Productive Development of Dynamic Program Analysis Tools with DiSL

Aibek Sarimbekov*, Yudi Zheng*, Danilo Ansaloni*, Lubomír Bulej†, Lukáš Marek†, Walter Binder*, Petr Tůma†, Zhengwei Qi‡

*University of Lugano, Switzerland
{firstname.lastname}@usi.ch
†Charles University, Czech Republic
{firstname.lastname}@d3s.mff.cuni.cz
‡Shanghai Jiao Tong University, China
{qizhwei}@sjtu.edu.cn

***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. The recently released Domain-Specific Language for Instrumentation (DiSL) was designed to boost the productivity of tool developers targeting the Java Virtual Machine, without impairing the performance of the resulting tools. DiSL offers high-level programming abstractions especially designed for development of instrumentation-based dynamic analysis tools. In this paper, we present a controlled experiment aimed at quantifying the impact of the DiSL programming model and high-level abstractions on the development of dynamic program analysis instrumentations. The experiment results show that compared with a prevailing, state-of-the-art instrumentation library, the DiSL users were able to complete instrumentation development tasks faster, and with more correct results.**

***Index Terms*—Dynamic program analysis, bytecode instrumentation, development productivity, controlled experiment**

I. 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], [2], debugging [3]–[6], and program comprehension [7], [8].

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 difficult and is usually performed using libraries such as BCEL [9], ASM [10], Soot [11], Shrike [12], or Javassist [13]. 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 low-level nature of the Java bytecode, the resulting code is often verbose, complex, and difficult to maintain or to extend.

The complexity associated with manipulating Java bytecode can be sometimes avoided by using aspect-oriented programming (AOP) 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 [14], the RacerAJ data race detector [15], or the Senseo Eclipse plugin for augmenting static source views with dynamic metrics [8], 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 [16].

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 [17], [18], 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 [8] to DiSL and ASM and compared the source code size and performance of the recasted instrumentations to the original. While the results suggested that, compared to ASM, DiSL indeed raised the abstraction level without impairing performance, the case study only covered a single 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 thus address the following research question:

RQ: Does DiSL improve developer productivity in writing instrumentations for DPA?

To answer this research question, we conduct a controlled experiment to determine if the use of DiSL instead of ASM increases developer productivity. Specifically, we measure how DiSL affects both the time needed to implement bytecode instrumentations and the correctness of the resulting instrumentations.

The remainder of the paper is structured as follows: Section II gives an overview of DiSL, focusing on key concepts needed to make this paper self-contained. Section III illustrates the conciseness and flexibility of the DiSL programming model compared to AspectJ, and ASM. Section IV introduces the experiment design, including task and subject descriptions. Section V presents the experiment results, followed by a discussion of the threats to validity of the study in Section VI. Section VII discusses related work, and Section VIII concludes the paper.

II. Background: DiSL Overview

While not an original contribution of this paper, this section provides the necessary background on DiSL to make the reader familiar with the key concepts. For a more detailed description we refer the reader to [17].

DiSL is a domain-specific language that provides developers of DPA tools with high-level concepts similar to those found 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., operand 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. Since guards can access any static context information, they provide a very powerful and efficient selection mechanism that avoids expensive runtime checks by performing static analysis at instrumentation time.

Snippets annotated with markers, scopes, and guards are collocated in a class referred to as DiSL *instrumentation*, which is similar to an *aspect* in AOP.

At a lower level, DiSL carefully avoids the shortcomings that usually result in low performance of AOP-based instrumentations. These concern efficient access to static and dynamic context information as well as efficient data passing between snippets.

In the case of static context, DiSL calculates the requested information on demand at instrumentation time and inlines the result directly into the code that requires it. In the case of dynamic context, DiSL provides a simple reflective API to access local variables and the operand stack, as well as an API to access the method arguments. Every use of those APIs within a snippet is replaced at instrumentation time with actual code that obtains the requested information at runtime.

For efficient sharing of data between snippets, DiSL provides synthetic local variables [19], and an implementation of thread-local variables that is more efficient than its counterpart in the Java Class Library.[1]

III. DiSL: Bridging Two Worlds

In this section, we illustrate the design goals of the DiSL framework on an example of a simple, but realistic tracing tool that is often used in the domain of dynamic program analysis. We first show how the AOP-inspired programming model of DiSL simplifies the instrumentation development by comparing alternative implementations of the same tool using AspectJ, DiSL, and ASM. Then we show that a slight, but common change in requirements makes the approach based on AspectJ unsuitable, which would normally force developers to adopt a more flexible (but usually lower-level) approach based on ASM, or other bytecode manipulation libraries. With DiSL, the developers can retain both the conciseness provided by a high-level AOP-based approach, and the flexibility provided by a low-level bytecode manipulation library.

