis part implements algorithms and data structures, and determines what points in the execution of the analyzed program must be observed. The instrumentation part is responsible for inserting code into the analyzed program. The inserted code then notifies the analysis part whenever the execution of the analyzed program reaches any of the points that must be observed.

There are many ways to instrument a program, but the focus of this paper is on Java bytecode manipulation. Since Java bytecode is similar to machine code, manipulating it is considered difficult and is usually performed using libraries such as BCEL [7], ASM [8], Soot [9], Shrike [10], or Javassist [11]. However, even with those libraries, writing the instrumentation part of a DPA tool is error-prone and requires advanced expertise from the developers. Due to the low-level nature of the Java bytecode, the resulting code is often verbose, complex, and difficult to maintain or to extend.

Email addresses: `aibek.sarimbekov@usi.ch` (Aibek Sarimbekov), `yudi.zheng@usi.ch` (Yudi Zheng), `danilo.ansaloni@usi.ch` (Danilo Ansaloni), `lubomir.bulej@usi.ch` (Lubomír Bulej), `lukas.marek@d3s.mff.cuni.cz` (Lukáš Marek), `walter.binder@usi.ch` (Walter Binder), `petr.tuma@d3s.mff.cuni.cz` (Petr Tůma), `qizhenwei@sjtu.edu.cn` (Zhengwei Qi)

The complexity associated with manipulating Java bytecode can be sometimes avoided by using aspect-oriented programming (AOP) [12] to implement the instrumentation part of a DPA tool. This is possible because AOP provides a high-level abstraction over predefined points in program execution (join points) and allows inserting code (advice) at a declaratively specified set of join points (pointcuts). Tools like the DJProf profiler [13], the RacerAJ data race detector [14], or the Senseo Eclipse plugin for augmenting static source views with dynamic metrics [6], are examples of successful applications of this approach.

AOP, however, is not a general solution to DPA needs—mainly because AOP was not primarily designed for DPA. AspectJ, the de-facto standard AOP language for Java, only provides a limited selection of join point types and thus does not allow inserting code at the boundaries of, e.g., basic blocks, loops, or individual bytecodes. Another important drawback is the lack of support for custom static analysis at instrumentation time, which can be used, e.g., to precompute static information accessible at runtime, or to select join points that need to be captured. An AOP-based DPA tool will usually perform such tasks at runtime, which can significantly increase the overhead of the inserted code. This is further aggravated by the fact that access to certain static and dynamic context information is not very efficient [15].

To leverage the syntactic conciseness of the pointcut-advice mechanism found in AOP without sacrificing the expressiveness and performance attainable by using the low-level bytecode manipulation libraries, we have previously presented DiSL [16, 17], an open-source framework that enables rapid development of efficient instrumentations for Java-based DPA tools. DiSL achieves this by relying on AOP principles to raise the abstraction level (thus reducing the effort needed to develop an instrumentation), while avoiding the DPA-related shortcomings of AOP languages (thus increasing the expressive power and enabling instrumentations that perform as well as instrumentations developed using low-level bytecode manipulation libraries).

Since DiSL is an AOP-inspired abstraction layer built on top of ASM, it is natural to question whether such a layer is actually worth having. In previous work, we have followed the community practice of demonstrating the benefits of DiSL on a case study, in which we recasted the AOP-based instrumentation of Senseo [6] to DiSL and ASM and compared the source code size and performance of the recasted instrumentations to the original. While the results indicated that, compared to ASM, DiSL indeed raised the abstraction level without impairing performance, the case study only covered a single DPA tool and did not *quantify* the impact of the higher abstraction level on the development of DPA instrumentations. To the best of our knowledge, no such quantification is present in the literature concerning instrumentation of Java programs.

The purpose of this paper, therefore, is to *quantify* the usefulness of DiSL when developing DPA instrumentations, and to extend the evaluation to other tools. Specifically, we aim to address the following research questions:

RQ1 Does DiSL improve developer productivity in writing instrumentations for DPA?

RQ2 Do DiSL instrumentations perform as fast as their equivalents written using low-level libraries?

To answer the research questions, we conduct a controlled experiment to determine if the use of DiSL instead of ASM increases developer productivity. We also perform an extensive evaluation of 10 existing open source DPA tools, in which we reimplement their instrumentation parts using DiSL. We compare reimplemented and the original instrumentation parts of those 10 DPA tools. With respect to RQ1, the controlled experiment provides evidence of increased developer productivity, supported by the evidence of more concise expression of equivalent instrumentations obtained by comparing the sizes of the original and DiSL-based instrumentations in terms of logical lines of code. Regarding RQ2, we compare the overhead of the evaluated DPA tools on benchmarks from the DaCapo [18] suite using both the original and the DiSL-based instrumentation.

The paper makes the following scientific contributions:

1. We present a controlled experiment in which we measure how DiSL affects the *time* needed to implement bytecode instrumentations and the *correctness* of the resulting instrumentations.
2. We present an evaluation of ten existing DPA tools, in which we recast their instrumentation parts in DiSL, and compare the size and performance of the original and the recasted instrumentations.

While the study on the controlled experiment was previously published [19], this paper contains unpublished material on performance comparison of ten different DPA tools.

```
pointcut executionPointcut () : execution (* HelloWorld.* (..));

before (): executionPointcut () {
  System.out.println ("On "+ thisJoinPointStaticPart.getSignature () +" method entry");
}

after (): executionPointcut () {
  System.out.println ("On "+ thisJoinPointStaticPart.getSignature () +" method exit");
}
```

Figure 1: Tracing tool implemented using AspectJ.

```
@Before (marker = BodyMarker.class, scope = "*.HelloWorld.*")
void onMethodEntry (MethodStaticContext msc) {
  System.out.println ("On "+ msc.thisMethodFullName () +" method entry");
}

@After (marker = BodyMarker.class, scope = "*.HelloWorld.*")
void onMethodExit (MethodStaticContext msc) {
  System.out.println ("On "+ msc.thisMethodFullName () +" method exit");
}
```

Figure 2: Tracing tool implemented using DiSL.

The remainder of the paper is structured as follows: Section 2 gives an overview of DiSL, focusing on key concepts needed to make this paper self-contained. Section 3 provides a detailed description of the controlled experiment. Section 4 presents the recasts and evaluation of ten different DPA tools. Section 5 discusses related work, and Section 6 concludes the paper.

2. Background: DiSL Overview

DiSL[1] is a domain-specific language that provides developers of DPA tools with high-level concepts similar to those in AOP, without taking away the expressiveness and performance that can be attained when developing instrumentations using low-level bytecode manipulation libraries.

The key concepts raising the level of abstraction in DiSL instrumentations are *markers* and *snippets*, complemented by *scopes* and *guards*. A marker represents a class of potential instrumentation sites and is similar to a *join point* in AOP. DiSL provides a predefined set of markers at the granularity of methods, basic blocks, loops, exception handlers, and individual bytecodes. Since DiSL follows an open join point model, programmers can implement custom markers to represent the desired instrumentation sites.

Snippets contain the instrumentation code and are similar to *advice* in AOP. Snippets are inlined *before* or *after* an instrumentation site, with the usual semantics found in AOP. The snippet code can access any kind of static context information (e.g., class and method name, basic block index), and may also inspect the dynamic context of the executing method (e.g., stack and method arguments).

Scopes and guards restrict the application of snippets. While scopes are based on method signature matching, guards contain Java code capturing potentially complex conditionals evaluated at instrumentation time. Snippets annotated with markers, scopes, and guards are colocated in a class referred to as DiSL *instrumentation*, which is similar to an *aspect* in AOP.

