Science of Computer Programming 95 (2014) 344–358

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

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*

H I G H L I G H T S

- We have created the Domain-Specific Language for Instrumentation (DiSL).
- We assess whether DiSL boosts developer productivity.
- We perform a controlled experiment.

A R T I C L E I N F O

Article history:
Received 20 January 2014
Accepted 11 March 2014
Available online 8 April 2014

Keywords:
Dynamic program analysis
Bytecode instrumentation
Development productivity
Controlled experiment

A B S T R A C T

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.

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 [2–4], and program comprehension [5,6].

* Corresponding authors.
E-mail addresses: aibek.sarimbekov@usi.ch (A. Sarimbekov), yudi.zheng@usi.ch (Y. Zheng), danilo.ansaloni@usi.ch (D. Ansaloni), lubomir.bulej@usi.ch (L. Bulej), lukas.marek@d3s.mff.cuni.cz (L. Marek), walter.binder@usi.ch (W. Binder), petr.tuma@d3s.mff.cuni.cz (P. Tůma), qizhenwei@sjtu.edu.cn (Z. Qi).

http://dx.doi.org/10.1016/j.scico.2014.03.014
0167-6423/© 2014 Elsevier B.V. All rights reserved.

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.

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 recast the AOP-based instrumentation of Senseo [6] to DiSL and ASM and compared the source code size and performance of the recast 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 recast 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.

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