[1]http://docs.oracle.com/javase/6/docs/api/java/lang/ThreadLocal.html.

```
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 onMethonEntry(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 development of a simple tracing tool starts with rather trivial requirements. On each method entry and method exit, the tool should output the full name of the method and its signature.

Considering the requirements, a tool developer familiar with AspectJ would just write a few lines of code to implement such a tool as an aspect, as shown in Figure 1. 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.

Developing the simple tracing tool using DiSL is very similar to AspectJ, with the DiSL-based implementation shown in Figure 2. Instead of advices, 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.

Comparing the tool implementations in AspectJ and DiSL, we can see much similarity between the two approaches, with very little added complexity. Both implementations are concise and easy to understand. The DiSL version is slightly more verbose in duplicating the `marker` and `scope` parameters, because the concept of annotated snippets makes it difficult to express related snippets in a simple way. This is a consequence of embedding the DiSL language in Java.

Now we consider implementing the tool using ASM, the de-facto standard library for bytecode manipulation. Users of ASM can choose between two different programming models depending on the API they use. The core (visitor) API provides a streaming, event-based programming model which requires users to implement visitors to perform bytecode transformations. The tree API provides a random-access programming model in which each class file is represented by nested collections of objects. The core API allows writing very efficient transformation tools, because it avoids building an object representation of a class. However, the programming model is difficult to grasp. The tree API provides higher level of abstraction, and is more convenient to use—and pays for it with longer transformation times. In situations, where the performance of the transformation itself is not important, the tree API is usually preferred.

The implementation of the tracing tool using ASM is shown in Figure 3. Even though we use the more convenient tree API, we notice a signicant drop in the abstraction level, followed by a significant increase in the amount of code that needs to be written. In short, the code iterates over all methods of a class, and inserts the instrumentation code at the beginning and at every exit point of the method code. The instrumentation code itself must be expressed as a sequence of bytecode instructions, and must be generated for each method. In Figure 3, we only show the part of the code responsible for inserting the instrumentation into method code. Additional code responsible for registering the Java transformer[2] and processing of the `HelloWorld` class was omitted for the sake of readability.

We note that the ASM-based code—if it was to be maintained—would have to be refactored, e.g., to eliminate the near-duplicate generation of code that prints the method name on method entry and method exit. The use JVM internal representation of Java types can be another source of errors—for instance, every reference type starts and ends with the "*L*" and "*;*" literals, and using the wrong type will cause the bytecode verifier to reject the code upon loading. A conscious developer will try to replace the string literals refering to Java types with class literals, and perform conversions on the fly—where possible. However, this is a recurring pattern with using low-level bytecode manipulation libraries—every developer has to come up with her own set of helper classes and abstractions to reduce the complexity of the code, and to make the actual

[2]http://docs.oracle.com/javase/6/docs/api/java/lang/instrument/ClassFileTransformer.html

```
for (MethodNode method : classNode.methods){
  if ((method.access & (Opcodes.ACC_NATIVE | Opcodes.ACC_ABSTRACT)) != 0)
    continue;

  InsnList instructions = method.instructions;
  String methodName = method.name + "." + method.desc;

  InsnList instrumentation = new InsnList();
  instrumentation.insert(new MethodInsnNode(Opcodes.INVOKEVIRTUAL,
    "java/io/PrintStream", "println", "(Ljava/lang/String;)V"));
  instrumentation.insert(new LdcInsnNode("On " + methodName + " method entry"));
  instrumentation.insert(new FieldInsnNode(Opcodes.GETSTATIC,
    "java/lang/System", "out", "Ljava/io/PrintStream;"));

  instructions.insert(instrumentation);

  instrumentation.clear();
  instrumentation.insert(new MethodInsnNode(Opcodes.INVOKEVIRTUAL,
    "java/io/PrintStream", "println", "(Ljava/lang/String;)V"));
  instrumentation.insert(new LdcInsnNode("On " + methodName + " method exit"));
  instrumentation.insert(new FieldInsnNode(Opcodes.GETSTATIC,
    "java/lang/System", "out", "Ljava/io/PrintStream;"));