To illustrate the basic DiSL concepts and their similarity to AOP, Figures 1 and 2 show the source code of a simple tracing tool implemented using AspectJ and DiSL, respectively. On each method entry and method exit, the tool should output the full name of the method and its signature.

[1]http://disl.ow2.org

In the AspectJ version, the executionPointcut() construct selects method executions restricted to the desired class, while the before() and after () constructs define the advice code that should be run before and after method execution. Within the advice code, the thisJoinPointStaticPart pseudo-variable is used to access static information, e.g., method name, related to each join-point where the advice is applied.

In the DiSL version, we define two code snippets, represented by the onMethodEntry() and onMethodExit() methods, that print out the method name and signature before and after executing a method body. The method name and signature is obtained from a method static context, which is accessed through the msc method argument. To determine when—relative to the desired point in program execution—the snippets should be executed, we use the @Before and @After annotations. The annotation parameters determine where to apply the snippets. The marker parameter selects the whole method body, and the scope parameter restricts the selection only to methods of the HelloWorld class.

To demonstrate the more advanced features of DiSL, Figure 3 shows a DiSL-based implementation of the instrumentation part of a field-immutability analysis tool, which identifies fields that were never written to outside the dynamic extent of the constructor [20]. This notion of immutability is dynamic by nature, and while it differs from the classic notion of immutability found in the literature [21, 22], it still provides a developer with valuable insights. The analysis (not shown) driven by the instrumentation tracks all field accesses and object allocations, and keeps a small state machine for each instance field. Every field can be in one of the three states: *virgin* (i.e., not read or not written to), *immutable* (i.e., read or was written to inside the dynamic extent of its owner object's constructor), or *mutable* (otherwise).

To implement the instrumentation for such an analysis in DiSL, we define two snippets, beforeFieldWrite and beforeFieldRead, which intercept the putfield and getfield bytecodes, respectively. Inside the snippets, we extract the reference to the instance in which the field is being accessed from the operand stack, and pass it along with the field identifier and a queue of objects under construction to the analysis class, ImmutabilityAnalysis[2], using the onFieldWrite and onFieldRead methods, respectively.

To extract values from the operand stack (in this case the object reference), we use the DynamicContext API, which allows obtaining the values from arbitrary (valid) operand stack slots. The type of access to a field is determined by the bytecode instruction to which the snippets are bound, and the instruction in turn determines the operand stack layout we can expect when the corresponding snippet is executed. For field reads we therefore extract the object refrence from the top of the operand stack, while for field writes we extract the reference from the second position from the top. The field identifier is obtained through a custom MethodStaticContext.

After each object allocation, we use the afterInitialization snippet to pass the newly allocated object, along with the identification of its allocation site, to the analysis runtime class using the onObjectInitialization method. As in the case of field accesses, the DynamicContext API is used to extract the reference to the newly allocated object from the operand stack.

The ThreadLocal static variable objectsUnderConstruction holds a stack of currently executing constructors, which the analysis uses to determine whether the owner of a field being accessed is under construction. To maintain the stack, the beforeConstructor snippet pushes the object under construction on the stack, whereas the afterConstructor snippet pops the stack. The ConstructorsOnly guard is used at instrumentation time to restrict the application of the two stack-maintenance snippets to constructors only.

3. Quantifying the Impact of DiSL

In this section we present the controlled experiment conducted to answer the first research question, i.e., whether using DiSL improves developer productivity in writing DPA instrumentations. We first introduce the experiment design, including task and subject descriptions, and then present the results of the experiment followed by a discussion of threats to validity of the study.

3.1. Experiment Design

The purpose of the experiment is to quantitatively evaluate the effectiveness of using DiSL for writing instrumentations for DPA tools compared to the use of a low-level bytecode manipulation library. We claim that using DiSL,

[2]For sake of brevity, we omit the description and the source code of the analysis runtime class, because it is not important in the context of instrumentation—it merely defines an API that the instrumentation will use to notify the analysis about events in the base program.

```java
/** INSTRUMENTATION CLASS **/
public class DiSLClass {
  @ThreadLocal
  private static Deque <Object> objectsUnderConstruction;

  /** STACK MAINTENANCE **/
  @Before (marker = BodyMarker.class, guard = ConstructorsOnly.class)
  public static void beforeConstructor (DynamicContext dc) {
    try {
      if (objectsUnderConstruction == null) {
        objectsUnderConstruction = new ArrayDeque <Object> ();
      }

      objectsUnderConstruction.push (dc.getThis ());
    } catch (Throwable t) {
      t.printStackTrace ();
    }
  }

  @After (marker = BodyMarker.class, guard = ConstructorsOnly.class)
  public static void afterConstructor () {
    ImmutabilityAnalysis.instanceOf ().popStackIfNonNull (objectsUnderConstruction);
  }

  /** ALLOCATION SITE **/
  @AfterReturning (marker = BytecodeMarker.class, args = "new")
  public static void afterInitialization (MyMethodStaticContext sc, DynamicContext dc) {
    ImmutabilityAnalysis.instanceOf ().onObjectInitialization (
      dc.getStackValue (0, Object.class), // the allocated object
      sc.getAllocationSite () // the allocation site
    );
  }

  /** FIELD ACCESSES **/
  @Before (marker = BytecodeMarker.class, args = "putfield")
  public static void beforeFieldWrite (MyMethodStaticContext sc, DynamicContext dc) {
    ImmutabilityAnalysis.instanceOf ().onFieldWrite (
      dc.getStackValue (1, Object.class), // the accessed object
      sc.getFieldId (), // the field identifier
      objectsUnderConstruction // the stack of constructors
    );
  }

  @Before (marker = BytecodeMarker.class, args = "getfield")
  public static void beforeFieldRead (MyMethodStaticContext sc, DynamicContext dc) {
    ImmutabilityAnalysis.instanceOf ().onFieldRead (
      dc.getStackValue (0, Object.class), // the accessed object
      sc.getFieldId (), // the field identifier
      objectsUnderConstruction // the stack of constructors
    );
  }
}

/** GUARD CLASS **/
class ConstructorsOnly {
  @GuardMethod
  public static boolean isApplicable (MethodStaticContext msc) {
    return msc.thisMethodName ().equals ("<init>");
  }
}
```

Figure 3: Field-immutability analysis tool implemented in DiSL.

Task	Description
0	a) On method entry, print the method name. b) Count the number of NEW bytecodes in the method. c) On each basic block entry, print its index in the method. d) Before each lock acquisition, invoke a given method that receives the object to be locked as its argument.
1	On method entry, print the number of method arguments.
2	Before array allocation, invoke a given method that receives the array length as its argument.
3	Upon method completion, invoke a given method that receives the dynamic execution count of a particular bytecode instruction as its argument.
4	Before each AASTORE bytecode, invoke a given method that receives the object to be stored in the array together with the corresponding array index as its arguments.
5	On each INVOKEVIRTUAL bytecode, invoke a given method that takes only the receiver of the invoke bytecode as its argument.
6	On each non-static field write access, invoke a given method that receives the object whose field is written to, and the value of the field as its arguments. Invocation shall be made only when writing non-null reference values.

Table 1: Description of instrumentation tasks

developers of DPA tools can improve their productivity and the correctness of the resulting tools. In terms of hypothesis testing, we have formulated the following null hypotheses:

$H1_0$: Implementing DPA tools with DiSL does not reduce the development time of the tools.

$H2_0$: Implementing DPA tools with DiSL does not improve the correctness of the tools.

We therefore need to determine if there is evidence that would allow us to refute the two null hypotheses in favor of the corresponding alternative hypotheses:

H1: Implementing DPA tools with DiSL reduces the development time of the tools.

H2: Implementing DPA tools with DiSL improves the correctness of the tools.

The rationale behind the first alternative hypothesis is that DiSL provides high-level language constructs that enable users to rapidly specify compact instrumentations that are easy to write and to maintain. The second alternative hypothesis is motivated by the fact that DiSL does not require knowledge of low-level details of the JVM and bytecodes from the developer, although more advanced developers can extend DiSL for special use cases.

To test the hypotheses $H1_0$ and $H2_0$, we define a series of tasks in which the subjects, split between a control and an experimental group, have to implement different instrumentations similar to those commonly found in DPA tools. The subjects in the control group have to solve the tasks using only ASM, while the subjects in the experimental group have to use DiSL.

The choice of ASM as the tool for the control group was driven by several factors. The goal of the experiment was to quantify the impact of the abstractions and the programming model provided by DiSL on the development of instrumentations for DPA tools. We did a thorough research of existing bytecode manipulation libraries and frameworks, and ASM came out as a clear winner with respect to flexibility and performance, both aspects crucial for development of efficient DPA tools. In addition, ASM is a mature, well-maintained library with an established community. As a result, ASM is often used for instrumentation development (and bytecode manipulation in general) both in academia and industry. We maintain that when a developer is asked to instrument an application by manipulating Java bytecode, ASM will most probably be the library of choice.

DiSL was developed as an abstraction layer on top of ASM precisely because of the above reasons, but with a completely different programming model inspired by AOP, tailored for instrumentation development. Using ASM as the baseline allowed us to *quantify* the impact of our abstraction layer and programming model on the instrumentation development process, compared to a lower-level, but commonly used programming model provided by ASM as the de-facto standard library.

3.2. Task Design

With respect to the instrumentation tasks to be solved during the experiment, we maintain two important criteria: the tasks shall be representative of instrumentations that are used in real-world applications, and they should not be biased towards either ASM or DiSL. Table 1 provides descriptions of the instrumentation tasks the subjects have to implement. Those are examples of typical instrumentations that are used in profiling, testing, reverse engineering, and debugging.

To familiarize themselves with all the concepts needed for solving the tasks, the subjects first had to complete a bootstrap task 0.

3.3. Subjects and Experimental Procedure

In total, we had 16 subjects—BSc., MSc., and PhD students from Shanghai Jiao Tong University—participate in the experiment on a voluntary basis. Prior to the experiment, all subjects were asked to complete a self-assessment questionnaire regarding their expertise in object-oriented programming (OOP), Java, Eclipse, DPA, Java bytecode, ASM, and AOP. The subjects rated themselves on a scale from 0 (no experience) to 4 (expert), and on average achieved a level of 2.6 for OOP, 2.5 for Java, 2.5 for Eclipse, 0.75 for DPA, 0.6 for JVM bytecode, 0 for ASM, and 0.12 for AOP. Our subjects can be thus considered average (from knowledgeable to advanced) Java developers who had experience with Eclipse and with writing very simple instrumentation tasks, but with little knowledge of DPA and JVM in general, and with no expertise in ASM and AOP. Based on the self-assessment results, the subjects were assigned to the control and experimental groups so as to maintain approximately equal distribution of expertise.

The subjects in both groups were given a thorough tutorial on DPA, JVM internals, and ASM. The ASM tutorial focused on the tree API, which is considered easier to understand and use. In addition, the subjects in the experimental group were given a tutorial on DiSL. Since the DiSL programming model is conceptually closer to AOP and significantly differs from the programming model provided by low-level bytecode manipulation libraries, including ASM, we saw no benefit in giving the tutorial on DiSL also to the subjects in the control group. The tutorial was based on the authors' experience with dynamic program analysis and was given in form of an informal 3-hour lecture. The subjects were free to ask clarification questions. The experiment was performed in a single session in order to minimize the experimental bias (e.g., by giving different tutorials on the same topic) that could affect the experimental results. The session was supervised, allowing the subjects to ask clarification questions and preventing them from cheating. The subjects were not familiar with the goal of the experiment and the hypotheses.

We provided the subjects with disk images for VirtualBox,[3] which was the only piece of software that had to be installed. Each disk image contained all the software necessary to complete the tasks: Eclipse IDE, Apache Ant, and ASM installed on an Ubuntu 10.4 operating system. In addition, the disk images for the experimental group also contained an installation of DiSL. All subjects received the task descriptions and a debriefing questionnaire, which required the subjects to rate the perceived time pressure and task difficulty. The tasks had to be completed in 180 minutes, giving a 30 minutes time slot for each task.

3.4. Variables and Analysis

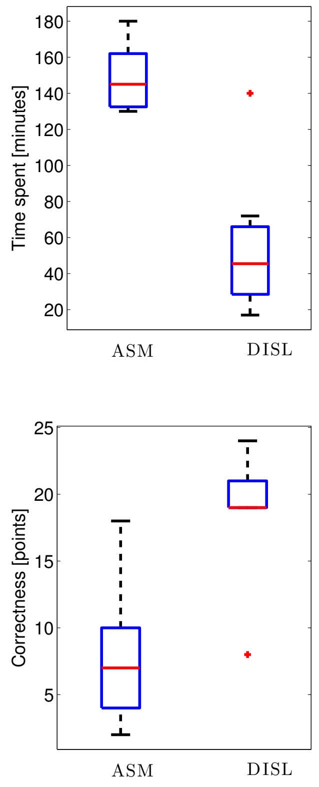
The only independent variable in our experiment is the availability of DiSL during the tasks.

The first dependent variable is the time the subjects spend to implement the instrumentations, measured by having the subjects write down the current time when starting and finishing a task. Since the session is supervised and going back to the previous task is not allowed, there is no chance for the subjects to cheat.

The second dependent variable is the correctness of the implemented solutions. This is assessed by code reviews and by manual verification. A fully correct solution is awarded 4 points, no solution 0 points, partially correct solutions can get from 1 to 3 points.

For hypothesis testing, we use the parametric one-tailed Student's t-test, after validating the assumptions of normality and equal variance using the Kolmogorov-Smirnov and Levene's tests. We maintain the typical confidence level of 99% ($\alpha = 0.01$) for all tests. All calculations were made using the SPSS statistical package.

[3]http://www.virtualbox.org/



	Time [minutes]		Correctness [points]	
	ASM	DiSL	ASM	DiSL
Summary statistics				
mean	148.62	54.62	8.75	18.75
difference	-63.2%		+46.6%	
min	130	17	2	8
max	180	140	18	24
median	145	45.5	7	19
stdev.	18.73	38.96	5.92	4.68
Assumption checks				
Kolmogorov-Smirnov Z	0.267	0.203	0.241	0.396
Levene F	1.291		1.939	
One-tailed Student's t-test				
df	14		14	
t	6.150		-3.746	
p-value	<0.001		0.002	

Figure 4: (a) Box plots for development time spent and correctness of the tools. The red dot represents an outlier. (b) Descriptive statistics of the experimental results

3.5. Experimental Results

3.5.1. Development Time

On average, the DiSL group spent 63% less time writing the instrumentations. The time spent by the two groups to complete the tasks is visualized as a box plot in Figure 4 (a).