  for (AbstractInsnNode instruction : instructions.toArray()) {
    int opcode = instruction.getOpcode();
    if (opcode >= Opcodes.IRETURN && opcode <= Opcodes.RETURN) {
      instructions.insert(instruction.getPrevious(), instrumentation);
    }
  }
}
```

Figure 3. Instrumentation part of a tracing tool implemented using ASM.

instrumentation code stand out from the low-level details.

However, even adherence to good coding practice cannot lift the limitation of the low-level code representation, where each method of a class consists of a set of instructions represented by `AbstractInsnNode` stored in a doubly linked list. As a result, any code modifications must be expressed as operations on the list of instructions. When inserting or removing instructions from the list, the developer has to be aware of the effects of each instruction on the JVM operand stack to avoid corrupting the stack at runtime. Developing, debugging, and maintaining code written using ASM is therefore a non-trivial task which requires the developer to possess advanced knowledge of JVM internals.

We now change the requirements on the tracing tool so that instead of method-level tracing, we now require tracing on the level of basic blocks. This requirement is quite common in DPA tools—it allows constructing basic-block profilers or code coverage analysis tools.

Unfortunately, we cannot use (pure) AspectJ to implement such a tool anymore, because AspectJ does not provide joint points at the level of basic blocks. Moreover, the join point model is closed, and cannot be easily extended—except by extending the language itself.

In case of ASM, the developer will have to implement the control flow analysis algorithm to first identify the basic blocks in method code, and then iterate over the basic blocks to insert the instrumentation code.

In contrast, the control flow analysis algorithm is already a part of DiSL, made available to the developer—the developer only needs to use it. Details have changed, but the programming model remains the same. Consequently, the implementation of the basic-block tracing tool shown in Figure 4 is very similar to the method-level tracing tool shown in Figure 2. The methods containing snippets have different names reflecting the different purpose, and we access a different kind of static context through the `bbsc` method parameter. Finally, the `marker` parameter in the annotations refers to the `BasicBlockMarker` class instead of the `BodyMarker` class, which causes DiSL to apply snippets at code locations corresponding to basic blocks, instead of method bodies.

To conclude—the design of DiSL attempts to merge the advantages of two different approaches for writing DPA tool instrumentations. DiSL extracts the complexity resulting from using low-level bytecode manipulation libraries into an extensible framework, but retains the flexibility that usually comes from using such libraries. By providing a high-level AOP-inspired programming model, DiSL allows expressing instrumentations in a concise manner. We therefore believe that using DiSL has a significant impact on productivity in development of DPA instrumetations, and we attempt to quantify the impact in the following section.

IV. Quantifying the Impact of DiSL

In this section we present the controlled experiment conducted to answer the 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

```
@Before(
  marker = BasicBlockMarker.class,
  scope = "*.HelloWorld.*")
void onBBentry(BasicBlockStaticContext bbsc) {
  System.out.println("On "+
    bbsc.getBBindex() +" basic block entry");
}