To assess whether the positive impact on the development time observed with DiSL has a statistical significance, we test the null hypothesis $H1_0$, which says that DiSL does not reduce the development time. Neither Kolmogorov-Smirnov nor Levene's tests indicate a violation of the Student's t-test assumptions. The application of the latter gives a p-value[4] of 0.001, which is one order of magnitude less than $\alpha = 0.01$ (see Figure 4 (b)). We therefore reject the null hypothesis in favor of the alternative hypothesis, which means that the time spent is statistically significantly reduced by the availability of DiSL.

We attribute the substantial time difference to the fact that instrumentations written using ASM are very verbose and low-level, whereas DiSL allows one to write instrumentations at a higher level of abstraction.

3.5.2. Instrumentation Correctness

As reported in Figure 4 (b), the average amount of points scored by a subject in the experimental (DiSL) group is 46.6% higher than in the control (ASM) group. A box plot for the results is shown in Figure 4 (a).

[4]In statistical testing, the p-value is the probability of obtaining a result at least as extreme as the one that was observed, provided that the null hypothesis is true. Generally, if the p-value is less than the chosen significance level (e.g., 0.01), then obtaining such a result with the null hypothesis being true is highly unlikely, and the null hypothesis is therefore rejected.

To test the null hypothesis $H2_0$, which says that there is no impact of using DiSL on instrumentation correctness, we again apply the Student's t-test, since neither Kolmogorov-Smirnov nor Levene's tests indicate a violation of the normality assumptions. The t-test gives a p-value of 0.002 (see Figure 4 (b)) which is again an order of magnitude less than $\alpha = 0.01$. We therefore reject the null hypothesis in favor of the alternative hypothesis $H2$, which means that the correctness of instrumentations is statistically significantly improved by the availability of DiSL.

3.6. Threats to Validity

3.6.1. Internal Validity

There is a chance that the subjects participating in the experiment may not have been competent enough. To minimize the impact of this uncertainty, we ensured that the subjects had expertise at least at the level of average Java developers by using a preliminary self-assessment questionnaire. Moreover, we ensured that the subjects were equally distributed among the control group and the experimental group according to their expertise. Also both groups were given a thorough tutorial on DPA and had to complete task 0 before starting to solve the evaluated tasks.

The instrumentation tasks were designed by the authors of this paper, and therefore could be biased towards DiSL. To avoid this threat, we created instrumentations that are representative and used in some real-world scenarios. We asked other faculty members, who are not involved in the DiSL project but are familiar with DPA, to provide their comments on the tasks. Additionally, the tasks may have been too difficult and the time slot of 30 minutes may have been insufficient. To address this threat, we conducted a pilot study at Charles University in Prague and collected feedback about the perceived task difficulty and time pressure, which allowed us to adjust the difficulty of the instrumentation tasks. Moreover, the pilot study allowed us to adjust the tutorial and refine the documentation of DiSL.

3.6.2. External Validity

One can question the generalization of our results given the limited representativeness of the subjects and tasks. Even though the literature [23] suggests to avoid using only students in a controlled experiment, we could not attract other subjects to participate in our experiment.

Another threat to external validity is the fact that we compare high-level (DiSL) and low-level (ASM) approaches for instrumentation development. This choice is a necessary consequence of considering DPA tools to be the primary targets for DiSL-based instrumentations—high-level bytecode manipulation frameworks typically have limitations with respect to flexibility of instrumentation and control over inserted code, both of which are crucial for development of efficient DPA tools.

While low-level libraries can be typically used for a wide range of tasks, it is the focus on a specific purpose that allows high-level libraries to hide the low-level details common to that particular purpose. In this sense, DiSL was specifically designed for instrumentation development, while other high-level frameworks often target general code manipulation and transformation tasks. Our study quantifies the impact of introducing a high-level, AOP-inspired API on the developer productivity compared to the common practice. An additional user study involving DiSL and other high-level bytecode manipulation frameworks could explore the suitability of various high-level interfaces for instrumentation development, but that would not invalidate our study.

3.6.3. Results Summary and Future Work

In summary, the experiment confirms that DiSL improves developer productivity compared to ASM, the library of choice for bytecode manipulation in DPA tools. In terms of development time and instrumentation correctness, the improvement is both practically and statistically significant.

However, our study can be possibly improved in several ways. The controlled experiment presented in this paper has a "between" subject design. Conducting a similar study with a "with-in" subject design, where subjects from the control and the experimental groups swap the tasks after the first experiment might provide new insights and strengthen the results. Moreover, comparing DiSL with other high-level approaches for performing instrumentations (e.g., AspectJ) would be an interesting continuation of this work.

4. DiSL Tools Are Concise and Efficient

In this section we present the second contribution of this paper—an extensive evaluation of DiSL in the context of 10 existing open-source DPA tools. We have identified and recasted the instrumentation part of each tool in DiSL, without touching the analysis part. We then compared the amount of code required to implement the DiSL-based instrumentation, as well as its performance, to the original instrumentation. In the following text, we will refer to the unmodified version of a tool as *original*, and to the version using an equivalent DiSL-based instrumentation as *recasted*.

4.1. Overview of Recasted Analysis Tools

To establish a common context for both parts of the evaluation, we first present a short overview of each tool. To improve clarity, all descriptions adhere to a common template: we start with a high-level overview of the tool, then we describe the instrumentation logic used to trigger the analysis actions, and finally we point out the DiSL features used to reimplement the instrumentation.

The original instrumentations were implemented mostly using ASM, or AspectJ, with C used in one case. Most of the ASM-based and AOP-based tools rely on a Java agent, which is part of the java.lang.instrument API, and perform load-time instrumentation of all (loaded) classes.

Cobertura[5] is a tool for Java code coverage analysis. At runtime, Cobertura collects coverage information for every line of source code and for every branch.

Cobertura uses an ASM-based instrumentation to intercept branch instructions and blocks of bytecode corresponding to lines of code, and to invoke the method corresponding to the intercepted event on the analysis class to update the coverage information.

The recasted instrumentation uses three custom markers to capture the same join points as Cobertura, and a synthetic local variable to indicate whether a branch occurred.

EMMA[6] is a tool for Java code coverage analysis. During instrumentation, EMMA analyzes the classes and collects various static information, including the number of methods in a class, and the number of basic blocks in a method. At runtime, EMMA collects coverage information for every basic block of every method in every class.

EMMA uses ASM for both static analysis and instrumentation to intercept every method entry, where it associates a two-dimensional array with each class, and every basic block exit, where it updates the array to mark a basic block visited.

The recasted instrumentation uses the DiSL method body marker to intercept method entry, where it registers the two-dimensional array with EMMA, and a basic block marker to intercept basic block entry, where it triggers the update of coverage information. The array is stored in a per-class synthetic static field, which can be shared among snippets executed in different methods. A guard is used to filter out interface classes.

HPROF [24] is a heap and CPU profiler for Java distributed with the HotSpot JVM. Since HPROF is a native JVM agent implemented in C, we have reimplemented one of its features in Java to enable comparison with a DiSL-based tool. Our tool only provides the heap allocation statistics feature of HPROF, and uses a DiSL-based instrumentation to collect data. We therefore use *HPROF** as a designation for "HPROF with only the heap allocation statistics feature" in the following text, and all comparisons against the original HPROF only concern that single feature.

To keep track of allocated objects, the *HPROF** agent uses the Java Virtual Machine Tool Interface (JVMTI) [25] to intercept object allocation and death events, and collects type, size, and allocation site for each object.

The recasted instrumentation uses the DiSL bytecode marker to intercept object and array allocations, and a dynamic context API to obtain the references to newly allocated objects from the operand stack.

JCarder[7] is a tool for finding potential deadlocks in multi-threaded Java applications. At runtime, JCarder constructs a dependency graph for threads and locks, and if the graph contains a cycle, JCarder reports a potential deadlock.

To maintain the dependency graph, JCarder uses an ASM-based instrumentation to intercept acquisition and release of locks. To simplify the instrumentation, synchronized methods are converted to normal methods with explicit locking.

The recasted instrumentation uses the DiSL bytecode marker to intercept the lock acquisition/release bytecodes, and a method body marker with a guard to intercept synchronized method entry and exit. A custom static context is used to precompute the static method description required by the analysis class, and the dynamic context API is used to extract the lock reference from the stack.

[5]http://cobertura.sourceforge.net/
[6]http://emma.sourceforge.net
[7]http://www.jcarder.org/

JP2 [26] is a calling-context profiler for Java. For each method, JP2 collects various static metrics (i.e., method names, number and sizes of basic blocks) and dynamic metrics (i.e., method invocations, basic block executions, and number of executed bytecodes), and associates them with a corresponding node in a calling-context tree (CCT) [27], grouped by the position of the method call-site in the caller.

JP2 uses an ASM-based instrumentation to intercept method entry and exit, basic block entry, and execution of bytecodes that may trigger method invocation or execution of class initializers upon loading a class. Static information is collected during instrumentation.

The recasted instrumentation uses default DiSL markers (method body, basic block, and bytecode) to update the CCT upon method entry and exit. Thread-local variables are used to access the CCT instance and call-site position in the bytecode, while synthetic local variables are used to cache and share CCT nodes and call-site position between snippets. Static information is collected at instrumentation time using the method body, basic block, and bytecode static contexts.

JRat[8] is a call graph profiler for Java. For each method, JRat collects the execution time of each invocation, grouped by the caller. The data is used to produce a call graph with execution time statistics, which allows to attribute the time spent in a particular method to individual callers.

To measure method execution time and to determine the caller, JRat uses an ASM-based instrumentation to intercept each method entry and exit.

The recasted instrumentation uses the DiSL method body marker to intercept method invocations, a synthetic local variable to share time stamps between snippets, and a per-method synthetic static field to store the instance of an analysis class. A guard is used to avoid instrumentation of constructors and static class initializers.

RacerAJ [28] is a tool for finding potential data races in multi-threaded Java applications. At runtime, RacerAJ monitors all field accesses and lock acquisitions/releases, and reports a potential data race when a field is accessed from multiple threads without holding a lock that synchronizes the accesses.

To maintain various per-thread and per-field data structures, RacerAJ uses an AOP-based instrumentation to intercept all field accesses, and all lock acquisitions/releases, both explicit and implicit due to synchronized methods.

The recasted instrumentation uses the DiSL bytecode marker to intercept the lock acquisition/release and field access bytecodes, and a method body marker together with a guard to intercept synchronized method entry and exit. A custom static context is used to obtain a field identifier and the name of the class owning the field that is being accessed. A class static context is used to identify a field access site, while the dynamic context API is used to extract the lock reference from the stack.

ReCrash [3] is a tool for reproducing software failures. During program execution, ReCrash snapshots method arguments leading to failures and uses them to generate unit tests that reproduce the failure.

ReCrash uses an ASM-based instrumentation to intercept method entry, where it snapshots the method arguments, normal method exit, where it discards the snapshot, and abnormal method exit (only in the `main` method), where it uses the snapshot to generate a test case.

The recasted instrumentation uses the DiSL method body marker to intercept method entry and normal/abnormal method exit. A per-method synthetic static field is used to cache method static information, while a synthetic local variable is used to share the index of the argument snapshot between snippets on method entry and exit. The dynamic context API is used to take a snapshot of method arguments, and a guard filters out private methods, methods without arguments, empty methods, static class initializers, and constructors.

Senseo [6] is a tool for profiling and code comprehension. For each method invocation, Senseo collects calling-context specific information, which a plugin[9] then makes available to the user via enriched code views in the Eclipse IDE.

Senseo uses an AspectJ-based instrumentation, and intercepts each method entry and exit to count method invocations and uses join point API to collect statistics on method arguments and return types. Within methods, it also intercepts object allocations to count the number of allocated objects.

The recasted instrumentation uses the DiSL method body marker to intercept method invocation, and a bytecode marker to intercept object allocations. Method arguments and newly allocated objects are accessed via the dynamic

[8] http://jrat.sourceforge.net/
[9] http://scg.unibe.ch/research/senseo

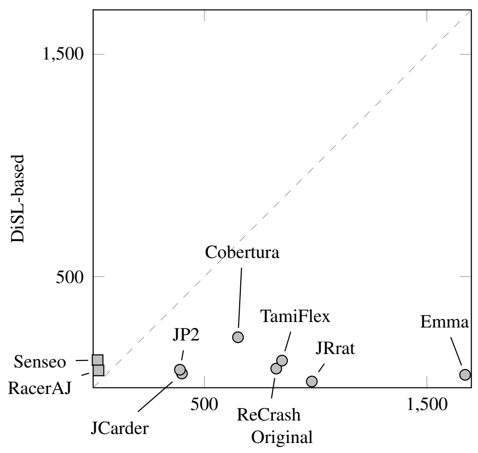


Figure 5: Logical source lines of code (SLOC) counts for original and DiSL-based instrumentations.
We use ● to denote instrumentations originally based on ASM, and ■ to denote those based on AspectJ.

context API. Guards are used to differentiate between methods that only accept and return primitive types and methods that work with reference types.

TamiFlex [29] is a tool that helps other (static analysis) tools deal with reflection and dynamically generated classes in Java. TamiFlex logs all reflective method calls and dumps all loaded classes, including those that are dynamically generated, to the disk. The collected information can be used to perform static analysis either with a TamiFlex-aware tool or, after transforming all reflective calls to actual method calls, with any other static analysis tool.

TamiFlex uses an ASM-based instrumentation in its *Play-out* agent to intercept method invocations on the instances of the Class, Constructor, and Method reflection classes.

The recasted instrumentation of the *Play-out* agent uses the DiSL method body marker restricted by a scope to intercept exits from methods in the aforementioned reflection classes. A corresponding snippet is used for each transformation from the *Play-out* agent. Support for dumping both the original and instrumented classes is present in DiSL and has been used.

4.2. Instrumentation Conciseness Evaluation

Based on the experience with recasting the instrumentation parts of the tools in this evaluation, we usually expect the instrumentations implemented using DiSL to require less logical source lines of code (SLOC) than their ASM-based equivalents. Even though “less code” does not generally mean “better code”, we assume that in the same context and for the same task, a shorter implementation can be considered more concise, and thus easier to write, understand, and maintain, if it also enables increased developer productivity.

Since the study in Section 3 shows that DiSL indeed positively impacts productivity compared to ASM, we use the SLOC count as a metric to compare different implementations of equivalent instrumentations in DPA tools. As a result, we provide quantitative evidence that DiSL-based instrumentations are more concise than their ASM-based equivalents, but not as concise as the AOP-based variants.

The plot in Figure 5 shows the SLOC counts[10] of both the DiSL-based and the original instrumentations for each tool. Each data point in the plot corresponds to a single tool, with the SLOC count of the original instrumentation on the x-axis, and the SLOC count of the DiSL-based instrumentation on the y-axis. Due to space limitations, we do not present the raw experimental data, as they would provide no additional insights.