@After(
  marker = BasicBlockMarker.class,
  scope = "*.HelloWorld.*")
void onBBexit(BasicBlockStaticContext bbsc) {
  System.out.println("On "+
    bbsc.getBBindex() +" basic block exit");
}
```

Figure 4. Basic block profiling tool implemented using DiSL.

the experiment followed by a discussion of threats to validity of the study.

A. 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, 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 manipulate 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.

Table I
DESCRIPTION OF INSTRUMENTATION TASKS

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.

B. 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 I 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.

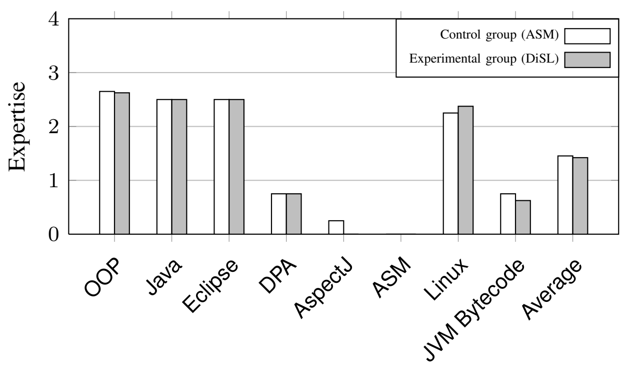

Figure 5. Distribution of the subjects among the control and the experimental groups. Expertise level 0 stands for no experience, while level 4 represents expert knowledge in the topic.

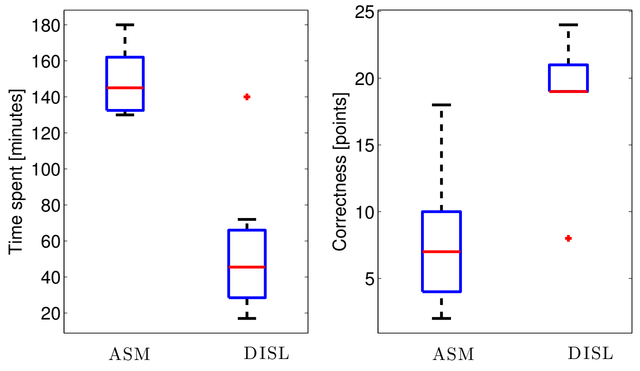

Figure 6. Box plots for development time spent and correctness of the tools. The red dot represents an outlier.

C. *Subjects and Experimental Procedure*

In total, we had 16 subjects—BSc., MSc., and PhD students from Shanghai Jiao Tong University—participate in the experiment. 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, as shown in Figure 5.

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 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]http://www.virtualbox.org/

D. *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% (α = 0.01) for all tests. All calculations were made using the SPSS statistical package.

V. Experimental Results

A. *Time results*

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 6.

To assess the significance of the result that DiSL has an impact on the time to write the instrumentations, 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 allows us to 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. The p-value of 0.001 is considerably lower than α = 0.01 (see Table II).

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.

B. Correctness results

As reported in Table II, the average amount of points scored by a subject in the experimental (DiSL) group is 46.6% higher than in the control (ASM) group. Such results are also presented as a box plot in Figure 6.

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 Table II) which is below α = 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.

VI. Threats to Validity

A. 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.

Table II
Descriptive statistics of the experimental results

	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	

B. External Validity

One can question the generalization of our results given the limited representativeness of the subjects and tasks. Even though the literature [20] 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.

C. 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 contolled 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. In the study presented here we did not consider the performance differences between the ASM and the DiSL code, therefore conducting an empirical study for observing such differences is another subject of the future work.

VII. Related Work

Java bytecode instrumentation is often performed using low-level bytecode manipulation libraries such as ASM [10], BCEL [9], Shrike [12], Soot [11], or Javassist [13]. 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 [21] 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, split, and routed to the analyses.

RoadRunner [22] 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 [23] 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 [17]. 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 [24] 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 [13]. Compared to DiSL, the join point model of Josh does not include arbitrary bytecodes and basic blocks of code.

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

Cornelissen et al. [25] 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. [8] 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 [26], 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.

VIII. Conclusions

The contribution of this paper is a thorough evaluation and assessment of DiSL [17], [18], a new abstraction layer on top of the well-known bytecode manipulation library ASM [10]. 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 implementation of any instrumentation-based dynamic analysis tool. In this paper we explored whether DiSL met this goal.

We conducted a controlled experiment to compare DiSL with ASM, measuring development time and correctness of the developed tools for six common dynamic analysis tasks. We showed that the use of DiSL reduced development time and improved tool correctness with both practical and statistical significance. We conclude that DiSL is a valuable abstraction layer on top of ASM which indeed succeeds in boosting the productivity of tool developers.

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

Acknowledgments

We thank Dimitri Igdalov and Olga Kaiser for their help with statistics, Vlastimil Babka and Michal Malohlava for conducting the pilot study, Matthias Hauswirth, Dmitri Makarov, Nathaniel Nystrom and Andreas Sewe for their feedback on a draft of this paper. We are grateful to the anonymous reviewers for their helpful comments. 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] M. Jovic, A. Adamoli, and M. Hauswirth, "Catch me if you can: performance bug detection in the wild," in *Proceedings of the 2011 ACM International Conference on Object oriented programming systems languages and applications (OOPSLA'11)*. ACM, 2011, pp. 155–170.

[2] W. Binder, J. Hulaas, P. Moret, and A. Villazón, "Platform-independent profiling in a virtual execution environment," *Software: Practice and Experience*, vol. 39, no. 1, pp. 47–79, 2009.

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

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

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

[6] F. Chen, T. F. Serbanuta, and G. Rosu, "jPredictor: A Predictive Runtime Analysis Tool for Java," in *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*. ACM, 2008, pp. 221–230.

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

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

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

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

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

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

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

[14] D. J. Pearce, M. Webster, R. Berry, and P. H. J. Kelly, "Profiling with AspectJ," *Software: Practice and Experience*, vol. 37, no. 7, pp. 747–777, June 2007.

[15] E. Bodden and K. Havelund, "Aspect-oriented Race Detection in Java," *IEEE Transactions on Software Engineering*, vol. 36, no. 4, pp. 509–527, 2010.

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

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

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

[19] W. Binder, D. Ansaloni, A. Villazón, and P. Moret, "Flexible and efficient profiling with aspect-oriented programming," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 15, pp. 1749–1773, 2011.

[20] M. Di Penta, R. E. K. Stirewalt, and 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.

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

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

[23] Naik, M., et al., "Chord: A static and dynamic program analysis framework for Java." [Online]. Available: http://pag.gatech.edu/chord/

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

[25] B. Cornelissen, A. Zaidman, and A. van Deursen, "A controlled experiment for program comprehension through trace visualization," *IEEE Transactions on Software Engineering*, vol. 37, pp. 341–355, 2011.

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