Figure 5 indicates that DiSL-based instrumentations generally require less code than their ASM-based counterparts, because bytecode manipulation, even when using ASM, results in more verbose code. The extreme savings in the case of EMMA are due to EMMA having to implement its own static analysis, whereas the DiSL-based instrumentation can use the static context information provided by DiSL.

[10]Calculated using Unified CodeCount by CSSE USC, rel. 2011.10, http://sunset.usc.edu/research/CODECOUNT.

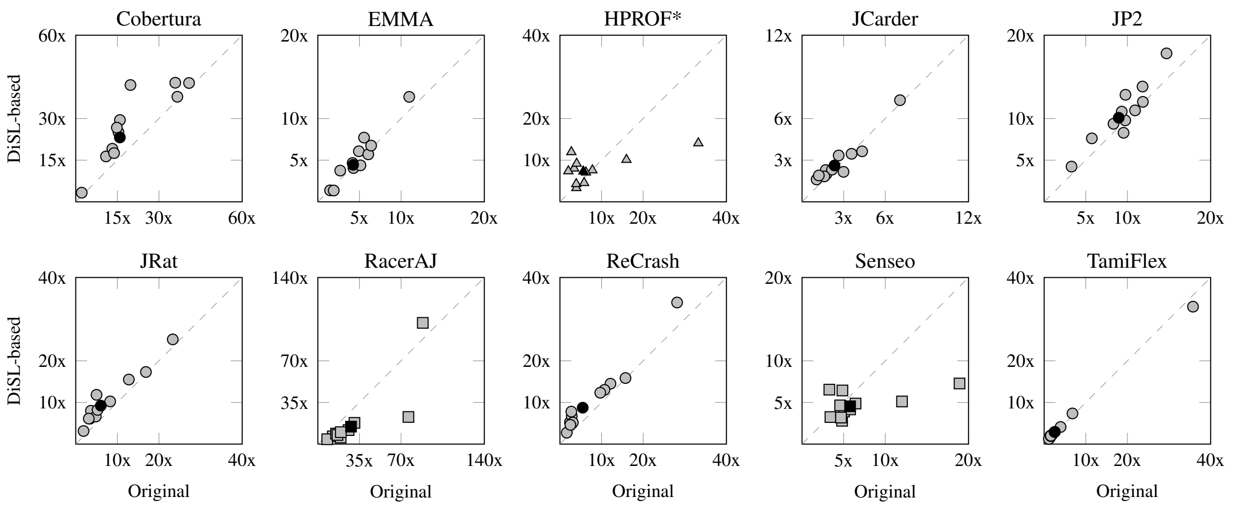


Figure 6: Startup phase – Overhead factor of original and DiSL-based applications compared to the baseline. Data points refer to a selection of 11 DaCapo 9.12 benchmarks and to the geometric mean of the overhead factors. We use ● to indicate applications originally based on ASM, ■ for those based on AspectJ, and ▲ for those based on C.

In line with our expectations, the DiSL-based instrumentations require more code than their AOP-based equivalents. This can be partially attributed to DiSL being an embedded language hosted in Java, whereas AOP has the advantage of being a separate language. Moreover, DiSL instrumentations also include code that is evaluated at instrumentation time, which increases the code size, but provides significant performance benefits at runtime. However, in the context of instrumentations for DPA, DiSL is more flexible and expressive than AOP without impairing the performance of the resulting tools.

The results for HPROF were intentionally omitted from the plot, because we were unable to isolate the instrumentation code for *HPROF** from the rest of the application. In total, HPROF consists of more than 9000 SLOC written in C, whereas our version of *HPROF** written in Java consists of 168 SLOC, of which 39 is the DiSL-based instrumentation.

4.3. Instrumentation Performance Evaluation

In this section, we conduct a series of experiments to provide answer to RQ2, i.e., whether DiSL-based instrumentations perform as fast as the equivalent instrumentations written using low-level bytecode manipulation libraries.

To evaluate the instrumentation performance, we compare the execution time of the original and the recasted tools on benchmarks from the DaCapo [18] suite (release 9.12). Of the fourteen benchmarks present in the suite, we excluded tradesoap, tradebeans and tomcat due to well known issues[11] unrelated to DiSL. All experiments were run on a multicore platform[12] with all non-essential system services disabled.

We present results for startup and steady-state performance in Figure 6 and Figure 7, respectively. Both figures contain a separate plot for each of the evaluated tools, displaying the overhead factor of a particular tool during the execution of each individual DaCapo benchmark. The data points marked in gray represent the execution of a single DaCapo benchmark, with the overhead factor of the original and the recasted tool on the x-axis and the y-axis, respectively. The single black data point in each plot represents the geometric mean of overhead factors from all benchmarks. The diagonal line serves to indicate the data points for which the overhead factor of the original and the recasted tool is the same.

To determine the startup overhead, we executed 3 runs of a single iteration of each benchmark and measured the time from the start of the process till the end of the iteration to capture the instrumentation overhead. We relied on the

[11]See bug ID 2955469 (hardcoded timeout in tradesoap and tradebeans) and bug ID 2934521 (StackOverflowError in tomcat) in the DaCapo bug tracker at http://sourceforge.net/tracker/?group_id=172498&atid=861957.

[12]Four quad-core Intel Xeon CPUs E7340, 2.4 GHz, 16 GB RAM, Ubuntu GNU/Linux 11.04 64-bit with kernel 2.6.38, Oracle Java HotSpot 64-bit Server VM 1.6.0_29.

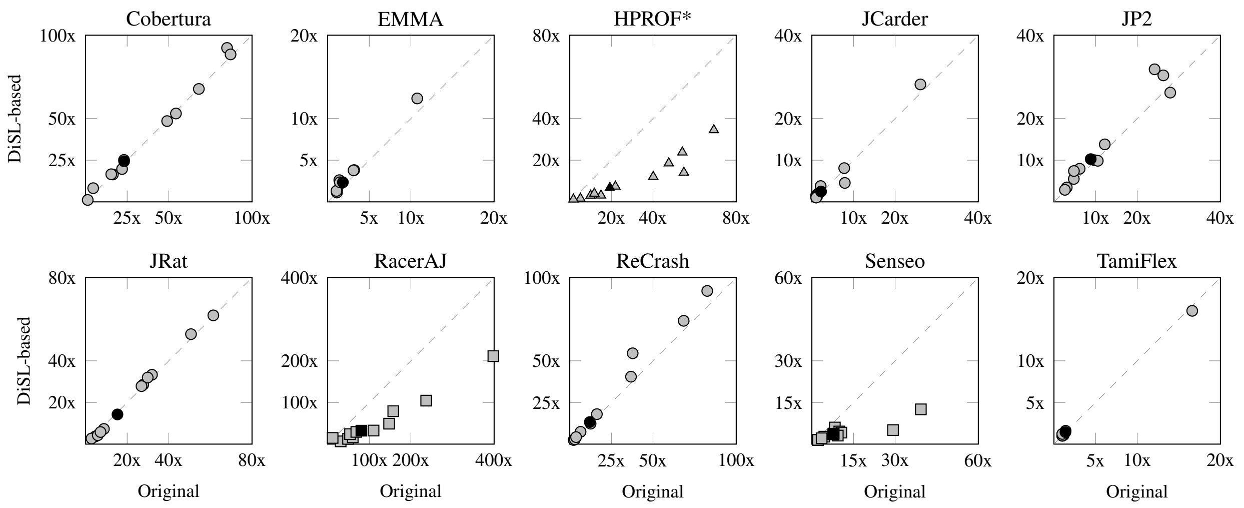


Figure 7: Steady-state phase – Overhead factor of original and DiSL-based applications compared to the baseline. Data points refer to a selection of 11 DaCapo 9.12 benchmarks and to the geometric mean of the overhead factors. We use ○ to indicate applications originally based on ASM, □ for those based on AspectJ, and △ for those based on C.

filesystem cache to mitigate the influence of I/O operations during startup. To determine the steady-state overhead, we made a single run with 10 iterations of each benchmark and excluded the first 5 iterations to minimize the influence of startup transients and interpreted code. The number of iterations to exclude was determined by visual inspection of the data from the benchmarks.

Concerning the startup overhead, the results in Figure 6 indicate that DiSL often slows down the startup phase of a benchmark. This can be partially attributed to DiSL using ASM to first create a tree representation of every class and only applying the exclusion filters at the level of methods. In this case, DiSL could be improved to decide early whether a class needs to be instrumented at all and thus avoid processing classes that need not be touched. Another source of overhead is evaluating guards and computing static context information for snippets that require it. In this case, the higher overhead at instrumentation time is traded for a lower overhead at runtime, as discussed below. The startup phase overhead is only important for applications where the amount of class loading relative to other code execution is high. We believe these cases to be rare.

Concerning the steady-state overhead, the results in Figure 7 show that the recasted tools are typically about as fast as their original counterparts, sometimes much faster, but never much slower. Performance improvements can be observed in the case of AOP-based tools (RacerAJ and Senseo both use AspectJ for instrumentation), and in the case of HPROF. The improved performance can be attributed mainly to the fact that DiSL allows to use static information at instrumentation time to precisely control where to insert snippet code, hence avoiding costly checks and static information computation (often comprising string concatenations) at runtime. The need for runtime checks is extremely pronounced with HPROF, which needs to filter the events related to program execution emitted by the JVM. Additional performance gains can be attributed to the ability of DiSL snippets to efficiently access the constant pool and the JVM operand stack, which is particularly relevant in comparisons with AOP-based tools.

5. Related Work

Java bytecode instrumentation is often performed using low-level bytecode manipulation libraries such as ASM [8], BCEL [7], Shrike [10], Soot [9], or Javassist [11]. In this section, we compare DiSL with state-of-the-art frameworks that build on top of such libraries to raise the abstraction level for specifying custom instrumentations. Then, we compare our methodology to assess developer's productivity with other related studies.

Prevailing high-level frameworks for expressing custom instrumentations usually generate streams of events that are composed at runtime and processed by the registered analyses. As a consequence, resulting tools suffer from the additional runtime overhead due to event generation and dispatching.

Sofya [30] is a framework that allows rapid development of DPA tools. It has a layered architecture in which the lower levels act as an abstraction layer on top of BCEL, while the top layers hide low-level details about the bytecode format and offer a publish/subscribe API that promotes composition and reuse of analyses. An Event Description Language (EDL) allows programmers to define custom streams of events, which can be filtered, splitted, and routed to the analyses.

RoadRunner [31] is a framework for composing DPA tools for checking safety and liveness properties of concurrent programs. Each analysis is essentially a filter over a set of event streams. While multiple analyses can be chained, it is only possible to specify a single chain at a time. Since analyses can filter events in an incompatible way, RoadRunner does not allow combination of arbitrary analyses. In contrast, DiSL is not tailored for a specific dynamic analysis task and offers fine-grained control over the inserted bytecode.

Chord [32] is a framework that provides a set of common static and dynamic analyses for Java. Moreover, developers can specify custom analyses, possibly on top of the existing ones. Similar to DiSL, Chord provides a rich and extensible set of low-level events that can be intercepted.

In contrast to these frameworks, DiSL is based on the pointcut/advice mechanism of AOP, which allows programmers to specify where to insert the instrumentations. Another advantage of DiSL is its built-in support for instrumenting also the classes from the Java Class Library [16]. As a consequence, DiSL is unique in reconciling a high-level API with important properties desired in the resulting tools, such as runtime efficiency and full code coverage.

Josh [33] is an AspectJ-like language that allows developers to define domain-specific extensions to the pointcut language. Similar to DiSL, Josh provides static pointcut designators that can access reflective static information at instrumentation time. Josh is built on top of Javassist [11]. Compared to DiSL, the join point model of Josh does not include arbitrary bytecodes and basic blocks of code.

In all the listed related work, the performance evaluation is limited to a few case studies. In constrast, we provide a detailed evaluation of both conciseness and runtime performance of the instrumentations written in DiSL, using an extensive set of existing DPA tools as case-studies.

Regarding the related work in assessing the developer productivity, the most related studies are in the area of program comprehension.

Cornelissen et al. [34] present a controlled experiment to evaluate completion time and correctness of solving several program comprehension tasks with the help of EXTRAVIS, an execution trace visualization tool.

Röthlisberger et al. [6] describe a controlled experiment to show that integrating dynamic information into the Eclipse IDE significantly decreases the amount of time required to perform typical software maintenance tasks while increasing the correctness of the solutions.

A similar experiment is performed in [35], in which 41 participants from both academia and industry are asked to solve a set of program comprehension tasks with, respectively without, using CodeCity.

These works all share a common approach to conducting controlled experiments that we have followed as well in our study to assess developer's productivity when implementing custom DPA tasks in DiSL.

6. Conclusions

The contribution of this paper is a thorough evaluation and assessment of DiSL [16, 17], a new abstraction layer on top of the well-known bytecode manipulation library ASM [8]. DiSL is a domain-specific aspect language especially designed for instrumentation-based dynamic program analysis. The design of DiSL aims at reconciling (1) a convenient high-level programming model to reduce tool development time, (2) high expressiveness to enable the implemention of any instrumentation-based dynamic analysis tool, and (3) efficiency of the generated code to ensure good tool performance. In this paper we explored whether DiSL meets this goal.

First, we conducted a controlled experiment to compare DiSL with ASM, measuring development time and correctness of the developed tools for 6 common dynamic analysis tasks. We showed that the use of DiSL reduced development time and improved tool correctness with both practical and statistical significance. Second, we recasted 10 open-source software engineering tools with DiSL, showing quantitative evidence that DiSL-based tools are (1) considerably more concise than equivalent tools based on ASM and (2) only slightly more verbose than equivalent tools implemented in AspectJ. Regarding performance, DiSL-based tools incur higher startup overhead than ASM-based tools but yield comparable steady-state performance; DiSL-based tools significantly outperform tools implemented in

AspectJ, both in terms of startup and steady-state performance. We conclude that DiSL is a valuable abstraction layer on top of ASM which indeed succeeds in boosting the productivity of tool developers. In contrast to AspectJ, DiSL neither limits expressiveness nor impairs performance of the resulting tools.

The open-source release of DiSL is available for download from OW2 Forge at http://disl.ow2.org.

Acknowledgements

The research presented in this paper has been supported by the Swiss National Science Foundation (project CRSII2_136225), by the Scientific Exchange Programme NMS–CH (project code 11.109), by a Sino-Swiss Science and Technology Cooperation (SSSTC) Exchange Grant (project no. EG34–092011) and Institutional Partnership (project IP04–092010), by the European Commission (Seventh Framework Programme grant 287746), by the Czech Science Foundation (project GACR P202/10/J042), and International Cooperation Program of Shanghai (No.11530700500).

References

[1] W. Binder, J. Hulaas, P. Moret, A. Villazón, Platform-independent profiling in a virtual execution environment, Software: Practice and Experience 39 (2009) 47–79.

[2] M. Eaddy, A. Aho, W. Hu, P. McDonald, J. Burger, Debugging aspect-enabled programs, in: Proceedings 6th International Conference on Software Composition (SC'07), volume 4829 of *LNCS*, Springer-Verlag, 2007, pp. 200–215.

[3] S. Artzi, S. Kim, M. D. Ernst, ReCrash: Making Software Failures Reproducible by Preserving Object States, in: Proceedings 22nd European Conference on Object-Oriented Programming (ECOOP'08), volume 5142 of *LNCS*, Springer-Verlag, 2008, pp. 542–565.

[4] G. Xu, A. Rountev, Precise memory leak detection for Java software using container profiling, in: Proceedings 30th International Conference on Software Engineering (ICSE'08), ACM, 2008, pp. 151–160.

[5] NetBeans, The NetBeans Profiler Project, Web pages at http://profiler.netbeans.org/, 2013.

[6] D. Röthlisberger, M. Härry, W. Binder, P. Moret, D. Ansaloni, A. Villazon, O. Nierstrasz, Exploiting dynamic information in IDEs improves speed and correctness of software maintenance tasks, IEEE Transactions on Software Engineering 38 (2012) 579–591.

[7] The Apache Jakarta Project, The Byte Code Engineering Library (BCEL), Web pages at http://jakarta.apache.org/bcel/, 2013.

[8] OW2 Consortium, ASM – A Java bytecode engineering library, Web pages at http://asm.ow2.org/, 2013.

[9] R. Vallée-Rai, E. Gagnon, L. J. Hendren, P. Lam, P. Pominville, V. Sundaresan, Optimizing Java bytecode using the Soot framework: Is it feasible?, in: Proceedings 9th International Conference on Compiler Construction (CC'00), Springer-Verlag, 2000, pp. 18–34.

[10] IBM, Shrike Bytecode Instrumentation Library, Web pages at http://wala.sourceforge.net/wiki/index.php/Shrike_technical_overview, 2013.

[11] S. Chiba, Load-time structural reflection in Java, in: Proceedings 14th European Conference on Object-Oriented Programming (ECOOP'00), volume 1850 of *LNCS*, Springer Verlag, 2000, pp. 313–336.

[12] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, J. Irwin, Aspect-oriented programming, in: Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97), volume 1241 of *Lecture Notes in Computer Science*, Springer, 1997, pp. 220–242.

[13] D. J. Pearce, M. Webster, R. Berry, P. H. J. Kelly, Profiling with AspectJ, Software: Practice and Experience 37 (2007) 747–777.

[14] E. Bodden, K. Havelund, Aspect-oriented Race Detection in Java, IEEE Transactions on Software Engineering 36 (2010) 509–527.

[15] W. Binder, A. Villazón, D. Ansaloni, P. Moret, @J - Towards rapid development of dynamic analysis tools for the Java Virtual Machine, in: Proceedings 3rd Workshop on Virtual Machines and Intermediate Languages (VMIL'09), ACM, 2009, pp. 1–9.

[16] L. Marek, A. Villazón, Y. Zheng, D. Ansaloni, W. Binder, Z. Qi, DiSL: A Domain-specific Language for Bytecode Instrumentation, in: Proceedings 11th International Conference on Aspect-oriented Software Development (AOSD'12), ACM, 2012, pp. 239–250.

[17] Y. Zheng, D. Ansaloni, L. Marek, A. Sewe, W. Binder, A. Villazón, P. Tuma, Z. Qi, M. Mezini, Turbo DiSL: Partial Evaluation for High-Level Bytecode Instrumentation, in: Proceedings 50th International Conference on Objects, Models, Components, Patterns (TOOLS'12), volume 7304 of *LNCS*, Springer-Verlag, 2012, pp. 353–368.

[18] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, B. Wiedermann, The DaCapo benchmarks: Java benchmarking development and analysis, in: Proceedings 21st ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA'06).

[19] A. Sarimbekov, Y. Zheng, D. Ansaloni, L. Bulej, L. Marek, W. Binder, P. Tuma, Z. Qi, Productive development of dynamic program analysis tools with DiSL, in: Proceedings of 22nd Australasian Software Engineering Conference, pp. 11–19.

[20] A. Sewe, M. Mezini, A. Sarimbekov, D. Ansaloni, W. Binder, N. Ricci, S. Z. Guyer, new Scala() instance of Java: a comparison of the memory behaviour of Java and Scala programs, in: Proceedings of the 2012 international symposium on Memory Management, ISMM '12, ACM, New York, NY, USA, 2012, pp. 97–108.

[21] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, D. Lea, Java Concurrency in Practice, Addison-Wesley Longman Publishing Co., Inc., 2006.

[22] J. Gosling, B. Joy, G. Steele, G. Bracha, Java Language Specification, Addison-Wesley Longman Publishing Co., Inc., 3rd edition, 2005.

[23] M. Di Penta, R. E. K. Stirewalt, E. Kraemer, Designing your next empirical study on program comprehension, in: Proceedings 15th IEEE International Conference on Program Comprehension (ICPC'07), IEEE CS, 2007, pp. 281–285.

[24] K. O'Hair, HPROF: A Heap/CPU Profiling Tool in J2SE 5.0, 2004.

[25] Oracle Corp., JVM Tool Interface (JVMTI), version 1.2, 2007.

[26] A. Sarimbekov, A. Sewe, W. Binder, P. Moret, M. Mezini, JP2: Call-site aware calling context profiling for the Java Virtual Machine, Science of Computer Programming 79 (2014) 146 – 157.

[27] G. Ammons, T. Ball, J. R. Larus, Exploiting hardware performance counters with flow and context sensitive profiling, in: Proceedings 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'97), ACM, 1997, pp. 85–96.

[28] E. Bodden, K. Havelund, Racer: Effective Race Detection Using AspectJ, in: Proceedings 2008 International Symposium on Software Testing and Analysis (ISSTA'08), ACM, 2008, pp. 155–165.

[29] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, M. Mezini, Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders, in: Proceedings 33rd International Conference on Software Engineering (ICSE'11), ACM, 2011, pp. 241–250.

[30] A. Kinneer, M. B. Dwyer, G. Rothermel, Sofya: Supporting rapid development of dynamic program analyses for java, in: Companion Proceedings 29th International Conference on Software Engineering (ICSE'07), IEEE CS, 2007, pp. 51–52.

[31] C. Flanagan, S. N. Freund, The roadrunner dynamic analysis framework for concurrent programs, in: Proceedings 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'10), ACM, 2010, pp. 1–8.

[32] Naik, M., et al., Chord: A static and dynamic program analysis framework for Java, 2012.

[33] S. Chiba, K. Nakagawa, Josh: An Open AspectJ-like Language, in: Proceedings 3rd International Conference on Aspect-oriented Software Development (AOSD'04), ACM, 2004, pp. 102–111.

[34] B. Cornelissen, A. Zaidman, A. van Deursen, A controlled experiment for program comprehension through trace visualization, IEEE Transactions on Software Engineering 37 (2011) 341–355.

[35] R. Wettel, M. Lanza, R. Robbes, Software systems as cities: a controlled experiment, in: Proceedings 33rd International Conference on Software Engineering (ICSE'11), ACM, 2011, pp. 551–560.